Methode und Technologie zur modellbasierten Automatisierung von Tests kontextsensitiver mobiler Anwendungen

D I S S E R T A T I O N

zur Erlangung des akademischen Grades eines

DOKTORS DER NATURWISSENSCHAFTEN
(Dr. rer. nat.)

durch die Fakultät für Wirtschaftswissenschaften der

Universität Duisburg-Essen
Campus Essen

vorgelegt von

Dipl.-Inf. Tobias Griebe
geboren am 16. Oktober 1981 in Strausberg (Deutschland)

Essen (2016)

Tag der mündlichen Prüfung:	25.04.2017
Erstgutachter:	Prof. Dr. Volker Gruhn
Zweitgutachter:	Prof. Dr. Tobias Hoßfeld

Zusammenfassung

Smartphones und Tablet Computer haben sich zu universalen Kommunikations- und Unterhaltungsplattformen entwickelt, die durch ständige Verfügbarkeit mobilen Internets die Verwendung mobiler, digitaler Dienste und Anwendungen immer mehr zur Normalität werden lassen und in alle Bereiche des Alltags vordringen. Die digitalen Marktplätze zum Vertrieb von Apps, sogenannten App Stores, sind Blockbuster-Märkte, in denen wenige erfolgreiche Produkte in kurzen Zeitintervallen den Großteil des Gesamtgewinns des Marktes erzielen. Durch dynamische, summative Bewertungssysteme in App Stores wird die Qualität einer App zu einem unmittelbaren Wert- und Aufwandstreiber. Die Qualität einer App steht in direktem Zusammenhang mit der Anzahl Downloads und somit mit dem wirtschaftlichen Erfolg.

Mobile Geräte zeichnen sich gegenüber Desktop-Computern vorrangig dadurch aus, dass sie durch Sensoren in der Lage sind, Parameter ihrer Umgebung zu messen und diese Daten für Anwendungsinhalte aufzubereiten. Anwendungsfälle für solche Technologien sind beispielsweise ortsbasierte digitale Dienste, die Verwendung von Standortinformationen für Fahrzeug- oder Fußgängernavigation oder die Verwendung von Sensoren zur Interaktion mit einer Anwendung oder zur grafischen Aufbereitung in Augmented Reality-Anwendungen.

Anwendungen, die Parameter ihrer Umgebung messen, aufbereiten und die Steuerung des Kontrollflusses einfließen lassen, werden als kontextsensitive Anwendungen bezeichnet. Kontextsensitivität hat prägenden Einfluss auf die fachliche und technische Gestaltung mobiler Anwendungen. Die fachliche Interpretation von Kontextparametern ist ein nicht-triviales Problem und erfordert eine sorgfältige Implementierung und gründliches Testen.

Herausforderungen des Testens kontextsensitiver, mobiler Anwendungen sind Erstellung und Durchführung von Tests, die zum einen die zu testende Anwendung adäquat abdecken und zum anderen Testdaten bereitstellen und reproduzierbar in die zu testende Anwendung einspeisen. In dieser Dissertation wird eine Methode und eine Technologie vorgestellt, die wesentliche Aspekte und Tätigkeiten des Testens durch modellbasierte Automatisierung von menschlicher Arbeitskraft entkoppelt. Es wird eine Methode vorgestellt, die Tests für kontextsensitive Anwendungen aus UML-Aktivitätsdiagrammen generiert, die durch Verwendung eines UML-Profils zur Kontext- und Testmodellierung um Testdaten angereichert werden. Ein Automatisierungswerkzeug unterstützt die Testdurchführung durch reproduzierbare Simulation von Kontextparametern.

Durch eine prototypische Implementierung der Generierung von funktionalen Akzeptanztests, der Testautomatisierung und Kontextsimulation wurde Machbarkeit des vorgestellten Ansatzes am Beispiel der mobilen Plattform Android praktisch nachgewiesen.

Inhaltsverzeichnis

Kapitel 1

Einleitung

Die Entwicklung mobiler Plattformen hat in den vergangenen Jahrzehnten eine beispiellose Entwicklung durchlebt. Mobiltelefone haben sich in Form von Smartphones und Tablet-Computern zu universalen Kommunikations- und Unterhaltungsplattformen entwickelt, die in alle Bereiche des Alltags vordringen (vgl. Anand et al. [8]). Die ständige Verfügbarkeit mobiler Internetverbindung in Kombination mit den regelmäßig nah am Körper getragenen mobilen Geräten lassen die Verwendung mobiler, digitaler Dienste und Anwendungen immer mehr zur Normalität werden. Ichikawa et al. [191] sowie Böhmer et al. [52] wiesen in empirischen Studien aus den Jahren 2005 bzw. 2011 nach, dass vom *Causual Game* (engl. Gelegenheitsspiel) bis hin zur Altersvorsorge kaum Lebensbereiche existieren, die nicht durch mobile *Apps* berührt werden. Mittlerweile sind Apps zu einem wichtigen Wirtschaftsfaktor geworden (vgl. Schönberger [316]) und die Entwicklung und der Betrieb von Apps ist für viele Unternehmen und Systemhäuser Alltagsgeschäft.

Mobile Geräte, insbesondere Smartphones und Tablet-Computer (Tablets), unterscheiden sich in einigen Aspekten stark von Desktop-Geräten. Unterscheidungsmerkmale sind hierbei nicht nur auf technische Eigenschaften oder die Gerätebauform beschränkt. Von Bedeutung ist vielmehr die Einbettung von Software (sogenannten Apps) in das Gerät auf der einen Seite und die Einbettung des Geräts in eine vom Plattformbetreiber vorgegebene Infrastruktur auf der anderen Seite. Aus diesen Merkmalen ergeben sich insbesondere für das Testen mobiler Anwendungen neue Herausforderungen, die bereits vielfältig zum Gegenstand wissenschaftlicher Untersuchungen geworden sind (vgl. Grønli et al. [158] und Haller [164]).

Viele für das Testen mobiler Anwendungen relevante Parameter entziehen sich dem direkten Zugriff von Entwicklern und Testern. Ursache hierfür ist einerseits, dass Komponenten mobiler Anwendungen häufig stark in die Infrastruktur von Plattformbetreibern eingebunden sind und andererseits die unscharf verlaufende Grenze zwischen App, Betriebssystem und Betreiberinfrastruktur aus der Perspektive funktionaler Anwendungsstests. Beispiele hierfür sind neben *Push*-Benachrichtigungs-Diensten oder Bezahlfunktionen innerhalb einer App, die eine Monetarisierung von App-Inhalten nach dem initialen *Rollout* (engl. Markteinführung) ermöglichen, insbesondere solche Komponenten, die besondere Geräte-Hardware wie z. B. das *Global Positioning System* (GPS) Modul oder Sensoren des Geräts verwenden. Solche Komponenten liegen außerhalb des Einflusses von Entwicklern bei der Erstellung von Apps, so dass sie nicht zu Testzwecken manipuliert werden können.

Anbieter von Apps sind aufgrund der Heterogenität mobiler Geräte und der Unzugänglichkeit vieler Aspekte mobiler Plattformen gezwungen, stichprobenartig manuelle Tests durchzuführen und das Softwareprodukt dann in den *App Stores* (digitale Marktplätze zum Softwarevertrieb, insb. Apps) anzubieten, ohne dass Tests in repräsentativem Umfang durchgeführt wurden. Hierzu wären Tests mit einer an der jeweils aktuellen Verteilung des Marktes ausgerichteten Auswahl mobiler Geräte erforderlich[1]. Weiterhin ist eine geeignete geografische Verteilung entsprechend des Zielmarktes notwendig, die zudem eine durch die Zielgruppe bestimmte Auswahl an Plattformen abdeckt. Um diese Bedingungen an repräsentative Tests zu erfüllen, entstehen hohe Kosten (vgl. Hoffman [183], Karhu et al. [212], Dantas et al. [87]).

Andere für das Testen relevante Parameter der Betriebsumgebung einer mobilen Anwendung liegen zwar im Einflussbereich eines Softwareentwicklers oder Testers, sind aber aufgrund technischer Beschränkungen der Softwarentwicklungs- und Testwerkzeuge für mobile Plattformen nur mit erheblichem Aufwand in Testfällen zu berücksichtigen. Hierzu gehören u. a. auch situative Faktoren des Anwendungsbetriebs oder des Anwenders. Diese bilden den sogenannten Kontext der Anwendnungsnutzung (vgl. Begriffsdefinition Kontext in Abschnitt 3.1.1.4) und haben einen großen Einfluss auf Testaktivitäten. Der Kontext kann unterschiedliche Parameter von einfachen Sensormesswerten bis hin zu komplexen Situationsbeschreibungen umfassen. Denkbar sind beispielsweise Standortinformationen, Sensormesswerte oder komplexe Zusammenhänge, die aus mehreren atomaren Messwerten abgeleitet werden. Die Erfassung von kontextorientierten Signalen kann durch Sensoren innerhalb eines Geräts erfolgen (z. B. Beschleunigungssensor, Magnetfeldsensor), durch Berechnung aus extern bereitgestellten Signalquellen (z. B. Standortbestimmung via GPS, Informationserhebung via *Near-Field-Communication* (NFC), usw.) oder durch Algorithmen die Informationen aus Backend-Systemen, ggf. unter Berücksichtigung weiterer Anwender, errechnen.

Je nach implementiertem Anwendungsfall kann der Kontext beliebig komplex werden, wodurch die Erstellung von Testfällen und insbesondere deren Durchführung einen sehr großen manuellen Aufwand verursacht. Dieser Aufwand steht der Absicht der Gewinnerzielung von Softwareproduzenten entgegen. Die Programmierung von Apps erfolgt unter einem hohen Konkurrenzdruck, der durch kurze Versionszyklen mobiler Plattformen zusätzlich verschärft wird. Im Marktsegment mobiler Apps kann Gewinn i. d. R. nicht durch einen hohen Einzelpreis pro Verkaufseinheit erzielt werden, sondern lässt sich nur durch eine große Anzahl verkaufter Produkteinheiten realisieren. Hierzu ist es notwendig, den Anforderungen der Zielgruppe mit hoher Produktqualität gerecht zu werden und in kurzer Zeit auf Fehlentwicklungen (d. h. negative App-Bewertungen in den App Stores) zu reagieren (vgl. Khalid Khalid [220], Khalid et al. [222, 221], Haller [164]). Grundlage hierfür sind effiziente Softwareprozesse, die Produktinkremente in kurzen Zyklen ermöglichen.

Eine Möglichkeit, um Softwareprozesse effizienter zu gestalten, ist die Reduzierung manuell zu erbringender Tätigkeiten durch Automatisierung (vgl. Wiklund et al. [363]). Insbesondere im Umfeld mobiler Anwendungen entsteht im Bereich Qualitätssicherung gegenwärtig ein hoher manueller Arbeitsaufwand für die Erstellung und Durchführung von Softwaretests. Ein

[1] Eine solche Analyse kann beispielsweise auf der Grundlage der von OpenSignal Inc. [271] ermittelten Daten erfolgen. OpenSignal Inc. analysieren die weltweite geographische Abdeckung mit 3G und 4G Netzwerken, wobei zusätzlich eine geographische Zuordnung der Verteilung des Marktes mobiler Geräte auf Hersteller und konkrete Gerätemodelle erzeugt wird.

Grund hierfür ist die nur rudimentär ausgebildete Unterstützung von Aktivitäten der Qualtitätssicherung in *Software Development Kits* (SDKs) mobiler Plattformen, die insbesondere Kontextparameter weitgehend unberücksichtigt lassen (vgl. Zhang und Adipat [376]).

In dieser Dissertation wird untersucht, ob Automatisierungstechnologien Aktivitäten der Qualitätssicherung für mobile Anwendungen unter technischen und wirtschaftlichen Aspekten unterstützen können. Hierzu wird die Problemstellung zunächst in Abschnitt 1.1 unter Berücksichtigung der zum Zeitpunkt der Erarbeitung dieser Dissertation verfügbaren Technologie zur Softwareentwicklung für mobile Geräte motiviert und die Praxisrelevanz des Testens mobiler Anwendungen basierend auf dem technischen Entwicklungsstand mobiler Plattformen dargestellt. Insbesondere wird die Bedeutung von Mobilität und Kontextsenitivität für die Entwicklung und für das Testen mobiler Software herausgestellt. Abschnitt 1.2 beleuchtet die Problemstellung des Testens mobiler, kontextsensitiver Software im Detail und formuliert die konkret bearbeitete Forschungshypothese, gefolgt von der Darlegung des untersuchten Lösungsansatzes zur Reduzierung manueller Arbeit durch Verwendung von Automatisierungstechnologie zum Erstellen und Durchführen von Tests für mobile Software.

1.1 Motivation

Bis vor wenigen Jahren unterlag die Entwicklung von Software für mobile Plattformen nichtfunktionalen Anforderungen hinsichtlich der Leistungsfähigkeit dieser Geräte, wie etwa Rechenleistung, Speicherkapazität und Netzwerkanbindung. Weitere limitierende Faktoren waren die eingeschränkte Displaygröße und schwach ausgeprägte Interaktionsmöglichkeiten mit dem Gerät (vgl. Joorabchi et al. [204]). Erst die sich über die gesamte Gerätevorderseite erstreckenden berührungsempfindlichen Displays separieren Smartphones endgültig von mobilen Geräten der Prä-Smartphone-Ära. Einschränkungen dieser Art sind mit der Entwicklung neuer Generationen von Smartphones weitgehend obsolet geworden. Einerseits ist die Leistungsfähigkeit aktueller Geräte signifikant gestiegen, so dass Geräteeigenschaften wie Speicherkapazität oder Rechenleistung nicht länger limitierende Faktoren in der Anwendungsentwicklung für mobile Geräte sind. Ganz im Gegenteil sind die Leistungsfähigkeit und die Vielseitigkeit aktueller Gerätegenerationen in Verbindung mit dem ubiquitären Internet zum treibenden Faktor in der Entwicklung neuartiger Dienstleistungen und Softwareanwendungen geworden. Zudem haben sich zeitgleich auch der zumindest in Ballungsgebieten flächendeckende Ausbau des drahtlosen Internets und die Entwicklung sozialer Netzwerke zu einem Faktor entwickelt, der auf mobil verfügbarem Internet basierende Geschäftsmodelle ermöglicht. Unternehmen aus allen Marktsektoren sind nicht nur in der Lage, sondern geradezu in der Pflicht, Software und Dienstleistungen auf mobilen Plattformen anzubieten, um Konkurrenzfähigkeit und Marktstellung langfristig zu sichern.

Mit dem technischen Fortschritt der Geräte-Hardware haben sich ebenfalls neuartige Vertriebs- und Distributionsmodelle für Software entwickelt. Anbieter von Geräteplattformen bieten neben fortschrittlichen SDKs integrierte Vertriebsplattformen an, die es Softwareentwicklern ermöglichen, Anwendungen dem Markt zugänglich zu machen, ohne hierfür eigene Vertriebs- und Distributionsinfrastrukturen zu betreiben. Die bisher den Betreibern mobiler Plattformen oder hochspezialisierten Anbietern vorbehaltene Entwicklung mobiler Anwen-

dungen (vgl. Holzer und Ondrus [186]) ist aufgrund dieses Angebots der Allgemeinheit ohne wesentliche Barrieren zugänglich. Jeder Entwickler, der über entsprechende Sachkenntnis verfügt, ist in der Lage, Anwendungen für mobile Systeme zu entwickeln und zu vertreiben. Diese wesentlich vereinfachte Entwicklung von Anwendungen für mobile Systeme bei gleichzeitiger Konvergenz von Eigenschaften der Geräte-Hardware ermöglicht es, traditionelle Methoden des Software Engineering aus der Server- und Desktopumgebung auf die Entwicklung mobiler Systeme zu übertragen.

Der weitgehende Entfall technischer Barrieren bei der Entwicklung mobiler Anwendungen sowie die praktisch unbegrenzte Zugänglichkeit von SDKs bei gleichzeitig stark anwachsender Nachfrage mobiler Anwendungen machen deren Entwicklung und Betrieb unter wirtschaftlichen Aspekten attraktiv (vgl. Abschnitt 1.1.2). Einerseits existiert ein Markt mit einer breiten Zielgruppe deren Angehörige unabhängig von Ort und Zeit ständig erreichbar sind. Andererseits steigt der wirtschaftliche Druck auf Unternehmen Anwendungen in kurzer Zeit und mit geringem Aufwand zu entwickeln und am Markt zu platzieren. Hierbei spielen sozioökologische Seiteneffekte ebenfalls eine Rolle. Die Präsenz in den einschlägigen Distributionsplattformen der jeweiligen Anbieter ist zum wichtigen Marketingfaktor geworden (vgl. Berger [39], Kotler et al. [230]). Beispielsweise wird durch geschickte Platzierung von Apps (u. U. auch fachfremd oder ohne differenzierten, domänenspezifischen Nutzinhalt versucht, Nähe zur Zielgruppe von Produkten aufzubauen. Ebenso fließen Aktivitäten von Konkurrenzunternehmen in die Entscheidung für die Entwicklung mobiler Anwendung ein (*"Me Too!"* als Marketingstrategie, vgl. Beckschebe [36], Lukas und Ferrell [241]). Aufgrund der ausgeprägten Konkurrenzsituation des Marktes (vgl. Abschnitt 1.1.2) steigen die Qualitätsansprüche. Die durch dynamische und soziale Faktoren geprägte Vertriebsstruktur mobiler Apps diktiert besondere Qualitätsanforderungen. Mangelhafte Produktqualität gefährdet nicht nur die Rentabilität individueller Produkte, sondern die Reputation des Entwicklers bzw. Betreibers und somit das Erfolgspotenzial aller zukünftigen Produkte (vgl. Kim et al. [223], Khalid [220]).

Mobile Systeme weisen jedoch eine Reihe technischer Besonderheiten auf, die qualitätssichernde Maßnahmen solcher Anwendungen im Vergleich zu nicht-mobilen Anwendungen stark verkomplizieren. Diese Eigenschaften werden unter anderem durch Mobilität der Anwendungen, Fragmentierung der Gerätelandschaft (vgl. Joorabchi et al. [204]) sowie durch die ausgeprägten externen Abhängigkeiten dieser Systeme charakterisiert. Zu diesen Abhängigkeiten gehören unter anderem die Verfügbarkeit von Netzwerkdiensten aber auch situative Faktoren, durch welche die Ausführungsumgebung der Anwendung wesentlich geprägt ist.

Neben Kontextinformationen, die über Netzwerkdienste erhoben werden, zählen hierzu ebenfalls Messdaten lokaler Sensoren des Geräts (Netzwerkverfügbarkeit, Standortinformation, Beschleunigungssensor, Magnetfeldsensor usw.). Diese technischen Möglichkeiten charakterisieren das Funktionsspektrum mobiler Anwendungen signifikant. Obgleich sie wesentliche Attraktoren bei der Entwicklung mobiler Anwendungen sind (z. B. ortsbasierte Dienstleistungen), erschweren diese Technologien die Qualitätssicherung in besonderem Maße, da sie nicht nur das Funktionsspektrum von Apps erweitern, sondern auch neue Fehlerquellen einführen und die Komplexität aller Qualitätssicherungsaktivitäten stark erweitern (vgl. Kapitel 3).

Die mit den plattformspezifischen SDKs ausgelieferten Testwerkzeuge sind i. d. R. nicht ausreichend, um die alleinstellenden Faktoren mobiler Geräte (mobiler Netzwerkzugriff, mul-

timodale Interaktionsmodi, Sensoren zur Messung physikalischer Parameter der Ausführungsumgebung) adäquat abzudecken. Insbesondere der Bereich des Testens sensorbasierter Anwendungen wird von gegenwärtig verfügbarer Technologie nur rudimentär unterstützt, so dass ein großer Teil der technischen Möglichkeiten aktueller Smartphone-Generationen nur durch manuelle Testprozesse bedient werden kann. Manuell durchgeführte Testaktivitäten, von der initialen Testplanung bis hin zur eigentlichen Testdurchführung, sind jedoch durch hohe Kosten geprägt. Nach einer Studie von Tassey [341] aus dem Jahr 2002 kann der für das Testen aufgewendete Arbeitsanteil bereits in Softwareprojekten, in denen Mobilitätsanforderungen keine Kosten- und Aufwandstreiber sind, mehr als 50% betragen. Insbesondere das Testen orts- oder sensorbasierter Anwendungen erfordert bei einer vollständig manuellen Durchführung das Aufsuchen testrelevanter Orte (vgl. Muccini et al. [258]). Analog erfordert manuelles Testen sensorbasierter Anwendungen ein planmäßiges und reproduzierbares Herstellen von Eigenschaften der Ausführungsumgebung (vgl. Broens und van Halteren [59]).

Das Aufsuchen testrelevanter Orte (unter technologisch bedingten Einschränkungen hinsichtlich der Präzision der Standortinformation) ist unter Inkaufnahme hoher Kosten – insbesondere hoher Personalkosten – grundsätzlich möglich (vgl. Agarwal et al. [5]). Die Situation stellt sich für andere physikalische Kontextfaktoren, wie etwa Beschleunigung oder Magnetfeldstärke, jedoch grundsätzlich anders dar. Ohne die Verwendung hochspezialisierter Laborumgebungen ist es praktisch nicht möglich, physikalische Parameter der Betriebsumgebung exakt zu reproduzieren. Der Betrieb einer solchen Laboreinrichtung ist jedoch i. d. R. nicht rentabel, da hierzu neben einer entsprechenden Laborausrüstung ebenfalls Personal mit sachdienlicher Kompetenz vorgehalten werden muss. Dieses wirtschaftliche Risiko kann durch Softwareunternehmen im Normalfall nicht getragen werden, wenn das spezialisierte Testen sensorbasierter Anwendungen nicht Kerngeschäftsfeld ist.

Die Notwendigkeit Softwaresysteme außerhalb einer Laborumgebung zu testen wurde bereits von Rooksby et al. [304] diskutiert. Die Autoren untersuchten die Testpraxis in unterschiedlichen Softwareprojekten und erkannten Mängel in Testprozessen, die auf unzweckmäßige Teststrategien zurückzuführen sind. Zu diesen gehört u. a., Softwaretests nur in Laborumgebungen von den Entwicklern des Systems durchführen zu lassen.

Alternativ kommt nur Technologie in Frage, die direktes Manipulieren von Sensoren mobiler Geräte direkt (auf Hardware-Ebene) oder indirekt (durch Eingriffe in Software) ermöglicht. Die Verwendung von Automatisierungstechnologie stellt durch Verbesserung von Qualitätssicherungsprozessen deshalb eine mögliche Alternative dar, um die Qualität von Software zu erhöhen. Automatisierungstechnologie, insbesondere solche, die möglichst nahe an den Entwicklungswerkzeugen für mobile Plattformen angelegt ist, könnte sicherstellen, dass bestimmte Eigenschaften der Betriebsumgebung (d. h. der Kontext) eines *System Under Test* (SUT) reproduzierbar während der Testdurchführung hergestellt werden können. Dies kann beispielsweise realisiert werden, indem bestimmte Systemkomponenten (z. B. das *Application Programming Interface* (API) einer mobilen Plattform zum Zugriff auf Sensoren) während des Testens künstlich stimuliert werden. Weiterhin bietet Automatisierungstechnologie die Möglichkeit, die Kosten des Testens durch Entkopplung von menschlicher Arbeitskraft zu senken. Zwar verursacht die Einführung und Instandhaltung sowie der Betrieb von Testautomatisierungstechnologie in Softwareprojekten zunächst zusätzliche Kosten, jedoch stellt sich

Rentabilität bereits nach wenigen Zyklen der iterativen Softwareentwicklung ein. Nach einer Studie von Berner et al. [40] aus dem Jahr 2005 ist der Einsatz von Testautomatisierung bereits dann lohnenswert, wenn ein einzelner Test häufiger als zehnmal durchgeführt wird. Einerseits können durch Automatisierungstechnologie die effektiven Durchführungskosten direkt gesenkt werden, andererseits wird die ansonsten in manuellen Testaktivitäten gebundene menschliche Arbeitskraft frei und kann an anderer Stelle profitabel in den Softwareprozess reinvestiert werden, beispielsweise in die qualitative Entwicklung weiterer Testfälle.

Insbesondere in der Entkopplung der Testdurchführung von menschlicher Arbeitskraft liegt ein herausragender Vorteil der Verwendung von Automatisierungswerkzeugen gegenüber manuellen Tätigkeiten. Nach einer Studie von Baur und Groth [30] aus dem Jahr 2013 zu den Folgen psychischer Belastungsfaktoren am Arbeitsplatz repräsentieren Arbeitssituationen, die einerseits ein vollständiges und abschließendes Bearbeiten einer Tätigkeit nur schwer zulassen, andererseits aber nur wenig Raum zur gedanklichen Auseinandersetzung mit der Tätigkeit erlauben einen reizarmen Zustand der Monotonie. Er wird ausgelöst durch gleichförmige Anforderungen an eine sich wiederholende Tätigkeit, die nur wenig Entscheidungsspielräume und geringen Anreiz zur intellektuellen Entwicklung der Tätigkeit bieten. Das Testen von Softwareprodukten erfüllt diese Kriterien in allen Phasen des Softwareprozesses, in denen bei Aktivitäten der Qualitätssicherung auf menschliche Arbeitskraft zurückgegriffen wird. Die Folgen erstrecken sich laut Baur und Groth [30] nicht nur unmittelbar auf den Akteur, der seine Tätigkeit als zunehmend frustrierend wahrnimmt, sondern ebenfalls auf wirtschaftliche Konsequenzen, wie etwa Herabsetzung der Arbeitsproduktivität und der Qualität der erzielten Ergebnisse. Maschinen sind deshalb bei der Durchführung sich wiederholender Aufgaben regelmäßig kostengünstiger und insbesondere zuverlässiger als menschliche Akteure, die bei monotonen Aufgaben zu ermüdungsbedingten Fehlern neigen.

Roman et al. [303] identifizierten und prognostizierten in ihrer Arbeit aus dem Jahr 2000 eine Reihe von Herausforderungen an das Software Engineering für mobile Systeme. In einer kritischen Revision dieser Arbeit reflektieren die Autoren in Picco et al. [282] aus dem Jahr 2014 diese Prognosen und kommen zu dem Schluss, dass einige der Herausforderungen aus den frühen Anfängen der Smartphone-Ära durch technologische Entwicklungen obsolet geworden sind. Einige der Herausforderungen haben jedoch auch im Zeitraum 2000 - 2014 nicht an Aktualität verloren. Hierzu gehören unter anderem die softwaretechnischen Implikationen physikalischer Mobilität und das Konzept des Kontexts, welches alle direkt oder indirekt messbaren Größen der Betriebsumgebung eines Softwareproduktes vereint. Insbesondere Technologien, die den Kontext eines Systems in der Modellierung berücksichtigten, verbleiben nach Picco et al. [282] als bedeutende ungelöste Herausforderungen des Software Engineering für mobile Systeme. Insbesondere wird in [282] die Bedeutung des Standortes des Anwenders als Schlüsselinformation in der Realisierung digitaler Dienstleistungen hervorgehoben. Neben anderen Aufgaben des ingenieursmäßigen Erstellens von Software, wie etwa Modellierung eines Systems, identifizieren Picco et al. [282] insbesondere das Testen mobiler Softwaresysteme als wesentliche Herausforderung.

Laut einer im Jahr 2013 von Joorabchi et al. durchgeführten Studie [204] gehören die Fragmentierung der Gerätelandschaft, die unzureichende Unterstützung für Monitoring, Analyse und Test, Pro- und Kontrafaktoren von *Open-Source-Software* im Vergleich mit *Closed-Source-*

Software, zunehmendes Datenaufkommen in mobilen Apps, schnelle Versionszyklen mobiler Plattformen sowie technische Herausforderungen von *Cross-Platform*-Technologie weiterhin zu den signifikanten Herausforderungen des Software Engineering für mobile Systeme. Insbesondere die Herausforderungen des Testens sind nach Joorabchi et al. und Picco et al. [282] sowohl in der technischen als auch der ökonomischen Perspektive besonders signifikant. Zum Zeitpunkt der Studie wurden laut Joorabchi et al. noch mehr als 60% aller Softwaretests manuell durchgeführt, bei ca. 30% der Befragten kommen hybride Ansätze zwischen manuellem und automatisiertem Testen zum Einsatz und nur ca. 3% stützen sich überwiegend auf Testautomatisierung. Hinzu kommt, dass bei über 80% der 188 von Joorabchi et al. befragten Unternehmen Aufgaben des Testens von Entwicklern übernommen werden, während in traditionellen Softwareprozessen dedizierte Testabteilungen Standard sind. Hierdurch werden wesentliche Kapazitäten dieser Entwickler in Aktivitäten gebunden, die nicht der unmittelbaren inhaltlichen Weiterentwicklung einer App dienen. Eine weitere Hürde ist die Notwendigkeit, Apps aufgrund fehlender Automatisierungstechnologie separat pro Plattform und ggf. geographisch verteilt zu testen.

Die Mobilisierung von IT-Technologie in Form von Smartphones und Tablets sowie die voranschreitende Mobilisierung von IT-Produkten stellt die Industrie vor neue Herausforderungen. Die bisherigen Anforderungen an Softwarequalität werden durch Mobilität (vgl. Begriffsdefinition Mobilität in Abschnitt 3.1.1.1) und Kontextsensitivität (vgl. Begriffsdefinition Kontextsensitivität in Abschnitt 3.1.1.5) ergänzt. Die Möglichkeit, den gegenwärtigen Standort des Nutzers und Kontext in Softwareprodukten und Dienstleistungen zu berücksichtigten, schlägt sich einer erheblichen Erhöhung der Komplexität von Testprozessen nieder. Tests für mobile, kontextsensitive Software müssen den Standort des Nutzers und andere Kontextfaktoren berücksichtigen. Das heißt insbesondere, dass sowohl manuelle Tests als auch durch Testautomatisierungstechnologie durchgeführte Tests diese Faktoren einerseits in Testmodellen abbilden müssen und andererseits, dass bei der Testausführung so auf das SUT eingewirkt werden muss, dass sich die Testumgebung aus der Sicht des SUT entsprechend den im Testmodell fixierten Kontextinformationen darstellt. Im Vergleich zur Geschwindigkeit in welcher sich mobile IT weiterentwickelt, verläuft die Entwicklung adäquater Testtechnologien langsamer. Methoden und Werkzeuge zum Testen werden i. d. R. als Reaktion neuer Technologien entwickelt und nur selten nach dem Paradigma *Design for Testability* (engl. sinngemäß auf Testbarkeit optimierter Entwurf, vgl. Pettichord [281]). Die Anbieter mobiler Kommunikationsgeräte ziehen sich auf die Rolle von Hardware- und Plattformanbietern zurück. Die Entwicklung von Software, insbesondere von Apps, fällt durch geöffnete Vertriebskanäle App-Entwicklern zu, die auch aus anderen Industriezweigen stammen können.

Eine mögliche Lösung die Problemstellung des Testens mobiler Anwendungen zu adressieren, ist die Ablösung menschlicher Arbeitskraft durch Automatisierungstechnologie in geeigneten Phasen des Softwareprozesses. Nach Berner et al. [40] sind Tests, die schwer manuell durchzuführen sind, meistens auch schwer automatisierbar. Dies liegt hierin begründet, dass sich besonders mehrwertstiftende Aktivitäten des Testens, wie z. B. die Erstellung adäquater Testfälle, nur in geringem Umfang von gegenwärtiger Technologie unterstützen lassen.

In dieser Dissertation wird untersucht, wie Technologien des *Model Driven Testing* (MDT) angewendet werden können, um auch Aktivitäten des Testens jenseits der bloßen Testdurch-

führung zu automatisieren. Im konkreten Lösungsansatz werden Modelle des Systementwurfs, die dynamische Aspekte des SUT abbilden, als Grundlage zur Testfallerstellung verwendet. Hierzu werden diese Modelle durch eine Erweiterung der Modellierungssprache *Unified Modeling Language* (UML) um testrelevante Informationen ergänzt und durch ein Werkzeug in automatisierbare Testfälle überführt. Im Besonderen wird ein Konzept untersucht, wie kostenintensive manuelle Tätigkeiten der Qualitätssicherung bei der Entwicklung von Anwendungen für mobile Plattformen kostensenkend weitgehend von menschlicher Arbeitskraft entkoppelt werden können. Zu automatisierende Prozesse erstrecken sich hierbei einerseits auf die Generierung von Testfällen aus Dokumenten des Systementwurfs und andererseits auf die automatisierte Testdurchführung.

1.1.1 Status Quo mobiler Plattformen und Softwaresysteme

Telefonie hat die Kommunikation der Menschen nachhaltig verändert. Wenngleich die traditionelle kabelgebundene Telefonie noch nicht völlig aus dem Alltag verschwunden ist, ist ein deutlicher Trend zur Ablösung kabelgebundener Kommunikationstechnologien erkennbar. Beispielsweise wird gegenwärtig bei der Erschließung ländlicher Regionen der Mobilfunktechnik *Long Term Evolution* (LTE) der Vorzug vor kabelgebundenen Technologien wie etwa DSL oder Glasfaserleitungen gegeben. Seit der Erfindung der Telefonie durch Graham Bell und Phillip Reis (vgl. Thompson [343], Graham [155]) ist die Geschichte der Telekommunikation durch sprunghafte Technologieentwicklungen und Paradigmenwechsel geprägt. Die Ablösung des Telegraphen durch das Telefon hat erstmals die Übertragung von Sprache über weite Entfernungen ermöglicht. Die Ablösung der Leitungsvermittlungstechnologie durch Paketvermittlungstechnologie in der Digitalisierung der Telefonie ermöglichte eine effizientere Nutzung von Ressourcen und eine weitgehende Abstraktion von der physikalischen Topologie eines Leitungsnetzes. Die Ablösung kabelgebundener Telefonie durch drahtlose Mobilfunktechnologie stellte einen weiteren markanten Schritt der Entwicklung der Telefonie dar, ohne den heutige Smartphones und Tablets nicht denkbar wären. Während dieser Phasen der technologischen Entwicklung von der Erfindung des elektrischen Telegraphen hin zur Entwicklung des ersten Mobilfunktelefons im Jahr 1973 (vgl. Teixeira [342]) stand zunächst die Weiterentwicklung der Technologie zur Übertragung von Sprache im Vordergrund. Die seit den 1970er Jahren parallel verlaufende Entwicklung des Internets sowie der Entwicklungsfortschritt der Computertechnologie hat dazu geführt, dass diese drei Technologien zunehmend miteinander verschmelzen und eine Vielzahl von Anwendungsfällen ermöglichen, die über die Telefonie weit hinausgehen.

Mit den anwachsenden technischen Möglichkeiten hat sich die Gerätelandschaft seitdem vom Mobiltelefon hin zum Smartphone entwickelt. Anwendungen, die in der Prä-Smartphone-Ära lediglich als Mehrwert-Funktionen implementiert wurden (z. B. Kalender, Kontaktverwaltung, kleinere Spiele), die aber keinesfalls die Kernfunktionalität des Gerätes darstellten, haben die Telefonie in der Smartphone-Welt verdrängt. Je nach Zielgruppe und Generation sind andere Funktionen deutlich in den Vordergrund gerückt. Mit dieser Verschiebung der Kernfunktionalität von der Telefonie hin zum ganzheitlichen Medienkonsum durch Verwendung mobiler Software hat sich ebenfalls die Marktprofilierung der Produzenten mobiler Geräte gewandelt. Zuvor marktbeherrschende Hersteller von Mobilfunktelefonen und Netzbetreiber sind kaum unter den Branchenriesen der Smartphone-Produzenten vertreten. Hier sind es

interessanterweise Hersteller von Hardware- und Softwareprodukten, die den Markt mobiler Geräte für sich erobert und entscheidend geprägt haben (vgl. Holzer und Ondrus [186]. Prognosen erwarten für das Jahr 2018 mehr als 2,5 Milliarden Smartphone-Nutzer weltweit (vgl. eMarketer [108]), davon mehr als 41 Millionen Smartphone-Nutzer allein in Deutschland (vgl. Bundesverband Digitale Wirtschaft (BVDW) e.V. [64]). Rechnerisch besitzt dann also jeder zweite Deutsche ein Smartphone, wobei aufgrund demographischer Effekte eher angenommen werden kann, dass die Mehrzahl aller deutschen Erwachsenen dann über mindesten ein Smartphone verfügen wird.

In den Anfängen der Smartphone-Ära (d. h. vor der Etablierung von Smartphones ab ca. 2007) erstreckten sich die Herausforderungen des Software Engineering für mobile Systeme über nahezu alle Aspekte und Teildisziplinen. Aufgrund der sich erst entwickelnden Technologie mobiler Geräte waren zukünftige Schwerpunkte der inhaltlichen Entwicklung mobiler Systeme kaum absehbar. Der Schwerpunkt der Forschung lag auf dem Umgang mit den geringen Leistungsressourcen mobiler Geräte und der geringen Verfügbarkeit mobilen Internets. Eine im Jahr 2000 von Roman et al. [303] durchgeführte Untersuchung des Staus Quo mobiler Systeme identifizierte neue Herausforderungen des Software Engineering vor allem im Bereich der Anpassung von Software auf die durch die Gerätetechnologie der Zeit vorgegebenen Möglichkeiten. Die Landschaft mobiler Anwendungen wurde klar von der technischen Machbarkeit dominiert. Eine typische Fragestellungen war, ob ein Anwendungsfall unter den gegebenen technischen Umständen sinnvoll in mobiler Software realisiert werden kann. Im Jahr 2016 ist es eher die Fragestellung, welche Geschäftsprozesse aus ökonomischer Perspektive mobilisiert werden sollten. Während um 2000 die dominierenden Themen der Forschung *Middleware*-zentrierte Architekturen, Systemadaptivität, *Code-On-Demand*-Technolgien und Algorithmen zum effizienten *Caching* von Daten in unzuverlässigen mobilen Netzwerken waren, lautet um das Jahr 2016 das Paradigma „*Design for mobile first*“ – Geschäftsprozesse werden so weiterentwickelt, dass eine Mobilisierung von vornherein grundsätzlich möglich ist (vgl. Voas et al. [355]). Diese Entwurfsentscheidung gilt selbst dann, wenn ein direkter Einsatz in einem mobilen Umfeld gegenwärtig nicht erforderlich ist.

Die Leistungsfähigkeit von Smartphones und Tablets hat sich inzwischen in wesentlichen Aspekten Desktop-Computern angenähert. Während in den 2000er Jahren das Internet noch per *Wireless Access Protocol* (WAP), einem auf die speziellen Bedürfnisse mobiler Geräte dieser Zeit zugeschnittenen Kommunikationsprotokoll, übertragen wurde, weil diese aufgrund geringer CPU-Leistung keine gewöhnlichen HTML-Seiten darstellen konnten (vgl. Picco et al. [282]), stellt sich die Welt der mobilen Geräte im Jahr 2016 vollkommen anders dar. Ähnlich dem Mooreschen Gesetz für Computertechnik im Allgemeinen (vgl. *Moore's Law* in Moore [255]) ist zu erwarten, dass sich die Leistungsfähigkeit von Smartphones weiterhin steigert. Die grundsätzliche Problematik der Bereitstellung ausreichender Rechenleistung auf geringer Baugröße scheint gemessen an den im Jahr 2016 aktuellen und zukünftig zu erwartenden Anwendungsfällen zunächst gelöst zu sein. Ebenfalls erkennbar ist bereits, dass mobile Geräte und Desktop-Computer zu konvergieren beginnen. Smartphones beginnen sich nach einer Phase der Miniaturisierung in ihren Abmessungen wieder auszudehnen, um auch den Anforderungen der Darstellung komplexer Anwendungsinhalte auf dem Display gerecht zu werden.

Plattformbetreiber stellen Vertriebsplattformen bereit, auf welchen Softwareprodukte entgeltlich oder unentgeltlich an Verbraucher abgegeben werden können. Hierzu zählen z. B. Apple App Store [15], Windows Phone Store [248] und Google Play Store [154]. Sowohl für den Vertrieb von Softwareprodukten als auch für Bezahlfunktionen innerhalb von Apps behalten sich Betreiber mobiler Plattformen den Einbehalt eines prozentualen Satzes der Verkaufssumme vor. Somit werden die Plattformbetreiber unmittelbar zum Wertschöpfer an Drittanbieterprodukten. Es liegt deshalb im Interesse des jeweiligen Plattformbetreibers, dass zu jedem Zeitpunkt eine große Anzahl Softwareprodukte kostenpflichtig auf dem plattformspezifischen Marktplatz im Angebot ist. Um die Angebotsvielfalt zu fördern, sind Plattformbetreiber bemüht ihre jeweilige mobile Plattform, d. h. sowohl mobile Betriebssysteme als auch die zugehörige Betriebsinfrastruktur und auch mobile Geräte selbst, in kurzen Zyklen an neue technische Möglichkeiten und neue Anforderungen des Marktes anzupassen. Hierzu gehört u. a. die Bereitstellung eines SDKs, welches öffentliche Teile der plattformspezifischen APIs für Anbieter von Softwareprodukten verfügbar macht.

Diese Mechanismen der mobilen Plattformen ermöglichen kurze Produktzyklen (vgl. Picco et al. [282], Holzer und Ondrus [186]) und versprechen aus Sicht der Plattformbetreiber einen hohen *Return On Investment* (ROI). Plattformbetreiber streben deshalb an, ähnliche Infrastrukturen ebenfalls für Desktop-Systeme zu etablieren. Auch hier sollen Softwareproduzenten und Anwendern leicht zugängliche Marktplätze geboten werden, die es Plattformbetreibern ermöglichen, Teil der Wertschöpfungskette von Anwendungen und Dienstleistungen zu werden. Im Idealfall können die Marktplätze für mobile Plattformen und Desktop-Systeme durch eine gemeinsame Infrastruktur bedient werden. Dieses Konzept erfordert jedoch eine Konsolidierung der für die jeweilige Zielplattform verwendeten Technologien. Das Unternehmen Microsoft nimmt hier im Jahr 2016 eine Vorreiterrolle ein. Waren die Microsoft Desktop- und Server-Betriebssysteme und Microsofts frühe mobilen Plattformen (z. B. Microsoft Windows CE) zwar oberflächlich aneinander angelehnt, waren Softwareprodukte beider Systeme jedoch nicht kompatibel zueinander. Mit der Version Microsoft Windows 10 wird hingegen angestrebt, sowohl Desktop-Systeme als auch Smartphones und Tablets gemeinsam in einem einheitlichen System zusammenzuführen. Die Apple-Systeme Apple iOS und Mac OS werden in ähnlicher Weise weiterentwickelt. Hierdurch soll Nutzern die Geborgenheit eines vollständig integrierten Kontinuums mobiler und nicht-mobiler Geräte gegeben werden, in welchem technologische und funktionale Grenzen zwischen einzelnen Geräten aufgehoben werden. Entwickler werden zukünftig ebenfalls von dieser Entwicklung profitieren, weil konsolidierte Plattformen langfristig einen geringeren Entwicklungsaufwand versprechen.

Die Probleme der Bereitstellung von Rechenleistung, Softwareprodukten und *User Experience* (UX, engl. Anwendererfahrung, Anwendererlebnis) sind mit der gegenwärtig verfügbaren mobilen Technologie ausreichend gelöst, um digitale Geschäftsmodelle effektiv und effizient zu unterstützen. Die klassischen Probleme des Software Engineering hingegen werden durch diese Entwicklungen hingegen nur marginal tangiert. Zwar sehen sich Programmierer nicht mehr länger mit Problemen zu geringer Leistungsressourcen mobiler Geräte konfrontiert, mobile Softwareprodukte werden jedoch zunehmend komplexer und interaktiver und sind durch einen hohen Grad der Integration mit anderen Systemen und Produkten (z. B. Backend-Systeme, *Social-Media*-Dienste, Cloud-Dienste oder Dienst- und Technologie-*Mash-Up*) charakterisiert

(vgl. Picco et al. [282], Zhang et al. [377]). Während das Erstellen performanter und zuverlässiger Software durch bessere SDKs und APIs auf der einen Seite einfacher wird, werden die Anforderungen an Entwicklungsprojekte auf der anderen Seite durch Fragmentierung der mobilen Gerätelandschaft und die zunehmende Anzahl technischer Features in mobilen Geräten höher. Entwickler sehen sich im Vergleich mit der Landschaft mobiler Geräte aus den Anfängen der Smartphone-Ära mit anderen Herausforderungen konfrontiert.

1.1.2 Ökonomische Bedeutung mobiler Plattformen und Softwaresysteme

Das iPhone, erster Vertreter der neuen Generation Smartphones, wurde im Jahr 2007 vorgestellt. Wesentliche Neuerung war allerdings nicht die Hardware, sondern das Geschäftsmodell. Der Apple App Store wurde im März 2008 zugänglich und hat die Welt des Vertriebs von Software nachhaltig verändert. Das erste Android Gerät wurde im Oktober 2008 vorgestellt, der Google Play Store wurde im selben Zeitraum zugänglich. Zum ersten Mal war App-Entwicklern ein weltweiter Marktplatz ohne wesentliche Investitionskosten zugänglich. Mussten bislang zum Vertrieb von Software eigene Vertriebskanäle erschlossen werden, wurden mit den App Stores Vertriebskanäle von den Betreibern mobiler Plattformen bereitgestellt, die gegen einen prozentualen Einbehalt des Erlöses aus der Kommerzialisierung ohne wesentliche Zugangshürden zum Vertrieb von Software verwendet werden konnten.

Im Jahr 2012 verwendeten bereits mehr als eine Milliarde Menschen weltweit Smartphones (vgl. eMarketer [108], Statista Inc. [332]). Mit zunehmenden Zahlen für Smartphone-Verkäufe nehmen auch die Zahlen der aus den App Stores heruntergeladenen Apps zu. Bereits während des ersten Monats seiner Existenz hat der Apple App Store mehr als eine Million US-Dollar Umsatz täglich erzielt (vgl. Wingfield [365]). Während 2010 ca. 10,9 Milliarden Apps aus den App Stores der verschiedenen Plattformbetreiber geladen wurden, werden laut einer Studie aus dem Jahr 2010 des Marktforschungs- und Beratungsunternehmens IDC für 2014 76,4 Milliarden App-Downloads mit einem Ertrag von etwa 35 Milliarden US-Dollar vorhergesagt [65, 107]. Tatsächlich erreicht wurde laut einer Statista [333] Studie ein Umsatz von ca. 35 Milliarden US-Dollar für das Jahr 2014. In derselben Studie wird für das Jahr 2017 ein Umsatz von ca. 76 Milliarden US-Dollar prognostiziert.

Im Juni 2014 standen im Apple App Store zirka 1,2 Millionen Apps zum Download bereit [331], im Google Play Store waren es zur selben Zeit etwa 1,3 Millionen [13]. Im Google Play Store kommen monatlich etwa 23.000 neue Apps hinzu. Im Apple App Store wächst die Anzahl Apps etwa auf einem vergleichbaren Niveau. Im Windows Phone Store waren es allein im Februar 2013 ca. 130.000 neue Apps [247]. Im Google Play Store ist mit mehr als 85% die überwiegende Anzahl von Apps ohne direkte Kosten für den Download erhältlich [12], die Zahlen für den Apple App Store sind vergleichbar. Eine Studie des US Wirtschaftsmagazins Forbes aus dem Jahr 2013 kommt zu dem Ergebnis, dass Apple im Jahr der Studie etwa 5 Milliarden US-Dollar an ca. 235.000 App-Entwickler ausgeschüttet hat, bei Google wurden im selben Zeitraum etwa 900 Million US-Dollar an 150.000 Entwickler ausgeschüttet [116]. Verglichen mit den Zahlen der heruntergeladenen Apps pro Plattform kann ein Apple iOS-Entwickler durchschnittlich mit 4.000 US-Dollar Einnahmen pro App und pro Jahr rechnen, während ein Google Android-Entwickler lediglich 1.125 US-Dollar pro App pro Jahr erwarten kann. Hierbei handelt es sich allerdings um Durchschnittswerte, durch welche die Realität nur

unzureichend abgebildet wird. Tatsächlich ist es so, dass der wesentliche Anteil dieser Umsätze durch wenige Entwickler erzielt wird, während der überwiegende Anteil der Entwickler, die Apps für die Direktvermarktung produzieren, kaum Erlös erzielen kann.

Eine im Jahr 2014 vom Unternehmen Kinvey, Inc. durchgeführte Studie unter 100 Personen in der Rolle *Chief Information Officer* (CIO, engl. Leitungsrolle der strategischen und operativen Führung der Informationstechnologie) in US-Unternehmen mit mehr als 500 Beschäftigten analysiert die Erwartungshaltung von Unternehmen und die wirtschaftlichen Bedingungen der App-Entwicklung. Im Ergebnis kommen die Autoren zum dem Schluss, dass CIOs vom Potenzial der Mobilisierung digitaler Geschäftsprozesse überzeugt sind. Mehr als 60% der Befragten erwarten durch Mobilisierung von Geschäftsprozessen neue Erlösquellen erschließen zu können. Noch 12% verfolgen das Ziel, durch mobile Softwareprodukte ein Alleinstellungsmerkmal generieren zu können und so das Marktgefüge zu verändern. Auf der anderen Seite zeigt sich die Mehrheit der Befragten unzufrieden mit dem Prozess der App-Entwicklung. Ursächlich hierfür sind zu lange Entwicklungszeiten und zu hohe Produktionskosten. Über die Hälfte der Befragten (56%) gab an, dass die Entwicklungszeit pro App zwischen 7 Monaten bis hin zu mehr als einem Jahr beträgt. Damit liegt die Dauer der Entwicklung über der durchschnittlichen Dauer von Versionszyklen mobiler Plattformen, wodurch ein App-Projekt in einer späten Phase mit neuen Anforderungen der Plattform belastet werden kann.

Die Untersuchung der wirtschaftlichen Bedeutung mobiler Systeme lässt sich also in den folgenden Aussagen zusammenfassen: (1) Mobile Systeme werden von vielen Unternehmen zur zukünftigen Unterstützung der Kerngeschäftsprozesse als herausragend bedeutsam wahrgenommen, (2) die Entwicklung mobiler Systeme ist eine wirtschaftliche Herausforderung, die bei geringem ROI einen hohen Ressourceneinsatz erforderlich macht und (3) die wichtigsten Strategieziele für die Zukunft der Entwicklung mobiler Systeme ist die Reduzierung der Betriebskosten und die Erhöhung der Produktivität der Mitarbeiter in der Entwicklung dieser Systeme. Die Dynamik der App Stores führt zu einer Verschärfung dieser Probleme. Aufgrund der hohen Zuwachszahlen bei den Nutzern von Smartphones und in der Konsequenz im Zuwachs bei den App-Nutzern sind – insbesondere bei populären Apps – schnell sehr viele Nutzer gleichzeitig von Softwaredefekten betroffen. Aktualisierungen bereits publizierter Apps werden unmittelbar für die Gesamtheit aller Nutzer verfügbar, die eine App auf ihrem Gerät installiert haben. Defekte Softwareprodukte werden von Anwendern i. d. R. nur schlecht akzeptiert, so dass ein defektes Softwareprodukt seine ökonomische Zielsetzung nicht mehr erfüllen kann (vgl. Zhong und Michahelles [379], Khalid [220, 222, 221]). Softwaredefekte stören deshalb unmittelbar die Wertschöpfungskette selbst dann, wenn nicht der Erlös aus dem Vertrieb der App das primäre Ziel der App-Entwicklung ist, sondern andere Geschäftsziele durch Einsatz mobiler Software gestützt werden sollten.

Solche Störungen in der Wertschöpfungskette können reduziert werden, indem die Entwicklung mobiler Software durch Verkürzung der Entwicklungszeit effektiver gestaltet wird und die Anzahl der Defekte durch die Verwendung zuverlässiger Teststrategien und -werkzeuge reduziert wird. Beide Ziele werden durch das in dieser Dissertation erarbeitete Konzept zur Testautomatisierung unterstützt. Die Verwendung von Automatisierungstechnologie zur Erzeugung von Testfällen und deren Durchführung ermöglicht eine Erhöhung der Produktivität der App-Entwicklung bei gleichzeitiger Erhöhung der Zuverlässigkeit der Testergebnisse.

1.1.3 Auswirkungen auf den Softwareentwicklungsprozess

Traditionelle Entwicklungsprozesse realisieren Produktvorhaben unter der Voraussetzung einer Menge von Annahmen über das zukünftige Betriebsumfeld des zu entwickelnden Produkts. In der Regel steht beispielsweise zu Beginn der Konzeptionsphase fest, welches Zielsystem das Produkt unterstützen soll. Hierbei wird spezifiziert, welche Hardware-Voraussetzungen das geplante Softwareprodukt an ein Computersystem stellt. Ebenfalls wird in der Spezifikation eines Softwareprodukts festgelegt, welches Betriebssystem in welcher Version durch das geplante Softwareprodukt bedient werden soll. Diese Anforderungen, der Betriebskontext, sind i. d. R. so ausgelegt, dass sie für Endanwenderprodukte mit marktüblicher Hardware erfüllt werden können. Ein Produkt wird so geplant, dass eine Vielzahl zukünftiger Anwender das Produkt verwenden können, ohne in neue Hardware zu investieren. Im Umfeld von Enterprise-Apps stellt sich die Situation so dar, dass entweder die zu unterstützende Hardware-Landschaft verbindlich vorgegeben wird (z. B. durch ein Legacy-Altsystem in *Mainframe*-Architektur), so dass eine Modifikation der Hardware-Landschaft nicht in Frage kommt oder die Softwareentwicklung einem Geschäftsprozess dient, welchem die Beschaffung und der Betrieb von Hardware ökonomisch untergeordnet wird.

Im Umfeld von Desktop- oder Serversystemen auch Projektlaufzeiten von mehreren Jahren verhältnismäßig unkritisch. Einerseits haben Betriebssysteme für Desktop- oder Serversysteme einen im Vergleich zu mobilen Plattformen langen Lebenszyklus, andererseits ist es für Anbieter von Desktop- oder Serversystemen aufgrund einer großen Marktdurchdringung und definierten *Service Level Agreements* (SLA, engl. Dienstgütevereinbarung) von großer Bedeutung Rückwärtskompatibilität zu gewährleisten. Softwareprodukte für Desktop- oder Serversysteme sind im Normalfall auch in der nächsten oder sogar übernächsten Version eines Betriebssystems ohne Anpassungen lauffähig. Veränderungen der Systemarchitektur oder grundlegende Änderungen von Programmierparadigmen sind in diesem Umfeld selten.

Mobile Plattformen haben hingegen vergleichsweise kurze Lebenszyklen. Eine neue Version eines mobilen Betriebssystems ist für die prävalenten Systeme Apple iOS und Android bereits nach durchschnittlich sechs Monaten zu erwarten. Im Gegensatz zu Desktop- oder Serversystemen ist es im Umfeld mobiler Plattformen ebenfalls nicht unüblich, dass eine Weiterentwicklung der Plattform grundlegende Änderungen der Systemarchitektur, von Programmierparadigmen[2,3] oder der Gestaltung des *User-Interface* (UI) des Betriebssystems einführt. Anbieter von Apps sehen sich in diesem Fall mit der Situation konfrontiert, dass bisher fehlerfreie Anwendungen nicht länger lauffähig sind und überarbeitet werden müssen.

Die Schnelllebigkeit mobiler Plattformen gepaart mit den Auswirkungen von Mobilität und Kontextsensitivität auf den Entwurf, die Entwicklung und den Betrieb von Softwaresystemen (vgl. Abschnitt 3.2) stellen Softwareentwicklungsprozesse vor neue Herausforderungen (vgl. Murphy et al. [259], Picco et al. [282], Roman et al. [303], Dehlinger und Dixon [90]). Für alle Phasen der Anforderungsermittlung gelten für mobile Plattformen wesentliche Einschränkungen, die für traditionelle Softwareprojekte in dieser Form nicht existieren. Insbesondere gilt, dass Annahmen über die zukünftige Betriebsumgebung eines Softwareproduktes nur noch

[2] z. B. Einführung der Programmiersprache Swift für Apple iOS Version > 8 im Jahr 2014

[3] z. B. Einführung des Konzepts der *Fragments* in Android, eine zusätzliche Komponente der Anwendungsarchitektur, die das bisherige Programmiermodell in wesentlichen Aspekten, insb. dem Testen, erweitert

in Grundlagenaspekten getroffen werden können. Beispielsweise kann lediglich vorausgesetzt werden, dass eine bestimmte Plattform (z. B. Apple iOS, Android oder Microsoft Windows Phone) zu unterstützen ist. Bereits definitive Aussagen über die zu unterstützende Version einer Plattform ist hingegen kritisch. Liegt der Zieltermin für ein Softwareprodukt in einem Zeitraum, der sich über den durchschnittlichen Lebenszyklus dieser Plattform erstreckt, können keine sicheren Annahmen über den Zustand dieser Plattform getroffen werden. Für einige Plattformen ist aufgrund technischer Vorgaben des Betreibers eventuell sogar ausgeschlossen, Rückwärtskompatibilität ab einer bestimmten Plattformversion zu gewährleisten.

Dieser Effekt wird durch die Entwicklungszyklen mobiler Geräte noch verstärkt. Hersteller von Smartphones und Tablets produzieren neue Geräte in einer Frequenz, die noch über derer mobiler Betriebssystemversionen liegt. Zusätzlich zu den Unsicherheiten über zukünftige Eigenschaften mobiler Plattformen als solche können keine Annahmen über die technischen Eigenschaften mobiler Geräte getroffen werden. In der Regel ist zwar davon auszugehen, dass die nächsten Generationen existierender Gerätefamilien der gegenwärtigen Generation in technischen Parametern überlegen sein werden. Es muss jedoch auch damit gerechnet werden, dass neue Gerätefamilien hinzukommen, die existierenden Gerätefamilien in ihren Eigenschaften unterlegen sind, z. B. aus Gründen einer zielgruppenorientierten Preisgestaltung, sogenanntes *Target Costing* (engl. Zielkostenrechnung). Weiterhin ist davon auszugehen, dass mobile Gerätefamilien ständig um neuartige Funktionen (z. B. weitere Displays, weitere Sensortypen, zusätzliche Interaktionsmodi[4]) erweitert werden. Zukünftige Anwender erwarten, dass diese neuen Funktionen schnellstmöglich adäquat in existierende Anwendungen integriert werden. Hierdurch sehen sich App-Anbieter außerplanmäßig zu Wartungsarbeiten an ansonsten fehlerfreien Anwendungen gezwungen.

Insbesondere für große Softwareprojekte mit Laufzeiten, die sich wesentlich über den durchschnittlichen Lebenszyklus mobiler Plattformen erstrecken, ist damit zu rechnen, dass sich bereits während der Projektlaufzeit wesentliche Neuerungen an der Zielplattform ergeben, durch welche die Projektplanung in Teilen obsolet wird. Traditionelle nicht-agile, nicht-iterative, plangetriebene Vorgehensmodelle wie etwa V-Modell oder Wasserfallmodell sind deshalb für Softwareprojekte im mobilen Umfeld nur bedingt geeignet (vgl. Dantas et al. [87]). Die Unbeständigkeit technischer Eigenschaften mobiler Plattformen erzwingt in dieser Hinsicht ebenfalls kurze Produktzyklen für mobile Anwendungen. Apps, die nicht schnell auf neue Versionen der Plattform angepasst werden, müssen mit Reputationseinbußen und konsequent mit einer Verkleinerung des Anwenderkreises rechnen. In der Regel wird ein Anwendungsfall durch eine Vielzahl an konkreten Produkten innerhalb einer Plattform abgedeckt, so dass Anwender nur mit geringem Aufwand zu einem alternativen Produkt wechseln können. Neben einer herausragenden UX ist deshalb die Zuverlässigkeit einer App ein zentrales Instrument der Kundenbindung.

Die Faktoren Mobilität und Kontextsensitivität fügen zu den plattformgetriebenen Unwägbarkeiten weitere Erschwernisse hinzu. Es können zum Entwurfszeitpunkt nur auf abstrakten Ebenen Annahmen über das Verhalten des Anwenders und den Nutzungskontext einer Anwendung getroffen werden. Die Existenz einer funktionierenden Netzwerkverbindung

[4] Etwa Apple Pay als zeitgenössisches Beispiel, in dem eine Gerätefamilie um ein Smartphone-basiertes Zahlungssystem erweitert wurde.

kann beispielsweise ebenso wenig vorausgesetzt werden wie die Verfügbarkeit von Standortinformationen. Es ergeben sich für den Softwareentwicklungsprozess aus diesen Gründen eine Reihe neue Herausforderungen, die mobile Anwendungen gegenüber traditionellen Anwendungen alleinstellen. Um auf die Herausforderungen der schnellen Plattform- und Gerätezyklen reagieren zu können, ist es von zentraler Bedeutung, dass Softwareprozesse schnell und flexibel auf Veränderungen in den Ökosystemen der jeweiligen Plattformen angepasst werden. Da Überschneidungen der Planungs- und Umsetzungsperiode mit den Entwicklungszyklen mobiler Plattformen nicht generell vermeidbar sind, müssen Softwareprozesse ausreichend agil gestaltet werden, um auch noch während der Entwicklung mit Anpassungen an der Produktspezifikation umgehen zu können. Durch diese Anforderungen werden nicht-agile, nichtiterative Vorgehensmodelle quasi vom Einsatz in Softwareprojekten in mobilen Umgebungen ausgeschlossen. Die Mobilität der zukünftigen Anwender einer Software macht zudem aufwendige Teststrategien erforderlich. Zur Durchführung von Tests müssen u. U. unterschiedliche Standorte aufgesucht werden, um das Nutzerverhalten realistisch durch Testfälle abzubilden (vgl. Haller [164], Joorabchi et al. [204]).

Ein zentrales Werkzeug zur Sicherstellung der Zuverlässigkeit einer Anwendung auch über Versionsgrenzen der Plattform hinweg ist das sorgfältige Testen aller Berührungspunkte der App mit dem umgebenden Ökosystem der Plattform und dem Gerät. Testen ist jedoch ein signifikanter Aufwandstreiber, der bei Durchführung durch einen menschlichen Akteur in einem hohen Maß Ressourcen bindet. Dies wiederum ist schlecht mit den Anforderungen an kurze, flexible Entwicklungsprozesse zu vereinbaren. Die Automatisierung von Testaktivitäten, die nicht zwingend durch einen menschlichen Akteur durchgeführt werden müssen, kann dazu beitragen, den Entwicklungsprozess effizienter und effektiver zu gestalten. Für den Softwareentwicklungsprozess bedeutet dies, dass das Testen in einem besonderen Maß berücksichtigt werden muss. Hier ergibt sich erneut ein unmittelbarer Vorteil agiler, iterativer Vorgehensmodelle, die eine intensive Testphase am Ende jeder Iteration vorsehen (vgl. Dantas et al. [87]). Der hierdurch zusätzlich entstehende Aufwand muss allerdings aufgrund ökonomischer Überlegungen durch die Verwendung von Testautomatisierungstechnologie reduziert werden.

1.2 Problemstellung und Lösungsansatz

Unabhängig vom verwendeten Vorgehensmodell (plangetrieben oder agil) ist die Absicherung der Qualität von Softwareprodukten eine zentrale Aufgabe im Softwareprozess, die sich in Aktivitäten manifestiert, die jeweils eine spezifische Perspektive auf die Software zum Gegenstand haben. Nach ihrer Markteinführung durchleben Softwareprodukte bis zu ihrer Außerdienststellung mehrere Evolutionsphasen. Während dieser Evolution werden Softwareprodukte inhaltlich an neue und sich verändernde Anforderungen der Anwender oder des technischen Umfelds angepasst. Hierbei treten eine Reihe von Seiteneffekten auf, die Auswirkungen auf Funktionalität, Qualität und Komplexität von Softwaresystemen haben.

Einige dieser Effekte wurden von Lehman [234] bereits im Jahr 1980 untersucht und sind als Lehmans Gesetze der Softwareevolution in die Fachliteratur eingegangen. Diese Gesetze beschreiben unter anderem, dass Softwaresysteme einem anhaltenden Wandel unterliegen und ständig angepasst werden müssen, um die Zufriedenheit der Anwender zu erhalten (Lehmans

1. Gesetz [234]), dass sich die Komplexität von Softwaresystemen ohne aktive Gegenmaßnahmen exponentiell zur Lebensdauer entwickelt (Lehmans 2. Gesetz [234]) und dass die Qualität von Softwaresystemen mit der Lebensdauer abnimmt, es sei denn dem Qualitätsverlust wird durch konsequente Wartung entgegengewirkt (Lehman 7. Gesetz [234]). Obgleich Lehmans Gesetze weit vor der Entwicklung der ersten mobilen App formuliert wurden, haben sie auch in der Domäne der mobilen Systeme nicht an Aktualität oder Gültigkeit verloren. Zhang et al. [377] weisen in einer empirischen Studie aus dem Jahr 2013 über die Evolution mobiler Sofwaresysteme nach, dass insbesondere Lehmans 1. Gesetz (kontinuierliche Anpassung), Lehmans 2. Gesetz (zunehmende Komplexität) und Lehmans 7. Gesetz (abnehmende Qualität) unverändert auch für mobile Systeme gelten.

Mit steigender Komplexität eines Softwaresystems (im Vergleich zu Softwaremetriken, wie z. B. *Lines of Code*, *Function Points* oder zyklomatische Komplexität, vgl. Boehm [50], Fenton und Bieman [112]) steigt der Aufwand des Testens überproportional an, d. h. der Aufwand zur Absicherung der Qualität einer Software wächst schneller als die Komplexität der zu testenden Software. Je nach Quelle und Periode der Formalisierung der Softwaresystementwicklung wird der Anteil des Testens zwischen 30% und 90% des Gesamtaufwands beziffert (vgl. Myers et al. [261], Tassey [341]). Deshalb werden an die Qualitätssicherung neben fachlich-inhaltlichen und technischen Anforderungen insbesondere auch ökonomische Ansprüche gestellt. Ein Softwareprodukt muss nicht nur mit einem geeigneten Vorgehen so getestet werden, dass seine Qualität objektiv beurteilt werden kann, sondern dieses Vorgehen muss mit dem verfügbaren Budget vereinbar sein. Im Hinblick auf technische und ökonomische Aspekte ist das Testen von Softwaresystemen Gegenstand zahlreicher Forschungsarbeiten (vgl. Kapitel 2), die sich mit Konzepten und Methoden zur Formalisierung und Automatisierung von Testprozessen befassen. Ziel dieser Arbeiten ist es, Testaktivitäten unter formalen Aspekten stärker in den Softwareprozess einzubetten und menschliche Akteure weitgehend durch Werkzeuge und Maschinen abzulösen. Weiterhin soll die Aussagekraft, die Qualität und die vertragliche Belastbarkeit von Testergebnissen unter ökonomischen und juristischen Aspekten greifbar gemacht werden. Das Bestehen von Akzeptanztests (engl. *Acceptance Tests*, auch Abnahmetest) ist nicht selten eine formale Bedingung für den erfolgreichen Abschluss eines Softwareprozesses.

Beschränkungen des Budgets betreffen qualitätssichernde Aktivitäten ebenso wie andere Aktivitäten des Softwareprozesses. Deshalb ist es i. d. R. nicht möglich, ein Softwareprodukt vollumfänglich unter Berücksichtigung aller möglichen Bedingungen zu testen. Im Umfeld mobiler, kontextsensitiver Anwendungen erzeugen Permutationen unterschiedlicher Eigenschaften der Betriebsumgebung einer Software einen hochkomplexen Zustandsraum, der sich aus den verschiedensten Kontextparametern (z. B. Zeit, Ort und Persona der Anwendungsnutzung, Eigenschaften von Mobilfunknetzen usw.) zusammensetzt und zudem Beschränkungen unterliegt, die sich dem technischen Zugriff auf Anwendungsebene entziehen (z. B. bei Beteiligung der Betriebsinfrastruktur eines Betreibers mobiler Plattformen). Das Testen von Softwaresystemen ist deshalb eine nicht-triviale Tätigkeit, die eine sorgsame Planung und Durchführung erfordert, um zu gewährleisten, dass alle Aspekte einer Software unter adäquater Auswahl von Methoden und Werkzeugen und mit ausreichender Detailtiefe hinsichtlich ihrer Konsistenz gegenüber der Spezifikation geprüft werden. Über technische und methodische Zwänge hinaus werden Testaktivitäten durch wirtschaftliche Erwägungen beschränkt. So

ist es ökonomisch selten sinnvoll, Testprozesse so zu gestalten, dass ihre Planung und Durchführung gegenüber der Inkaufnahme eines Schadens durch Systemdefekte unrentabel ist. Eine Ausnahme bilden hier Softwaresysteme deren Defekt potenziell den Verlust von Gesundheit oder Leben zur Folge haben könnte.

Um Testaktivitäten unter technischen, methodischen und ökonomischen Aspekten so zu gestalten, dass ein optimaler Schnitt zwischen Umfang der Testaktivitäten und der Rentabilität der Testdurchführung erreicht wird, müssen auch Testaktivitäten unter ökonomischen Perspektiven im Rahmen einer Teststrategie in den Softwareprozess eingebettet werden. Die Teststrategie bestimmt welche Teile, Aspekte oder Funktionen mit welchem Aufwand, welcher Intensität und mit welchen Mitteln zu testen sind. Sie bestimmt ebenfalls, auf welche Arten von Tests die Konsistenz eines Softwaresystems gegenüber einer Spezifikation überprüft wird. Weiterhin definiert sie, welche Artefakte die Testdurchführung prägen. Hierzu gehört unter anderem die Spezifikation bedarfsgerechter Testfälle. Deren konkrete Ausprägung bestimmt den Testaufwand wesentlich. Dieser Aufwand kann durch unterschiedliche Maßnahmen aufgefangen werden. Zur Durchführung von Tests, die einen hohen personellen Aufwand erfordern kommen beispielsweise *Outsourcing*-Strategien in Frage, um die Personalkosten durch hinzukaufen günstiger Arbeitskraft zu reduzieren. Ergänzend oder als Alternative kann Automatisierungstechnologie angewendet werden, die menschliche Arbeitskraft in Testaktivitäten durch Maschinen oder Werkzeuge ablöst. Hierdurch wird insbesondere bei wiederholter Durchführung einer großen Anzahl von Testfällen nicht nur der Aufwand reduziert, sondern menschliche Fehlerquellen können für bestimmte Arten von Tests ausgeschlossen werden. Es ergeben sich neben unmittelbaren ökonomischen Vorteilen deshalb zusätzlich qualitative Vorteile aus der Verwendung von Automatisierungstechnologien.

Aufgrund der Komplexität von Softwareanwendungen stellt bereits die Erstellung von Testfällen eine Herausforderung dar. Einzelne Funktionen von Softwareanwendungen stehen ggf. in Relation zueinander (d. h. eine Funktion setzt voraus, dass zunächst eine andere Funktion ausgeführt wird usw.). Es ergeben sich deshalb verzweigte oder zyklische Pfade durch eine Anwendung. Eine adäquate Testabdeckung (in Abhängigkeit eines gewählten Testabdeckungskriteriums) erzeugt deshalb schnell sehr viele, ggf. ähnliche Testfälle. Diese Komplexität zu beherrschen ist menschlichen Akteuren häufig nicht möglich. Im Gegensatz hierzu steht die Möglichkeit, Testfälle nicht manuell zu erstellen, sondern zu modellieren. Die durch Modellierung erreichte Abstraktion von technischen Details hilft dabei, auch komplexe Zusammenhänge für menschliche Akteure beherrschbar zu machen. Diese Modelle können dann anschließend als Eingabe für eine Automatisierungstechnologie zur Testdurchführung verwendet werden.

1.2.1 Testen mobiler Softwaresysteme

Mobilität im engeren Sinne bezeichnet die Eigenschaft von Entitäten, ortsungebunden und beweglich zu sein. Im Kontext des Software Engineering bezeichnet Mobilität bezogen auf Softwaresysteme die Eigenschaft dieser Systeme auf verschiedenen Abstraktionsebenen ortsungebunden und beweglich zu sein. Zu unterscheiden ist zwischen physikalischer und logischer Mobilität (vgl. Roman et al. [303]), wobei sowohl Geräte auf denen eine Anwendung ausgeführt wird als auch Anwendungen oder einzelne Komponenten einer Anwendung oder eines Anwendungssystems mobil sein können.

Physikalische Mobilität bedeutet, dass eine Anwendung tatsächlich von einem Ort zu einem anderen Ort bewegt wird [303]. Im Normalfall wird diese Art der Mobilität durch die Bewegung des Geräts verursacht. Das ist dann der Fall, wenn ein Benutzer ein Smartphone oder ein Tablet-Computer mit sich trägt, während er sich selbst von einem Ort zum anderen bewegt. Das heißt, die Anwendung selbst verhält sich relativ zur Ausführungsplattform weitgehend wie eine Desktop-Anwendung. Daraus folgt, dass für eine Reihe von Systemparametern verhältnismäßig sichere Annahmen über die Unveränderlichkeit von Kerneigenschaften des Systems gemacht werden können. In der Regel ist davon auszugehen, dass etwa der verfügbare Arbeitsspeicher, Größe und Auflösung des Displays, Vorhandensein von spezieller Gerätehardware (GPS, Sensoren usw.) über den Lebenszyklus der Anwendung hinweg konstant bleiben. Ebenfalls ist es jedoch möglich, dass Anwendungen oder einzelne Komponenten von einem Gerät auf ein anderes Gerät bewegt werden. Denkbar ist hierfür der Anwendungsfall, dass ein Benutzer die Bearbeitung eines Geschäftsprozesses auf einem Desktop-Computer begonnen hat – beispielsweise in einer Cloud-Anwendung – und die Bearbeitung nach verlassen der Arbeitsstelle (z. B. während einer Zugfahrt) auf einem mobilen Gerät fortsetzen möchte. Annahmen über Kerneigenschaften des Geräts können dann nicht mehr vorausgesetzt werden. Zwar ist im Einzelfall davon auszugehen, dass individuelle Anwender sich immer derselben Gerätefamilie bedienen werden, für die Gesamtheit der Anwender muss jedoch mit einem breiten Spektrum an mobilen Geräten gerechnet werden.

Logische Mobilität bezeichnet den Wechsel des Ausführungsortes einer Anwendung oder einer Anwendungskomponente, wobei weder der Anwender noch die Ausführungsplattform physikalisch ihren Standort wechseln, wohl aber ausführbarer Programmcode den Standort seiner Ausführung wechselt oder Ort der Codeausführung zwischen unterschiedlichen Orten wechselt [303]. Ein exemplarischer Anwendungsfall für diese Art der Mobilität ist, dass die Bearbeitung eines Geschäftsprozesses durch eine lokale Anwendung/Komponente abgebrochen und anschließend durch eine Anwendung/Komponente auf einem entfernten Gerät (zum Beispiel ein Web-Service) fortgesetzt wird. Mobilität des Anwenders ist hier keine zwingende Voraussetzung. Gründe für den Wechsel des Bearbeitungsortes eines Geschäftsprozesses können hier rein technischer Natur sein (z. B. *Load-Balancing*), es muss nicht einmal zwingend ein menschlicher Akteur beteiligt sein.

Im Fokus dieser Arbeit steht der Einfluss physikalischer Mobilität von Anwendungen im Rahmen der physikalischen Standortveränderungen mobiler Geräte, d. h. des Nutzers, der sich mit seinem Gerät zusammen bewegt. Mobilität von Anwendungen über die Grenzen der Ausführungsplattform hinweg werden im Rahmen dieser Arbeit nicht untersucht. Eine ausführliche Diskussion des Begriffs Mobilität erfolgt in Abschnitt 3.1.1.1.

Durch die Modularisierung von Anwendungen einerseits und eventueller Abhängigkeit mobiler Anwendungen von Netzwerkdiensten andererseits verhalten sich mobile Anwendungen in vielen Aspekten ähnlich wie verteilte Systeme. Das heißt, funktionale Komponenten werden nicht ausschließlich lokal vorgehalten, sondern sind über Netzwerkdienste verfügbar. Mobile Geräte der Kategorie Smartphone oder auch Tablet-Computer sind in ihrer Mobilität i. d. R. unbeschränkt. Es können deshalb zu keinem Zeitpunk im Lebenszyklus der Anwendung sichere Annahmen über die Verfügbarkeit von Ressourcen wie zum Beispiel Netzwerk- oder Lokalisierungsdienste gemacht werden. Das funktionale Verhalten einer Anwendung ist deshalb geprägt

von wechselnder Resourcenverfügbarkeit, wodurch sich besondere Herausforderungen für den Entwurf, die Entwicklung und insbesondere das Testen mobiler Anwendungen ergeben.

In der Entwurfsphase muss die teilweise unplanbare Nichtverfügbarkeit von Ressourcen bereits in der Anwendungsarchitektur berücksichtigt werden. Es sind hierbei Entscheidungen darüber zu treffen, in welchem Maß angestrebt wird, die Anwendungsfunktionalität auch bei Nichtverfügbarkeit von Ressourcen zu erhalten. Für das Testen mobiler Anwendungen stellen Mobilitätsfaktoren eine besondere Erschwernis dar. Ist das SUT in seiner Funktion in wesentlichen Aspekten von Mobilitätsfaktoren abhängig, so müssen diese beim Entwurf von Testplänen und Testfällen berücksichtigt werden. Ortsbasierte Anwendungen die geographische Standortinformationen in den Daten- oder Kontrollfluss einbeziehen, müssen beispielsweise auf korrekte Funktion unter realistischen Bedingungen untersucht werden. Dem steht gegenüber, dass in der Planungsphase häufig unrealistische Annahmen über die Verfügbarkeit von Ressourcen unter Mobilitätsbedingungen gemacht werden. So wird beispielsweise regelmäßig davon ausgegangen, dass GPS-basierte Standortinformationen immer verfügbar und korrekt sind. Beide Annahmen entsprechen normalerweise nicht der Realität: GPS-Daten sind i. d. R. ungenau (z. B. durch Benutzung des letzten bekannten Standorts bei temporärer Nichtverfügbarkeit des tatsächlichen Standorts) und häufig nicht verfügbar. Für das Beispiel Ortsinformationen kommt hinzu, dass in der Entwurfsphase häufig implizit Annahmen über den zu erwartenden Wertebereich von Messdaten gemacht werden, die sich in der Praxis als unzutreffend herausstellen.

Neben geographischen Ortsinformationen spielen andere Faktoren ebenfalls eine Rolle. So schwankt etwa die Verfügbarkeit und die Qualität von Netzwerkverbindungen stark standortabhängig und ist i. d. R. nicht planbar (vgl. Kleinrock [226], Grassi et al. [156]). Neben regionalen Verfügbarkeitsschwankungen spielen bei gegenwärtiger Netzwerktechnologie ebenfalls eine Reihe dynamischer Faktoren eine Rolle, da die punktuelle Leistungsfähigkeit zellulärer Netzwerke stark durch die aktuelle Anzahl von Kommunikationsteilnehmern beeinflusst wird. Beispielsweise beeinflussen bei der Netzwerktechnologie um das Jahr 2016 (z. B. *General Packet Radio Service* (GPRS), *Universal Mobile Telecommunications System* (UMTS)) hohe Reisegeschwindigkeiten bei Zug- oder Autobahnfahrten durch den Dopplereffekt instabile Netzwerkverbindungen. In der Testphase gilt es, den Einfluss von Mobilitätsfaktoren auf das SUT korrekt einzuschätzen, kritische Komponenten zu identifizieren und im Rahmen der Testdurchführung entsprechend zu stimulieren.

Mit aktuellen Technologien ist es nur mit erheblichem Aufwand möglich, Kontextfaktoren in die Planung und Durchführung von Softwaretests einzubeziehen. Einerseits sind aktuelle Entwicklungs- und Testwerkzeuge in diesem Aspekt stark unterentwickelt, andererseits existieren keine standardisierten Vorgehensmodelle und Konzepte, um Mobilitätsfaktoren in die Testplanung zu integrieren. Entwicklungswerkzeuge für mobile Plattformen bieten kaum die Möglichkeit, den Standort des Gerätes für die Testdurchführung anzupassen. Auf Simulatoren/Emulatoren ist das zwar teilweise möglich (z. B. Apple iOS und Android), allerdings sind Tests auf Simulatoren nur bedingt aussagekräftig. Auf mobilen Geräten selbst kann eine von der Realität abweichende Standortinformation nur mit erheblichem Aufwand hergestellt werden, sofern das durch die Plattform überhaupt ermöglicht wird. In Standardtestwerkzeugen ist diese Funktion bislang nicht integriert. Auf der Plattform Android kann eine künstliche

Standortinformation erzeugt werden, indem das Gerät in einen besonderen Modus geschaltet wird und durch eine spezielle Software, die jedoch nicht zum Standardumfang des SDK gehört, das Verhalten des GPS-Subsystems imitiert wird. Auf der Plattform Apple iOS ist das hingegen nicht möglich, so dass Tests für ortsbasierte Anwendungen entweder nur auf dem Simulator oder nur durch tatsächliches Aufsuchen der in der Testspezifikation angegebenen Orte möglich ist. Testfälle müssen deshalb i. d. R. manuell erstellt und durchgeführt werden. Im Fall einer Anwendung deren Funktion abhängig von Ortsinformationen ist, müssten im Rahmen der Testdurchführung mit verschiedenen mobilen Geräten in verschiedenen Konfiguration (Netzverbindung, Plattform, Betreibergesellschaft usw.) unterschiedliche Orte aufgesucht werden, um den Testfall im eigentlichen Sinn *in loco*, also vor Ort, durchzuführen, sofern keine adäquate Kontextsimulationstechnologie verfügbar ist (vgl. Haller [164], Vieira et al. [352]). Zwar bieten Emulatoren/Simulatoren eingeschränkte Möglichkeiten Ortsinformationen zu simulieren, jedoch sind diese nicht durchgehend automatisiert, so dass auch hier ein großer Anteil der Arbeit nur manuell durchgeführt werden kann.

1.2.2 Testen kontextsensitiver Softwaresysteme

Kontextsensitivität bezeichnet die Eigenschaft einer Anwendung, Faktoren der Betriebsumgebung in den Kontrollfluß der Anwendung einzubeziehen. Das heißt, Umweltfaktoren bedingen zum einen, wie sich eine Anwendung verhält und zum anderen werden Eigenschaften der Betriebsumgebung dazu genutzt, die Funktionalität der Anwendungen auch in Stresssituationen weiterhin zu gewährleisten. Der (Betriebs-)Kontext einer Anwendung umfasst dabei alle Umgebungsparameter der Anwendung, des Geräts oder des Benutzers, sofern diese entweder direkt durch Messung erfasst werden können oder der Anwendung durch geeignete Maßnahmen (z. B. Web-Services) zur Verfügung stehen. Die Auswertung dieser Kontextinformationen ermöglicht es, das Verhalten der Anwendung zu steuern. Aus der Perspektive der Softwareentwicklung ist der Anwendungskontext also als zusätzlicher Kanal der Interaktionsschnittstelle der Anwendung zu verstehen. Der Kontext einer Anwendung erstreckt sich hierbei über mehrere Abstraktionsebenen. Individuelle Kontextinformationen können über mehrere Abstraktionsebenen hinweg zu komplexen Situationen aggregiert werden, die das Umfeld des Benutzers und der Anwendung detailliert beschreiben (vgl. Vieira et al. [352]). Eine atomare Ortsinformation in Form geodätischer Koordinaten ist also ebenso Teil der Kontextbeschreibung einer Anwendung wie die abstrakte Situation “im Zug” (im Sinne einer Reise mit der Eisenbahn), die aus atomaren Kontextinformationen wie etwa Abwesenheit von bekannten Wi-Fi-Netzwerken und einer stetigen Standortveränderung bei einer eisenbahntypischen Geschwindigkeit entlang einer möglicherweise bekannten Reiseroute aggregiert wird. Die Erfassung und Analyse von Kontextinformationen ist hierbei nicht auf eine einzelne Anwendungsinstanz oder ein einzelnes Gerät beschränkt. Durch Vernetzung von Diensten können ebenfalls dezentral erhobene Informationen in die Situationsanalyse einbezogen werden.

Kontextsensitivität beeinflusst das Verhalten von Anwendungen, durch Analyse individueller Kontextparameter und einer darauf basierenden Interpretation, in welcher Situation eine Anwendung benutzt wird. Anwendungsbeispiele sind etwa die Anzeige ortsbezogener Wetter- oder Verkehrsinformationen (sogenannte ortsbasierte Dienstleistungen). Basierend auf der Situationshistorie einer Anwendung ist es ebenfalls möglich, die zukünftige Verfügbarkeit von

Netzwerkverbindung oder Web-Services vorherzusagen und die Anwendung so frühzeitig auf zukünftige Situationen anzupassen (etwa durch präventives Herunterladen zukünftig benötigter Daten oder Anwendungskomponenten).

Auf die Entwicklung und vor allem das Testen mobiler Anwendungen hat Kontextsensitivität weitreichende Auswirkungen. Hinsichtlich der Entwicklung ist die Anwendungsarchitektur so zu konzipieren, dass Kontextfaktoren den Kontrollfluss der Anwendungen derart beeinflussen, dass das gewünschte Anwendungsverhalten eintritt. Die eigentliche Herausforderung bei der Entwicklung mobiler kontextsensitiver Anwendungen ist jedoch, Testprozesse und Testfälle so zu gestalten, dass einerseits der durch Kontextsensitivität zusätzlich entstehende Mehraufwand angemessen reduziert wird, andererseits aber auch eine hinreichende Testabdeckung der Anwendung gewährleistet bleibt. Kontextsensitivität verursacht einen erheblichen Mehraufwand beim Testen von Anwendungen, da Kontextfaktoren den Raum für mögliche Anwendungsfehler wesentlich erweitern (vgl. Abschnitt 4.2, Abschnitt 5.5.3). Im Kontrast zu nicht-kontextsensitiven Anwendungen muss hier nicht nur die korrekte Funktion einer Anwendung getestet werden, sondern die korrekte Funktion unter dem Einfluss relevanter Kontextfaktoren. Für kontextsensitive Anwendungskomponenten heißt das, dass diese im Rahmen des Testprozesses mit kontextbezogenen Testdaten stimuliert werden müssen, um mögliche Defekte zu erkennen.

Mit verfügbaren Methoden und Werkzeugen ist es aktuell nicht oder nur stark eingeschränkt möglich, ein SUT mit künstlichen kontextbezogenen Testdaten zu stimulieren. Testfälle müssen deshalb manuell durchgeführt werden, wobei sich für viele Kontextparameter jedoch die Beschränkung ergibt, dass ein menschlicher Akteur nicht in der Lage ist, einen einzelnen Testfall mehrfach zu reproduzieren. Einige Einschränkungen ergeben sich aus der mangelnden Präzision der menschlichen Bewegungsfähigkeit (z. B. wenn ein Geräte in einer exakt vorgegebenen Orientierung relativ zur Erdoberfläche positioniert werden muss), andere ergeben sich aus der technischen Nichtmachbarkeit der Kontrolle physikalischer Umgebungsparameter (z. B. Richtung und Stärke des Erdmangnetfeldes, gemessen durch einen Magentfeldsensor und verwendet zur Bestimmung der Orientierung eines Geräts relativ zur Erdoberfläche).

1.2.3 Testautomatisierung kontextsensitiver mobiler Softwaresysteme

Aktuell verfügbare *Integrated Development Environments* (IDE) sind primär auf die Entwicklung von Anwendungen ausgerichtet, das Testen wird jedoch nur rudimentär unterstützt. Im Bereich Desktop- und Serversysteme existieren bereits zahlreiche Konzepte, um das Testen von Anwendungen und Anwendungssystemen zu automatisieren (vgl. Abschnitt 2.3.3, Abschnitt 4.3.2). Diese Konzepte umfassen Methoden und Werkzeuge, den Testprozess durch geeignete Automatisierungstechnologien zu unterstützen und somit durch Reduzierung manueller Tätigkeiten unter wirtschaftlichen Aspekten zu optimieren und menschliche Fehlerquellen weitgehend auszuschließen. Diese Konzepte bedienen sich modellbasierter Methoden, die aus Artefakten des Systementwurfs (d. h. Modellen, die statische und dynamische Systemeigenschaften abbilden) Testfälle und Testpläne generieren. Diesen generierten Testfällen und Testplänen wird unterstellt, ein zu entwickelndes System weitgehend vollständig abzudecken und gleichzeitig Redundanzen zu vermeiden, wodurch der manuell zu erbringende

Aufwand für die Erstellung und Durchführung von Softwaretests maßgeblich reduziert wird. Unter Anwendung dieser Technologien können Testfälle automatisch aus Entwurfsartefakten erstellt und automatisiert durchgeführt werden. Als Eingabe in einen solchen automatisierten Testprozess dienen beispielsweise UML-Diagramme. Ausgabe kann z. B. ein Testreport sein, der Fehler im SUT und deren Ursache gezielt aufzeigt. Insbesondere in Zusammenarbeit mit *Continuous Integration* (CI)-Systemen, die eine zur Auslieferung geeignete Version eines Softwaresystems automatisch erstellen, werden automatisierte Testwerkzeuge eingesetzt.

Im Bereich mobiler, kontextsensitiver Anwendungen existieren gegenwärtig keine geeignete Testwerkzeuge, Mobilitäts- oder Kontextfaktoren in die Erstellung von Testfällen oder deren automatische Durchführung zu integrieren. Die Relevanz der in dieser Dissertation bearbeiteten Problemstellung ist deshalb sowohl wirtschaftlicher als auch wissenschaftlicher Natur. Die wissenschaftliche Relevanz des diskutierten Themas ist in mehreren Dimensionen angesiedelt. Es existiert bislang kein wissenschaftlich fundiertes und belastbares Gemeinverständnis des Kontextbegriffs (vgl. Jung et al. [206]). Eine auf den konkreten Anwendungsfall Testen mobiler, kontextsensitiver Anwendungen bezogene Definition des Kontextbegriffs ist unumgänglich, um den Kontext einer mobilen Anwendung für Testaktivitäten greifbar zu machen.

Die im Bereich Desktop- und Serversysteme in zahlreichen Forschungsarbeiten behandelten Konzepte zur Automatisierung von Softwaretests basieren auf einem umfangreichen theoretischen Verständnis von Softwaresystemen und deren Modellen. Sie bilden die formalen Grundlagen für die Entwicklung von Werkzeugunterstützung. Existierende theoretische Grundlagen beinhalten aktuell keine Formalismen den Kontext einer Anwendung in Modellierungs-, Entwurfs-, Entwicklungs- und Testaktivitäten einzubeziehen. Die wissenschaftliche Herausforderung liegt also darin, zu untersuchen ob existierende Konzepte dazu geeignet sind, um durch Kontextfaktoren erweitert zu werden oder ob grundlegend neue Konzepte zum Verständnis und zur Interpretation von Kontextfaktoren notwendig sind, um den Anwendungskontext verstärkt in Entwurfs- und Testaktivitäten einzubeziehen.

In der wirtschaftlichen Perspektive sind Automatisierungstechnologien für das Testen mobiler Anwendungen deshalb interessant, weil sie den Aufwand – und damit unmittelbar die Kosten – für die Testdurchführung wesentlich reduzieren können (vgl. Ramler und Wolfmaier [291]). Nach gegenwärtigem Stand der Technik sind nur automatisierte Technologien dazu in der Lage, Kontextfaktoren mobiler Anwendungen in einem ausreichenden Wertebereich zu simulieren, ohne dass hierzu die tatsächliche Mobilität des Testers im Rahmen der Testdurchführung notwendig ist (vgl. Bylund und Espinoza [67], Broens und van Halteren [59]).

Es ist i. d. R. nicht davon auszugehen, dass die Entwicklung eines Softwaresystems nach dem ersten Rollout vollständig abgeschlossen ist. Vielmehr ist zu erwarten, dass Pflege und Wartung eines Softwaresystems einen Großteil des Gesamtaufwandes verursachen (vgl. Lebenszyklen mobiler Plattformen, Abschnitt 1.1.1). Insbesondere beim Testen kommt es hierbei zu wiederkehrenden Tätigkeiten, um Regressionen (Rückgang oder Rückschritt, lat. *regredi*: umkehren, zurückgehen) bei der Weiterentwicklung oder der korrektiven Pflege eines System zu vermeiden. Für den speziellen Anwendungsfall des automatisierten Testens kontextsensitiver mobiler Anwendungen sind aktuell keine Werkzeuge verfügbar, durch welche der gesamte Prozess vom Testfallentwurf bis zur Durchführung abgedeckt wird. Die Entwicklungsumgebungen einiger Plattformen bieten jedoch Werkzeuge auf Basis von JUnit/CU-

nit [34, 326] an, mit denen Softwaretests zumindest teilweise automatisiert werden können. Kontextfaktoren werden durch diese Werkzeuge gegenwärtig nicht unterstützt, ebensowenig die modellbasierte Generierung von Testfällen aus Artefakten des Systementwurfs.

Das Testen mobiler, kontextsensitiver Anwendungen stellt Entwickler und Tester vor neue Probleme, die im Desktop- oder Server-Umfeld typischerweise nicht auftreten. Im Desktop- oder Server-Umfeld sind die Charakteristika der zukünftigen Einsatzumgebung eines Softwaresystems weitgehend bekannt oder werden durch Anforderungs- und Spezifikationsdokumente definiert. Weiterhin kann die Abhängigkeit von Kontextfaktoren, wie sie im Umfeld mobiler Anwendungen verwendet werden, als gering eingeschätzt werden. Bezug zu Kontexteigenschaften tritt bei diesen Systemen i. d. R. nur in den verarbeiteten Daten auf und ist nur selten funktionsbedingender Parameter.

Bei Entwicklung und Betrieb mobiler, kontextsensitiver Anwendung lassen sich konzeptuell und technisch bedingte Probleme unterscheiden. Die Klasse der konzeptuell bedingten Probleme basiert auf der Problematik, alle für einen Testfall notwendigen externen Abhängigkeiten kontrollieren zu können. Die Klasse der technisch bedingten Probleme wird durch die Eigenschaften der zu entwickelnden Systeme, ihrer Zielplattformen und deren Entwicklungsumgebungen charakterisiert. Sie umfasst plattformunabhängige und plattformabhängige Fragestellungen, also den gesamten Entwicklungsprozess von Anforderungsanalyse, Systementwurf, Modellierung bis zur Implementierung.

1.2.3.1 Konzeptuelle Problemstellungen

Die Klasse der konzeptuell bedingten Probleme beim Testen mobiler Anwendungen umfasst alle Faktoren, die das Testen von Anwendungen methodisch erschweren oder unmöglich machen. Hierzu zählen u. a. Abhängigkeiten von externen Diensten, wie zum Beispiel der Zugriff auf Webservices, deren Verfügbarkeit durch Dritte reguliert wird. Insbesondere zeitabhängige Dienstleistungen, wie etwa die Wiedergabe bestimmter Medieninhalte zu einer bestimmten Tageszeit (z. B. uhrzeitabhängiges Live-Streaming), führen dazu, dass Funktionen, die auf diese Dienste angewiesen sind, nicht oder nur mit erheblichem Mehraufwand außerhalb der regulären Verfügbarkeitsintervalle getestet werden können. Zu den zu diskutierenden Fragestellungen gehört beispielsweise die Problematik festzulegen, welche Eigenschaften des SUT mit welcher Priorität und Intensität getestet werden müssen. Problematisch ist ebenfalls die Feststellung der Verfügbarkeit relevanter Schnittstellen externer Dienste und notwendiger Schritte diese Verfügbarkeit planmäßig herbeizuführen. Diese konzeptuell bedingten Probleme stellen eine erhebliche Herausforderung für das Testen mobiler Anwendungen dar, deren Lösung jedoch in erster Linie auf Prozessebene und im Rahmen des organisatorischen Projektmanagements erarbeitet werden muss (vgl. Vieira et al. [352]). Zur Verdeutlichung der Problematik werden an dieser Stelle zwei Beispiele angeführt:

Mobile Infotainment Anwendung Funktionaler Bestandteil einer Infotainment-Anwendung ist die Wiedergabe von Audio- und Videoinhalten. Seitens des Inhalteanbieters wird je nach Tageszeit ein Standardprogramm oder ein uhrzeitspezifisches Themenprogramm angeboten. Zeitintervalle, in denen das Themenprogramm angeboten wird, werden der mobilen Anwendung über einen Web-Service verfügbar gemacht. Zum Zeitpunkt der

Wiedergabe wird algorithmisch entschieden, welcher Inhalt tatsächlich wiedergegeben wird. Für den Testfall ergibt sich hieraus die Notwendigkeit entweder den Test der Komponente zur Medienwiedergabe zu entsprechenden Tageszeiten durchzuführen oder aber seitens des Inhalteanbieters eine Testschnittstelle anzubieten, die einen Test des uhrzeitbezogenen Themenprogramms auch außerhalb des standardmäßig vorgesehenen Zeitintervalls zulässt. Die erste Variante ist aus organisatorischen Gründen (Projektmanagement, Arbeitszeiten, Arbeitsrecht, usw.) eventuell nur mit Einschränkungen zu realisieren. Die zweite Variante ist aus ökonomischen Gründen seitens des Inhalteanbieters ebenfalls nur mit Einschränkungen möglich, da es sich um ein Produktivsystem handelt, welches ebenfalls von weiteren Kanälen genutzt wird. Die Anpassung der Uhrzeit auf dem Testgerät ist für diesen Anwendungsfall weitgehend ungeeignet, da die Wiedergabe des Themenprogramms serverseitig gesteuert wird. Automatisierungstechnologie kann hier dabei helfen, zumindest die erste Einschränkung durch die Entkopplung von menschlicher Arbeitskraft aufzulösen.

Mobile Depotverwaltung Eine mobile Anwendung ermöglicht dem Nutzer die Verwaltung von Aktiendepots. Der Datenbestand ist über Web-Services erreichbar. Aus Sicherheitsgründen kommt der Test in der Produktivumgebung des Anbieters nicht in Frage. Stattdessen wird eine realitätsnahe Kopie der Produktivumgebung erstellt, um während der Entwicklung Testdaten bereitzustellen. Die Erstellung dieser Testumgebung ist mit erheblichem Aufwand verbunden, da nicht nur Datenbestände zu Testzwecken generiert werden müssen, sondern auch die zur sicheren Kommunikation zwischen Client und Server via *Secure Socket Layer* (SSL) Schnittstelle notwendigen Signaturzertifikate beschafft werden müssen. Die Anonymisierung von Testdaten ist in der Realität nicht nur von technischen Einschränkungen abhängig, sondern wird auch durch juristische Rahmenbedingungen (z. B. Verarbeitung personenbezogener Daten) wesentlich erschwert.

In beiden Beispielen sehen sich Entwickler und Tester mit Problemen konfrontiert, die durch das komplexe Zusammenspiel der mobilen Anwendung mit dem Server-Backend bedingt werden. Ausschließlich mit technischen Mitteln kann diesem Problem nicht begegnet werden. Selbst wenn sich einige dieser Probleme mit technischen Mitteln lösen ließen, würde eine solche Lösung die Beteiligung von Dritten erforderlich machen. Im Beispiel der mobilen Infotainment Anwendung käme hier eine separate Version des Backend-Systems in Frage. Durch hohe Investitionskosten könnte der Betreiber die Bereitstellung eines solchen Systems jedoch ablehnen. Selbst wenn ein solches Testsystem bereitsteht, wäre durch eine Entkopplung der Medienwiedergabe von der Uhrzeit ein vom Original abweichender Anwendungsfall gegeben, so dass ein Test auf dem Testsystem eben kein valider Test der Anwendung wäre.

1.2.3.2 Technische Problemstellungen

Die Klasse der technisch bedingten Probleme im Umfeld des Testens mobiler, kontextsensitiver Anwendungen umfasst alle Faktoren, die auf Mobilität und Kontextsensitivität als allgemein plattformunspezifische Eigenschaften eines Softwaresystems sowie auf plattform- und implementierungsspezifische Probleme der Softwareentwicklung zurückzuführen sind.

Mobile Anwendungen werden i. d. R. in monolithischen Betriebssystemen ausgeführt. Interna dieser Betriebssysteme sind häufig unbekannt und i. d. R. unzugänglich für externe Manipulation. Für den bestimmungsgemäßen Betrieb handelt es sich hierbei zwar um eine wünschenswerte Eigenschaft, im Umfeld des Testens bleiben dem Testbetrieb hierdurch jedoch einige der etablierten Methoden und Werkzeuge aus dem Umfeld von Desktop-Systemen grundsätzlich verschlossen. Im Desktop-Umfeld gibt es Software, die dazu eingesetzt werden kann, das SUT im laufenden Betrieb zu analysieren (Inhalte von Variablen, Speicheradressen, Auswertung von Benachrichtigungsströmen, usw.). Für eingebettete Systeme (mobile Plattformen sind in dieser Hinsicht als solche zu verstehen) gestaltet sich die Anwendung solcher Software wesentlich komplizierter, da das SUT i. d. R. nicht auf dem Entwicklungsgerät ausgeführt wird, sondern in einem physikalisch unabhängigen, autonomen und abgeschlossenen System. Die im Umfeld der Entwicklung eingebetteter Systeme gängigen Testwerkzeuge (z. B. JTAG) sind jedoch ebenfalls nicht einsetzbar, da diese Schnittstellen auf Smartphones im Normalfall nicht verfügbar sind. Während für technische Einrichtungen aus dem Bereich Maschineningenieurwesen i. d. R. speziell für Test- und Diagnosezwecke vorgesehene Schnittstellen existieren (etwa in der Kraftwerksanlagen- oder Motorentechnik), ist das Vorhandensein solcher Schnittstellen in der Softwaretechnik unüblich (vgl. Pettichord [281]). Testwerkzeuge müssen aus diesem Grund so entworfen werden, dass einerseits die angestrebten Ergebnisse erzielt werden, andererseits der regelmäßige Betrieb des SUT und anderer Apps auf dem Gerät oder die Plattform an sich durch die Anwesenheit einer Testumgebung nicht gestört wird bzw. Testergebnisse nicht durch die Testdurchführung selbst verfälscht werden. Beim Einsatz von Debuggern ist das nicht grundsätzlich gewährleistet. Eine Software im Debug-Modus zu kompilieren und auszuführen stellt insofern einen erheblichen Eingriff in das System dar.

Trotz zunehmender Konvergenz der Kerneigenschaften zeitgemäßer Smartphone-Technologie im Hinblick auf Hardware und Benutzerschnittstellen stellt die Varianz konkreter Geräteeigenschaften sowohl der Hardware als auch der Software ein weiteres signifikantes Problem in der Entwicklung und dem Testen von Softwaresystemen dar. Dazu zählen beispielsweise Displayauflösung, individuelle Gerätetreiber für Hardware-Komponenten, Verhalten des Geräts bei Änderung der Orientierung (Hochformat, Querformat) oder unterschiedliche Interpretation von Sensormesswerten.

Beim Entwurf von Konzepten zur automatisierten Generierung von Testfällen sind diese technisch bedingten Probleme insofern zu berücksichtigen, als dass auf technischer Ebene auftretende Probleme nicht grundsätzlich durch technische Hilfsmittel behoben werden können. Für den Entwurf modellbasierter Automatisierungslösungen müssen deshalb bereits auf Konzeptebene generalisierte und flexible Strategien zur Lösung technisch motivierter Probleme gefunden werden. Als Beispiel hierfür seien etwa Plattform-Defekte genannt, die sich dem Zugriff eines Anwendungsentwicklers generell entziehen, da sie zwar durch geeignete Maßnahmen lokal behoben werden könnten, z. B. durch einen *Workaround* (engl. Umgehungslösung), diese Änderung aber unmöglich auf alle bereits ausgelieferten Geräte übertragen werden können (z. B. Speicherzugriffsfehler in Android-Betriebssystemkomponente). Ebenfalls unter diese Kategorie fallen etwa Zugriffe auf die Sensoren eines Smartphones. Messwerte können zwar gelesen, grundsätzlich aber nicht geschrieben werden. Demnach ist es unmöglich Sensordaten so zu manipulieren, dass sie Messwerte liefern, die nicht der Realität entsprechen. Genau eine

solche Funktion wäre jedoch für das Testen mobiler Anwendungen notwenig, um für Testfälle sensorbasierter Anwendungen reproduzierbare Tests zu realisieren.

Anwendungsfälle mobiler Anwendungen umfassen häufig die Anforderung, die Lage des Geräts relativ zur Erdoberfläche festzustellen. Typische Anwendungsfälle sind etwa Fahrzeug- oder Fußgängernavigation, Geo-Caching-Apps oder der virtuelle Kompass. Kein individueller Sensor ist in der Lage, Informationen der Orientierung eines Geräts im dreidimensionalen Raum (Roll-, Nick- und Gier-Winkel) direkt zu messen. Aus diesem Grund kommt i. d. R. eine Kombination unterschiedlicher Hardware-Sensoren zum Einsatz. Gebräuchlich ist z. B. die Verwendung des linearen Beschleunigungssensors und des Magnetfeldsensors. Aus den Werten dieser Sensoren können Roll-, Nick- und Gier-Winkel des Gerätes errechnet werden. Beim Testen dieser Funktionen entstehen jedoch mehrere Probleme. Zum einen ist ein menschlicher Tester nicht in der Lage während des manuellen Testens eine lineare Beschleunigung des Gerätes (in diesem Anwendungsfall wird lediglich die Richtung der Gravitation der Erde benötigt) reproduzierbar zu erzeugen. Selbst wenn das Gerät auf einer Tischoberfläche ruht, beeinflussen Signalrauschen des Sensors und Vibrationen den Signalverlauf in einem Maß, dass Resultate für einen reproduzierbaren Testfall unbrauchbar werden. Die Sensoren in gegenwärtigen Smartphone-Generationen sind ausreichend genau, um auch die durch vorbeifahrende Lkw verursachte Vibration am Sensorsignal feststellbar zu machen. Zum anderen ist der Magnetfeldsensor eines Smartphones empfänglich für Störungen durch externe Quellen. Eigentlicher Zweck dieses Sensors ist die Messung des Erdmangetfelds. Allerdings können ferromagnetische Objekte in der Nähe des Geräts diesen Sensor stören (z. B. Geräteschutzhüllen mit magnetischem Verschluss). Während der Testdurchführung kann die Abwesenheit solcher Störgrößen nicht garantiert werden. Das heißt mehrmalige manuelle Testdurchführung ist nicht geeignet die fehlerfreie Funktion einer App nachzuweisen, da Identität physikalischer Eigenschaften der Testumgebung bei aufeinanderfolgenden Tests nicht gewährleistet ist.

1.2.3.3 Problemstellungen durch Verwendung von Stellvertretertechnologie

Die Entwicklung mobiler Anwendungen unterscheidet sich in mehreren Aspekten von der Entwicklung von Anwendungen für Desktop- oder Server-Systeme. Mobile Geräte sind aus Entwicklungssicht als eingebettete Systeme zu betrachten (vgl. Wasserman [359]) und unterliegen im Vergleich zu Desktop- oder Server-Systemen deshalb einer Reihe von Einschränkungen. Hierzu zählen unter anderem Aspekte der Leistungsfähigkeit der Geräte-Hardware und Schnittstellen zur Benutzerinteraktion aber auch Aspekte der Leistungsfähigkeit und des Umfangs von APIs.

Typisch für alle Arten eingebetteter Systeme ist, dass Anwendungen nicht direkt auf dem Zielsystem programmiert werden. Ursache hierfür ist weniger die Leistungsfähigkeit der Gerätehardware, sondern vielmehr die gering ausgeprägten Interaktionsschnittstellen für Anwender und Entwickler (z. B. vollständige Tastatur, ausreichend großer Bildschirm). Aus diesem Grund erfolgt die Entwicklung i. d. R. stellvertretend für tatsächliche mobile Geräte auf Desktop-Systemen in entsprechenden IDEs. Diese umfassen Editoren, Compiler, Linker und Emulatoren bzw. Simulatoren, um Fragmente der zu entwickelnden Anwendung auf dem Entwicklungsgerät erproben zu können. Auf diese Weise wird ein häufiger und oftmals zeitaufwändiger Transfer der Anwendung vom Entwicklungssystem auf das Zielsystem vermieden.

Die Verwendung von Emulatoren vereinfacht die Anwendungsentwicklung in vielerlei Hinsicht. Einerseits können Entwickler die Anwendung in gewohnter Umgebung mit gewohnten Werkzeugen programmieren. Für Teams mit mehreren Entwicklern sind auf diese Weise ebenfalls *Source Code Management Systems* (SCM) nutzbar, die mobilen Geräten im Allgemeinen verschlossen bleiben. Andererseits kann so ebenfalls in Teams an der Entwicklung gearbeitet werden, wenn nur wenige Endgeräte zur Erprobung der entstehenden Anwendung zur Verfügung stehen. Ein weiterer Vorteil in der Verwendung von Emulatoren in der Anwendungsentwicklung ist, dass diese in einem wesentlich umfangreicheren Maß konfigurierbar sind, als es bei tatsächlichen Geräten der Fall ist. So kann etwa die Größe des verfügbaren Gerätespeichers durch den Entwickler bestimmt werden oder auch die Auflösung und die Größe des Geräte-Displays – Faktoren durch welche die Anwendungsfunktion u. U. erheblich mitbestimmt wird. Auf diese Weise ist es möglich, ein breites Spektrum von zu erwartenden Gerätecharakteristika in der Entwicklungsumgebung zu modellieren und zu Testzwecken nachzubilden.

Emulatoren weisen im Vergleich zu den tatsächlichen Geräten eine Reihe weiterer Besonderheiten auf, durch welche die Anwendungsentwicklung einerseits in einigen Aspekten vereinfacht wird, die aber andererseits die Zuverlässigkeit von Testergebnissen erheblich einschränken. Zudem reflektieren Emulatoren die Eigenschaften der Hard- und Software nicht realistisch (Sensormesswerte unterliegen z. B. Signalrauschen, das durch Emulatoren nicht nachempfunden wird), so dass einige Aspekte des SUT inhärent schwer oder unmöglich getestet werden können (vgl. Dantas et al. [87]). Auf eine unvollständige Liste dieser Besonderheiten wird im Folgenden kurz eingegangen, da sie i. d. R. sichere Indikatoren dafür sind, dass ein Test auf einem Emulator nicht ausreicht, um die Spezifikationskonformität einer Anwendung zu bewerten.

User Interface Emulatoren bilden die Gerätehardware gewöhnlich einschließlich des UI nach. Während es so einerseits möglich ist die Anwendung auf einer großen Anzahl von ähnlichen Geräten zu testen (mehrere Modellreihen des Herstellers innerhalb einer emulierten Umgebung), sind die Testresultate jedoch nur bedingt aussagefähig. Die mit dem Emulatoren ausgelieferten Betriebsysteme sind i. d. R. stark generalisiert und entsprechen normalerweise dem *Factory Default* (d. h. Basisversion des Betriebssystems wie durch den Plattformanbieter ausgeliefert ohne Modifikationen) des Plattformanbieters. Das auf einem handelsüblichen Gerät installierte Betriebssystem (eventuell mit weiterer Software) ist jedoch oftmals speziell auf spezifische Betreibergesellschaften oder Gerätehersteller zugeschnitten (z. B. AT&T, Telekom, Samsung, HTC), wobei Plattformanbieter und Gerätehersteller nicht zwingend identisch sind. Diese sogenannten *Custom-tailored ROMs* (auf den jeweiligen Geräteanbieter oder Mobilefunkbetreiber zugeschnittenen Editionen des Betriebssystems) weisen u. U. andere Eigenschaften auf als das Standardsystem, mit der Folge, dass sich betroffene Geräte im Betrieb von Anwendungen nicht erwartungskonform verhalten. Gerätehersteller von Android-Geräten modifizieren beispielsweise regelmäßig die Benutzungsoberfläche des Systems und/oder ersetzen Systemanwendungen durch eigene Implementierungen. Aus der Perspektive eines Softwareentwicklers oder Testers ist das insofern problematisch, als dass für eine Untermenge aller im Umlauf befindlichen Geräte mit Fehlern gerechnet werden muss (z. B. verfälschte Farbschemata, unerwartetes Scroll-Verhalten), die das Bestehen von Akzeptanztests unmöglich

machen. Emulatoren bieten keine Möglichkeit, hersteller- oder betreiberspezifische Systembesonderheiten zu berücksichtigen. Um die korrekte Funktion einer Anwendung auf Custom-tailored ROMs zu testen, müssen entsprechende Geräte beschafft werden.

Netzwerk Ein weiterer signifikanter Unterschied zwischen Emulatoren und tatsächlichen Geräten liegt im Verhalten der Netzwerkanbindung. Mobile Geräte verfügen i. d. R. über ein Vielzahl von Optionen zur Kommunikation mit Netzwerken (GPRS, *Enhanced Data Rates for GSM Evolution* (EDGE), UMTS usw.), die jeweils einzigartige Parameter hinsichtlich Latenzzeiten, Übertragungsraten und *Quality of Service* (QoS) (engl. Dienstgüte) aufweisen. Hinzu kommen betreiberspezifische Konfigurationsmerkmale (Port-Sperren, *Traffic-Shaping*). Diese Parameter beeinflussen die Performanz von Netzwerkschnittstellen signifikant, können jedoch mit aktueller Emulatortechnologie nicht hinreichend nachgebildet werden. Emulatoren benutzen zur Kommunikation mit Netzwerken grundsätzlich die Netzwerkschnittstellen des Backend-Systems, d. h. desjenigen Rechensystems in welchem der Emulator ausgeführt wird. Wesentliche Qualitätsmerkmale können deshalb für Testzwecke nicht nachgebildet werden. Betreiberspezifische Eigenschaften finden ebenfalls keine Berücksichtigung.

Medienwiedergabe Ein weiterer kritischer Faktor im Bereich Testen mobiler Anwendungen ist, dass Emulatoren nur beschränkte Möglichkeiten zur Wiedergabe von Medien haben. Dieses Problem steht in engem Zusammenhang mit der Netzwerkunterstützung. Die Wiedergabe einzelner Inhalte ist unter Umständen sensibel gegenüber bestimmten Transportprotokollen. Die Infrastruktur der Plattform BlackBerry (insbesondere der BlackBerry Enterprise Server) beispielsweise lässt eine präzise Konfiguration zulässiger Protokolle zu. Hier kann etwa die Verwendung des *Real-Time Streaming Protocol* (RTSP) durch den Administrator innerhalb der BlackBerry Infrastruktur untersagt werden. Sofern eine mobile Anwendung Komponenten zur Wiedergabe von Medien beinhaltet, ist das als sicherer Indikator zum verpflichtenden Testen auf einer angemessenen Anzahl von Geräten zu bewerten.

Ortsinformation/Sensordaten Sind Anwendungen zu testen, deren Funktion die Feststellung und Auswertung von Ortungsinformationen beinhaltet, so muss diese einem Emulator künstlich zugeführt werden. Je nach Plattform ist aktuelle Emulatortechnologie unterschiedlich gut für diese Aufgaben geeignet, insgesamt jedoch auf niedrigem Niveau. Kernproblem ist, dass die aus dem Emulator gelesene Ortsinformation immer gültig und immer korrekt ist. Das entspricht jedoch nicht der Realität, da Ortsinformationen technologiebedingt in der Praxis grundsätzlich nur Näherungswerte sind. Mit Abweichungen und Mehrdeutigkeit muss i. d. R. gerechnet werden. Gründe hierfür sind, dass schlechte Netzabdeckung oder enge Straßen zwischen Häusern (sogenanntes *Urban Canyoning*) die Empfangsqualität von GPS-Komponenten negativ beeinflussen. Die Qualität einer mobilen, kontextsensitiven Anwendung ist stark abhängig von der Fähigkeit, mit fehlerhaften oder mehrdeutigen Ortsinformationen umzugehen.

Für Sensoren zur Messung physikalischer Eigenschaften gilt analog, dass diese i. d. R. sensorspezifischem Rauschen unterliegen, das durch Emulatoren nicht nachgebildet wird.

Algorithmen zur Verarbeitung solcher Sensordaten müssen hingegen genau dieser Anforderung gegenüber robust sein, d. h. auch verrauschte Messwerte sinnvoll zu interpretieren. Zudem existiert keine Schnittstellen in den APIs mobiler Plattformen, um Sensordaten zu Testzwecken künstlich in das SUT einzuführen.

Sicherheit An mobile Geräte werden i. d. R. besondere Sicherheitsansprüche gestellt. Viele Tätigkeiten, die mit einem mobilen Gerät durchgeführt werden können, sind für den Benutzer kostenpflichtig oder sind sogar dazu geeignet den Betrieb der Kommunikationsinfrastruktur zu stören. Neben ökonomischen Aspekten sind Überlegungen zur Betriebssicherheit der ausschlaggebende Faktor, die Ausführung von nicht durch vertrauenswürdige Instanzen in ihrer Unbedenklichkeit bestätigte Software auf dem Gerät zu verhindern. Dieser Prozess der Codesignierung verursacht einerseits Kosten, andererseits ist er oftmals an einen aufwändigen organisatorischen Prozess geknüpft. Emulatoren erwarten nicht, dass der auf ihnen ausgeführte Code mit einem gültigen Zertifikat signiert wird. Hierdurch wird es Entwicklern ermöglicht, das entstehende System ohne Berücksichtigung eventueller Sicherheitsaspekte hinsichtlich der Vertrauenswürdigkeit der Software zu entwickeln. Das gilt auch für SSL-Zertifikate, die sich auf tatsächlichen Geräten stark von Emulatoren unterscheiden können. Fehlt auf einem Gerät ein Zertifikat, das auf einem Emulator existiert, wird das gesamte Testergebnis hinfällig.

Die technischen Problemstellungen des Testens mobiler Anwendungen sind vielseitig und erstrecken sich über alle Aspekte der Anwendungsentwicklung, von der Implementierung der Anwendungslogik bis hin zum UI. Vielen dieser technischen Schwierigkeiten kann unter Inkaufnahme erhöhten Aufwands mit manuellem Testen begegnet werden. Eine Ausnahme hiervon bilden alle Tests von Anwendungskomponenten die Kontextparameter verarbeiten. Standortinformationen sowie Sensordaten können nur näherungsweise in manuellen Tests berücksichtigt werden. Es ist außerhalb hochspezialisierter Laborumgebung nicht möglich, bestimmte Umgebungsparameter für Testzwecke reproduzierbar nachzubilden. Zum einen ist die menschliche Fähigkeit Bewegungsabläufe exakt zu wiederholden eingeschränkt, zum anderen entziehen sich bestimmte Parameter der Umwelt, insbesondere physikalische Parameter wie z. B. Magnetfelder, dem Einfluss eines menschlichen Testers. Nur die maschinelle Simulation solcher Parameter kann hier die Reproduzierbarkeit von Testfällen gewährleisten.

1.2.4 Forschungshypothese

In Abschnitt 1.1 wurde dargelegt, dass mobile Softwaresysteme zukünftig weiter an wirtschaftlicher Bedeutung gewinnen werden und die Unterstützung von Geschäftsprozessen durch mobile Lösungen zur Kernaufgabe der IT-Strategie digitaler Unternehmen geworden ist. Die Berücksichtigung von Kontextinformationen im Systementwurf durch Bereitstellung geeigneter Modellierungsmethoden und Werkzeuge wird dabei helfen, die Abhängigkeiten individueller Softwareartefakte von Kontextinformationen herauszustellen und deren Bedeutung für die Funktion einer entstehenden Anwendung zu verdeutlichen. Die formale Berücksichtigung von Kontextinformationen in der Systemmodellierung bildet die Grundlage zur automatisierten Generierung kontextsenstiver Testfälle, die den Prozess der Testfallerstellung in einigen kritischen Aspekten von menschlicher Arbeitskraft entkoppeln kann. Hierdurch wird nicht nur die

zur Testfallerstellung aufgewendete Arbeitszeit reduziert, sondern es können auch menschliche Fehler bei der Testfallerstellung vermieden werden. Ein Werkzeug zur automatisierten Durchführung von Tests unter Berücksichtigung von Kontextinformationen (z. B. durch Injektion von Standort- oder Sensordaten in das SUT) hilft dabei, den Aufwand der Testdurchführung wesentlich zu reduzieren. Tester sind nicht länger gezwungen, ihren Arbeitsplatz zur Durchführung von Tests zu verlassen.

Zur Unterstützung dieser Forschungshypothesen umfasst diese Dissertation die Bearbeitung folgender Schwerpunkte:

Entwurf eines Kontextmodells / Testmodells: Zur Integration von Kontextfaktoren in übliche Systementwurfsmodelle (z. B. UML-Klassenmodelle, UML-Verhaltensdiagramme) werden Metamodelle zur Modellierung von Kontextmodellen / Testmodellen entworfen. Dies ermöglicht es, den Kontext einer Anwendung bereits zur Entwurfszeit formal zu berücksichtigen. Weiterhin bildet das Kontextmodell die Grundlage für die Generierung von Testfällen und deren automatische Durchführung.

Neben dem Kontextmodell ist es ebenfalls zweckmäßig, Systemmodelle zur nachfolgenden Generierung von Testfällen in dedizierte Testmodelle zu überführen. Ein entsprechendes Metamodell zur Testfallmodellierung stellt die für das Testen relevanten Eigenschaften eines Systems in den Vordergrund und bildet zugleich eine technologieunabhängige Grundlage für die Weiterverarbeitung zu automatisierbaren Testfällen.

Verfahren zur Generierung von Testfällen: Zur automatischen modellgetriebenen Erzeugung von Testfällen aus Artefakten des Systementwurfs wird ein Verfahren zur Testfallgenerierung erarbeitet. Die Generierung von Testfällen aus Artefakten des Systementwurfs setzt eine sorgfältige Modellierung voraus, kann jedoch gewährleisten, dass kritische Komponenten nicht bei der manuellen Testfallerstellung vernachlässigt werden.

Verfahren zur automatischen Durchführung: Zur automatischen Durchführung von Tests und zur Generierung von Fehlerreports wird ein Verfahren und begleitende Werkzeugunterstützung entwickelt. Die Durchführung von Testfällen ist zeitaufwändig und fehleranfällig. Automatisierung gestaltet den Prozess reproduzierbar, sicher und ökonomisch.

Die Konzepte des *Model Driven Architecture* (MDA)/*Model Driven Software Development* (MDSD) haben zum Ziel, die Erstellung von Softwaresystemen über den gesamten Prozess hinweg durch Systemmodelle zu unterstützen und hierbei besonders arbeitsintensive und fehleranfällige manuelle Tätigkeiten durch geeignete Automatisierungstechologien abzulösen.

Situative Faktoren der Ausführungsumgebung, der Plattform oder des Benutzers, haben bei kontextsensitiven, mobilen Anwendungen starken Einfluss auf Kontroll- und Datenflüsse in Softwaresystemen. Um den Testprozess zu automatisieren muss Kontext deshalb formal modellierbar gemacht werden. Nur so ist gewährleistet, Kontextfaktoren kontrollierbar in die Testautomatisierung einzubeziehen.

1.2.5 Lösungsansatz

Zum Nachweis der in Abschnitt 1.2.4 beschriebenen Forschungshypothese wird in dieser Dissertation eine Reihe von Lösungen für die einzelnen Fragestellungen erarbeitet. Die individuelle Bearbeitung der einzelnen Forschungsfragestellungen zur Kontextmodellierung, zur Testfallgenerierung und zur Testautomatisierung wird im Rahmen dieser Dissertation zu einer umfassenden Lösungsstrategie zusammengeführt, die das Testen mobiler Softwaresysteme durch Verwendung von Werkzeugen des MDT unter qualitativen und quantitativen Aspekten verbessern. Die einzelnen Lösungselemente werden in den Abschnitten 1.2.5.1 bis 1.2.5.3 erläutert.

1.2.5.1 Lösungsansatz zur Kontextmodellierung

Plangetriebene Softwareprozesse sehen i. d. R. vor, dass ein System vor der eigentlichen Implementierung unter Verwendung von Modellierungssprachen und -werkzeugen modelliert wird. Dies dient der Abstraktion von technischen Details und soll den Stakeholdern ein einheitliches Verständnis der zu entwickelnden Software ermöglichen. Die Modellierung erfolgt i. d. R. auf unterschiedlichen Abstraktionsniveaus und bildet das System aus verschiedenen Perspektiven ab. Gebräuchlich sind sowohl statische als auch dynamische Modelle, durch welche strukturelle Eigenschaften und Verhaltenseigenschaften zur Laufzeit beschrieben werden.

Den Grundstein zur automatisierten Generierung von Testfällen bildet die Integration von Kontextartefakten in die Systemmodellierung. Hierzu wird zunächst ein Metamodell entworfen, welches notwendige Komponenten und Elemente zur Modellierung von Kontextartefakten (z. B. Standort, Sensordaten) bereitstellt. Um die Verwendung von Kontextmodellen in üblichen Modellierungssprachen zu ermöglichen, erfolgt die technische Ausführung des Metamodells als UML-Profil. Die UML ist der de facto Standard zur Modellierung von Softwaresystemen. Zwar existieren neben der UML auch andere Modellierungssprachen wie beispielsweise *Business Process Model and Notation* (BPMN) oder *Ereignisgesteuerte Prozessketten* (EPK), jedoch bietet die UML eine Reihe von Vorteilen gegenüber anderen Modellierungssprachen, die sie für den in dieser Dissertation verwendeten Zweck besonders hervorhebt. Zum einen genießt die UML sowohl in der Industrie als auch in der Wissenschaft eine große Verbreitung. Die unterschiedlichen Modelle der UML sind deshalb auf der einen Seite bereits gut erforscht und werden auch in anderen MDSD-Technologien bevorzugt verwendet. Auf der anderen Seite stellt die UML eine Reihe unterschiedlicher Modelltypen bereit, um ein Softwaresystem aus unterschiedlichen Aspekten zu beschreiben. Für diese Arbeit von besonderem Interesse sind UML-Aktivitätsdiagramme (ACD), die geeignet sind Verhaltensaspekte eines Softwaresystems in beliebiger Detailtiefe abzubilden. Das UML-Profil zur Kontextmodellierung wird deshalb so ausgeführt, dass es auf UML-Aktivitätsdiagramme angewendet werden kann, um den Einfluss von Kontextfaktoren auf das Systemverhalten zu modellieren.

Das UML-Profil ist geeignet unterschiedliche Kontextparameter (vgl. Abschnitt 3.1.2) in Systemmodelle zu integrieren. Hierzu werden für relevante Kontextparameter im Sinne der Entwicklung mobiler Anwendungen entsprechende Modellierungselemente bereitgestellt. Übrige Modelleigenschaften werden durch die Anwendung des UML-Profils nicht beeinträchtigt und die Anreicherung des Modells mit Kontextparametern erfolgt sowohl in menschen- als auch maschinenlesbarem Format. Somit ist einerseits gewährleistet, dass Systemmodel-

le weiterhin für andere MDSD-Anwendungen verwendet werden können und andererseits ist gewährleistet, dass das Modell einer weiteren maschinellen Verarbeitung, nämlich der Testfallgenerierung, zugeführt werden kann.

1.2.5.2 Lösungsansatz zur Testfallgenerierung

Die Erstellung von Testfällen ist eine nicht-triviale Aufgabe, die für einen menschlichen Akteur bereits bei mittlerer Systemkomplexität kaum noch überschaubar ist. Software bietet i. d. R. UIs an, die das Zurückkehren zu früheren Zuständen des Dialogflusses erlaubt. Durch Push-Benachrichtigungen und andere Systemereignisse kann eine mobile Anwendung auf den gegenwärtigen mobilen Plattformen über zusätzliche Einstiegspunkte verfügen, die sich nicht unmittelbar aus der Anwendungsspezifikation ergeben.

Der in dieser Dissertation verfolgte Lösungsansatz zur automatisierten Generierung von Testfällen basiert auf Technologien der Modelltransformation. In einem ersten Schritt werden die unter Verwendung des UML-Profils zur kontextsensitiven Testfallmodellierung angereicherten Modelle durch Modell-zu-Modell Transformation (M2M) in ein intermediäres plattformunabhängiges Testmodell überführt. Dieses enthält alle zur Testdurchführung relevanten Informationen wie etwa Einstiegspunkte in das SUT, die Reihenfolge der auszuführenden individuellen Interaktionen mit dem SUT, zu tätigende Eingaben und insbesondere die Vorbedingungen der Testdurchführung für jede individuelle Interaktion mit dem SUT. In diesem Schritt werden die durch die Testspezifikation geforderten Abdeckungskriterien berücksichtigt. Durch Zyklen im Kontrollfluss kann hierbei auch für Anwendungen mit nur wenigen Funktionen eine sehr große Anzahl an Testfällen entstehen, deren manuelle Durchführung aufgrund ökonomischer Rahmenbedingungen ausgeschlossen ist.

In einem zweiten Schritt wird das so erzeugte Testmodell durch eine weitere Modelltransformation in technologiespezifische Testfälle überführt. Hier könnte durch Verwendung einer Modell-zu-Text Transformation (M2T) beispielsweise ein Testplan zur manuellen Durchführung durch einen menschlichen Akteur entstehen. Ebenfalls kann hier auch Quellcode für eine Automatisierungstechnologie entstehen, durch welche die Testdurchführung vollständig von menschlicher Arbeitskraft abgelöst wird. In dieser Dissertation wird hier konkret die M2T-Transformation zu Testfällen für eine modifizierte Variante des Automatisierungsrahmenwerks *Calabash-Android* realisiert.

1.2.5.3 Lösungsansatz zur Testautomatisierung

Die manuelle Durchführung von Tests ist i. d. R. mit hohen Kosten verbunden. Zudem handelt es sich um eine repetitive Tätigkeit, die u. U. erfordert eine hohe Anzahl von ähnlichen Testfällen durchzuführen. Aufgrund von Ermüdungseffekten sind menschliche Akteure für solche Tätigkeiten im Vergleich zu Maschinen nicht gut geeignet (vgl. Baur und Groth [30]). Die Herstellung spezifischer Vorbedingungen individueller Testfälle liegt zudem i. d. R. außerhalb menschlicher Möglichkeiten (vgl. Abschnitt 1.2.3.1 und Abschnitt 1.2.3.2). Je nach Plattform können Kontextparameter nur in minimalem Umfang auf Emluatoren nachgebildet werden.

Im Rahmen dieser Arbeit wurde deshalb eine Automatisierungstechnologie erarbeitet, die eine vollständig automatisierte Durchführung von Tests für Android-Anwendungen ermög-

licht. Die Einschränkung auf die Android-Plattform musste getroffen werden, da aufgrund der Verschiedenheit mobiler Plattformen eine plattformübergreifende Implementierung einer Automatisierungstechnologie unter Beibehalt von *Black-Box-Test* Fähigkeiten technologiebedingt ausscheidet. Die Plattform Android wurde gewählt, da es sich um OSS handelt, die es ermöglicht Veränderungen am Betriebssystem vorzunehmen.

Lösungselemente der Testautomatisierung umfassen die Modifikation des Android-Betriebssystems und die Modifikation des existierenden Testautomatisierungsrahmenwerks *Calabash*. Das Automatisierungsrahmenwerk Calabash (vgl. Abschnitt 2.5.2.6) ist ein komplexer Verbund unterschiedlicher Technologien, der nach Paradigmen des *Behavior-Driven-Development* (BDD) dazu geeignet ist, Testfälle auf der Ebene von Akzeptanztests in einer an die natürliche Sprache angelehnten Syntax zu formulieren. Hierdurch sind auch Stakeholder ohne technischen Hintergrund in der Lage, Testfälle zu verstehen und zu formulieren. Das Rahmenwerk übersetzt natürlichsprachliche Testfallbeschreibungen in Maschineninstruktionen, die lokal auf der Plattform des SUT unter Verwendung plattformspezifischer Instrumentierungstechnologie das SUT ausführen, testfallspezifische Stimuli erzeugen und das Verhalten des SUT beobachten.

Originär enthält der Calabash Instruktionssatz keine Befehle, um dem SUT während der Testdurchführung unter Ausblendung der tatsächlich durch die Gerätesensoren erzeugten Werte Sensordaten zuzuführen. Für das Testen kontextsensitiver Anwendungen ist das jedoch eine Schlüsselfunktion, da nur reproduzierbare Testdaten gewährleisten, dass durch einen Test zuverlässige und aussagefähige Ergebnisse erzielt werden. Die tatsächlich von den Gerätesensoren gemessenen Werte unterliegen immer natürlichen Schwankungen (d. h. Signalrauschen), so dass die mehrmalige identische manuelle Testdurchführung nicht möglich ist. Im Rahmen dieser Dissertation wurde der Calabash Instruktionssatz deshalb um Schlüsselwörter zur Manipulation von Sensordaten ergänzt. Durch diese Erweiterung ist es möglich, auch kontextsensitive Apps mit Calabash automatisiert zu testen.

Calabash stellt jedoch lediglich einen Befehlssatz bereit, der lokal auf dem SUT in Maschineninstruktionen übersetzt wird. Diese sind zunächst auf den Instruktionssatz der plattformspezifischen Instrumentierungstechnologie beschränkt. Im Fall der Android Plattform existieren auch hier keine adäquaten Mittel, um dem SUT zur Laufzeit des Tests Sensordaten zuzuführen. Auf Android Geräten stellt das Betriebssystem keine Schnittstellen bereit, um die Hardware-Sensoren zur Testdurchführung vom System zu entkoppeln und stattdessen Surrogat zu verwenden, die auf Black-Box-Test-Ebene die Zuführung spezieller Testdaten erlauben. Nur wenn für die Testdurchführung wiederholt identische Testdaten verwendet werden, sind aussagefähige Testergebnisse zu erwarten. Deshalb wurde im Rahmen dieser Dissertation eine Modifikation am Android-Betriebssystem erarbeitet, die es erlaubt die Hardware-Sensoren für die Durchführung von Tests durch Surrogat zu ersetzen, die Testdaten von externen Quellen bereitstellen. Aus der Sicht der zu testenden Anwendung erscheinen diese Testdaten so, als wären sie von den Hardware-Sensoren erzeugt worden. Es handelt sich also um einen Black-Box-Test-Ansatz, bei dem das SUT nicht modifiziert werden muss.

1.3 Zusammenfassung

Im Kapitel 1 wurde die in dieser Dissertation bearbeitete Problemstellung definiert und in einen wissenschaftlich-wirtschaftlichen Kontext eingebettet. Abschnitt 1.1 motiviert die Problemstellung und identifiziert das Testen mobiler Anwendungen als praxisrelevantes Problem, dass mit der im Jahr 2016 verfügbaren Technologie nicht gelöst wird. Das Testen mobiler Anwendungen ist im Vergleich zum Testen nicht-mobiler Software aus konzeptuellen und technischen Gründen besonders herausfordernd (vgl. Vieira et al. [351], Ridene et al. [297], Muccini et al. [258]). Dem durch das Testen verursachten Aufwand können gegenwärtig nur manuelle Testmethoden entgegen gestellt werden. Die Marktsituation mobiler Anwendungen erfordert jedoch schnelle Entwicklungszyklen bei gleichzeitig geringen Entwicklungskosten. Der hohe manuelle Aufwand des Testens steht der Kostenreduktion jedoch entgegen.

Der hohe Aufwand für das Testen wird durch technische Besonderheiten mobiler Plattformen verursacht, die insbesondere Mobilität und Kontextsensitivität als algorithmische Artfakte in die Anwendungsentwicklung einführen. Die konzeptuellen und technischen Herausforderungen des Testens mobiler Anwendungen werden in Abschnitt 1.2 im Detail diskutiert. Hieraus wird in Abschnitt 1.2.4 die Forschungshypothese abgeleitet, dass Automatisierungstechnologien in den Bereichen Testfallgenerierung und Testautomatisierung dabei helfen können, die Kosten für die Entwicklung mobiler Anwendungen zu reduzieren. Zur Realisierung dieser Kostenreduktionen werden in Abschnitt 1.2.5 Lösungsansätze vorgestellt, wie der manuelle Aufwand zur Durchführung einiger Aktivitäten in Qualitätssicherungsprozessen durch die Ablösung menschlicher Arbeitskraft reduziert werden kann. Kernelemente der Lösung sind hierbei die Verwendung von MDSD-Technologien zur Generierung von Testfällen aus Modellen des Systementwurfs. Zur automatisierten Durchführung von Softwaretests wird eine Technologie vorgeschlagen, die eine Simulation von Kontext (d. h. situative Faktoren des Softwarebetriebs, z. B. Standort des Anwenders, Orientierung des Gerätes, Sensormesswerte usw.) erlauben und somit das Testen am Arbeitsplatz erlauben. Dem gegenüber steht das Testen eines Softwareproduktes direkt in seiner zukünftigen Betriebsumgebung. Dies kann ein Aufsuchen testrelevanter Umgebungen erfordern und verursacht deshalb hohe Kosten.

1.4 Aufbau der Dissertationsschrift

Diese Dissertation ist in sieben Kapitel unterteilt. Kapitel 1 diskutiert die Grundlagen und die Motivation für den in dieser Dissertation untersuchten Ansatz zur Testautomatisierung mobiler, kontextsensitiver Anwendungen. Dieses Kapitel geht auf die historische Entwicklung mobiler Plattformen und deren Werkzeuge zur Softwareentwicklung ein. Insbesondere wird der Mangel an geeigneten Methoden und Technologien zur Automatisierung von Testaktivitäten aufgezeigt und hieraus die in Abschnitt 1.2.4 vorgestellte Forschungshypothese abgeleitet.

Als Grundlage und Einbettung der in dieser Dissertation entwickelten Methode und Technologie in das wissenschaftliche Umfeld werden in Kapitel 2 themenverwandte Arbeiten zu den Themen Qualität und Erfolg mobiler Anwendungen, Entwicklung und Testen mobiler Anwendungen, Generierung von Softwaretests sowie das existierende technologische Umfeld analysiert.

Die besondere Rolle von Mobilität und Kontextsensitivität für den Entwurf und die Entwicklung mobiler, kontextsensitiver Anwendungen werden in Kapitel 3 adressiert. Neben einer Einordnung der wesentlichen in dieser Dissertation verwendeten Begriffe wird insbesondere auf relevante Kontextparameter unter dem Aspekt des Testens eingegangen.

Kapitel 4 hat das Thema Testen kontextsensitiver, mobiler Anwendungen zum Gegenstand und diskutiert insbesondere die Auswirkungen von Kontextsensitivität auf den Softwaretest, wobei insbesondere auf den konkreten Einfluss spezifischer Kontextparameter eingegangen wird. In diesem Kapitel wird dargelegt, welche Aktivitäten des Testens vom Grundsatz her automatisierbar sind. Weiterer Inhalt von Kapitel 4 ist die Vorstellung unterschiedlicher Lösungsansätze zur Testautomatisierung, insb. im Umfeld mobiler Anwendungen.

Der Kerninhalt dieser Dissertation, die Entwicklung einer Methode und eines Werkzeugs zur modellbasierten Testautomatisierung für mobile, kontextsensitive Anwendungen, wird in Kapitel 5 dargelegt. Dort wird zunächst die Methode zur modellbasierten Testautomatisierung vorgestellt und im Anschluss die Modellierung von Testfällen für kontextsensitive, mobile Anwendungen thematisiert. Weiterhin werden die Generierung plattformunabhängiger Testfälle, die Generierung von plattformspezifischen Testfällen und deren automatisierte Ausführung mit modifizierten Implementierungen der Technologie Calabash und des Android-Betriebssystems im Detail diskutiert.

In Kapitel 6 erfolgt eine Validierung des in dieser Dissertation vorgestellten Ansatzes zur Testautomatisierung. Neben einer akademischen Untersuchung der Korrektheit der einzelnen Lösungselemente wird im Rahmen mehrerer Fallstudien die Methode und das Werkzeug zur Testautomatisierung auf Praxisbeispiele angewendet und evaluiert.

Eine abschließende Bewertung des Ansatzes und ein Ausblick auf fortführende Forschungsarbeiten erfolgt in Kapitel 7.

Kapitel 2

Stand der Wissenschaft und Technik zu Präsenz, Entwicklung, Modellierung und Testautomatisierung mobiler, kontextsensitiver Systeme

Testen ist wesentliches Werkzeug der Qualitätssicherung und nimmt im Softwareprozess eine hervorgehobene Rolle ein. Je nach gewähltem Vorgehensmodell entfallen bis zu 50% des Gesamtaufwands auf das Testen (vgl. Tassey [341], Ramler und Wolfmaier [291]), womit das Testen zum wesentlichen Kostenfaktor wird. Eine Reduzierung des Testaufwands mit dem Ziel die Gesamtkosten eines Softwareprojekts zu senken, kann durch unterschiedliche Strategien erreicht werden. In Frage kommt beispielsweise die Verlagerung von Testaktivitäten in Wirtschaftszonen, in denen Arbeitskraft im Vergleich zum Standort des IT-Unternehmens preiswert eingekauft werden kann (z. B. *Offshoring*, Outsourcing, *Crowdsourcing / Crowdtesting*). Diese Maßnahmen sind trotz eines erhöhten Organisationsaufwands geeignet, die Kosten für Softwaretests zu senken (vgl. Aspray et al. [20]). Sie sind hingegen nicht dazu geeignet, die Qualität von Testprozessen zu steigern. Die Qualität von Testprozessen ist jedoch für die Qualität und den Erfolg von Softwareprodukten von wesentlicher Bedeutung. Es gilt deshalb, diese Prozesse zu standardisieren und von schwer zu kalkulierenden Risikofaktoren wie z. B. der Qualität menschlicher Arbeitskraft zu entkoppeln. Beide Ziele können durch Automatisierungstechnologien erreicht werden (vgl. Harrold [170]).

Ein Computer oder ein Softwarewerkzeug das einen Test ausführt, ist in der Lage, diesen bei jeder Ausführung identisch zu wiederholen. Dies ist für das Testen von entscheidender Bedeutung, erfordert aber von menschlichen Akteuren ein Höchstmaß an Disziplin. Da Tests häufig unter Zeitdruck durchgeführt werden, ist mit menschlichen Fehlern zu rechnen, die den gesamten Softwaretest kompromittieren können (z. B. wegen psychischer Belastung und Beanspruchung durch monotone Tätigkeiten, Baur und Groth [30]). Weiterhin kann Automatisierungstechnologie durch Verwendung von CI-Werkzeugen so in den Softwareprozess einge-

führt werden, dass ein Mindestmaß an Softwarequalität bereits zum Zeitpunkt des Einpflegens von Anpassungen in die Codebasis erzwungen wird. Das heißt, die erfolgreiche Ausführung von Tests oder eine vorgegeben Testabdeckungsquote ist eine Bedingung zur Einpflege in die Codebasis. CI-Werkzeuge verhindern, dass defekter Quellcode in die Codebasis gelangt. Insbesondere ist die Ausführung von Tests durch Maschinen i. d. R. deutlich günstiger und schneller als eine Testdurchführung durch menschliche Akteure.

IT-Unternehmen haben deshalb bereits früh damit begonnen, Testprozesse zu standardisieren und zu automatisieren. Hierzu wurden entsprechende Methoden und Werkzeuge erarbeitet, durch welche garantiert werden soll, dass ein Softwareprodukt zum Zeitpunkt der Auslieferung definierten Qualitätsanforderungen genügt. Hierzu ist es notwendig, Automatisierungswerkzeuge mit dem Soll-Zustand einer Software zu parametrisieren, damit ein Werkzeug im Rahmen einer definierten, reproduzierbaren Ausführung des SUT Abweichungen des Ist-Zustands vom Soll-Zustand erkennen kann. Es müssen Testdaten und ein Plan zur Testdurchführung, d. h. Art und Abfolge von Instruktionen, festgelegt und in einem maschinenlesbaren Format aufbereitet werden. Diese Testdaten und die Abfolge der Instruktionen die während der Testdurchführung im SUT ausgeführt werden, bilden das sogenannte Testmodell.

In den folgenden Abschnitten 2.1 bis 2.5 wird der aktuelle Stand der Wissenschaft und Technik unter den Aspekten Testautomatisierung im Allgemeinen, Modellierung von Mobilität und Kontextsensitivität in Softwaresystemen und Testautomatisierung mobiler, kontextsensitiver Systeme im Besonderen diskutiert. Es wird insbesondere aufgezeigt, dass Testautomatisierung ein Thema ist, dem seit der allgegenwärtigen Marktpräsenz von Smartphones und Tablets zwar zunehmend Aufmerksamkeit in Wissenschaft und Technik gewidmet wird, bislang aber nur wenige Erfolge erzielt wurden, die rasch durch voranschreitende Weiterentwicklung mobiler Geräte obsolet werden.

2.1 Qualität und Erfolg mobiler Anwendungen

Im kurzlebigen Markt mobiler Anwendungen kann ein Softwareprodukt nur dann erfolgreich realisiert werden, wenn die Qualität eines Softwareproduktes in einem Maß sichergestellt werden kann, dass Anwendungsdefekte das Nutzungserlebnis nicht einschränkten. Mobile Anwendungen im verbraucherorientierten Mark werden i. d. R. zu einem geringen Preis in den App Stores angeboten. Die Generierung von Gewinn ist daher nur über eine hohe Anzahl von Verkäufen möglich. Die Dynamik der Märkte für mobile Anwendungen lässt jedoch nur wenig Spielraum für die Nachbesserung von Produkten, da alternative Produkte i. d. R. schnell und unkompliziert verfügbar sind. Die Anzahl von App-Downloads ist daher eine direkte Erfolgsmetrik. Die App Stores bieten Bewertungssysteme an, in denen Anwender ihre Zufriedenheit mit einem Produkt summativ bewerten können, i. d. R. mit einer Bewertung von 1-5 Sternen, wobei eine 5-Sterne-Bewertung das Maximum der Kundenzufriedenheit repräsentiert.

Eine im Jahr 2011 von Kim et al. [223] durchgeführte Studie kommt zu dem Ergebnis, dass die App-Bewertung einer der entscheidenden Faktoren bei der Kaufentscheidung ist. Zhong und Michahelles [379] untersuchen im Jahr 2013 den Zusammenhang zwischen der Bewertung einer App und deren Erfolg anhand der Metrik Anzahl Downloads. Die Autoren kommen übereinstimmend mit Kim et al. zu dem Ergebnis, dass Anwendungen mit niedriger

Bewertung – also geringer Anwenderzufriedenheit – signifikant weniger Downloads verzeichnen als Apps mit hohen Bewertungen. Der Zusammenhang zwischen der App-Bewertung und der App-Qualität wird durch eine ebenfalls im Jahr 2013 von Khalid [220] durchgeführte Studie bestätigt. Weiterhin suggeriert die von Zhong und Michahelles durchgeführte Studie, dass die App Stores keine *Long-Tail*-Märkte (vgl. Gladwell [124], engl. Märkte, die Erlös langfristig mit Nischenprodukten erwirtschaften) sind und nicht zu erwarten ist, wenigstens langfristig mit einer großen Anzahl wenig oder mittelmäßig erfolgreicher Produkte großen Profit zu erzielen. Im Gegenteil handelt es sich bei den App Stores um *Superstar*- oder *Blockbuster*-Märkte [379], in denen wenige erfolgreiche Produkte in kurzen Zeitintervallen den Großteil des Gesamtgewinns des Marktes erzielen.

Wenngleich eine positive App-Bewertung kein Garant für ein erfolgreiches Produkt ist, ist es umgekehrt jedoch so, dass ein Produkt mit niedriger Bewertung kaum eine Chance auf wirtschaftlichen Erfolg hat. Da der Zusammenhang zwischen App-Bewertung und ihrer Qualität (als Maß für die Abwesenheit von Defekten) von Khalid [220] nachgewiesen wurde, kann gefolgert werden, dass die Qualität eines Softwareproduktes im Umfeld mobiler Anwendungen unmittelbar auf die Chancen auf wirtschaftlichen Erfolg einwirkt. Hieraus erwächst die Motivation dieser Dissertation, eine Methode und Technologie bereitzustellen, die qualitätssichernde Aktivitäten bei der App-Entwicklung unterstützen.

2.2 Mobilität und Kontextsensitivität in mobiler Software

Die Fähigkeiten moderner Kommunikationsgeräte geht bei Weitem über Telefonie und den Versand von Textnachrichten hinaus. Geräte der Smartphone-Generation sind in ihrer Leistungsfähigkeit in etwa vergleichbar mit Desktop-Computern des vorigen Jahrzehnts. Die Realisierung von Anwendungsinhalten wird i. d. R. nicht durch Faktoren wie Speicherplatz oder Prozessorgeschwindigkeit eingeschränkt. Mobile Geräte zeichnen sich gegenüber Desktop-Computern vorrangig dadurch aus, dass sie durch ein umfangreiches Sortiment an Sensoren in der Lage sind, Parameter ihrer Umgebung zu messen und diese Daten für Anwendungsinhalte aufzubereiten. Anwendungsfälle für solche Technologien sind beispielsweise ortsbasierte Zugriffskontrolle für Softwaresysteme, die Verwendung von Standortinformationen für Fahrzeug- oder Fußgängernavigation oder die Verwendung von Sensormesswerten (z. B. Magnetfeldsensor oder Beschleunigungssensor) zur Interaktion mit einer Anwendung oder zur grafischen Aufbereitung in *Augmented Reality* (AR)-Anwendungen.

Die Betreiber mobiler Plattformen stellen zur Realisierung solcher Anwendungen einfach zu bedienende API-Schnittstellen bereit. Ein Mangel tut sich allerdings bei der Dokumentation kontextsensitiver Anforderungen in strukturierten Softwareentwicklungsprozessen auf, in denen Anforderungen an zu entwickelnde Anwendungen formal dokumentiert und in Vorbereitung der Implementierung in Systemmodellen fixiert werden. Modellierungssprachen und -werkzeuge wie z. B. die UML sehen in ihrem Vokabular keine Elemente zur Modellierung von Kontext vor. Im günstigsten Fall können generische Modellierungsartefakte, wie z. B. Nachrichtenflüsse in BPMN-Modellen, Informationsobjekte in EPK-Modellen oder Annotationen in UML-Modellen für diese Aufgabe herangezogen werden. Dedizierte Notationselemente existieren indes nicht in üblichen Modellierungssprachen. Es besteht ein Bedarf, diese Informationen

zukünftig in der Modellierung von Softwaresystemen und Testfällen formal zu berücksichtigen. Nur so ist gewährleistet, dass alle Stakeholder ein einheitliches Verständnis dafür entwickeln, welchen Einfluss Kontextinformationen auf ein Softwaresystem haben.

In der Vergangenheit wurde die Verwendung und die Modellierung von Standort- und Kontextinformation bereits in Forschungsarbeiten thematisiert. Anwendungsfälle sind hierbei u. a. Darstellung von Standortinformationen für AR-Anwendungen und auch Ansätze zum modellbasierten Testen kontextsensitiver Softwaresysteme. Eine Auswahl dieser Arbeiten wird in den folgenden Abschnitten diskutiert und hinsichtlich ihrer Relevanz für das in dieser Dissertation erarbeitete Konzept zur Generierung von Testfällen aus Systemmodellen bewertet.

2.2.1 Verwendung von Kontextinformationen in mobilen Anwendungen

Eine Reihe von Forschungsarbeiten beschäftigt sich mit der Auswertung von Kontextinformationen zur Feststellung des anwenderbezogenen Nutzungskontext des Geräts mit der übergeordneten Zielstellung, Aussagen über Situation, Zustand und Aktivitäten des Anwenders treffen zu können. Diese Arbeiten haben den Kontext des Anwenders und der Gerätenutzung unter unterschiedlichen Aspekten zum Gegenstand. Einige Arbeiten beschäftigen sich primär aus soziologischen Aspekten mit der Frage *wann*, *warum*, *wie* und *wo*, d. h. in welchem Kontext, der Anwender sein mobiles Gerät mit sich führt. Andere Arbeiten haben technische Aspekte der Kontextanalyse zum Gegenstand.

Wenngleich im Jahr 2000 die Leistungsfähigkeit mobiler Geräte deutlich hinter der von Geräten der Smartphone-Generation zurücklag, wurden kontextsensitive Anwendungen bereits produktiv eingesetzt. Cheverst et al. [78, 77] dokumentieren in ihrer Arbeit aus dem Jahr 2000 ihre Erfahrungen mit einem kontextsensitiven Touristenleitsystem in der Stadt Lancaster. Als Abgrenzung zu Entwicklungen in der Smartphone-Ära, die i. d. R. auf bereits existierender Infrastruktur (z. B. GPS) basieren, kam im von Cheverst et al. vorgestellten Projekt *GUIDE* ein proprietäres Kommunikationsnetzwerk zum Einsatz, über welches ebenfalls Standortinformation generiert werden konnten. Zu den Kernanforderungen an das System gehörte insbesondere Kontextsensitivität, d. h. situative Faktoren des Anwenders und dessen Umgebung (vgl. Abschnitt 3.1.1.5). Anwendungsinhalte wurden für den Anwender basierend auf dessen Kontext, d. h. Standort, Tageszeit (Berücksichtigung von Öffnungszeiten von Touristenattraktionen) und spezifischen Interessen, aufbereitet. Als besondere Herausforderung beschreiben die Autoren die Erstellung eines geeigneten Datenmodells, das in der Lage war, Inhalte in Verbindung mit zugehörigen Metadaten (z. B. geographischer Standort) abzubilden. Ebenfalls berücksichtigt die *GUIDE* Softwarearchitektur Eigenschaften drahtloser Kommunikationskanäle, d. h. schwankende Übertragungszeiten und ortsabhängig instabile Netzwerkverbindungen. Die technischen Voraussetzungen für mobile, kontextsensitive Anwendungen wie dem Projekt *GUIDE* haben sich seit 2000 nur in der konkreten Ausführung der verwendeten Technologien verändert. Mittlerweile kommen i. d. R. keine proprietären Kommunkationstechnologien zum Einsatz, sondern es werden vorrangig Standardtechnologien wie UMTS, LTE oder Wi-Fi für die Kommunikation und GPS zur Lokalisierung des Anwenders verwendet. Eine Ausnahme hiervon bildet die Lokalisierung innerhalb geschlossener Räume, bei der GPS technologiebedingt ausscheidet und stattdessen alternative Technologie, z. B. *Bluetooth Low Energy* (BLE)-*Beacons* verwendet werden müssen.

Darüber hinaus sehen sich Entwickler mobiler, kontextsensitiver Anwendungen im Jahr 2016 noch immer mit ähnlichen Problemen konfrontiert wie Cheverst et al. im Jahr 2000. Netzwerkverbindungen mobiler Geräte sind flüchtig und stellen Anforderungen an Anwendungen, ihren Bestimmungszweck auch ohne Netzwerkverbindung noch adäquat zu erfüllen. Auch sind kontextsensitive Anwendungen auch im Jahr 2016 noch nicht in der Lage, den Standort des Anwenders sicher (d. h. ohne Abweichung zwischen gemessenem und tatsächlichem Standort) zu bestimmen. Hieraus erwächst die Notwendigkeit, mit Nichtverfügbarkeit von Standortinformationen oder Netzwerkverbindungen umzugehen, ohne die UX einer Anwendung wesentlich zu beeinträchtigen. Intensives Testen kontextsensitiver Funktionen unter realistischen Bedingungen ist deshalb notwendig, um Softwarequalität sicherzustellen. Die Herstellung realistischer Testumgebungen mit ökonomisch sinnvollem Aufwand ist hingegen ein ungelöstes Problem. Automatisierungstechnologie – wie beispielsweise die in dieser Dissertation untersuchte – kann dabei helfen, den zum Testen notwendigen manuellen Arbeitsaufwand zu reduzieren.

Die Art und Anzahl der in Smartphones integrierten Sensoren ermöglichen Anwendungen, die das Umfeld und den Anwender aktiv und passiv beobachten und aus den so ermittelten Informationen algorithmisch oder heuristisch versuchen Rückschlüsse auf die Aktivität des Anwenders zu ziehen. Mögliche Anwendungsfälle sind vielgestaltig und die Verwendung von Gerätesensoren obligatorisch. Eine Reihe Forschungsarbeiten beschäftigt sich mit dem Thema *Activity Tracking* (engl. inferieren und protokollieren der Aktivität eines Akteurs), d. h. dem Versuch, aus Sensordaten Informationen über zu einem Zeitpunkt vom Anwender durchgeführte Aktivität zu erkennen.

Sun et al. [337] diskutieren in ihrer Forschungsarbeit eine Variante des Activity Tracking, bei welchem Anwender außer dem Smartphone keine weiteren Sensoren mitführen müssen. Im Kontrast zu vorherigen Forschungsarbeiten anderer Autoren verfolgen Sun et al. das Ziel, lediglich den im Smartphone vorhandenen Beschleunigungssensor zu verwenden. Das Smartphone darf zudem verhältnismäßig frei in einer Tasche des Trägers positioniert werden. Hierbei handelt es sich aufgrund der Ergebnisse der von Ichikawa et al. [191] durchgeführten Studie um eine Kernanforderung, da Anwender erwarten, dass sich ein Gerät zum Activity Tracking in das gewohnte Nutzungsverhalten des Smartphone integriert ohne weitere Hürden zu schaffen. Durch trainieren eines Algorithmus mit experimentell erworbenen Daten waren die Autoren in der Lage, die Aktivitäten Ruhe, Laufen, Rennen, Radfahren, Treppensteigen aufwärts, Treppensteigen abwärts und Fahren mit 94,8% Sicherheit aus den Werten des Beschleunigungssensors abzuleiten. Sun et al. haben somit gezeigt, dass Sensoren in Smartphones ein zuverlässiges Werkzeug sind, um aus passiver Beobachtung der Betriebsumgebung und situativen Faktoren (d. h. Kontext) des Anwenders Mehrwert zu generieren.

Sun et al. erarbeiteten ihre Ergebnisse durch manuelle Experimente, die über einen Zeitraum von drei Wochen Arbeitskraft von sieben Freiwilligen in Anspruch nahmen. Die Autoren thematisieren nicht, wie der Algorithmus im Anschluss an die Trainingsphase getestet wurde. Eine geeignete Automatisierungstechnologie – wie beispielsweise die in dieser Dissertation vorgestellte – hätte hier dabei helfen können, Algorithmen des Activity Tracking hinsichtlich Korrektheit und Robustheit zu überprüfen, ohne dass hierzu in hohem Maße menschliche Arbeitskraft gebunden werden muss.

Wiese et al. [361] untersuchen in ihrer Arbeit die Verwendung von Sensorinformationen zur Feststellung, wie ein mobiles Geräte unterwegs geführt wird. Basierend auf der intuiven Annahme, dass Anwender ihr Gerät in Hosen-, Jacken- oder Handtaschen tragen, führten die Autoren in einem ersten Schritt demographisch repräsentative Personenbefragungen zum Ablageort des Smartphones zum exakten Zeitpunkt der Befragung durch. Hierdurch wurden Klassen typischer Ablageorte des mobilen Geräts identifiziert: Schreibtisch, vordere Hosentasche, Handtasche, Aktentasche oder Rucksack, in der Hand gehalten, hintere Hosentasche, im Auto, Hemdtasche. Eine Anschlussbefragung zur Ablageort des Mobilfunktelefons innerhalb der letzten 24 Stunden ergab ähnliche Ergebnisse der Verteilung, so dass der Schluss naheliegt, dass Anwender deutliche Präferenzen zum Ablageort ihres Smartphones haben, die zudem in Korrelation mit der ausgeübten Tätigkeit (Gehen, Schlafen, Sport, Arbeit, etc.) stehen. Basierend auf diesen Erkenntnissen entwickelten Wiese et al. eine Anwendung, die über einen Versuchszeitraum die Daten des Beschleunigungssensors aufzeichnete und die Anwender zyklisch zum Ablageort des Telefons befragte. Aus diesen Daten extrahierten die Autoren ein mathematisches Modell, mit dem der Ablageort des Smartphones basierend auf der Auswertung von Sensordaten, insbesondere Beschleunigung, Annäherung und Helligkeitsniveau, mit einer hohen Genauigkeit berechnet werden konnte. Hieraus wiederum konnte auf situative Faktoren des Anwenders geschlossen werden.

Die Arbeit von Wiese et al. ist für diese Dissertation interessant, weil sie physikalische Kontextparameter (d. h. mit Sensoren messbare Eigenschaften der Umgebung) auf logische und fachliche Kontextparameter abbildet, d. h. auf Situation und Aktivität des Anwenders. Die so gewonnenen Erkenntnisse können in Software nutzbar gemacht werden, um das Verhalten von Anwendungen auf dem Smartphone zu steuern. Wiese et al. führen als Beispiel an, dass aufgrund des Nutzungskontextes des Geräts algorithmisch darauf geschlossen werden kann, ob das Gerät aktiv genutzt wird oder passiv an einem Ablageort aufbewahrt wird. Hieraus erwächst eine Vielzahl möglicher Anwendungsfälle (z. B. automatische Anpassung der Klingeltonlautstärke, Vermeidung unbeabsichtigter Interaktion durch Sperrung von Bedienelementen), die bei ihrer Implementierung hinsichtlich der Zuverlässigkeit des Inferierens des Nutzungskontextes getestet werden müssen.

Um den Aufwand für solche Tests zu minimieren und gleichzeitig den Nutzen zu maximieren kann Automatisierungstechnologie eingesetzt werden, die in der Lage ist Kontextfaktoren zu simulieren. Ohne den Einsatz einer geeigneten Automatisierungstechnologie ist die exakte Reproduzierbarkeit von Testdaten praktisch ausgeschlossen. Die in dieser Dissertation erarbeitete und in Kapitel 5 diskutierte Automatisierungstechnologie erfüllt diesen Zweck.

Jung et al. [206] untersuchen im Jahr 2015 einen weiteren Ansatz des Activity Tracking mit einem Fokus auf Sicherheitsthemen. Hier argumentieren die Autoren, dass Kontextinformationen dabei helfen können, Entscheidungen über Zugangsberechtigungen zu treffen, in dem situative Faktoren des Anwenders algorithmisch bewertet werden. Weitgehend losgelöst von diesem Anwendungsfall befassen sich Jung et al. damit, wie aus Kontextparametern, insbesondere aus Messwerten von Gerätesensoren, auf die Aktivität des Anwenders geschlossen werden kann. Hierzu analysierten die Autoren die in einem Experiment erhobenen Daten, um signifikante Zusammenhänge zwischen der konkreten Ausprägung von Kontextparametern und der Anwenderaktivität zu identifizieren. In diesem Vorgehen existieren wesentliche Par-

allelen zum bereits von Sun et al. untersuchten Ansatz des Activity Tracking. Auch bei Jung et al. wird nicht thematisiert, wie der resultierende Algrithmus zum Activity Tracking getestet wurde. Aufgrund der Nichtexistenz adäquater Testautomatisierungstechnologien, die in der Lage sind, Kontext zu simulieren, kommen deshalb nur manuelle Tests in Frage.

2.2.2 Modellierung von Kontextinformationen in mobilen Anwendungen

Die Erstellung kontextsensitiver Anwendungen in geordneten Softwareprozessen erfordert eine der Domäne adäquate Modellierungstechnologie. Bereits bevor Smartphones zu Alltagsgegenständen avancierten, die in nahezu jeden Lebensbereich vordringen, wurde das Problem der Modellierung und Verarbeitung von Kontextinformationen wissenschaftlich thematisiert.

Unter dem Begriff *Nomadic-aware Computing* (engl. nomadisches, d. h. umherziehendes, nicht standortgebundes Computerwesen im Sinne von Mobilität von Gerät und Anwender) betrachtete Kleinrock [225, 226] im Jahr 1996 zukünftige Anforderungen an Softwarearchitekturen mit der Erkenntnis, dass mobile, kontextsensitive Softwaresysteme nur dann effizient zu entwickeln sind, wenn Mobilität und Kontextsensitivität bereits zum Entwurfzeitpunkt berücksichtigt werden und nicht erst nachträglich in ein Softwareprodukt eingepflegt werden. Die tatsächliche Leistungsfähigkeit und vor allem die Vielfalt von Sensoren von Geräten der Smartphone-Generation konnte Kleinrock 1996 nicht abschätzen. Dennoch berücksichtigt er in seiner Forschungsarbeit bereits Kernkonzepte der Kontextsensitivität, nämlich Standort, Gerät, Netzwerkverbindung und Mobilität (vgl. Abschnitt 3.1.2), die auch im Jahr 2016 noch Forschungsfragen aufwerfen. Hierzu gehört unter anderem die von Kleinrock erkannte Notwendigkeit eines Referenzmodells, mit dem sich *Computational Nomadicity* [226] (engl. Computersysteme und Anwender betreffende Ortsungebundenheit, entspricht Mobilität im Sinne der in dieser Dissertation angewendeten Definition) in Softwaremodellen beschreiben lässt. Der Autor diskutiert eine Reihe existierender Architekurmodelle und Optionen, wie diese an die neuartigen Anforderungen angepasst werden können. Allerdings fokussieren diese Betrachtungen primär auf die in mobilen Szenarien unzuverlässigen Netzwerkverbindungen auf Ebene der Anwendungsarchitektur. Modelle, die Verhaltensaspekte einer Anwendung zum Gegenstand haben, werden von Kleinrock nicht betrachtet. Für das in dieser Dissertation verfolgte Ziel der automatischen Generierung von Testfällen aus Systemmodellen ist die Arbeit von Kleinrock deshalb nur insofern relevant, als dass sie eine Notwendigkeit der Modellierung von Kontextfaktoren bereits in frühen Phasen des Softwareprozesses voraussetzt. Der in dieser Dissertation untersuchte Ansatz zur Testfallgenerierung basiert ebenfalls auf der Annahme, dass Kontextfaktoren bereits früh berücksichtigt werden müssen. Dies gilt insbesondere für das Testen, für dessen effiziente Durchführung bei allen Stakeholdern ein Verständnis für den Einfluss von Mobilität und Kontextsensitivität auf das Laufzeitverhalten von Softwaresystemen vorhanden sein muss.

Mobile Geräte vor der Smartphone-Generation verfügten nicht über die Vielfalt an Sensoren wie Geräte der Gegenwart. Auch die Lokalisierung des Anwenders war noch ein technisches Problem, da GPS-Module in mobilen Geräten keine Selbstverständlichkeit waren. Im Jahr 1999 war jedoch bereits absehbar, dass die Technik zukünftiger Gerätegenerationen mehr Möglichkeiten bieten würde, als lediglich die Bestimmung des Standorts des Anwenders. Schmidt et al. [314, 315] argumentieren deshalb bereits 1999, dass Kontextsensitivität

die Art der Interaktion von Anwendern mit Anwendungen prägen wird. Die Autoren entwerfen ein Kontextmodell, in dem sowohl physikalische Faktoren (z. B. Standort, Temperatur, Beschleunigung) der Betriebsumgebung einer mobilen Anwendung, als auch menschliche Faktoren und das soziale Umfeld des Anwenders berücksichtigt waren. Weiterhin diskutieren Schmidt et al., inwiefern Kontextfaktoren verwendet werden können, um die *User Experience* (UX, engl. Anwendererfahrung, Anwendererlebnis) mobiler Anwendungen auch in mobilen Szenarien aufzuwerten. Hierbei wurden bereits erste Ideen zur Fusion bzw. Aggregation von Kontextinformationen thematisiert, die Schlussfolgerungen auf situative Faktoren des Anwenders aus physikalischen Parametern der Betriebsumgebung erlauben.

Annahmen über die zukünftige Entwicklung mobiler Geräte sind aus der Perspektive des Jahres 2016 in den wesentlichen Aspekten eingetreten. Nahezu alle mobilen Plattformen verfügen über eine Sensor-API, mit der Kontextparameter erfasst werden können. Einige Plattformen bieten ebenfalls APIs zur Detektion der gegenwärtig vom Anwender ausgeführten Aktivität an (z. B. die *ActivityRecognitionApi* [127] auf der Plattform Android). Fragestellungen der formalen Berücksichtigung von Kontext in formalisierten Softwareprozessen und in Aktivitäten der Qualitätssicherung sind jedoch auch im Jahr 2016 nicht vollständig beantwortet.

Im Jahr 2003 befassen sich Chen et al. [75, 76] mit einem ontologiebasierten Kontextmodell. Die Autoren unterstellen, dass die Verwendung von Kontext in Softwaresystemen eine semantische Wissensbasis zur Interpretation von Ortsinformationen und Daten von Sensoren erfordert. Die bisherige Praxis Kontext lediglich als Objekte im Sinne der objektorientierten Programmierung zu behandeln, wird den Anforderungen moderner kontextsensitiver Anwendungen nicht gerecht, da ein maschinelles Infererieren von höherklassigem Kontext (d. h. komplexe situative Beschreibungen, z. B. „in einer Besprechung“, „auf einer Zugfahrt“, „im Auto“) aus niederklassigen Kontextparametern (d. h. physikalische Parameter der Betriebsumgebung) hierdurch nicht ermöglicht wird. Weiterhin beschreiben die Autoren eine Architektur eines Kontext-*Brokers* (engl. Makler, Vermittler, Zwischenhändler), der Anwendungen ermöglicht, Kontextinformationen miteinander auszutauschen. Hierzu entwerfen Chen et al. eine Kontextontologie, die den Kontextparameter Standort in unterschiedlichen Perspektiven abbildet. Insbesondere ist die hierarchische Beziehung von Standortinformationen untereinander, z. B. regionales Enthaltensein von Standortspezifikationen, Gegenstand der Betrachtung. Aus diesem Grund sind die Arbeiten von Chen et al. für diese Dissertation insofern relevant, als dass die Anforderungen an ein Kontextmodell, insbesondere hinsichtlich der Standortmodellierung, teilweise auf das hier verwendete Metamodell zur Kontextmodellierung (Abschnitt 5.2.2.2) übertragbar sind. Darüber hinaus untersuchen Chen et al. Kontext mit einem Fokus auf die Interaktion von Anwendungen untereinander und mit dem Kontext-Brokers, nicht jedoch auf das Testen dieser Anwendungen.

Korpipää und Mäntyjärvi [229, 228] untersuchen in ihrer Arbeit aus dem Jahr 2003 ebenfalls einen ontologiebasierten Ansatz Kontextparameter zu modellieren. Der Ansatz der Autoren zielt nicht darauf ab, Kontext während der Systemmodellierung darstellbar zu machen, sondern vielmehr zur Laufzeit einer Anwendung unter Abstraktion von Rohdaten von Sensoren situative Faktoren des Anwenders in einer Anwendung nutzbar zu machen. Hierbei setzen die Autoren bereits ein Modell mobiler Anwendungen voraus, das weitgehend dem Modell von Apps der Smartphone-Ära entspricht, also Anwendungen die physikalische und

fachliche Parameter ihrer Ausführungsumgebung als Eingabevektoren in die Fachlogik verwenden. Korpipää und Mäntyjärvi setzen hierbei allerdings einen starken Fokus auf physikalische Kontextparameter, wenngleich das von ihnen vorgestellte Kontextmodell prinzipiell auch auf logische und fachliche Parameter erweiterbar ist. Insbesondere adressieren die Autoren Anforderungen an ein Metamodell zur Modellierung von Kontext. Zu diesen gehören Simpilizität der Verwendung, Flexibilität bzw. Erweiterbarkeit auf weitere Kontextparameter, Praktikabilität in der algorithmischen Verwendung, Generizität im Hinblick auf den domänenspezifischen Einsatz, Ausdrucksstärke, Effizienz und schließlich Unterstützung maschineller Folgerbarkeit von Fakten aus dem Kontextmodell.

Einige dieser Anforderungen sind ebenfalls relevant für das in dieser Dissertation zur Generierung von Testfällen verwendete Kontextmodell. Diese werden in Abschnitt 5.2.2.1 diskutiert und im Hinblick auf das hier untersuchte Konzept einer Testautomatisierungstechnolgie bewertet. Darüber hinaus definieren Korpipää und Mäntyjärvi bereits detaillierte Eigenschaften individueller Kontextparameter, die ebenfalls in dem in dieser Dissertation verwendeten Kontextmetamodell erneut aufgegriffen werden. Weiterhin diskutieren Korpipää und Mäntyjärvi eine mögliche Middleware zur Realisierung kontextsenstiver Anwendungen. Dieser Aspekt ist im Rahmen dieser Dissertation jedoch nicht relevant. Das hier untersuchte Konzept zur Testautomatisierung mobiler Anwendung hat das Ziel, einer getesteten Anwendung durch geeignete Simulationstechnologie einen Kontext bereitzustellen, der die wesentlichen Eigenschaften der zukünftigen Betriebsumgebung dieser Anwendung abbildet. Auf welche Art diese Anwendung Informationen zu ihrem Kontext interpretiert ist hier nicht Gegenstand der Betrachtung. Vielmehr liegt der Fokus darauf, diejenigen Kontextparameter, die eine Anwendung in der Lage ist, durch Zugriff auf Gerätesensoren wahrzunehmen, während des Testens gemäß der Testspezifikation in die Ausführungsumgebung einzubringen.

Einen UML-basierten Ansatz zur Modellierung von Kontext in Anwendungen untersuchen Baumeister et al. [29] im Jahr 2003. Die Autoren argumentieren, dass Werkzeuge zur Spezifikation von Sofwaresystemen nicht über die geeigneten Darstellungsmittel verfügen, Kontext in Systemmodellen abzubilden. Mit einem Fokus auf Standortinformationen entwerfen sie deshalb ein Metamodell, das Modellierungselemente bereitstellt, mit denen Ortsabhängigkeit von Komponenten eines Softwaresystems abgebildet werden kann. Konkret erstellen die Autoren ein UML-Profil mit dem Strukturmodelle durch UML-Stereotypen annotiert werden können, die eine Komponente einem Ort zuordnet. Das Metamodell zur Standortmodellierung nach Baumeister et al. unterscheidet hierbei unterschiedliche Arten der Standortdefinition. Standorte sind hierarchisch organisiert, können einander enthalten, können stationär sein oder selbst mobil sein. Hierdurch ist es möglich mobile Entitäten (z. B. Auto, Zug) als Standort anderer Entitäten zu spezifizieren. Das in dieser Dissertation verwendete Metamodell zur Kontextmodellierung verwendet einen ähnlichen Ansatz. Ebenfalls wird ein UML-Profil verwendet, um Kontext in Systemmodelle zu integrieren. Allerdings fokussiert diese Dissertation auf Verhaltensaspekte von Softwaresystemen, so dass grundlegend andere Metaklassen, nämlich die der UML-Verhaltensdiagramme erweitert werden.

Historisch hat das Software Engineering für kontextsensitive Softwaresysteme Wurzeln in der Verfügbarkeit von Standortinformationen in mobilen Geräten. Standortinformationen basierend auf geodätischen Formaten abzubilden, wie z. B. dem *World Geodetic System 1984*

(WGS84), ist für viele Anwendungen ausreichend. Allerdings ist es häufig unter unterschiedlichen Aspekten unpraktisch. Beispielsweise funktionieren GPS-Empfänger innerhalb von Gebäuden nur ungenau, so dass Standortinformationen in WGS84-Koordinaten innerhalb von Gebäuden nicht sinnvoll verwendet werden können. Selbst bei ausreichender Genauigkeit sind numerische Koordinaten für menschliche Akteure – die letztlich Konsumenten in Software aufbereiteter Inhalte sind – schwer fassbar. Es ist deshalb von Vorteil, Standortinformationen neben geodätischen Formaten auch in symbolischen Formaten abzubilden. Hu und Lee [189] erarbeiten hierzu ein Konzept, wie symbolische Ortsinformationen in einem Modell abgebildet werden können. Grundlage ist die Annahme, dass bisher kein Modell in der Lage ist, Ortsinformationen in einem adäquaten Format zu repräsentieren, während solche Anwendungen jedoch ein Format voraussetzen, welches von Algorithmen verarbeitet werden kann, um diese Informationen in Anwendungen nutzbar zu machen.

Mit dem Ziel zusätzlich semantische Informationen im Modell zu verankern, definieren Hu und Lee deshalb ein Konzept eines Metamodells, in welchem Entitäten neben beliebigen anderen Attributen eine begrenzte physikalische geografische Ausdehnung zugeordnet werden. Diese Entitäten stehen untereinander in Beziehung, wodurch eine semantische Navigation zwischen Entitäten ermöglicht wird. Hierbei handelt es sich allerdings lediglich um Aufbereitung von Standortinformationen, um Plätzen Namen zuzuweisen. Für eine sinnvolle Verwendung bleibt es auch in diesem Ansatz notwendig, symbolischen Orten numerische Koordinaten in einem definierten Referenzsystem zuzuordnen. Aus diesem Grund kann ein rein symbolisches Referenzsystem für die Testautomatisierung mobiler Anwendungen nicht ausreichend sein. Gegenwärtig verfügbare Ortungstechnologie in mobilen Geräten liefern immer Koordinaten im WGS84-Referenzsystem. Eine Technologie zur Automatisierung von Tests für ortsbasierte Anwendungen ist deshalb zwingend auf WGS84-Koordinaten angewiesen. Das in dieser Dissertation entworfene Metamodell zur Standortmodellierung basiert deshalb auf dem WGS84-Referenzsystem. Symbolische Standortinformationen werden lediglich insofern unterstützt, als dass sie eindeutig auf WGS84-Koordinaten abgebildet werden können.

Grassi et al. [156] untersuchten im Jahr 2004 ein Konzept, Mobilitätsaspekte in Softwaremodellen zu berücksichtigen. Die Autoren erkennen, dass Mobilität von Anwender und Gerät eine Umgebungsbedingung ist, auf die ein Softwaresystem keinen Einfluss nehmen kann. Die Erwartung eines Anwenders ist hingegen, dass Software die Mobilität von Anwender und Gerät mindestens ohne ungewollte Seiteneffekte toleriert oder sogar mehrwertstiftend in den Anwendungsfall einbezieht. Basierend auf einem Mobilitätsmodell, das zwischen logischer und physikalischer Mobiltität unterscheidet, entwerfen die Autoren ein UML-Profil, durch welches Mobilitätsaspekte einem Softwaresystem hinzumodelliert werden können. Grassi et al. versuchen durch die Integration von Mobilitätsaspekten in Softwaremodelle das Verständnis für die Auswirkungen von Mobilität auf ein Softwaresystem oder seine Komponenten beim Entwickler zu unterstützen. Zwar erweitern die Autoren durch ein UML-Profil das Metamodell der UML. Doch diese Erweiterung zielt lediglich auf informative Zwecke ab, so dass über eine Identifikation von Mobilitätspunkten im Modell keine weiteren Informationen aus dem Modell extrahiert werden können. Die Relevanz der Untersuchungen von Grassi et al. zur Kontextmodellierung für diese Dissertation erschöpft sich also in der Verwendung von UML-Profilen zur Erweiterung des UML-Metamodells zur Darstellung von Mobilität.

Becker und Dürr [35] diskutieren Ansätze zur Modellierung von Standortinformationen für Anwendungen des *Ubiquitous Computing* (UC). Mit speziellem Fokus auf ortsbasierte Anwendungen als Spezialfall des *Ubiquitous Computing* (UC) identifizieren die Autoren Anwendungsfälle und diskutieren daraus resultierende Anforderungen an Metamodelle zur Standortmodellierung. Die Autoren identifizieren die Abbildung sowohl geodätischer als auch symbolischer Standortinformationen als Kernanforderung an Standortmodelle. Diese Abbildung sollen unterschiedliche Referenzsysteme unterstützen, die sowohl eine absolute Positionsbestimmung einzelner Entitäten als auch eine Positionsbestimmung mehrerer Entitäten relativ zueinander ermöglicht. Hierzu muss es möglich sein, die Position einer Entität eindeutig zu bestimmen, ein Maß für den Abstand zweier Entitäten zu definieren, topologische Relationen zwischen Entitäten zu modellieren und zusätzlich zur Position einer Entität auch ihre Orientierung im Raum abzubilden. Nach der Identifikation von Anforderungen an Standortmodelle diskutieren Becker und Dürr unterschiedliche Varianten von Metamodellen mit denen geometrische, geodätische und symbolische Standortinformationen modelliert werden können.

Die von Becker und Dürr diskutierten Ansätze haben das in dieser Dissertation entworfene Metamodell zur Modellierung von Standortinformationen inspiriert. In dieser Dissertation werden die von Becker und Dürr diskutierten Anforderungen zur Modellierung relativer Positionen mehrerer Entitäten zueinander jedoch nur sekundär betrachtet. Weiterhin formulieren die Autoren lediglich generelle Anforderungen an ein Metamodell zur Standortmodellierung, eine technische Lösung wird nicht thematisiert. Im Rahmen dieser Dissertation ist eine solche technische Realisierung allerdings Voraussetzung zur Implementierung eines Werkzeuges zur Generierung von Testfällen aus mit Kontextinformationen angereicherten Systemmodellen.

Mit der Modellierung von Kontext in Softwaresystemen befassen sich auch Henricksen et al. [178, 176, 177] in mehreren Arbeiten zur Entwicklung kontextsensitiver Anwendungen. Kontext hat nach Henricksen et al. sowohl statische als auch dynamische Attribute. Die Unterscheidung erfolgt hierbei anhand der zu erwartenden Änderungshäufigkeit individueller Attribute. Attribute, deren Werte sich nie oder nur in sehr großen Zeitintervallen verändern, werden von den Autoren als statisch bezeichnet. Hierzu gehören beispielsweise nichtdynamische Informationen über Personen, wie etwa das Geburtsdatum. Zu den dynamischen Kontextattributen gehören umgekehrt solche, von denen erwartet wird, dass ihr Wert sich häufig verändert, beispielsweise der Aufenthaltsort einer Person. Statische Kontextattribute können etwa in einem Speicher persistiert werden, wohingegen dynamische Kontextattribute direkt zur Laufzeit durch Sensoren erfasst werden müssen. Henricksen et al. unterscheiden zudem drei Klassen von Quellen von Kontextinformationen: Sensoren, Personen und aus anderen Quellen inferierte Informationen.

Sensordaten fallen in die Kategorie der dynamischen Kontextattribute. Je nach Anwendungsfall sind wenige Sekunden ausreichend, bis ein Attribut dieser Klasse so stark gealtert ist, dass es nicht mehr sinnvoll verwendet werden kann. Direkt vom Anwender (d. h. Personen) gewonnene Kontextattribute sind der Kategorie statische Kontextattribute zuzuordnen. Einerseits kann vom Anwender nicht gefordert werden Informationen manuell hochfrequent zu aktualisieren, andererseits handelt es sich i. d. R. um Informationen, die keiner hochfrequenten Aktualisierung bedürfen. Weiterhin argumentieren die Autoren, dass Kontextmodelle in der Lage sein müssen Kontext unvollständig bzw. probabilistisch zu modellieren.

Kontextsensitive Anwendungen sind häufig mobile Anwendungen. Es gelten deshalb alle für mobile Anwendungen typischen Einschränkungen im Bezug auf Netzwerkverbindung oder Qualität der Standortermittlung. Hieraus ergibt sich auch unmittelbar ein Problem mit zentralen Kontextaggregatoren, die ihre Dienste über Netzwerkschnittstellen anbieten. Das von Becker und Dürr [35], Dürr und Rothermel [102] und ebenfalls in dieser Dissertation in Abschnitt 3.1.2.1.3 thematisierte Konzept der geodätischen und symbolischen Standorte wird in abgeschwächter Form auch von Henricksen et al. diskutiert. Die Autoren fordern von Kontextmodellen, alternative Beschreibungen desselben Kontextobjekts zuzulassen. Hierbei sollten einerseits unterschiedliche Abstraktionsebenen unterstützt werden, andererseits müssen Relationen zwischen Kontextobjekten trotz dieser unterschiedlichen Abstraktionsebenen erhalten bleiben. Dies gilt beispielsweise für eine Wohnanschrift, die als geodätisches Datum in Form von WGS84-Koordinaten beschrieben werden kann oder auch als Kombination von Straßennamen, Hausnummer, Postleitzahl. Die Autoren entwerfen zur Modellierung von Kontext ein Metamodell, welches unterschiedliche Kontextattribute miteinander in Beziehung setzt. Es umfasst Personen, Geräte und Standortinformationen, nicht jedoch Sensordaten.

Henricksen et al. untersuchen ebenfalls die Option, Kontextmodelle durch Sprachmittel der UML abzubilden. Die Autoren kommen jedoch zu dem Schluss, dass die UML für die konkrete Anwendung nicht das geeignete Werkzeug ist, weil durch die Verwendung von UML-Klassendiagrammen eine Unterscheidung von Kontextobjekten unterschiedlicher Art und eine Unterscheidung von sonstigen Datenobjekten des modellierten Geschäftsprozesses nicht gewährleistet scheint. Diese Einschränkung ist für das in dieser Dissertation verfolgte Ziel nicht zutreffend, da Kontextobjekte lediglich als Metainformationen für Tests verwendet werden, nicht jedoch für die Implementierung einer Anwendung. Zum Zweck der Testautomatisierung unter Berücksichtigung von Kontextparametern ist es erforderlich, auf der Ebene der Kontextsimulation alle Parameter auf die detaillierteste Abstraktionsebene konkreter Sensor oder GPS Werte zu überführen. Henricksen et al. führen darüber hinaus jedoch an, dass die UML aufgrund ihrer höheren Expressivität eine bessere Alternative zu dem tatsächlich von Henricksen et al. verwendeten *Entity-Relationship-Model* (ER-Modell) ist. Die Autoren greifen in einer Arbeit aus dem Jahr 2006 diese Fragestellung erneut auf und entwerfen ein technologienäheres Modell. Die Autoren diskutieren in ihrer Arbeit zum Entwurf und zur Entwicklung von kontextsensitiven Anwendungen aus dem Jahr 2006 ebenfalls die Problematik des Testens. Bereits hier wurden die in Abschnitt 4.2 dieser Dissertation adressierten Probleme des Testens kontextsensitiver Anwendungen oberflächlich beleuchtet, ohne jedoch Automatisierungstechnologien zu thematisieren.

Castelli et al. [73] untersuchen im Jahr 2007 Möglichkeiten Kontext in der Anwendungsentwicklung zu modellieren. Zwar fokussieren die Autoren noch nicht auf die Art kontextsensitiver Anwendungen wie sie diese Dissertation zum Gegenstand hat. Dennoch sind die untersuchten Anforderungen und Konzepte ähnlich. Der von Castelli et al. untersuchte Ansatz ist auf die Verwendung von Kontextinformationen in Web-Anwendungen orientiert. Zugrunde liegt die Idee, dass der Fortschritt digitaler Technologie mit geringem Aufwand eine Vielzahl an Metainformationen über digitale und nichtdigitale Objekte verfügbar machen wird. Hierin erkennen Castelli et al. insbesondere die Aufgabe, entsprechende Metamodelle zur Integration von Kontextinformationen zu entwerfen. Diese sollen Kontext generisch abbilden, also

zunächst losgelöst vom spezifischen Verwendungszweck, aber dennoch einfach zu erstellen und zu verändern sein. Hierzu entwerfen die Autoren das W4-Metamodell, welches Modellierungselemente bereitstellt, um Fakten über Objekte zu modellieren. Die Namensherkunft der Bezeichnung W4 beruht auf den Kontextparametern Person, Ort, Aktivität und Zeit (*W4 – Who, What, Where, When*) und ist damit im Einklang mit den früher von Abowd et al. [2, 3] und Dey et al. [92, 93, 94, 95] identifizieren Kernkontextparametern.

Weiterhin entwerfen Castelli et al. eine Architektur einer Middleware, die Kontextinformationen für Anwendungen auf mobilen Geräten nutzbar macht. Hauptaufgabe des von den Autoren entworfenen W4-Metamodells ist es, heterogene Daten aus heterogenen Quellen homogen adressierbar zu machen und komplexe semantische Abfragen über große Datenmengen zu ermöglichen. In der Terminologie der Autoren wird ein W4-Tupel als *Knowledge Atom* (engl. Wissensatom, ≙ Kontextatom nach Korpipää und Mäntyjärvi [229], Korpipää et al. [228], Baldauf et al. [25]) als Wissenseinheit bezeichnet, die den Kontext eines individuellen Objekts hinreichend beschreibt, um in kontextsensitiven Anwendungen verwendet zu werden. Dieser Ansatz ist für das in dieser Dissertation untersuchte Konzept der automatischen Testfallgenerierung und Testausführung interessant, weil ein ähnliches Format der Kontextprovisionierung zum Einsatz kommt. Für Testfälle gemäß der in dieser Dissertation verwendeten Definition gilt, dass diese ebenfalls unter definierten Kontextbedingungen ausgeführt werden. Wer, was, wo und wann sind Kriterien, die grundsätzlich auch auf Vorbedingungen und Nachbedingungen von Testfälle angewendet werden können. In dieser Hinsicht ist die Arbeit von Castelli et al. auf das in dieser Dissertation untersuchte Konzept der Testautomatisierung übertragbar. Allerdings ist das W4-Metamodell auf einer Abstraktionsebene angesiedelt, die eine direkte Verwendung in der hier untersuchten Testautomatisierungstechnologie unmöglich macht. Beispielweise abstrahiert das W4-Metamodell vollständig von Signalquellen. Der Ort wird z. B. spezifiziert, ohne die Herkunft der Information ebenfalls zu spezifizieren. Beim Testen einer kontextsensitiven Anwendung gilt es jedoch, durch gezielte Manipulation der Gerätesensoren einen durch den Testfall spezifizierten Kontext zu simulieren. Das W4-Metamodell ermöglicht jedoch keine ausreichende Detailtiefe im Kontextmodell. Darüber hinaus bestünde allerdings die Möglichkeit, das W4-Metamodell in existierende Modellierungssprachen des Software Engineering, beispielsweise als Profil in die UML, zu integrieren.

Ayed et al. [21] befassen sich im Jahr 2007 mit dem Entwurf eines Metamodells zum Entwurf kontextsensitiver Anwendungen für die Verarbeitung durch MDSD-Technologien. Ziel ist es, Softwareartefakte (z. B. Modelle unterschiedlicher Abstraktionsebenen, Quellcode) durch sukzessive Verfeinerung abstrakter plattformunabhägiger Modelle zu plattformabhägigen Modellen und schließlich Quellcode zu generieren. Hierzu identifizieren die Autoren zunächst Phasen der Entwicklung kontextsensitiver Anwendungen. In einer ersten Phase gilt es zunächst solche Kontextinformationen zu identifizieren, die potenziell Einfluss auf die zu entwickelnde Anwendung haben. Ziel der ersten Phase ist eine Spezifikation relevanter Kontextparameter, an diese gestellte Qualitätsattribute und eine Spezifikation des Prozesses zur Erfassung dieser Kontextparameter. Eine zweite Phase identifiziert kontextabhängige Variabilitätspunkte in der Software. Sie dient der Spezifikation der Struktur und des Verhaltens von Kontextparametern, deren Veränderung zu einem angepassten Laufzeitverhalten der Anwendung führt. Die dritte Phase zielt darauf ab zu spezifizieren, wie Kontextparameter von

der Anwendung erfasst werden, beispielsweise durch die Verwendung spezifischer Sensoren. Die vierte Phase definiert Adaptionsmechanismen zur Anpassung des Laufzeitverhaltens der Anwendung auf sich verändernde Kontextparameter. Die Phasen fünf und sechs sind originäre Phasen des MDSD, in welchen das plattformunabhängige Modell unter Hinzuziehung eines Plattformmodells zu einem plattformspezifischen Modell verfeinert wird (Phase 5), aus welchem im Anschluss Code generiert wird (Phase 6). Diese Modellierungsaktivitäten erfordern die Verwendung geeigneter Werkzeuge und Modellierungssprachen. Kernbeitrag der Autoren ist die Bereitstellung einer solchen Modellierungssprache in Form eines UML-Profils.

Die Arbeit von Ayed et al. verfolgt ähnlich dem in dieser Arbeit untersuchten Konzept einen MDSD-Ansatz zur Integration von Kontext in die Systemmodellierung. Der Fokus liegt allerdings auf der Entwicklung, nicht auf dem Testen. Dennoch können zwischen dem von Ayed et al. betrachteten Konzept und dieser Dissertation Parallelen gezogen werden. Die Verwendung der Modellierungssprache UML und deren Erweiterung durch ein Profil wird auch im hier untersuchten Konzept zur Testfallgenerierung verwendet, um die Möglichkeit der Verwendung einer konsistenten Werkzeugkette innerhalb des Softwareprozesses zu gewährleisten.

Mit dem Fokus auf kontextsensitive Zugriffskontrolle auf Funktionen eines Softwaresystems entwirft Decker [89] im Jahr 2009 ein rudimentäres Metamodell zur Standortmodellierung. Es werden Modellelemente zur Modellierung einzelner Standortentitäten oder Regionen bereitgestellt. Diese Informationen werden in einm UML-Profil hinterlegt. Hintergrund des von Decker vorgestellten Konzepts ist die Unterstützung eines *Workflow-Management*-Systems durch Anreicherung von Workflow-Komponenten um Mobilität. Diese schlägt sich in der Verwendung mobiler Geräte zur Durchführung spezifischer Arbeitsschritte nieder, bei denen der aktuelle Aufenthaltsort des Anwenders das Systemverhalten beeinflusst. Insbesondere erlaubt das von Decker vorgestellte UML-Profil die Spezifikation der Zulässigkeit individueller Arbeitsschritte in Abhängigkeit vom Standort. Zusätzlich zu Standortinformationen wird eine Menge von *Location Constraints* (engl. ortsabhängige Rahmenbedingungen) spezifiziert, die in der Modellierung die Zulässigkeit einer Aktion unter gegebene Standortbedingungen beschreiben. In Abgrenzung zu dem in dieser Arbeit vorgestellten Konzept, angereicherte UML-Aktivitätsdiagramme zur Testfallgenerierung zu verwenden, dient das in Decker [89] vorgestellte UML-Profil jedoch lediglich zur Annotation von Systemmodellen. Eine tatsächliche Wirkung entfalten die modellierten *Location Constraints* jedoch nur dann, wenn sie durch den Programmierer bei der Implementierung berücksichtigt werden. Die Relevanz der Arbeit von Decker für diese Dissertation ergibt sich aus dem verwandten Konzept angereicherte UML-Aktivitätsdiagramme zur Modellierung von Standortinformationen zu verwenden.

Die Modellierung von Kontextinformationen ist integraler Bestandteil des in dieser Dissertation untersuchten Konzepts zur automatischen Generierung von Testfällen aus Systemmodellen. Voraussetzung hierfür ist eine Modellierungstechnologie, die es einerseits zulässt, anwendungsfallspezifisch relevante Kontextparameter zu modellieren, andererseits jedoch von irrelevanten zu abstrahieren. Weiterhin muss der Modellierungsaufwand dem Verwendungszweck angemessen sein. Diese Kriterien stellen Anforderungen an Kontextmodelle, die in Abschnitt 5.2.2.1 diskutiert werden. Technologien Kontext zu modellieren wurden bereits in früheren Forschungsarbeiten thematisiert. Bettini et al. [42] analysierten im Jahr 2010 die zu diesem Zeitpunkt verfügbaren Modellierungssprachen und -werkzeuge. Die Autoren definieren

hierzu zunächst eine Reihe von Anforderungen an Kontextmodelle. Unter der Voraussetzung, dass mobile Softwaresysteme häufig auch kontextsensitiv sind und insbesondere viele kontextsensitive Softwaresysteme auch mobil sind, identifizieren Bettini et al. die Unterstützung von Heterogenität und Mobilität als Kernanforderungen an Kontextmodelle. Heterogenität ist gefordert, damit ein Kontextmodell in der Lage ist auf Softwaresysteme angewendet zu werden, deren zukünftige Betriebsumgebung nicht vollständig bekannt ist. Für die mobilen Geräte der Smartphone-Generation kommt dieser Anforderung ein besonderer Stellenwert zu, da Entwickler von Anwendungen für die mobilen Plattformen der Smartphone-Generation kaum valide Annahmen für das tatsächlich von Anwender verwendete Gerät treffen können, die über die konkrete Plattform hinausgehen.

Die zweite Kernanforderung die Bettini et al. an Kontextmodelle stellen, ist die Fähigkeit, Abhängigkeiten von Kontextinformationen (vgl. Kontextatome, Korpipää und Mäntyjärvi [229], Korpipää et al. [228], Baldauf et al. [25]) untereinander abzubilden. Zugrunde liegt die Annahme, dass eine Veränderung eines Kontextatoms von einer Ursache ausgelöst wird, die auch Auswirkungen auf andere Kontextatome hat. Beispielsweise beeinflusst die Art der verfügbaren Netzwerkverbindung den Energieverbrauch des Geräts, der Standort des Anwenders beeinflusst die Art des verfügbaren Netzwerks, die Orientierung des Geräts schlägt sich in den Werten unterschiedlicher Sensoren nieder.

Als dritte Anforderung formulieren Bettini et al. die Eigenschaft eines Kontextmodells, den Kontext auch unvollständig modellieren zu können. Motiviert wird diese Anforderung durch die Tatsache, dass für konkrete Anwendungsfälle häufig nur Ausschnitte des Gesamtkontext relevant sind (z. B. werden zwar Orientierungssensoren verwendet, der Standort ist aber irrelevant) oder bestimmte Kontextinformationen sind nicht verfügbar.

Als vierte Anforderung nennen Bettini et al. Schlussfolgerbarkeit, d. h. es soll möglich sein, aus einem Kontextmodell Informationen abzuleiten, die zur Steuerung des Verhaltens einer Softwareanwendung verwendet werden. Als fünfte Anforderung führen Bettini et al. ein, dass Kontextmodelle in übliche Modellierungswerkzeuge des Software Engineering integrierbar sein sollten. Hierdurch soll gewährleistet werden, dass der ohnehin komplexe Softwareprozess nicht durch weitere Technologien zusätzlich verkompliziert wird. Die Integration in existierende Modellierungstechnologien stellt sicher, dass Kontextmodell und Systemmodell kohärent gepflegt werden, ohne zusätzliche Anforderungen an den Softwareingenieur zu stellen.

Schließlich wird noch effiziente Provisionierung gefordert, die einen performanten Zugriff auf Kontextinformationen in Anwendungen sicherstellen soll. Weiterhin untersuchen die Autoren die im Jahr 2010 verfügbaren Technologien zur Kontextmodellierung. Die Erkenntnisse werden in Abschnitt 5.2.2 im Detail diskutiert und in Bezug zu dem in dieser Dissertation untersuchten Konzept zur Testautomatisierung für kontextsensitive Anwendungen gesetzt.

Vieira et al. befassen sich im Jahr 2011 mit der Modellierung von Kontext aus unterschiedlichen Perspektiven. Die Autoren untersuchen einerseits generelle Eigenschaften des abstrakten Konzepts Kontext und andererseits die Integration von Kontext in die Systemmodellierung. Vieira et al. entwerfen ein Kontextmetamodell als UML-Profil mit dem Ziel, Kontextparameter bei der Modellierung von Verhaltensaspekten eines Softwaresystems zu berücksichtigen. Das von Vieira et al. erarbeitete UML-Profil erweitert hierzu das UML-Metamodell für Aktivitätsdiagramme. Dieses Konzept wird ebenfalls in dieser Dissertation untersucht. Allerdings steht

in dieser Dissertation das Ziel im Fokus, Aktivitäten des Testens durch eine modellbasierte Automatisierungstechnologie zu unterstützen. Die Komplexität individueller Kontextparameter wird vom UML-Profil nach Vieira et al. nicht ausreichend unterstützt. Die Autoren untersuchen ebenfalls den Einfluss von Kontextsensitivität auf das Testen in ihrer späteren Arbeit [352], ohne jedoch das UML-Profil auf die Verwendung beim Testen anzupassen, da eine wie in dieser Dissertation untersuchte modellbasierte Technologie zur Generierung von Testfällen nicht Gegenstand der Arbeit der Autoren ist.

2.3 Entwicklung und Testen mobiler Anwendungen

Die Herausforderungen der Entwicklung mobiler Software wurde in der jüngeren wissenschaftlichen Literatur vielfältig untersucht. Insbesondere die erste Generation mobiler Geräte konfrontierte Entwickler und Anwender gleichermaßen mit einer neuen Situation des Umgangs mit digitalen Dienstleistungen. Zunächst überwog noch die Anpassung von Standardanwendungen wie z. B. E-Mail und digitaler Kalender. Herausforderungen lagen hier in erster Linie in der Sicherstellung von Datenintegrität und im Umgang mit kleinen Display-Diagonalen, die nur wenig Raum zur Gestaltung einer adäquaten UX ließ.

Diese Herausforderungen haben sich gewandelt. In den folgenden Abschnitten werden unterschiedliche Forschungsansätze aus den Themenumfeldern Entwicklung, Testen und Testautomatisierung von Anwendungen für mobile Geräte untersucht und die Entwicklung des Forschungsthemas bis hin zur in dieser Dissertation diskutierten Problematik des Testens mobiler, kontextsensitiver dargelegt.

2.3.1 Entwicklung mobiler Anwendungen

Holzer und Ondrus [186] befassen sich im Jahr 2009 in ihrer Arbeit mit dem sich in der Entstehung befindlichen Ökosystem mobiler Plattformen. Zu diesem Zeitpunkt hatten sich die Unternehmen, die im Jahr 2016 den Markt dominieren, noch nicht herauskristallisiert. Die wirtschaftliche Tragfähigkeit dieser neuen Märkte stand ebenso in Frage wie die Zukunftsfähigkeit der im Jahr 2016 etablierten Infrastrukturen mobiler Plattform. Es zeichnete sich bereits ab, dass einige Unternehmen streng an ihrer Closed-Source-Software-Philosophie – der *Cathedral*-Strategie – festhalten würden (Apple), während andere sich auf OSS-Modelle – der *Bazaar*-Strategie – festgelegt hatten [186] (Terminologie in Anlehnung an Raymond [293]). Die Entscheidung für eines dieser Modelle wurde als Einflussfaktor in die Entscheidung von Entwicklern und Softwareanbietern bewertet, in die jeweilige mobile Plattform zu investieren. Typische Fragestellungen der Zeit berührten primär die Einbettung von Softwareprodukten in die entstehenden Ökosysteme mobiler Plattformen, weniger Fragestellungen der Softwaretechnik. Diese Kategorisierung nach *Bazaar* oder *Cathedral* ist im Jahr 2016 obsolet. Um die Marktfähigkeit mobiler Apps zu gewährleisten, sind Entwickler nun gezwungen alle relevanten mobilen Plattformen ungeachtet ihrer Philosophie zu bedienen.

Weitere Fragestellungen, etwa ob sich OSS Plattformen gegen Closed-Source-Software Plattformen durchsetzen werden oder ob der Trend zu dezentralen oder zu zentralen Marktplätzen strebt, sind mittlerweile irrelevant geworden. Vorherrschend geworden sind hingegen wirtschaftliche Aspekte der App-Entwicklung, die schnelles Reagieren auf die Anforderungen

des Marktes unter Ausnutzung aller technologischen Möglichkeiten erforderlich machen. Hierzu gehören vor allem effiziente Methoden der Softwaretechnik, die Automatisierungslösungen einschließen wo immer es sinnvoll ist kostenintensive menschliche Arbeitskraft durch günstige und zuverlässige Maschinen abzulösen, wie beispielsweise der in dieser Dissertation behandelte Ansatz zur Testautomatisierung.

Dehlinger und Dixon [90] untersuchen in ihrer Arbeit aus dem Jahr 2011 die zukünftigen Herausforderungen und Forschungsfragen im Umfeld mobiler Anwendungen. Die Autoren identifizieren die unterschiedlichen Entwicklungsgeschwindigkeiten mobiler Gerätetechnologie und der begleitenden wissenschaftlichen Forschung zum Software Engineering als problematisch. Mobile Geräte, so die Autoren, etablieren neue Paradigmen der Mensch-Maschine Interaktion (z. B. Multi-Touch-Displays, *Radio-Frequency-Identification* (RFID), *Augmented Reality* (AR)), für die ebenfalls neue Methoden und Werkzeuge des Software Engineering benötigt werden. Weiterhin stellt die zunehmende Fragmentierung des mobilen Gerätemarktes Entwickler vor die Aufgabe, Software auf die unterschiedlichen Eigenschaften verschiedener Geräte und Geräteklassen zuzuschneiden, wodurch Softwareprojekte für mobile Anwendungen implizit zu Produktlinien geraten. Die Autoren identifizieren die schnellen Evolutionszyklen mobiler Plattformen als Treiber für agile Vorgehensmodelle, da Software schnell an neue technische Entwicklungen und Anforderungen angepasst werden müssen, um die Akzeptanz des Anwenders auch längerfristig zu sichern. Weiterhin erkennen auch Dehlinger und Dixon [90] das Umfeld der kontextsensitiven Anwendungen als Forschungsbereich mit herausragender Bedeutung. Neuartige Technologien ermöglichen die Entwicklung von Anwendungen, die neben dem Standort des Anwenders und Sensorinformationen zahlreiche weitere Kontextparameter als Eingabevektoren nutzbar machen. Verfügbare Methoden und Werkzeuge des Software Engineering, beispielsweise Modellierungssprachen, sind diesen neuen Anforderungen nicht gewachsen. Insbesondere die in dieser Dissertation untersuchte Methode zur Testautomatisierung mobiler, kontextsensitiver Anwendungen entfaltet sich in diesem Spannungsfeld.

Joorabchi et al. [204] untersuchten in einer Studie aus dem Jahr 2013 durch Befragung von 188 Teilnehmern aus dem Umfeld der App-Entwicklung Aussagen zu den gegenwärtig signifikanten Herausforderungen bei der Entwicklung mobiler Systeme. Die Autoren identifizieren in ihrer Studie eine Reihe allgemeiner und spezifischer Herausforderungen bei der Entwicklung mobiler Anwendungen. Zu den allgemeinen Herausforderungen, die auch im Rahmen dieser Dissertation von besonderer Bedeutung sind, gehört die Fragmentierung der mobilen Plattform. Nach der von Joorabchi et al. durchgeführten Studie glauben 76% der Befragten im Umfeld mobiler Geräte einen Trend zur weiteren Fragmentierung der Gerätelandschaft zu erkennen. Die Autoren differenzieren hier unterschiedliche Arten der Fragmentierung. Zum einen tritt Fragmentierung über Plattformgrenzen hinweg auf. Das heißt, trotz einer prinzipiellen Angleichung der Hardware-Komponenten aller Geräte über die Plattformen hinweg kommt es auf der Ebene der SDKs und APIs zu einer zunehmenden Inkompatibilität. Mehrere Plattformen können deshalb nicht mit einer einzelnen App bedient werden, sondern es müssen inhaltsgleiche Apps individuell für jede zu unterstützende Plattform entwickelt werden. Innerhalb einiger Plattformen tritt ein ähnliches Phänomen auf. Insbesondere die Plattform Android ist hier betroffen. Da es sich bei Android um eine OSS handelt, können Hersteller von mobilen Geräten Modifikationen am Quellcode des Betriebssystems vornehmen. Neben

einem erhöhten Entwicklungsaufwand führt dies ebenfalls zu erhöhtem Aufwand im Bereich des Testens. Das Testen wurde von Joorabchi et al. [204] als eine besondere Herausforderung identifiziert. Nach den Resultaten der Studie liegen die wesentlichen Herausforderungen des Testens hierbei im Management von Testaktivitäten und im Fehlen geeigneter Automatisierungstechnologien. Demnach werden aktuell mehr als 60% aller Testaktivitäten manuell durchgeführt und nur bei etwa 31% kommen Automatisierungswerkzeuge in einem hybriden Ansatz zwischen manuellem und automatisiertem Testen zum Einsatz. In die Problematik des Testens mobiler Anwendungen fällt nach den Erkenntnissen der Studie ebenfalls, dass Testen eine Tätigkeit ist, die vorrangig von Entwicklern vorgenommen wird, ohne dass ein dediziertes Testmanagement existiert. Hierdurch werden in Testaktivitäten in vergleichsweise hohem Umfang Ressourcen gebunden, die dann anderen Aktivitäten des Softwareprozesses nicht mehr zur Verfügung stehen.

Weiterhin werden durch die Teilnehmer der Studie solche Probleme als herausragend identifiziert, die im Rahmen dieser Dissertation mit besonderem Fokus betrachtet werden. Insbesondere wird hier vor allem die unzureichende Werkzeugunterstützung für Tests solcher Features hervorgehoben, die eine Mobilisierung digitaler Geschäftsprozesse von nicht-mobiler Software abgrenzen. Hierbei geht es vorrangig um technische Features mobiler Geräte, wie etwa drahtlose Internetverbindung, GPS-Funtionalität und die Sensorik mobiler Geräte. Insofern besteht in der Praxis eine hohe Nachfrage für Automatisierungstechnolgien, die dabei helfen, das Testen von Apps einschließlich besonderer Geräte-Features zu professionalisieren. Durch die von Joorabchi et al. [204] durchgeführte Studie wird die Relevanz der in dieser Dissertation bearbeiteten Problemstellungen durch eine empirische Studie bestätigt.

Im Jahr 2013 waren die App Stores der dominanten Plattformbetreiber bereits seit einigen Jahren verfügbar. Die Bewertungssysteme der App Stores in Form einer einfachen summativen Bewertung von 1-5 Sternen hatten sich bereits als Standard etabliert und es war Konsens, dass der Erfolg einer App direkt an Anwenderbewertung in den App Stores gemessen werden kann. Der Zusammenhang zwischen der Anwenderbewertung und der Anzahl App-Downloads wurde bereits von Kim et al. [223], Harman et al. [167] und Zhong und Michahelles [379] nachgewiesen. Khalid [220, 221] untersuchen in ihrer Arbeit aus dem Jahr 2013 Qualitätskriterien für Apps aus der Anwenderperspektive mit dem Ziel, hieraus Strategien für eine optimierte Qualitätssicherung abzuleiten in einem ähnlichen Ansatz wie zuvor Haller [164]. Eine Kernerkenntnis der Autoren ist, dass der Einfluss von 1-2-Sterne Bewertungen (also am negativen Ende der Skala) einen höheren Einfluss auf den kommerziellen Erfolg einer App haben als positive Bewertungen. Die Autoren untersuchten eine Auswahl von 20 Apps mit insgesamt mehr als 250.000 1-2-Sterne Bewertungen. Ihre Analyse ergab, dass mehr als 50% der negativen Bewertungen auf funktionale Defekte, App-*Crashes* (engl. Absturz, unerwartete Termierung einer Software) und unvollständige Features zurückzuführen sind. Hierdurch wird deutlich, dass sorgfältiges Testen eines App-Produktes zwingende Voraussetzung für deren wirtschaftlichen Erfolg ist.

Die Qualität einer App kann durch konstruktive und destruktive Aktivitäten der Qualitätssicherung innerhalb eines Spielraums geplant werden. Testen im mobilen Umfeld verursacht jedoch aufgrund der hohen Heterogenität der Gerätelandschaft einen besonders hohen Aufwand, da Apps auf einer großen Anzahl unterschiedlicher Geräte getestet werden müssen

(vgl. Abschnitt 4.2.1, Joorabchi et al. [204]). Im Jahr 2014 untersuchen Khalid et al. [222] die Fragestellung, wie eine zum Testen zweckmäßige Auswahl repräsentativer Geräte für ein App-Projekt ermittelt werden kann. Insbesondere adressieren die Autoren diese Fragestellungen: Welcher Anteil spezifischer Geräte entfällt auf die Mehrheit der Anwenderbewertung? Wie können Entwickler Geräte identifizieren, die beim Testen mit hoher Priorität berücksichtigt werden sollten? Existieren signifikante Abweichungen in der Bewertung einer App für unterschiedliche Geräte?

Die erste Frage konnte durch eine quantitative Analyse des Google Play Store beantwortet werden. 33% der beobachteten Geräte zeichnen 80% aller Bewertungen. Diese Erkenntnis konnte direkt auf die zweite Frage übertragen werden. Es sind genau diese 33% der Geräte, die beim Testen mit Priorität berücksichtig werden sollten, da erwartet werden kann, dass diese Geräte auch den größten Anteil der Bewertung zukünftiger Anwendungen stellen werden. Die Untersuchung der dritten Frage ergab, dass spezifische Geräte existieren, mit denen eine App signifikant schlechter bewertet wurde als mit anderen Geräten. Hieraus ergeben sich zwei Handlungsoptionen: Entweder muss mit diesen Geräten zukünftig besonders sorgfältig getestet werden, auch wenn diese Geräte nicht die Mehrheit der Anwender repräsentieren oder die Unterstützung für diese Geräte wird insgesamt eingestellt. Diese Erkenntnisse sind für diese Dissertation insofern relevant, als dass sie die Signifikanz der Kontextparameter Plattform und Gerät (vgl. Abschnitt 3.1.2.1.1) in ihren Auswirkungen auf das Testen (vgl. Abschnitt 4.2.1) empirisch bestätigen. Die Auswahl geeigneter Geräte ist eine Aktivität der Qualitätssicherung, die einen maßgeblichen Einfluss auf den Erfolg der Qualitätssicherung hat, aufgrund ihrer Komplexität jedoch nur mit großen Aufwand automatisierbar ist.

2.3.2 Testen

Dantas et al. [87] untersuchten im Jahr 2009 Anforderungen an das Testen mobiler Anwendungen im Vergleich zu traditioneller Software. Die Autoren identifizierten die besonderen Eigenschaften mobiler Anwendungen und mobiler Geräte (u. a. ausschließlich drahtlose Netzwerkkonnektivität, unvorhersehbare Bandbreite, verfügbarer Arbeitsspeicher und Energievorrat) als Komplexitätstreiber des Testens. Zur Bestimmung dieser Anforderungen führten Dantas et al. eine Studie mit professionellen Entwicklern mobiler Anwendungen durch. Im Ergebnis wurde eine Liste charakteristischer Anforderungen erhoben. Diese Anforderungen beziehen sich sowohl auf den Softwareprozess als auch auf technische Aspekte des Testens. Eine der Anforderungen ist beispielsweise, dass für effizientes und effektives Testen mobiler Anwendungen der gesamte Softwareprozess auf diesen Aspekt fokussieren sollte. Der Vorschlag zur Erfüllung dieser Anforderung ist die Verwendung inkrementeller, agiler Vorgehensmodelle, die in kurzen Zyklen Produktinkremente erzeugt. Hierdurch stehen in kurzen Abständen testbare Softwareartefakte zur Verfügung. Entdeckte Defekte können dann bereits in der nächsten Entwicklungsiteration korrigiert werden.

Da manuelles Testen i. d. R. Regel einen großen Aufwand verursacht, bietet sich hier die Verwendung von Automatisierungstechnologie an, um die Kosten des Testens herabzusetzen. An eine solche Automatisierungstechnologie stellen sich allerdings eine Reihe von Anforderungen die technischen Besonderheiten mobiler Anwendungen zu berücksichtigen. Diese Dissertation untersucht eine Testautomatisierungstechnologie, die in der Lage ist, bestimmte Aspekte

der Betriebsumgebung mobiler Anwendungen (also den Kontext) zu simulieren, so dass Kontextparameter als Testdaten verwendet werden können. Als weitere Anforderungen an das Testen mobiler Anwendungen identifizieren Dantas et al. die Notwendigkeit, Anwendungen sowohl auf Geräten als auch in Emulatoren zu testen. Zugrunde liegt dieser Anforderung, dass Emulatoren nicht in der Lage sind, bestimmte Eigenschaften mobiler Geräte realitätsnah abzubilden. Wenngleich sich Emulatortechnologie aus dem Jahr 2009 wesentlich von der Technologie im Jahr 2016 unterscheidet, gilt diese Einschränkungen weiterhin unverändert für Emulatoren. Deshalb stellt sich an die in dieser Dissertation untersuchte Testautomatisierungstechnologie die Anforderung, sowohl auf mobilen Geräten als auch auf Emulatoren funktionsfähig zu sein.

Eine Besonderheit mobiler Plattformen ist, dass der Lebenszyklus von Anwendungen hoheitlich vom Betriebssystem gesteuert wird, welches einer Anwendung jederzeit und ohne Berücksichtigung ihres Zustands Ressourcen entziehen kann. Als Anforderung an das Testen mobiler Anwendungen formulieren Dantas et al. deshalb, dass Tests für mobile Anwendungen durchgeführt werden müssen, in denen das Betriebssystem der jeweiligen Plattform die getestete Anwendung unterbricht (z. B. wegen eingehenden Telefonanrufs) und das spezifikationskonforme Verhalten der Anwendung während und nach der Unterbrechung überprüft wird. Hieraus ergibt sich die Anforderung an eine Automatisierungstechnologie, dass sie nicht nur auf Funktionen der zu testenden Anwendung beschränkt sein darf, sondern in der Lage ist, Systemereignisse zu simulieren, die eine Programmunterbrechung verursachen.

Der Kontext der Verwendung ist bei mobilen Anwendungen ein besonders hervorgehobenes Kriterium. Deshalb identifizieren Dantas et al. den mobilen Kontext einer Anwendung als weitere Anforderung an das Testen. Diese Anforderung zielt auf alle Kontextparameter (vgl. Abschnitt 3.1.2) ab, die potenziell Einfluss auf eine mobile Anwendung haben können. Im Jahr 2009 hatte die Smartphone-Ära gerade begonnen und die technischen Möglichkeiten mobiler Geräte waren weniger ausgeprägt als im Jahr 2016. Die Kontextfaktoren Plattform- und Geräteheterogenität, Netzwerkverfügbarkeit und Mobilität hatte jedoch bereits Bedeutung. Hieraus ergibt sich die Anforderung an das Testen, Anwendungen sowohl unter Laborbedingungen als auch in ihren realen zukünftigen Einsatzumgebungen zu testen. Unmittelbar hieran schließt sich die Anforderung an, zu erkennen, für welche Features einer Anwendung ein Labortest ausreichend ist und für welche Features Feldtests erforderlich sind. An eine Automatisierungstechnologie stellt sich konsequent die Anforderung, diesen Betriebskontext simulieren zu können. Die Relevanz der Arbeit von Dantas et al. ergibt sich daraus, dass die Anforderungen an das Testen mobiler Anforderungen auch in der Smartphone-Ära noch gültig sind, aber aufgrund des weiteren technischen Fortschritts seit 2009 weitere Anforderungen, insbesondere aufgrund von Kontextsensitivität, hinzugekommen sind.

Mit dem Test mobiler Anwendungen, noch ohne einen besonderen Fokus auf kontextsensitive Anwendungen, befassen sich Agarwal et al. [5] im Jahr 2010. Die Autoren erkennen, dass die Diagnose von Anwendungen, die auf einer großen Anzahl mobiler Geräte installiert und ausgeführt wird, deren konkrete Eigenschaften bei der Entwicklung unbekannt sind, durch vielfältige Faktoren erschwert wird. Einerseits eröffnet bereits die Heterogenität der Zielgeräte den Raum für potenzielle Defekte, die durch Inkompatibilitäten mit einigen Geräten ausgelöst werden. Andererseits ist bei der Spezifikation und Entwicklung nur wenig über die zukünftige

Betriebsumgebung der Anwendung bekannt. Stellvertretend für abgesichertes Wissen müssen hier Annahmen über die Verfügbarkeit von Ressourcen (z. B. Art und Leistungsfähigkeit der Netzwerkverbindung) gemacht werden.

Gleichzeitig identifizieren Agarwal et al. einen Mangel in der Bereitstellung adäquater Diagnosetechnologien bei allen mobilen Plattformbetreibern. Fehlerprotokolle erreichen die Entwickler nur dann, wenn ein Anwendungsdefekt zu einem Crash der App geführt hat und der Anwender der Übermittlung des Crash-Reports zustimmt. Und auch in diesen Fällen sind in den Diagnoseinformationen wichtige Informationen nicht enthalten, die zur Identifikation eines Softwarefehlers und dessen Korrektur beitragen könnten. Zwar enthalten Crash-Reporte i. d. R. Informationen über die technische Ursache eines Defekts (z. B. Art des Fehler und die Stelle im Code der Anwendung), nicht aber über das Betriebsumfeld der App, z. B. Standort des Anwenders, zum Zeitpunkt des Crashs verfügbare Netzwerke und deren Parameter. Hieraus leiten Agarwal et al. ab, dass es kaum möglich ist, mobile Anwendungen adäquat unter Laborbedingungen zu testen, da die durch die Plattform bereitgestellten Entwicklungswerkzeuge nicht in der Lage sind, die Betriebsumgebung einer App zu Testzwecken zu simulieren.

Ohne die Existenz einer solchen Simulationstechnologie sind Feldtest (d. h. Testen einer Software innerhalb ihres zukünftigen Betriebsumfelds), oder *Testing in the Wild* (Terminologie auch bei Haller [164]), die einzige Option, mobile Anwendungen in realitätsnahen Umgebungen zu testen. Hier stellen sich allerdings Fragen der Reproduzierbarkeit von Anwendungsdefekten, da selbst die Durchführung von Feldtests mit Aufsuchen unterschiedlicher in der Testfallspezifikation fixierten Orte keine zuverlässige Reproduzierbarkeit der Betriebsumgebung garantiert. Hier kann eine Simulationstechnologie dabei helfen, die Betriebsumgebung einer mobilen Anwendung unter Laborbedingungen zu Reproduzieren. Im Rahmen dieser Dissertation wird eine solche Simulationstechnologie als Bestandteil einer Automatisierungslösung konzipiert. Das Problem der Analyse von Fehlern, die im realen Einsatz einer App auftreten, kann hierdurch jedoch nicht gelöst werden.

Im Jahr 2012 hatten sich die marktdominierenden Betreiber mobiler Plattformen bereits etabliert und es hatte sich ein deutlicher Trend zur Mobilisierung digitaler Dienstleistung herausgebildet. Es war klar, dass mobile Anwendungen ein konstituierender Bestandteil digitaler Geschäftsprozesse der Zukunft sein würden. Vor diesem Hintergrund befassen sich Muccini et al. [258] 2012 mit der Fragestellung, ob mobile Software gegenüber traditioneller Software ausreichend signifikante Alleinstellungsmerkmale ausbilden, um einen auf diese besonderen Anforderungen angepassten Testprozesss zu begründen. Konkret untersuchen Muccini et al. drei Forschungsfragen: (1) Unterscheiden sich mobile Anwendungen so stark von traditionellen Anwendungen, dass spezialisierte Testmethoden und -werkzeuge erforderlich sind? (2) Welches sind die neuen Herausforderungen beim Testen mobiler Anwendungen? (3) Welche Rolle spielen Automatisierungstechnologien bei der Bewältigung dieser neuen Herausforderungen? Zur Beantwortung der ersten Fragestellung diskutieren die Autoren zunächst alleinstellende Merkmale mobiler Anwendungen und deren potenzielle Implikationen für das Testen. Zu den Alleinstellungsmerkmalen gehören nach Muccini et al. mobile Netzwerkkonnektivität, begrenzte Hardware-Ressourcen (z. B. Speicher, Prozessorarchitektur), Autonomie der Energieversorgung, spezifische UIs, Kontextsensitivität und Kontextadaption, spezifische Programmiersprachen, kurze Evolutionszyklen mobiler Plattformen, Heterogenität und neu-

artige Eingabemodalitäten (z. B. berührungsempfindliche Displays). Diese Merkmale wirken sich jeweils spezifisch auf Qualitätsaspekte der Anwendung und somit auch auf das Testen aus. Die Implikationen dieser Alleinstellungsmerkmale sind in Tabelle 2.1 abgebildet.

Tabelle 2.1: Implikationen mobiler Anwendungen auf das Testen nach Muccini et al. [258]

Alleinstellungsmerkmal	Implikationen für das Testen
mobile Konnektivität	Zuverlässigkeit, Performanz, Sicherheit, funktionales Testen in verschiedenen Netzen
beschränkte Ressourcen	Monitoring der Performanz
autonome Energieversorgung	Monitoring der Energieverwendung
User-Interface (UI)	Testen des UI
Kontextsensitivität	funktionales Testen unter variierenden Kontextbedingungen
Kontextadaption	Korrektheit der Anpassung
Programmiersprachen	angepasste Byte-Code Analyse, neue *White-Box-Tests*
kurze Evolutionszyklen	Kompatibilitätstests
Heterogenität	Geräteabdeckungstests
Eingabemodalitäten	Usability Tests

Muccini et al. argumentieren, dass die Alleinstellungsmerkmale mobiler Anwendungen und deren Auswirkung auf das Testen dazu beitragen, dass sich das Testen mobiler Anwendungen vom Testen traditioneller Software unterscheidet. Die erste Forschungsfrage konnte eindeutig bejaht werden. Mobile Anwendungen weisen Eigenschaften auf, die sie stark von traditioneller Software abgrenzen. Zur Beantwortung der zweiten Forschungsfrage analysieren die Autoren Prozesse des Testens, Testartefakte, Testebenen und Testtypen und gelangen zu dem Ergebnis, dass das Testen mobiler Anwendungen durch eine Vielzahl neuer Herausforderungen gekennzeichnet ist. Hierzu gehört beispielsweise das Erstellen von Testfällen und die Ausführung von Testfällen unter adäquaten Kontextbedingungen. Insbesondere erkennen die Autoren, dass durch die Vielfalt möglicher Kontextfaktoren und einer hohen Geräteheterogenität mit einem signifikanten Anstieg des auf Testaktivitäten entfallenden Aufwands zu rechnen ist, welcher in erhöhten Kosten für qualitätssichernde Maßnahmen resultiert.

Um diesen Effekt abzumildern, identifizieren Muccini et al. die Verwendung von Automatisierungswerkzeugen als wirkungsvolle Maßnahme und beantworten damit die dritte Forschungsfrage. Das in dieser Dissertation untersuchte Konzept zur Automatisierung der Generierung von Testfällen und deren automatisierte Ausführung adressiert spezifisch die von Muccini et al. identifizierten Herausforderungen durch die Bereitstellung eines Werkzeugs zur Modellierung kontextsensitiver Anwendungen und eines Testautomatisierungswerkzeuges, das in der Lage ist, Kontext während der Testausführung zu simulieren. Beide Komponenten tragen dazu bei, die Komplexität des Testens kontextsensitiver Anwendungen sowohl fachlich-inhaltlich als auch wirtschaftlich beherrschbar zu machen.

Aus einer anderen Perspektive diskutieren Pathak et al. [274] im Jahr 2011 das Testen mobiler Anwendungen. Die Autoren analysierten ca. 40.000 Beiträge in Smartphone-orientierten Online-Foren auf Beschwerden von Anwendern über vorzeitige Erschöpfung des Energievor-

rats ihres mobilen Geräts. Die Autoren konnten basierend auf der Untersuchung der Daten Klassen von Defekten bilden, die zu einer frühzeitigen Unbenutzbarkeit des Geräts führen. In einige Klassen fallen Defekte, die auf fehlerhafte oder veraltete Hardware zurückzuführen sind. Diese liegen nicht in der Verantwortlichkeit von App-Entwicklern und sind deshalb für diese Dissertation nicht weiter relevant. Allerdings identifizierten Pathak et al. ebenfalls einige Klassen von Defekten, bei denen durch Mängel der Software der Energievorrat des Geräts vorzeitig erschöpft wird. Einige dieser Defekte haben ihre Ursache im Betriebssystem, welches seinerseits nicht völlig fehlerfrei ist. Andere der sogenannten *Ebugs* (*Energy Bugs*, [274]) hatten ihre Ursache jedoch in Endnutzeranwendungen, insbesondere wenn diese kontextsensitive Charakteristiken aufweisen. Häufig beobachtet wurden beispielsweise Defekte, die durch einen unsachgemäßen Umgang mit Sensoren oder dem GPS-Modul verursacht wurden. In diesen Fällen wurde die jeweilige Hardware zur Benutzung durch eine App aktiviert, nach Beendigung der Nutzung oder bei Eintritt der App in plattformspezifische Ruhezustände jedoch nicht wieder deaktiviert. Dies ist in zweierlei Hinsicht problematisch, da hierdurch einerseits der jeweilige Hardwarebaustein Energie konsumiert ohne einen objektiven Nutzen für den Anwender zu erfüllen und andererseits ebenfalls verhindert wird, dass die CPU in einen Energiesparmodus versetzt wird.

Die besondere Relevanz von Ebugs für das Testen mobiler Anwendungen ergibt sich daraus, dass sie i. d. R. kein unmittelbar beobachtbares Fehlverhalten der Anwendung, wie etwa Crashes, verursachen. Sie sind deshalb mit konventionellen Testtechniken nur schwer aufzudecken, da sich die Konsequenz des Defekts u. U. erst signifikant später einstellt und nicht mehr objektiv mit einer bestimmten App in Verbindung gebracht werden kann. Hier könnten möglicherweise Testautomatisierungstechnologien eingesetzt werden, um den Energieverbrauch einer bestimmten App über einen Zeitraum zu beobachten. Menschliche Tester sind hierfür ungeeignet, weil eine solche langfristige Beobachtung des SUT ein hohes Maß an Aufmerksamkeit auf einen unspektakulären Test bindet. Die Klasse der Ebugs wird deshalb zwar als von großer Bedeutung für das Testen mobiler Anwendungen mit adäquatem Automatisierungspotenzial betrachtet, kann jedoch aufgrund der zusätzlichen Komplexität im Rahmen dieser Dissertation nicht weiter berücksichtigt werden.

Ridene et al. [297, 296] identifizieren im Jahr 2010 die Abwesenheit von Testschnittstellen als wesentliches Problem des Testens mobiler Anwendungen. Die Möglichkeiten den Zustand einer Anwendung durch Verwendung von Analysewerkzeugen zur Laufzeit zu überwachen sind stark eingeschränkt und beschränken wesentliche Aspekte von Testaktivitäten auf die Beobachtung der Benutzungsoberfläche mobiler Anwendungen.

Zu diesem Zweck entwerfen und implementieren die Autoren einen Prüfstand, in welchem eine mobile Anwendung auf einem Smartphone unter realen Bedingungen getestet werden kann. Insbesondere wird auf die Verwendung von Emulatoren verzichtet; die Anwendung wird auf einem Smartphone installiert, welches über Anbindungen zu kommerziellen Netzwerkbetreibern verfügt. Hierzu wurden die technischen Gegebenheiten geschaffen, eine zu testende Anwendung auf einem Smartphone zu installieren und bei der automatisierten Ausführung von Testfällen durch eine Kamera und ein Mikrophon zu beobachten. Die Relevanz der Arbeit von Ridene et al. für diese Arbeit liegt in der Gemeinsamkeit der Verwendung eines modellbasierten Ansatzes zu Erstellung von Testfällen. Ridene et al. definieren ein Metamodell (auf

Basis des *Eclipse Modeling Framework* (EMF)) sowie eine *Domain Specific Language* (DSL) mit Hilfe derer Testmodelle in einer graphischen Umgebung modelliert werden können. Das Metamodell erlaubt die Spezifikation von Interaktionen des Anwenders mit der Anwendung wie z. B. Interaktionen mit der graphischen Benutzungsoberfläche. Aus dem Modell werden technologiespezifische Test-Skripte erstellt. Eine Middleware übernimmt die Abstraktion von gerätetypischen Besonderheiten (Smartphone-Modell, Verfügbarkeit von GPS, Art der Tastatur, Displayauflösung usw.). Ähnlich dem in dieser Arbeit verfolgten Ziel dient der Prüfstand dazu, aufwändige Tests reproduzierbar zu automatisieren.

Ebenfalls ist die Verwendung modellbasierter Technologien eine wesentliche Gemeinsamkeit zwischen der Arbeit von Ridene et al. und dieser Dissertation. Der in dieser Arbeit verfolgte Ansatz grenzt sich insofern von Ridene et al. ab, als dass auf die Verwendung eines technischen Prüfstands verzichtet wird. Weiterhin liegt der Fokus dieser Arbeit insbesondere auf der Integration von Kontext und Mobilität in die Testfallgenerierung. Das Problem der Simulation von GPS-Informationen oder anderer Kontextparameter liegt nicht im Fokus von Ridene et al., obgleich es neben der effizienten Reproduzierbarkeit von Testfällen eine signifikante Herausforderung des Testens mobiler Anwendungen ist. In weiterer Abgrenzung zielen die in dieser Arbeit erarbeiteten Methoden und Technologien hingegen nicht darauf ab, eine zu testende Anwendung per Fernzugriff auf einer Vielzahl von Geräten zu installieren.

Für das Testen mobiler Software ist deren wachsende Komplexität ein wesentliches Problem. Hierbei muss die Softwarekomplexität unter mehreren Perspektiven betrachtet werden. Einerseits werden mobile Anwendungen aufgrund des steigenden Nutzerbedürfnisses, auch komplexe Geschäftsprozesse mobil bearbeiten zu können, immer komplizierter, d. h. mobile Anwendungen bieten immer mehr Funktionalität an und sind stärker in Backend-Systeme integriert. Durch diese Abhängigkeiten ist bereits in der initialen Entwicklungsphase mit hohem Aufwand für Qualitätssicherungsaktivitäten zu rechnen. Andererseits unterliegen auch mobile Anwendungen Lehmans Gesetzen der Softwareevolution (vgl. Lehman [234]).

Zhang et al. [377] haben in einer Studie aus dem Jahr 2013 untersucht, inwiefern Lehmans Gesetze der Softwareevolution auch für mobile Systeme Anwendung finden. Hierzu wurden repräsentative mobile Anwendungen ausgewählt und durch Anwendung verschiedener Softwaremetriken und insbesondere durch Untersuchung des historischen Verlaufs der Weiterentwicklung anhand von Informationen aus den jeweils verwendeten SCM-Systemen analysiert. Als Ergebnis der Untersuchung wurde festgestellt, dass mobile Apps Lehmans 1., 2. und 7. Gesetzen folgen (die übrigen Gesetze wurden von Zhang et al. nicht untersucht).

Mobile Anwendungen unterliegen während ihres Lebenszyklus kontinuierlichen Veränderungen, zunehmender Komplexität und abnehmender Qualität. Kontinuierliche Veränderung und zunehmende Komplexität macht häufiges Testen erforderlich, um eine Reduzierung von Softwarequalität zu vermeiden. Da hierdurch ein erhöhter Aufwand für das Testen entsteht, ist es erstrebenswert, Testautomatisierungswerkzeuge zu verwenden. Hieraus ergibt sich die Relevanz der Untersuchung von Zhang et al. für diese Dissertation. Nur durch die strukturierte Anwendung von Testautomatisierung kann vermieden werden, dass die Effekte der Lehmanschen Gesetzte zu einem graduellen qualitativen Verfall mobiler Apps führen. Durch die anwachsende Komplexität mobiler Software ist nicht sichergestellt, dass menschliche Akteure die Komplexität des Testens ausreichend beherrschen, um die Produktqualität zu gewährleisten.

Haller [164] befassten sich im Jahr 2013 mit dem Testen mobiler Anwendungen aus der Perspektive agiler Softwareprozesse. Zugrunde liegt die Erkenntnis, dass die kurzen Evolutionszyklen mobiler Plattformen die Machbarkeit traditioneller Testprozesse in Frage stellen. Die plangetriebene Strukturierung von Tests in *Unit-Tests / Komponententests*, Integrationstests (engl. *Integration Tests*), *Systemtests (engl. System Tests)* und Akzeptanztests entlang der Softwareentwicklung in V-Modell-artigen Softwareprozessen ist für mobile Anwendungen nicht geeignet (ähnliche Argumentation auch bei Dantas et al. [87]). In der Regel erfordern die kurzen Iterationszyklen der mobilen Plattformen angepasste Softwareprozesse, in denen auslieferbare Softwareinkremente ebenfalls in kurzen Zyklen erstellt werden können (vgl. Dantas et al. [87]). Insbesondere induzieren die besonderen Anforderungen mobiler Software die Notwendigkeit, diese innerhalb ihres realen Betriebskontextes zu Testen (vgl. Rooksby et al. [304]). Dieses Vorgehen wird von Haller als *Testing in the Wild* (engl. Testen in der Wildnis, entspricht Feldtest) bezeichnet.

Zusätzlich identifiziert der Autor neue Auslöser für Testaktivitäten. Im Umfeld traditioneller Software waren solche Auslöser typischerweise Inkremente des Softwareprodukts, z. B. Implementierung weiterer funktionaler oder nicht-funktionaler Anforderungen. Im Umfeld mobiler Anwendungen können solche Auslöser ebenfalls Inkremente der jeweiligen Plattform sein, die ein erneutes Testen einer App erforderlich machen, ohne dass die App selbst verändert wurde. Veränderungen der Plattform könnten das Betriebsumfeld der App so beeinflussen, dass es seiner Spezifikation nicht länger entspricht, obwohl das zum Zeitpunkt der letzten Produktauslieferung der Fall war.

Der Autor analysiert weiterhin die Bewertungen und Anwenderkommentare von 54 Apps mit dem Ziel, charakteristische Kritikpunkte zu identifizieren. Im Ergebnis stellt Haller fest, dass Anwender besonders häufig Anwendungsdefekte, Crashes und die Beschaffenheit des Geschäftsmodells der App kritisieren. Insbesondere den ersten beiden Kritikpunkten kann durch einen angepassten Testprozess begegnet werden. Der Autor argumentiert, dass, um auch den Anforderungen agiler Softwareprozesse gerecht zu werden, Testautomatisierungswerkzeuge eine wesentliche Voraussetzung sind, da mobile Anwendungen innerhalb kurzer Entwicklungszyklen, u. U. in Integration mit CI-Technologien, sehr häufig getestet werden. Insbesondere in agilen Softwareprozessen werden daher zum Ende eines Iterationszyklus Tests auf allen Stufen wiederholt und die Anzahl der Tests wächst mit jeder Iteration, so dass die von Berner et al. [40] definierten Anforderungen an einen effizienten Einsatz von Testautomatisierungswerkzeugen erfüllt werden. Im Anschluss diskutiert der Autor ein Konzept zur Testautomatisierung für mobile Anwendungen, ohne hierbei jedoch Kontextsensitivität zu berücksichtigen.

Die in dieser Dissertation untersuchte Testautomatisierungstechnologie weist darüber hinaus Parallelen zur von Haller vorgestellten Technologie auf. Jeweils werden Testfälle auf einem Desktop-System interpretiert und steuern das SUT, welches auf einem mobilen Gerät, einem Emulator oder einer Gerätefarm (Emulatorfarm), d. h. auf vielen Geräten (Emulatoren) zur gleichen Zeit, ausgeführt wird. In Abgrenzung zu Haller adressiert diese Dissertation zusätzlich zur Testautomatisierung eine Methode zur Generierung von Testfällen aus Systemmodellen. Das Konzept von Haller impliziert eine manuelle Erstellung von Testfällen.

Aufgrund der besonderen Eigenschaften mobiler Anwendungen gegenüber Desktop- und Server-Software (Mobilität, Kontextsensititvität, Plattform- und Geräteheterogeniät, usw.)

unterscheiden sich auch Defekte in mobiler Software von Defekten in traditioneller Software. Holl und Elberzhager [185] untersuchen in ihrer Arbeit aus dem Jahr 2014, ob Defekte mobiler, kontextsensitiver Anwendungen in diskrete Kategorien klassifiziert werden können und welche Art von Defekten für die jeweilige Klasse typisch ist. Basierend auf einer Analyse wissenschaftlicher Literatur und den Erfahrungen aus Entwicklungsprojekten identifizieren die Autoren übergeordnete Klassen von Defekten: Verhalten der Software, Design der Benutzungsoberfläche und Inhalt. Innerhalb dieser auch für nichtmobile Software gültigen Klassen identifizieren die Autoren eine Reihe von Defektaspekten, die typisch für mobile Anwendungen sind. Hierbei handelt es sich um Konnektivität, Energie, Geräteeinstellungen, User Interface, außerplanmäßige Unterbrechung des Programmablaufs (z. B. durch Kurznachricht, oder Anruf), Schnittstellen und Verwendung von Ressourcen. Diese Defektklassen werden in dieser Dissertation adressiert und explizit im Entwurf der modellbasierten Testautomatisierung berücksichtigt. In dieser Dissertation wird eine gegenüber Holl und Elberzhager erweiterte Klassifikation verwendet, da Holl und Elberzhager physikalische Kontextparameter mit Ausnahme des Standorts des Anwenders nicht berücksichtigen.

2.3.3 Testautomatisierung

Mit Testautomatisierung im Allgemeinen befasste sich Pettichord [280, 281]. Der Autor argumentiert, dass der Erfolg von Testautomatisierungstechnologie wesentlich von der Testbarkeit des Softwareprodukts abhängig ist. Testbarkeit von Software ist das Resultat einer konstruktiven Beziehung zwischen Entwicklern und Testern, Selbstverpflichtung und Hingabe der gesamten Entwicklungsmannschaft und einer frühen Einbeziehung von Testern in den Softwareprozess. Es ist einfacher, ein Softwareprodukt durch Bereitstellung von Testschnittstellen auf eine gute Testbarkeit zu optimieren, als spät im Softwareprozess zu versuchen, Testbarkeit durch Werkzeugverwendung zu forcieren. Pettichord argumentiert, dass sich gute Testbarkeit einer Software in den Eigenschaften *Visibility* (engl. Sichtbarkeit) und *Control* (engl. Kontrolle) äußert. Mit Sichtbarkeit ist hier konkret die Zugänglichkeit von Programmzuständen, Ausgaben, Ressourcenverwendung und Seiteneffekten der Programmausführung gemeint. Der Ansatz fokussiert daher primär auf White-Box-Tests (vgl. Abschnitt 4.1.1.5.1), da diese Eigenschaften von Software i. d. R. nur für einen Test sichtbar werden, wenn der Quellcode beim Testen bekannt ist. Mit Kontrolle meint Pettichord, dass es für die Verwendung von Testautomatisierung notwendig ist, dass ein SUT maschinell mit Eingaben versorgt werden kann und Zustände des SUT extern steuerbar bar sind. In der Regel sind die Eigenschaften Sichtbarkeit und Kontrolle in Software nur dann erfüllt, wenn diese Software entsprechende Schnittstellen zum Testen implementiert. Die Existenz solcher Schnittstellen basiert auf Entscheidungen, diese im System zu realisieren. Zwar wird die Testbarkeit einer Software durch Testschnittstellen wesentlich verbessert, ebenfalls kann hierdurch jedoch ein Sicherheitsrisiko entstehen, wenn diese Schnittstellen im finalen Kompilat verbleiben. Darüber hinaus wird die Integration von Testschnittstellen nach dem Paradigma *Design for Testability* [281] zu einer Aufgabe, die in jedem Softwareprojekt erneut berücksichtig werden muss.

Das in dieser Dissertation untersuchte Konzept zum Testen mobiler, kontextsensitiver Anwendungen greift dieses Paradigma auf und verlagert die Implementierung von Testschnittstellen, insbesondere jener der Kategorie *Control*, vom SUT in die Zielplattform. Hierdurch

wird die Testbarkeit von Softwareprodukten in einigen Aspekten wesentlich vereinfacht. Testschnittstellen sind so konsistent über mehrere Softwareprodukte hinweg verfügbar, ohne ein Sicherheitsrisiko für individuelle Anwendungen zu implizieren. Bei Testschnittstellen, die auf diese Weise in die Zielplattform integriert werden können, handelt es vor allem um solche, die in der Lage sind den Zustand und insbesondere die Betriebsumgebung des SUT global zu manipulieren, wie beispielsweise den Standort von Gerät und Anwender, Verfügbarkeit von Netzwerk oder anderer Ressourcen oder Messwerte von Sensoren. Die Verwendung solcher Informationen hat insofern Einfluss auf die Kernfunktionalität mobiler Anwendungen, als dass sie ein besonders intensives Testen erforderlich macht.

Ebenfalls mit einer Untersuchung von Testautomatisierung im Allgemeinen ohne Fokus auf mobile, kontextsensitive Systeme befassten sich Berner et al. [40] im Jahr 2005. Die Autoren diskutieren in ihrem Erfahrungsreport über 12 Softwareprojekte die Nützlichkeit und den Aufwand von Testautomatisierung. In einem von den Autoren betrachteten Projekt belief sich der Anteil von Code, der auf Tests entfiel auf 25% des in diesem Projekt insgesamt erzeugten Codes. In einem anderen Projekt konnte eine außergewöhnlich lange Laufzeit einer Suite von *Regressionstests (engl. Regression Tests)* von drei Monaten beobachtet werden.

Während die Autoren die Verwendung von Testautomatisierung vom Grundsatz her befürworten, nehmen sie ihre Beobachtungen jedoch zum Anlass, kritisch über einige Eigenschaften und Anwendungsgebiete von Testautomatisierung zu reflektieren. Insbesondere diskutieren die Autoren Fehler bei der Implementierung von Testautomatisierung, durch welche sie die Rentabilität und Effizienz von Automatisierungslösungen in Frage gestellt sehen. An erster Stelle identifizieren die Autoren falsche oder unrealistische Erwartungen an Testautomatisierung, die zu einer vorzeitigen Aufgabe der Automatisierungsbemühungen führen, etwa wenn sich der ROI in einem geringeren Umfang oder später als erwartet einstellt.

Die Einführung von Testautomatisierungswerkzeugen erfordert von Entwicklern und Testern gleichermaßen Anpassungen der Arbeitsweise und Anpassungen an dem in Entwicklung befindlichen Produkt (vgl. *Design for Testability* [281]). Folglich ist zunächst einmal mit einem Anstieg des Aufwands für das Testen und Produktivitätseinbußen bei der Entwicklung zu rechnen, bevor sich die Vorteile von Testautomatisierung bemerkbar machen und der Aufwand für die Testdurchführung unter das Niveau des manuellen Testens absinkt. Hierzu zählt vor allem die Möglichkeit, Inkremente des Softwareprodukts in kürzeren Zyklen an die Stakeholder zu liefern, weil die Testphasen signifikant verkürzt, Tests häufiger ausgeführt und Defekte früher entdeckt und korrigiert werden können [40].

Weiterhin kann die Qualität der Testausführung verbessert werden, wenn Tester nicht mehr länger Aufwand in ermüdende, repetitive und kognitiv wenig fordernde Aufgaben investieren müssen (vgl. Baur und Groth [30] zu den psychologischen Auswirkungen monotoner Tätigkeiten auf die Arbeitsqualität), sondern die hinzugewonnenen Ressourcen nutzen können, um weitere und bessere Tests zu erstellen [40]. Als weitere Fehlerquelle bei der Einführung von Testautomatisierung identifizieren die Autoren die Ansiedlung von Tests auf der verkehrten Teststufe (vgl. Abschnitt 4.1.1.6) oder das Vergessen von Tests. Hierdurch kann die Effizient und die Qualität automatisierter Tests schlimmstenfalls unter das Niveau manueller Tests absinken. Als Ursache hierfür führen Berner et al. an, dass solche Tests, die manuell schwierig auszuführen sind, i. d. R. auch schwierig zu automatisieren sind.

Dieser Effekt wird im Kern von dem in dieser Dissertation untersuchten Konzept zur Testautomatisierung mobiler Anwendungen unter Bereitstellung eines künstlichen Betriebskontext adressiert. Gerade im Umfeld mobiler Anwendungen sind solche Tests, die Software oder ihre Komponenten hinsichtlich der korrekten Verarbeitung von Kontextinformationen prüfen, besonders schwierig manuell auszuführen, da menschliche Akteure im Normalfall nicht in der Lage sind, Kontextparameter bei jeder Testausführung spezifikationskonform zu reproduzieren (vgl. Abschnitt 4.2). Ein Automatisierungswerkzeug wie das in dieser Dissertation untersuchte, welches in der Lage ist, auch komplexe Kontextparameter zuverlässig zu reproduzieren, kann dazu beitragen die Effizienz von Tests signifikant zu erhöhen. Als weiteren möglichen Fehler bei der Einführung von Testautomatisierung diskutieren die Autoren unzureichende Diversifikation automatisierter Tests, z. B. zu frühe Fokussierung auf bestimmte Subsysteme. Beispielsweise gehört das UI einer Software zu denjenigen Komponenten, die gerade in frühen Phasen des Entwicklungszyklus häufig verändert werden. In der Konsequenz müssen automatisierte Tests ebenso häufig angepasst werden. Der durch die Verwendung von Automatisierungswerkzeugen entstehende Effizienzgewinn wird dann durch Mehraufwand durch häufige Anpassung von Tests aufgezehrt.

Der in dieser Dissertation untersuchte Ansatz des modellbasierten Testens kann diesen Effekt abmildern. In der Regel entsteht ein geringerer Aufwand bei der Modifikation von Modellen von Software, als bei der manuellen Anpassung einer großen Anzahl von Testfällen. Werden diese Testfälle direkt aus dem Systemmodell generiert, reduziert sich der Anpassungsaufwand auf die Modifikation des Modells, unabhängig davon wie viele individuelle Tests aus diesem Modell generiert werden. Weiterhin stellten die Autoren fest, dass die Ausführungshäufigkeit individueller Tests ein guter Indikator für die zu erwartenden Profitabilität einer Automatisierungslösung ist. Nach Berner et al. ist die Investition in Testautomatisierung bereits dann lohnenswert, wenn ein einzelner Test häufiger als zehn mal ausgeführt wird. Je nach Autor wird dieser Wert in der wissenschaftlichen Literatur zwischen 2 und 20 Testausführungen angegeben (vgl. Dustin et al. [103], Fewster und Graham [114]), liegt aber in einer so geringen Größenordnung, dass Testautomatisierung de facto in jedem Softwareprojekt lohnenswert ist. Dem gegenüber stehen jedoch substantielle Qualitätseinbußen beim Testen, wenn automatisierte Tests nur selten ausgeführt werden. Je seltener Tests ausgeführt werden, umso weniger sind Entwickler und Tester in späteren Phasen der Entwicklung in der Lage, den Inhalt dieser Tests nachzuvollziehen. Testautomatisierung ist also nicht nur ein probates Mittel zur Erhöhung der Effizienz des Testens, sondern verpflichtet zur weiteren Automatisierung, um die Einspareffekte langfristig abzusichern. Hieraus ergibt sich ein weiterer Vorteil des modellbasierten Generierens von Testfällen, weil sich das Verständnis eines Softwaremodells bei den Stakeholder mit voranschreitender Projektdauer verbessert, wohingegen das Verständnis von Code-basierten Tests umso mehr abnimmt, je weiter der Zeitpunkt des Erstellens des Codes in der Vergangenheit liegt [291].

Schon im Jahr 2001 befassen sich Bylund und Espinoza [67, 68] mit dem Testen kontextsensitiver Anwendungen. In ihrem Ansatz aus der Prä-Smartphone-Ära erkannten die Autoren die Notwendigkeit, Kontextfaktoren (insbesondere den Standort des Anwenders) beim Testen gezielt anpassen zu können, um die Korrektheit des SUT unter dynamischen Kontextbedingungen zu prüfen. Die Autoren unterscheiden zwei unterschiedliche Herange-

hensweisen des Testens: Durchführung von Anwendungstests in einer passiven Testsuite unter Simulation von Kontextbedingungen und die Durchführung interaktiver Tests. Letztere erlauben es dem Tester das SUT während des Testens zu beobachten und den Test während der Durchführung anzupassen.

Ebenfalls im Jahr 2001 entwarfen Bylund und Espinoza eine interaktive Testumgebung, die es dem Tester erlaubt, einen Avatar des Anwenders durch eine virtuelle Umgebung – eine modifizierte Quake III Arena – zu steuern, die Kontextparameter modelliert und simuliert. Weiterhin stellt die virtuelle Umgebung Schnittstellen bereit, durch welche die simulierten Kontextparameter an ein SUT herangeführt werden. Die Arbeit von Bylund und Espinoza ist jedoch ausschließlich für die Simulation von Kontextparametern konzipiert. Eine Testautomatisierung wie die in dieser Dissertation untersuchte war nicht Gegenstand der Untersuchungen. Darüber hinaus existieren jedoch Parallelen zu dem in dieser Dissertation untersuchten Konzept, insbesondere bei der Simulation von Kontextparametern.

Morla und Davies [256, 257] untersuchten in ihrer Arbeit aus dem Jahr 2004 eine ortsbasierte Anwendung aus dem Gesundheitssektor. Getrieben von der Erkenntnis, dass ortsbasierte Anwendungen vom Grundsatz her nur innerhalb ihrer zukünftigen Betriebsumgebung adäquate getestet werden können, fokussierten sich die Autoren spezifisch auf Methoden und Konzepte des Testens mobiler Anwendungen unter der Perspektive, den Aufwand für Testaktivitäten durch Verwendung einer Simulationsumgebung zu reduzieren.

Im Jahr 2004 war weder das technische Umfeld noch der Markt mobiler Anwendungen in großem Umfang ausgeprägt. Mobile Anwendungen waren deshalb i. d. R. auf spezifische Anwendungsdomänen beschränkt, in denen die Beschaffungskosten für mobile Geräte dem Anwendungsfall unter wirtschaftlichen Aspekten untergeordnet werden konnten. Im konkreten Fall untersuchen Morla und Davies den Anwendungsfall einer medizinischen Anwendung, die es erlaubt, dass Patienten mit Herz- und Lungenkrankheiten Sensoren für Vitalwerte ständig am Körper tragen zu können, ohne durch Kabel mit einer Infrastruktur zur Datenübertragung und -auswertung verbunden zu sein. Ziel der Anwendung war die kontinuierliche Übertragung von Vitaldaten des Patienten zusammen mit Informationen über dessen Aufenthaltsort an ein Backend-System. Als mobiles Gerät wurde ein *Personal Digital Assistant* (PDA) verwendet. Zwar ist dieses Gerät in seiner Leistungsfähigkeit kaum mit Geräten der Smartphone-Ära vergleichbar. Parallelen in der Loslösung von stationärer Infrastruktur sind jedoch gegeben, insbesondere im Hinblick auf die Softwarearchitktur, die im Vergleich mit mobilen Anwendungen der Smartphone-Ära ihre Funktion nur in Verbindung mit einem Backend-System erfüllen kann, welches das Gerät ausschließlich über drahtlose Netzwerktechnologien erreichen konnte. Hier identifizieren die Autoren insbesondere die Problematik, dass der Standort des Patienten (d. h. der Anwender der Software) die Funktion der Anwendung beeinflusst, woraus sich besondere Anforderungen an eine Testumgebung ergeben, die in dieser Form im Jahre 2016 noch immer gültig sind. Morla und Davies definieren Anforderungen an eine solche Testumgebung. Zu diesen gehört u. a. die Fähigkeit, den Standort des Anwenders und in Abhängigkeit davon unterschiedliche Eigenschaften der Netzwerkverbindung simulieren zu können.

Obgleich diese Anforderungen bereits im Jahr 2004 formuliert wurden, werden sie von Testtechnologien im Jahr 2016 noch immer nicht oder nur rudimentär erfüllt. Besondere Parallelen zum in dieser Dissertation untersuchten Konzept zur Testautomatisierung sind in der

von Morla und Davies verwendeten Architektur der Automatisierungslösung zu erkennen. In beiden Fällen liegt der Fokus darauf, manuellen Interaktion mit den SUT zu minimieren. Die von den Autoren vorgestellte Architektur ermöglicht es, dass die zu testende Anwendung automatisch innerhalb der Testumgebung installiert und ausgeführt werden kann. Eine zusätzlich installierte Komponente übernimmt während der Testdurchführung die Rolle des Anwenders und stimuliert das SUT mit Interaktionen und beobachtet das Verhalten der Anwendung, u. a. durch Inspektion des UI. Gleichtzeitig ist die Automatisierungslösung in der Lage, bestimmte Eigenschaften der Betriebsumgebung des SUT – also dessen Kontext – zu simulieren.

Das in dieser Dissertation untersuchte Konzept zur Testautomatisierung mobiler Anwendungen ist dem von Morla und Davies vorgestellten ähnlich. Allerdings haben sich seit 2004 die Anforderungen an mobile, kontextsensitive Anforderungen gewandelt. Hierzu beigetragen haben die technischen Möglichkeiten mobiler Geräte der Smartphone-Generation, die weit mehr Sensoren zur Wahrnehmung von Parametern der Betriebsumgebung bieten. Beispielsweise hat sich die Bedeutung des Standortes des Anwenders von einem Indikator der Netzwerkverfügbarkeit zu einem Kontextparameter erster Klasse entwickelt, bei dem nicht mehr primär Seitenaspekte der Kontextinformation von Bedeutung sind, sondern der Standort selbst die primäre Information ist. Darüber hinaus haben sich ebenfalls die technologischen Grundlagen mobiler Geräte verändert. PDAs wie die von Morla und Davies verwendet, führen mittlerweile ein Nischendasein und sind im Vergleich zu den Plattformen Android, Apple iOS und Microsoft Windows Phone wirtschaftlich obsolet. Das von Morla und Davies vorgestellte Konzept zur Bewältigung der Herausforderungen des Testens mobiler kontextsensitiver Anwendungen kann deshalb bis auf grundlegende Architekturkonzepte für Testautomatisierungswerkzeuge nicht auf das Umfeld mobiler Anwendungen des Jahres 2016 übertragen werden.

Broens und van Halteren [59] untersuchten im Jahr 2006 einen weiteren Ansatz zur Simulation von Kontext bei der Durchführung von Softwaretests. Basierend auf der Erkenntnis, dass Feldtests einerseits hohe Kosten verursachen und andererseits kaum geeignet sind, Tests zuverlässig zu reproduzieren (z. B. wegen Ungenauigkeit des GPS-Moduls), entwickelten Broens und van Halteren eine Simulationsumgebung mit den Zielen, die Kosten für Tests zu reduzieren, die Reproduzierbarkeit von Tests zu gewährleisten und die Zeit für Entwicklung und Testdurchführung zu reduzieren. Hierzu diskutierten die Autoren zunächst nichtfunktionale Anforderungen an eine Simulationsumgebung. Als erste Anforderung an eine solche Simulationsumgebung identifizierten die Autoren Generizität, d. h. sie muss in der Lage sein, eine Vielzahl unterschiedlicher Kontextfaktoren simulieren zu können, ohne hierbei auf eine spezifische Anwendungsdomäne festgelegt zu sein. Da die Menge von Kontextquellen mit der Weiterentwicklung mobiler Geräte anwächst, ist die zweite Anforderung Erweiterbarkeit. Das heißt, die Simulationsumgebung muss mit geringem Aufwand an weitere Kontextquellen angepasst werden können. Weiterhin identifizierten die Autoren eine Reihe funktionaler Anforderungen. Hierzu gehören Präzision, Korrektheit, Zuverlässigkeit, Auflösung und Aktualität.

Diese Anforderungen zu befriedigen ist eine Simulationsumgebung nur dann in der Lage, wenn das zugrundeliegende Metamodell zur Kontextmodellierung diese Anforderungen abbildet. In Abschnitt 5.2.2.1 werden deshalb Anforderungen an ein solches Metamodell im Hinblick auf die in dieser Dissertation untersuchte Testfallgenerierungs- und Testautomatisierungstechnologie unter Berücksichtigung der von Broens und van Halteren identifizierten

Anforderungen diskutiert. Von einer Simulationsumgebung fordern die Autoren die Fähigkeit, Kontextparameter während der Simulation bzw. während der Testdurchführung dynamisch anpassen zu können. Dies wird sowohl für konkrete Werte von Kontextparametern als auch für Metainformationen zu Kontextparametern gefordert. Beispielsweise soll es möglich sein, dass sich sowohl der simulierte Standort eines Anwenders ändert als auch die Präzision dieser Standortinformation. Aufgrund der technologischen Implementierung der Simulationsumgebung von Broens und van Halteren ist diese jedoch nicht zur Verwendung mit mobilen Anwendungen der Smartphone-Ära geeignet und ist deshalb nur in Teilen auf Konzeptebene auf den in dieser Dissertation untersuchten Ansatz zur Testautomatisierung übertragbar.

Unter der Bezeichnung *MobileTest* stellten Bo et al. [48] im Jahr 2007 ebenfalls einen Ansatz zur Testautomatisierung für mobile Anwendungen vor. Die Autoren diskutieren einen Black-Box-Test-basierten Ansatz, Interaktionen des Anwenders mit einer Anwendung zu simulieren und die Reaktion der Anwendung automatisiert zu beurteilen. Zwar zielt der vorgestellte Ansatz auf Technologien der Prä-Smartphone-Ära ab, berücksichtigt jedoch Faktoren, die auch im Jahr 2016 für das Testen mobiler Anwendungen der Smartphone-Generation gelten. So argumentieren die Autoren, dass es aufgrund der Ressourcenbeschränkung mobiler Geräte zweckmäßig ist, Teile der Testausführung vom mobilen Gerät abzutrennen und stattdessen auf einem leistungsstarken Desktop-Computer auszuführen. Hierdurch kann verhindert werden, dass die Testausführung selbst die zu testende Anwendung korrumpiert und die Testergebnisse obsolet werden. Dieses Risiko existiert auf Geräten der Smartphone-Generation unverändert. Wenngleich diese Geräte deutlich leistungsfähiger sind als frühe Vertreter ihrer Klasse, obliegt die Ressourcenverwaltung auch bei modernen mobilen Plattformen dem Betriebssystem. Die Ausführung des Tests direkt auf einem Gerät, einschließlich der Auswertung der Ergebnisse, könnte Lastspitzen verursachen, die das Betriebssystem veranlasst die zu testende Anwendung zu terminieren, wodurch ein Abschluss des Tests unmöglich wird. Bo et al. stellen deshalb eine Architektur einer Testautomatisierungstechnologie vor, die wesentliche Teile der Testausführung auf einen Desktop-Computer verlagert, insbesondere die Interpretation und Auswertung der Testergebnisse. Dieses Architekturmodell wird auch von dem in dieser Dissertation als Basistechnologie verwendeten Automatisierungswerkzeug Calabash aufgegriffen, um eine Störung der zu testenden Anwendung durch die Präsenz des Testwerkzeugs zu minimieren.

Hu und Neamtiu [188] untersuchten 2011 eine Technologie zum automatisierten Testen von Android-Anwendung. In einem ersten Schritt bestimmten die Autoren eine Reihe von Defektklassen (vgl. Abschnitt 4.1.2), die für Android-Anwendungen charakteristisch sind (*Activity*[1]-Defekte, *Event*-Defekte, dynamische Typisierungsdefekte, unbehandelte *Exceptions*, API-Defekte, *I/O*-Defekte und *Concurrency*-Defekte). Anschließend analysierten die Autoren zehn OSS-Anwendungen die öffentlich im Google Play Store erhältlich sind. Durch Inspektion der SCM-Repositorien der Anwendungen konnten die Autoren bereits bekannte Defekte der untersuchten Anwendungen den zuvor erstellen Defektklassen zuordnen. Für die Klassen *Activity*-Defekte, *Event*-Defekte, dynamische Typisierungsdefekte führten die Autoren im Anschluss einen Reihe automatisierter Tests durch, bei denen mit der verwendeten Technologie bereits

[1]Gemeint ist hier die durch das Android-SDK bereitgestellte Komponente *Activity* der Android-Anwendungsarchitektur.

bekannten Defekte wiederentdeckt sowie einige neue Defekte erkannt wurden. Als Automatisierungstechnologie verwendeten die Autoren ausschließlich Werkzeuge, die im Android-SDK bereits enthalten sind, konkret ein *JUnit-Framework* (Rahmenwerk) sowie die Technologie *UI/Application Exerciser Monkey* [146]. Für jede der untersuchten Apps wurde eine Reihe generischer Unit-Tests erstellt, deren einziger Gegenstand die Überprüfung einer korrekten Implementierung des Android Activity-Protokolls war, d. h. Tests, ob das SUT korrekt auf Zustandswechsel im Lebenszyklus der jeweiligen Activity reagiert. Das Werkzeug UI/Application Exerciser Monkey erfüllt die Aufgabe, ein SUT stellvertretend für einen Anwender mit einem Strom von Interaktionsereignissen (z. B. Texteingaben, Anklicken von Schaltflächen) zu stimulieren. Es handelt sich bei UI/Application Exerciser Monkey konkret um ein Werkzeug für sogenannte *Smoke-Test* (engl. Rauchtest), dessen einzige Funktion es ist festzustellen, ob ein Softwareartefakt bei einer bestimmten Sequenz von Interaktionsereignissen abstürzt. Funktionale Tests sind nicht möglich. Die Autoren erstellten mit Hilfe des UI/Application Exerciser Monkey Sequenzen von Interaktionsereignissen und führten diese jeweils den untersuchten Anwendungen zu. Für den Fall, dass eine der Anwendungen abstürzte, untersuchten die Autoren das Systemprotokoll nach dem verursachenden Fehler.

Wenngleich dieser Ansatz der Testautomatisierung weit hinter den technischen Möglichkeiten des Jahres 2016 zurückbleibt, gelang es den Autoren dennoch, in mehr als der Hälfte der untersuchten Anwendungen bislang unbekannte Defekte aufzudecken und den zuvor erstellten Defektklassen zuzuordnen. Der in dieser Dissertation untersuchte Ansatz zur Testfallgenerierung und zur Testautomatisierung geht weit über das von Hu und Neamtiu realisierte Konzept hinaus. Zwar werden auch beim hier untersuchten Ansatz Anwendungen mit künstlichen Interaktionsereignissen stimuliert, allerdings werden diese nicht zufällig durch das Werkzeug UI/Application Exerciser Monkey generiert, sondern entspringen einem Testmodell, welches auf Basis von Entwurfsmodellen der zu testenden Anwendung durch Modelltransformation erzeugt wurde. Darüber hinaus ist das von Hu und Neamtiu vorgestellte Konzept nicht in der Lage, den Betriebskontext des SUT zu berücksichtigen und wird damit den Anforderungen an das Testen mobiler, kontextsensitiver Anwendungen nicht ausreichend gerecht.

Ebenfalls am Beispiel der mobilen Plattform Android untersuchten Anand et al. [8] 2012 einen Ansatz zur Generierung von Tests auf Basis von *Events* (engl. Ereignissen) der UI-Interaktion (z. B. Anklicken berührungsempfindlicher Schaltflächen, Texteingabe, Einstellen von Schiebereglern). Die Autoren stellten fest, dass aufgrund der engen Verzahnung von App-Code mit dem Code des Betriebssystems, asynchroner Interprozesskommunikation, loser Kopplung von App-Komponenten und dem UI einer Android-App eine statische Analyse von Programmcode zur Fehlersuche de facto aussichtslos ist. Effektives Testen kann nur dynamisch durch Ausführen der App und Simulation von UI-Events erfolgen.

In der Regel bietet ein UI einer App mehrere interaktionsfähige UI-Elemente, die je nach Funktion und Zustand der App unterschiedliche Aufgaben erfüllen. Die Erstellung von Testfällen beinhaltet folglich auch die Bestimmung von UI-Event-Sequenzen, die Testfälle repräsentieren, da App-Funktionalität i. d. R. durch Interaktionen des Anwenders gesteuert wird.

Anand et al. adressieren in ihrer Arbeit die Fragestellung, wie Sequenzen von UI-Events mit geringem manuellen Arbeitsaufwand identifiziert werden können. Prinzipiell könnten neben vollständig manuellem Bilden von Interaktionssequenzen *Capture and Replay* (C'n'R)

(engl. Aufzeichen und Wiederabspielen) Ansätze oder modellbasierte Techniken verwendet werden. Die Autoren argumentieren gegen C'n'R-Techniken, da diese nach der Aufzeichnung auf eine bestimmte Kombination von Betriebssystemversion, Gerät und SUT festgelegt sind. Die ökonomischen Vorteile der Wiederverwendung von Tests durch Übertragung auf weitere Kombinationen von Gerät und SUT können dann nicht ausgeschöpft werden. Im Gegensatz zu dem in dieser Dissertation untersuchten Ansatz entscheiden sich Anand et al. ebenfalls gegen einen modellbasierten Ansatz und verwenden stattdessen eine explorative Lösung, die das UI des SUT zur Laufzeit analysiert und interaktive UI-Elemente identifiziert.

Die von den Autoren untersuchte Technologie fokussiert auf die Generierung von Interaktionssequenzen und ist deshalb zunächst auf einfache Klick-Events reduziert. Da ein Algorithmus nicht in der Lage ist festzustellen, welche Interaktionssequenzen gemäß der Spezifikation sinnvoll sind, erzeugt die Analyse des UI neben gültigen Sequenzen ebenfalls ungültige Interaktionssequenzen. Diese repräsentieren ebenfalls gültige Testfälle, da es ein Qualitätsmerkmal einer App ist, definiert und defektfrei auf ungültige Interaktionsequenzen zu reagieren. Insofern stellt der von Anand et al. vorgestellte Ansatz einen wesentlichen Vorteil gegenüber den erratisch generierten Interaktionssequenzen von Smoke-Test-Werkzeugen (z. B. UI/Exerciser Monkey, Abschnitt 2.5.1.2) dar, weil nur solche Sequenzen von Interaktionen erzeugt werden, die UI-Elemente gezielt und reproduzierbar adressieren.

In einer weiteren Abgrenzung zu dem in dieser Dissertation untersuchten Konzept verwendet der von Anand et al. vorgestellte Ansatz ein angepasstes Android-SDK, welches es ermöglicht auf der Plattform Android den Fluss von UI-Events durch das System zu beobachten. Das in dieser Dissertation untersuchte Konzept verwendet hingegen eine angepasste Version des Android-Betriebssystems. Während der prinzipielle Ansatz eine Android-App gezielt zur Testdurchführung zu instrumentalisieren auch von Anand et al. verfolgt wird, unterscheidet sich das in dieser Dissertation untersuchte Konzept in einigen Punkten signifikant von dem von Anand et al. vorstellten Ansatz. Anstelle der Exploration des UI der App wird hier ein modellbasierter Ansatz verfolgt, um Testfälle durch Modelltransformation aus Systemmodellen zu generieren. Zwar entsteht hierdurch ein erhöhter Modellierungsaufwand, aber es können so Aspekte des SUT adressiert werden, die mit dem von Anand et al. favorisierten Ansatz nicht realisierbar sind. Insbesondere kann das hier untersuchte Konzept auch solche Events in ein SUT einspeisen, die dem UI i. d. R. verborgen bleiben, wie etwa Aktualisierungen des Standortes via GPS oder Sensorinformationen. Die Abhängigkeit einer App von solchen Kontextparametern können durch eine Analyse des UI einer Anwendung nicht entdeckt werden. Hierdurch ist der von Anand et al. vorgestellte Ansatz nicht für kontextsensitive Apps nutzbar, da wesentliche funktionale Aspekte nicht berücksichtigt werden.

Kaasila et al. [209] untersuchten im Jahr 2012 einen C'n'R-Ansatz mit dem Namen *Testdroid* [45] zur Generierung von UI-Tests für die Plattform Android. Die Autoren erkennen die Notwendigkeit, Tests auf unterschiedlichen Geräten auszuführen, um sicherzustellen, dass trotz der Heterogenität der Android-Gerätelandschaft ein adäquates Qualitätsniveau erreicht wird und UIs von Apps auf allen Geräten nahezu identisch aussehen und eine gleichbleibende UX bieten sollen. Hierzu verwendet Testdroid Werkzeuge des Android SDK, um während der manuellen Ausführung des SUT die Hierarchie des UI des SUT zu analysieren, UI-Ereignisse aufzuzeichnen und anschließend in ein Testskript zu konvertieren.

Zur nachfolgenden automatisierten Ausführung dieser aufgezeichneten Tests werden diese Testskripts in Tests der Technologie *Robotium* (vgl. Abschnitt 2.5.2.1) transformiert. Diese können im Anschluss als regelmäßige JUnit Tests auf einem Gerät oder einem Emulator ausgeführt werden. Eine Besonderheit des von Kaasila et al. vorgestellten C'n'R-Ansatzes ist es, dass Tests nicht nur auf einem Gerät oder Emulator ausgeführt werden können, sondern über einen Web-Service auf einer Gerätefarm auf vielen unterschiedlich profilierten Geräten gleichzeitig ausgeführt werden können. Während der Ausführung von Tests zeichnet Testdroid alle Systemprotokolle sowie Bildschirmabdrucke auf. Diese werden dem Tester nach der Testausführung auf einem Web-Portal angezeigt. Der Vorteil dieses Ansatzes liegt in der Möglichkeit, Tests auf eine intuitive Weise zu definieren, nämlich der manuellen Ausführung des SUT sowie der Verfügbarkeit einer großen Anzahl unterschiedlicher Geräte in der Farm der Betreiber. Nachteilig ist hingegen, dass sowohl die Erstellung von Tests als auch die Analyse von Testergebnissen weiterhin einen hohen Anteil manuelle Arbeit erfordert. Insbesondere bei der Erstellung von Tests erfährt der Entwickler oder Tester keinerlei Unterstützung. Hierbei handelt es sich um eine besonders herausfordernde Aufgabe, da die Komplexität von Software schnell einen Grad erreicht, der für einen menschlichen Akteur schwer zu überblicken ist. Dieses Problem wird in Abgrenzung zu Kaasila et al. in dem in dieser Dissertation vorgestellten Konzept zur Testautomatisierung explizit durch einen modellbasierten Ansatz adressiert, in dem Tests nicht manuell erstellt, sondern aus Modellen des Systementwurfs generiert werden. Weiterhin adressiert das in dieser Dissertation untersuchte Konzept gezielt kontextsensitive Anwendungen, bei denen das Verhalten einer App nicht nur durch Interaktionen des Anwenders, sondern ebenfalls durch zahlreiche Kontextparameter (vgl. Abschnitt 3.1.2) gesteuert wird. Dieser Aspekt wird von Kaasila et al. nicht adressiert.

Einen weiteren C'n'R-Ansatz mit der Bezeichnung *RERAN* zur Testautomatisierung untersuchten Gomez et al. [126] im Jahr 2013. Die Autoren argumentieren, dass traditionelle C'n'R-Werkzeuge i. d. R. auf dem Paradigma *Point and Click* (P'n'C) basieren, in welchem Anwender mit einer Zeigeeinrichtung, z. B. Maus oder Stylus, diskrete Schaltflächen der UI manipulieren. Für Smartphone-Anwendungen, so die Autoren, ist dieses Paradigma nur eingeschränkt gültig, weil hier häufig alternative UIs implementiert werden, die nicht auf diskrete Schaltflächen beschränkt sind, sondern Möglichkeiten der Interaktion anbieten, die sich über die gesamte Displayfläche erstrecken und deshalb keinem UI-Element eindeutig zuordenbar sind, wie etwa *Swipe* (engl. gerichtete Wischgeste) oder *Pinch-Zoom* (engl. Geste mit mind. zwei Fingern zur Vergrößerung der Darstellung von Bildschirminhalt).

Darüber hinaus verfügen mobile Geräte über Sensoren, die ebenfalls als Eingabevektor zur gezielten Steuerung einer App verwendet werden können, etwa durch Neigen der Geräts. Um das Testen solcher Anwendungen zu automatisieren, entwickeln Gomez et al. eine Technologie, die Interaktions-Events auf einer hardware-nahen Ebene des Systems aufzeichnet und zu einem späteren Zeitpunkt erneut einspeisen kann. Der Ansatz ist an die Architektur und die SDK-Werkzeuge der Android-Plattform gebunden und nicht auf andere mobile Plattformen übertragbar. Allerdings erlaubt die Technologie Black-Box-Tests, da weder zur Aufzeichnung von Test noch zur späteren maschinellen Wiederholung Modifikationen am SUT notwendig sind. Die Technologie erlaubt nach Aussage der Autoren die Aufzeichnung und Wiedergabe von Sensordaten aller im Gerät vorhanden Sensoren. Das Sensorsortiment variiert jedoch von

Gerät zu Gerät, so dass eine Übertragbarkeit von Tests, die auf dieser Technologie basieren, auf andere Geräte als dem zur Aufzeichnung verwendeten nicht möglich ist.

Eine im Rahmen dieser Dissertation durchgeführte Evaluation der Technologie RERAN hat zudem ergeben, dass entgegen der Aussage der Autoren tatsächlich nicht alle Sensoren mobiler Geräte erfasst werden können. Das Gerät ASUS Nexus 7 aus dem Modelljahr 2013 (ASUS-1A019A) beispielsweise exponiert nicht alle Sensoren in der Weise, die durch die von Gomez et al. erprobte Technologie adressierbar sind. Dieses Problem ergibt sich unmittelbar aus der Fragmentierung der Android-Plattform, bei der Geräteherstellern die konkrete Implementierung hardwarenaher Funktionen jeweils individuell überlassen bleibt.

Dennoch stellten die Autoren einen interessanten Ansatz vor, der im Umfeld der im Jahr 2016 verfügbaren C'n'R-Technologien eine herausragende Stellung einnimmt. Allerdings ist gerade die Heterogenität der Android-Plattform ein wesentlicher Aufwandstreiber des Testens, der durch die von Gomez et al. entwickelte Technologie nicht adressiert wird. Hier sind modellbasierte Technologien wie die in dieser Dissertation untersuchte, dem RERAN-Ansatz überlegen, weil Tests ohne Anpassungen auf eine großen Anzahl unterschiedlicher Geräte übertragbar sind. Darüber hinaus wird die Idee, Sensordaten beim Testen in das SUT einzuspeisen, auch in dem in dieser Dissertation untersuchen Konzept zur Testautomatisierung aufgegriffen, allerdings anders realisiert.

Getrieben von der Erkenntnis, dass Defektfreiheit eine wesentliche Voraussetzung für den wirtschaftlichen Erfolg im kompetetiven App-Markt ist (vgl. Kim et al. [223], Harman et al. [167], Zhong und Michahelles [379], Khalid [220, 222, 221]), stellten Ravindranath et al. [292] im Jahr 2014 einen holistischen Ansatz zum Testen mobiler Anwendungen für die Plattform Microsoft Windows Phone vor, der anderen bis zu diesem Zeitpunkt verfügbaren Testwerkzeugen in Funktionalität und Benutzbarkeit um Längen voraus war. Der App Store der Plattform Microsoft Windows Phone aggregiert sogenannte *Stack Traces* (engl. Stapelzurückverfolgung, Werkzeug zur Identifikation und Lokalisierung von Defektursachen in Software) von solchen Defekten, die einen Crash einer App verursacht haben. Ravindranath et al. analysierten 25 Millionen solcher Crash-Reports für mehr als 100.000 Microsoft Windows Phone Apps aus dem Jahr 2012 und konnten dabei feststellen, dass 90% aller beobachteten Crashes auf nur etwa 10% der häufig beobachteten Defekte zurückzuführen sind und insbesondere ein signifikanter Anteil dieser Defekte aus externen Einflüssen resultierte. In der Terminologie der Autoren überschneiden sich diese externen Einflüsse mit einigen jener Faktoren, die in dieser Dissertation als Kontextparameter bezeichnet werden, z. B. unvorhersehbare Schwankungen der Qualität mobiler Netzwerke.

Die Autoren argumentieren, dass das Testen mobiler Anwendungen sich in wesentlichen Aspekten vom Testen traditioneller Anwendungen unterscheidet: Sie werden in unkontrollierten und unkontrollierbaren Umgebungen verwendet, der Standort von Gerät und Anwender ist nicht vorhersehbar, es kommen unterschiedliche Netzwerke verschiedener Betreiber zum Einsatz und über die Eigenschaften der vom Anwender verwendeten Hardware ist – wenigstens zur Entwurfszeit – nur wenig bekannt.

Diese Argumente bilden ebenfalls die Grundlage des in dieser Dissertation untersuchten Konzepts zur Testautomatisierung mobiler Anwendungen (vgl. Abschnitt 1.2.3, Abschnitt 1.2.4 und Abschnitt 4.2). Ebenfalls in Übereinstimmung mit dem in dieser Dissertation untersuchten

Konzept kommen Ravindranath et al. zu dem Schluss, dass aufgrund technischer Eigenschaften mobiler Plattformen eine statische Analyse von App-Code zur Defektsuche wenig nützlich ist, sondern nur dynamische Testverfahren geeignet sind, mobile Anwendungen adäquat auf Fehler zu überprüfen. Ebenfalls argumentieren die Autoren, dass auch symbolische Programmausführung kein adäquates Vorgehen zum Testen ist, weil die Modellierung der vollständigen Betriebsumgebung im sich schnell weiterentwickelnden technologischen Umfeld quasi unmöglich ist. Es ist der Anspruch der Autoren, Entwickler in die Lage zu versetzen, ein fertiges Kompilat einer App zum Testen in ein automatisiertes Werkzeug einzureichen und nicht etwa ein Modell oder Quellcode. Die von den Autoren entwickelte Technologie *VanarSena* fügt dem Binärcode der App weitere Funktionalität hinzu, unter anderem Schnittstellen zur Manipulation von Netzwerkkommunikation, Anwenderinteraktion oder Sensorinformationen. Diese nachträgliche Anreicherung der App mit Testschnittstellen hat einerseits den Vorteil, dass diese Schnittstellen im ausgelieferten Kompilat nicht existieren und deshalb kein potenzielles Sicherheitsrisiko darstellen. Andererseits erfolgt das Testen durch dieses Konzept an einem modifizierten Kompilat, so dass Testergebnisse u. U. nicht repräsentativ sind, weil faktisch eine andere App getestet wird, als am Ende an den Anwender ausgeliefert wird.

Trotz dieser Einschränkungen ist es Ravindranath et al. gelungen, mit ihrem Ansatz in einer empirischen Studie über 3000 Apps für Microsoft Windows Phone 1108 bislang unbekannte Defekte zu entdecken, die zu insgesamt 2969 Crashes führten. Konkret wurden in 60% der getesteten Apps ein oder zwei Defekte entdeckt, die zum Absturz der jeweiligen App führten. Als unmittelbaren Vorteil der Automatisierungstechnologie stellen die Autoren heraus, dass die 3000 Apps in etwa 4500 Maschinenstunden, verteilt auf 12 Desktop-Computer getestet werden konnten. Das entspricht etwa 1,5 Stunden pro App. Allerdings versäumen es die Autoren die Komplexität der jeweiligen Tests zu dokumentieren und die Ausführungsdauer in Relation zur einer manuellen Ausführung zu setzen.

In dem in dieser Dissertation untersuchten Konzept der Testautomatisierung wird in Abgrenzung zur Ravindranath et al. nicht das SUT um Testschnittstellen erweitert, sondern das Betriebssystem der mobilen Plattform. Weiterhin räumen die Autoren ein, dass ihre Technologie nicht in der Lage ist, alle Arten von Defekten zu erkennen, sondern lediglich solche, die bei der Analyse von Protokollen aus dem App Store als Klassen häufiger Defekte identifiziert wurden. Insbesondere ist anzumerken, dass es sich beim von Ravindranath et al. vorgestellten Ansatz um eine Smoke-Test-Technologie handelt, die lediglich in der Lage ist festzustellen, ob eine App unerwartet terminiert. Unmöglich ist mit dieser Technologie hingegen die Durchführung funktionaler Tests, die neben der technischen Korrektheit ebenfalls die fachliche Spezifikationskonformität der App beurteilen. Dies ist ein klares Abgrenzungskriterium zu dem in dieser Dissertation untersuchten Konzept zur Testautomatiserung für mobile Anwendungen.

Darüber hinaus existieren allerdings eine Reihe von Gemeinsamkeiten des hier untersuchten Konzepts der Testautomatisierung und dem von Ravindranath et al. vorgestellten. Zunächst betrachten beide Konzepte anwenderzentrierte Tests, also Akzeptanztests aus der Perspektive des zukünftigen Benutzers. Weiterhin identifizieren die Autoren ebenfalls für mobile Anwendungen typische Klassen von Defekten, die sich insbesondere aufgrund der technischen Besonderheiten mobiler Plattformen ergeben (vgl. Abschnitt 4.1.2 sowie Abschnitt 4.2). Anders als im hier untersuchten modellbasierten Ansatz fokussieren sich Ravindranath et al.

jedoch auf einen explorativen Ansatz der App-Analyse, bei dem versucht wird mögliche Interaktionssequenzen zur Laufzeit der App zu ermitteln. Die Autoren analysieren hierzu das UI des SUT und versuchen interaktive UI-Elemente zu identifizieren. Ein randomisierter *UI Automator* (Technologie zur Simulation von Interaktion eines menschlichen Akteurs, vgl. Abschnitt 2.5.2.3) stimuliert diese dann und versucht gezielt einen Defekt der zuvor erstellten Defektklassen zu verursachen, beispielsweise durch Eingabe technisch unzulässiger Werte in Eingabefelder (z. B. Text in ein Eingabefeld, dass in der UI-Auszeichnungssprache XAML als Eingabefeld für numerische Werte deklariert wurde).

Durch das randomisierte Vorgehen ergeben sich allerdings schwerwiegende Folgen für die Testabdeckung des SUT. Ravindranath et al. entwicklen hierzu eine Abdeckungsmetrik, die bewertet, wie viele der einzelnen UI-Seiten bei der Testausführung tatsächlich erreicht wurden. Es besteht hier durch den randomisierten Ansatz das Risiko, dass gemäß dieser Metrik eine schlechte Abdeckung erzielt wird, weil Teile der App unberücksichtigt bleiben. In einem modellbasierten Ansatz wie dem in dieser Dissertation vorgestellten besteht dieses Risiko hingegen nicht, da beim Entwurf von testspezifischen Systemmodellen eine Abdeckung gemäß einer solchen Metrik gezielt gewählt werden kann.

Zu den Gemeinsamkeiten des hier vorgestellten Konzepts und dem von Ravindranath et al. entworfenen gehört die Verwendung einer dedizierten Komponente, die es erlaubt, den Kontext des SUT während der Laufzeit zu manipulieren. So können beispielsweise Eigenschaften der Netzwerkverbindung gezielt während der Testausführung manipuliert werden, um festzustellen, ob das SUT auch unter diesen Bedingungen korrekt funktioniert. In der Terminologie der Autoren wird diese Komponente als *Fault Inducer* (engl. Defektinduzierer) bezeichnet. In dieser Dissertation übernimmt diese Aufgabe der Kontextsimulator (vgl. Abschnitt 5.5.1), der in Abgrenzung zu Ravindranath et al. jedoch in der Lage ist, neben der Netzwerkverfügbarkeit weitere Kontextparameter, wie etwa Sensordaten und Standortinformationen, zu steuern.

Liu et al. [240] untersuchen im Jahr 2014 einen weiteren Ansatz des C'n'R-Testens für Android-Anwendungen mit dem Ziel, den Aufwand für die Testfallerstellung zu reduzieren. Die Autoren argumentieren, dass selbst in teilautomatisierten Testprozessen signifikanter Aufwand in die Erstellung von Tests, beispielsweise Unit-Tests oder Skripte zur Steuerung alternativer Automatisierungstechnologien wie etwa Calabash, investiert werden muss. Als besondere Herausforderung des C'n'R-Testens stellen Liu et al. die neuartigen Interaktionsmodalitäten mobiler Software heraus. Während traditionelle Software für Desktopgeräte beinahe exklusiv mit Tastatur und Maus bedient werden, unterstützen mobile Geräte eine Vielfalt von Interkationsmodalitäten (Bedienung via berührungsempfindlichem Bildschirm, Gesten mit einem oder mehreren Fingern auf der Bildschirmfläche, Sprachinteraktion, Bewegungsgesten mit dem Geräte) und eine Vielfalt spezialisierter UI-Elemente.

Die Autoren entwerfen eine Technologie zur Aufzeichnung des Ereignisstroms bei einer manuellen Ausführung einer Android-Anwendungen. Hierzu implementieren Liu et al. ein besonderes UI-Element, dass direkt unterhalb des Wurzel-UI-Elements einer Android App eingefügt wird und alle UI-Events (d. h. Interaktionsereignisse) abhört und in ein Protokoll exportiert. Im Anschluss wird aus diesem Protokoll Quellcode für einen Test für das Robotium-Framework (vgl. Abschnitt 2.5.2.1) generiert. In diesem sind Informationen zu den zu manipulierenden UI-Elementen zusammen mit den Testdaten kodiert.

Bei Robotium handelt es sich um ein Framework der JUnit-Familie, die Überprüfung von Nachbedingungen erfolgt im Test durch JUnit-*Assertions*. Diese werden in der von Liu et al. vorgestellten Technologie durch eine Touch-Geste zum Ereignisprotokoll hinzugefügt. Die automatisierte Wiederholung des Tests erfolgt, indem der aus dem Ereignisprotokoll gewonnene Quellcode zu ausführbaren Testfällen kompiliert und durch Verwendung der Android Testwerkzeuge ausgeführt wird. Es wäre hier auch möglich gewesen, unter Auslassung des Robotium-Frameworks eine Technologie zu verwenden, die direkt UI-Events in das SUT injiziert. Allerdings ist das bei der Aufzeichnung entstandene Ereignisprotokoll gerätespezifisch und zudem abhängig von der Orientierung des Geräts (Hochformat, Querformat). Damit wäre eine direkte Wiederverwendung der UI-Events nur auf demselben Gerät in identischer Orientierung möglich. Die Verwendung von Robotium ermöglicht hier eine Abstraktion von Gerät und Orientierung und somit eine Wiederverwendung der Testfälle auf anderen Geräten.

Weiterhin erzeugt die von Liu et al. vorgestellte Technologie menschenlesbaren Quellcode für Testfälle, wodurch diese leicht manuell angepasst werden können. Das von den Autoren vorgestellte Konzept ist deshalb im Kontext dieser Dissertation interessant, weil menschenlesbare Testfälle (in Form von Quellcode) generiert werden, die im Anschluss durch eine Automatisierungstechnologie mit geringem manuellen Aufwand beliebig oft wiederholt und in CI-Umgebungen integriert werden können. Das in dieser Dissertation untersuchte Konzept zur modellbasierten Testfallgenerierung und automatisierten Ausführung erfüllt diese Anforderungen ebenfalls. Die Aufgabe der initialen Erstellung von Testfällen wird von Liu et al. jedoch nicht adressiert. Es verbleibt weiterhin in der Verantwortung eines menschlichen Akteurs eine adäquate Menge von Tests zu erzeugen, die gewählte Abdeckungskriterien erfüllen. Der in dieser Dissertation untersuchte Ansatz unterstützt in Abgrenzung zu Liu et al. ebenfalls den Prozess der Testfallerstellung durch Abstraktion auf Modellebene. Darüber hinaus adressieren die Autoren Fragestellungen kontextsensitiver Anwendungen nicht. Es wäre hier denkbar gewesen, neben dem UI-Ereignisstrom ebenfalls Kontextparameter (z. B. Ort, Zeit, Sensorwerte) im Ereignisprotokoll zu erfassen und bei der Wiederholung ebenfalls in das SUT zu injizieren. Hierzu wäre allerdings eine Simulationsumgebung notwendig, die in der Lage ist, Kontextparameter bei der Testdurchführung künstlich herzustellen. Neben der modellbasierten Generierung von Testfällen adressiert diese Dissertation eine solche Technologie ebenfalls.

In ihrer Arbeit aus dem Jahr 2015 befassen sich Vieira et al. [352] mit den Herausforderungen des Testens kontextsensitiver Anwendungen in einer nahezu mit den Inhalten dieser Dissertation identischen Ausrichtung. Die Autoren identifizieren die Bereitstellung eines testadäquaten Kontext als Kernherausforderung des Testens. Testen *in loco*, d. h. in realer Einbettung in das zukünftige Betriebsumfeld, ist aus wirtschaftlichen Gründen häufig nicht möglich oder ist aus konzeptuellen Gründen unmöglich. Vieira et al. argumentieren insbesondere, dass komplexe Kontexte Kompositionen aus atomaren (physikalischen) Kontextparametern sind (vgl. Kontextatome, Korpipää und Mäntyjärvi [229], Korpipää et al. [228], Baldauf et al. [25]), die unter wirtschaftlicher (und in einigen Fällen technischer) Perspektive nur beim Testen berücksichtigt werden können, wenn sie in Laborumgebungen simuliert werden können.

Auf dieser Annahme begründen die Autoren ihre untersuchte Forschungsfrage, wie komplexe Kontexte klassifiziert, modelliert und zum Testen simuliert werden können. Ähnlich dem in dieser Dissertation untersuchten Konzept zur Testautomatisierung entwerfen auch Vi-

eira et al. ein Konzept zur modellbasierten Simulation von Kontext bei der Testdurchführung. Zur Bereitstellung des Kontext bedienen sich die Autoren eines 4-lagigen Kontextmodells, in welchem Kontextinformationen von abstrakten Situationsbeschreibungen (Szenarien) bis zu Kontextatomen auf der Ebene von Sensordaten verfeinert werden. Dieses Kontextmodell war bereits Gegenstand ihrer Forschungsarbeiten aus dem Jahr 2011 [353], die in dieser Arbeit in Abschnitt 2.2.2 und Abschnitt 5.2.2 diskutiert werden. Zur Bereitstellung von Kontextinformationen im Testbetrieb evaluieren Vieira et al. drei Optionen: Sandboxing des SUT (d. h. Ausführung in einem Container, der den Kontext bereitstellt), Verwendung eines modifizierten Android-Systems und die Verwendung angepasster Android-SDK-Bibliotheken. In allen Fällen werden alle Funktionen, die Kontextparameter von Sensoren des Android-Betriebssystems ermitteln auf einen Kontextsimulator verwiesen.

Vieira et al. verwenden die dritte Option, Bereitstellung alternativer Android-SDK-Bibliotheken. Die Autoren begründen ihren Entschluss mit der Option, Teile des OpenIntents SensorSimulator™ [269] wiederverwenden zu können, erkennen aber an, dass hierdurch eine Quellcodeanpassung des SUT notwendig wird. In der Konsequenz ist das getestete Softwareartefakt ein Anderes als dasjenige, das mit Projektabschluss an den Kunden ausgeliefert wird. Eine Übertragbarkeit der Testergebnisse zwischen diesen beiden Softwareartefakten kann nicht gewährleistet werden. Darüber hinaus ist die Simulation einiger Kontextparameter mit diesem Vorgehen auf Emulatoren beschränkt, z. B. GPS oder Akkuladestand. In dem in dieser Dissertation untersuchten Konzept wird deshalb in Abgrenzung zu Vieira et al. [352] die Option des modifizierten Android-Systems verwendet, wo diese Einschränkungen nicht gelten. In weiterer Abgrenzung zum in dieser Dissertation untersuchten Konzept der Testautomatisierung ist die modellbasierte Generierung von Testfälle nicht Gegenstand der Arbeit von Vieira et al.

2.4 Generierung von Softwaretests

Die Erstellung von Softwaretests gehört zu den besonders aufwändigen und anspruchsvollen Aktivitäten in Softwareprozessen. Zudem ist Testen i. d. R. keine Aktivität, die unmittelbar zum Fortschritt eines Softwareprojekts beiträgt. Um Kosten für die Erstellung von Softwaretest zu reduzieren und gleichzeitig die Testqualität zu erhöhen, werden seit langer Zeit Technologien zur automatischen Generierung von Tests erforscht. Hierbei wurden eine Reihe unterschiedlicher Ansätze verfolgt, die beispielsweise auf DSLs oder Modellen von Software basieren. Ziel ist es jeweils, entweder bereits existierende Artefakte des Systementwurfs oder der Implementierung zur Generierung von Tests wiederzuverwenden oder spezifische Tests aus abstrakten Beschreibungen zu generieren.

Der Ansatz, Tests aus Entwurfsdokumenten eines Softwaresystems zu generieren, hat sich in der Wissenschaft als *Model Driven Testing* (MDT) etabliert. Ziel ist es, Tests aus denselben Systemmodellen zu erzeugen, aus denen ebenfalls die Implementierung abgeleitet wird, um die Kongruenz zwischen Systemspezifikation, Implementierung und Tests zu maximieren. Aufgrund der herausragenden Stellung der UML (vgl. Abschnitt 5.2.1) basieren MDT-Technologien i. d. R. auf UML-Modellen. Das in dieser Dissertation untersuchte Konzept zur Testautomatisierung mobiler, kontextsensitiver Anwendungen basiert ebenfalls auf UML-Modellen und ist deshalb in das Umfeld der UML-basierten MDT-Ansätze einzuordnen.

Apfelbaum und Doyle [11] befassten sich bereits im Jahr 1997 mit der Generierung von Tests aus Entwurfsmodellen von Softwaresystemen. Die Autoren argumentieren, dass die Modellierung eines Softwaresystems gegenüber alternativen Repräsentierungen einschlägige Vorteile bietet. Insbesondere sind Modelle aus der ökonomischen Perspektive vorteilhaft, weil sie das System auf einer Abstraktionsebene repräsentieren, die für viele Aktivitäten in Softwareprozessen einen adäquaten Nutzinhalt bietet und deshalb geeignet ist, über den gesamten Prozess hinweg wiederverwendet zu werden. Insbesondere für Aktivitäten der Qualitätssicherung, die in traditionellen Vorgehensmodellen i. d. R. spät im Prozess angesiedelt sind, stellen Modelle einen Fundus an Informationen über das erwartete Systemverhalten dar, die von Testern zur Erstellung von Tests verwendet werden können.

Die traditionelle Repräsentierung dieses Wissen in natürlicher Sprache hat hingegen den Nachteil, dass häufig nur typische Szenarien der Nutzung ausformuliert werden, untypische oder problematische Nutzungsszenarien hingegen ausgelassen oder nicht erkannt werden. Dieser Mangel existiert während des gesamten Prozesses und führt zu einer unvollständigen oder inkorrekten Implementierung. Insbesondere argumentieren Apfelbaum und Doyle, dass ebenfalls das Risiko besteht, dass diese Mängel auch während des Testens nicht erkannt werden, da durch eine unvollständige natürlichsprachliche Anforderungsspezifikation auch Tests nicht mit einer adäquaten Abdeckung entwickelt werden.

Modellbasiertes Vorgehen kann hier helfen, mögliche Lücken in der Systemspezifikation zu schließen und Ambiguitäten auszuschließen. Tests, die aus solchen Modellen generiert werden, haben das Potenzial das zu testende System in einem besseren Maß abzudecken, als solche, die manuell aufgrund einer textuellen Spezifikation erzeugt werden. Insbesondere kann basierend auf einem Modell eine automatische Testfallgenerierung erfolgen, die sicherstellt, dass während der Ausführung nur zulässige Sequenzen von Interaktionen mit dem SUT generiert werden. Insbesondere gewährleistet eine automatische Generierung von Tests aus Modellen eine Komplexitätsbeherrschung, die einem menschlichen Akteur unter Aufwendung kognitiver Leistung nur schwer möglich ist.

Cavarra et al. [74] untersuchten im Jahr 2002 einen Ansatz zur automatisierten Generierung von Testfällen aus UML-Modellen. Im untersuchten Konzept verwenden die Autoren Modelltransformationstechnologie, um UML-Modelle in Modelle zur Repräsentierung von Testfällen und Testdaten zu überführen. Zur Integration testrelevanter Daten in UML-Modelle entwerfen die Autoren ein UML-Profil, welches das Metamodell der UML erweitert.

Die Autoren fundieren ihren Ansatz auf der Annahme, dass Software schnell Komplexität erreicht, die von menschlichen Akteuren nicht mehr vollständig erfasst werden kann. Modellbildung soll hier helfen, vom technischem Detail zu abstrahieren und Komplexität zu reduzieren. Cavarra et al. argumentieren weiter, dass diese Notwendigkeit der Komplexitätsreduktion insbesondere im Umfeld des Testens notwendig ist, da der Zustandsraum möglicher Kombinationen von Eingaben in ein Softwaresystem rasch auf ein unüberschaubares Maß anwächst. Die Autoren argumentieren weiterhin, dass für den Prozess der Testgenerierung aus Modellen prinzipiell jede Modellierungssprache verwendet werden könnte. Entscheidendes Kriterium für die Akzeptanz beim Anwender ist jedoch, dass zum Erlernen der Modellierungssprache kein zusätzlicher Aufwand entsteht. Da es sich bei der UML um ein Standardwerkzeug der Softwaretechnik handelt, verwenden Cavarra et al. sie als Grundlage ihrer Forschungsarbeiten.

Während zwischen dem von Cavarra et al. untersuchten Konzept zur Testgenerierung und dem in dieser Dissertation untersuchten zahlreiche Parallelen existieren, insbesondere in der Modellierung von Testdaten als UML-Objektdiagramme zur Sicherstellung einer adäquaten Abdeckung des SUT, existieren ebenfalls signifikante Unterschiede, die eine direkte Wiederverwendung der von Cavarra et al. untersuchten Technologie zur Generierung von Tests für mobile, kontextsensitive Anwendungen ausschließen. Zum einen verwenden die Autoren UML-Klassen- und Zustandsdiagramme als Basis für die Testgenerierung. Im Vergleich zu UML-Aktivitätsdiagrammen sind diese Diagramme hinsichtlich der Darstellung des dynamischen Systemverhaltens unterlegen. Die in dieser Dissertation als Grundlage verwendeten UML-Aktivitätsdiagramme bilden das Verhalten des zukünftigen Anwenders auf einer beliebigen Abstraktionsebene ab und erfüllen daher eher die Anforderungen an Akzeptanztests, die das Verhalten des Anwenders in einem kontrollierten Versuchsaufbau simulieren. Weiterhin ist das von Cavarra et al. entworfene UML-Profil nicht in der Lage, den Kontext des SUT neben sonstigen Testdaten abzubilden.

Zu den Gemeinsamkeiten gehört hingegen die Verwendung der UML als Grundlage der Modellierung, sowie der Ansatz, UML-Modelle zunächst in ein technologieunspezifisches Testfallmodell zu transformieren, welches seinerseits die Grundlage zur Erzeugung technologiespezifischer Tests bildet. So ergibt sich der Vorteil, die tatsächlich verwendete Testautomatisierungstechnologie erst spät im Prozess festlegen zu müssen. Weiterhin entwerfen die Autoren ein UML-Profil, welches es erlaubt, testrelevante Attribute eines Klassendiagramms explizit hervorzuheben und somit relevante Testdaten von nichtrelevanten zu trennen. Dieser Ansatz wird auch in dieser Dissertation verfolgt, allerdings basierend auf UML-Aktivitätsdiagrammen, in denen nicht individuelle Attribute einer Komponente als testrelevant gekennzeichnet werden, sondern Aktionen und Aktivitäten in Aktivitätsdiagrammen. Auch liegt die Annahme zugrunde, dass nicht alle Aktivitäten in Aktivitätsdiagrammen testrelevant sind, beispielsweise weil sie keine Schnittstelle zum Anwender implementieren.

Das Konzept des MDT umfasst nach Heckel und Lohmann [173] drei wesentliche Aufgaben: (i) Die Generierung von Testfällen aus Modellen, (ii) die Generierung eines Orakels zur Bestimmung erwarteter Resultate und (iii) die Ausführung von Tests in entsprechenden Testumgebungen. Heckel und Lohmann [173] beschreiben die Aufgaben i und ii als prinzipiell plattformunabhängig. Aufgabe iii ist spezifisch für die Zielplattform. Kernaufgabe des MDT ist die Automatisierung der Aufgaben i bis iii.

Mit dem Fokus auf Web-Anwendungen propagieren Heckel und Lohmann die Verwendung von Modellen als Orakel. Diese dienen dem Vergleich der tatsächlich gemessenen Testresultate mit den jeweilig zughörigen Erwartungswerten. Heckel und Lohmann [173] identifizieren UML-Modelle als ungeeignet für diese Aufgabe, da insbesondere statische Systemmodelle (wie etwa UML-Klassendiagramme) nicht abbilden können, wie Klasseninstanzen durch die Ausführung von Funktionen des zu testenden Systems beeinflusst werden.

Die Autoren entwerfen eine auf Modelltransformation basierende Technologie, durch welche sowohl die Testumgebung als auch eine technische Implementierung eines Orakels mit demselben UML-Modell der zu testenden Anwendung initialisiert werden. Dieses Modell repräsentiert das zu testende System, insbesondere Eingabe- und Ausgabeschnittstellen, die während der Testdurchführung mit Testdaten gespeist werden. Ausgabedaten des zu test-

enden Systems werden mit jenen verglichen, die das Orakel erzeugt hat. Stimmen die tatsächlich durch das System erzeugten Ausgabedaten nicht mit den Orakel-Daten überein, ist das ein Indikator für einen möglichen Defekt im zu testenden System.

Die Relevanz der Arbeit von Heckel und Lohmann [173] für diese Dissertation liegt in dem gemeinsam verwendeten Konzept, Systemmodelle für die Generierung von Testfällen heranzuziehen. Ebenfalls ist das in dieser Dissertation verwendete Metamodell zur Testfallmodellierung in Übereinstimmung mit Aufgabe ii zunächst technologieunabhängig. Hieraus ergibt sich unmittelbar der Vorteil, dass Testfälle für mobile Anwendungen, die mehrere Plattformen bedienen sollen, nicht mehrfach modelliert werden müssen.

Im Jahr 2003 befasste sich Robinson [299] mit den Herausforderungen und Chancen der modellbasierten Testgenerierung. Der Autor fundiert seine Argumentation für die Generierung von Tests aus Modellen anstelle der manuellen Erstellung mit der Beobachtung, dass Tester häufig weniger technisch orientiert sind als Softwareentwickler und zudem erst verhältnismäßig spät im Softwareprozess involviert sind. Mangelnde technische Einbettung in die zu testende Software und ein geringer Bezug zur Projekthistorie setzen die Qualität von Tests herab. Ein modellbasiertes Vorgehen kann auch weniger technisch versierten Testern dabei helfen, zügig ein abstraktes Verständnis des SUT zu entwicklen und die Testentwicklung daraufhin besser auf wesentliche Produktmerkmale zu fokussieren. Grundlegende Idee ist hier, Tests nicht länger als parallel zum Softwareprodukt entwickelte Artefakte zu begreifen, sondern als ausführbare Spezifikation.

Der Autor argumentiert, dass natürlichsprachliche Spezifikationsdokumente für eine effektive Testfallerstellung häufig zu unpräzise sind, um eine adäquate Testabdeckung zu gewährleisten. Weiterhin diskutiert Robinson die Vorteile der modellbasierten Testgenerierung hinsichtlich der Anzahl individueller Testfälle, die aus Managementperspektive als Metrik der Produktivität herangezogen wird. Für komplexe Software entsteht schnell eine hohe Anzahl von Tests. Eine Änderung der Spezifikation des Softwareprodukts kann einen Teil oder alle dieser Tests obsolet machen und das Erstellen neuer Tests erforderlich machen. Eine manuelle Erstellung erfordert hier u. U. einen hohen Arbeitsaufwand zur Revision existierender Tests, während sich der Aufwand bei einer Generierung von Tests aus einem Modell auf die Anpassung des Modells reduziert.

Das in dieser Dissertation untersuchte Konzept der Testautomatiserung adressiert genau diese Herausforderungen durch die in Kapitel 5 diskutierte Methode zur Generierung von Tests aus Modellen. Insbesondere das Vorgehen, aus einem Systemmodell zunächst ein plattformunabhängiges Testmodell zu erzeugen und dieses erst in einem späteren Schritt zu technologispezifischen Tests zu transformieren adressiert die von Robinson geforderte Separierung von Modell und Technologie.

Ebenfalls in Übereinstimmung mit der in einer Fallstudie erworbenen Erkenntnis von Robinson ist die hier untersuchte Testautomatisierungstechnologie so angelegt, dass Tests nach ihrer Generierung zu einer wiederholten Ausführung persistiert werden, anstelle einer *On-the-Fly* Ausführung von Tests auf Grundlage des Modells (also Generierung und Ausführung in einem Arbeitsgang), wodurch die Identifikation von Fehlern in den Tests selbst erschwert würde. Allerdings argumentiert Robinson ebenfalls, dass durch die Verschiebung des Fokus der Testerstellung von traditionellen zu modellbasierten Methoden ebenfalls andere Anforderungen an

Tester entstehen. Die Generierung von Tests aus Modellen erfordert eine Auseinandersetzung mit dem SUT auf einer höheren Abstraktionsebene und eine frühere Integration von Testern in den Softwareprozess. Der in dieser Dissertation untersuchte Ansatz zur Testautomatisierung entkräftet dieses Argument nicht. Allerdings wird eine frühe und intensive Einbettung von Testern in den Softwareprozess im Licht fortschreitend agiler werdender Vorgehensmodelle als vorteilhaft eingeschätzt.

Diese Dissertation untersucht einen modellbasierten Ansatz zur automatisierten Generierung von Softwaretests. Dieses Vorgehen setzt grundsätzlich die Existenz von Modellen des zu testenden Systems voraus. Hartman et al. [171] unterscheiden hier den Begriff des modellbasierten Testens explizit von Technologien des *Model Driven Testing* (MDT). Nach der Argumentation der Autoren ist nur bei Letzterem eine Manifestation des Softwaremodells als ein technisches Artefakt obligatorisch, während beim modellbasierten Testen auch mentale Modelle des SUT zulässig sind (ähnliche Argumentation bei Apfelbaum und Doyle [11]).

Dem in dieser Dissertation untersuchten Konzept der Testautomatisierung liegt das MDT-Konzept zugrunde. Es müssen zwingend maschinenlesbare Artefakte vorhanden sein, in denen sich das Modell des SUT manifestiert. Weiterhin argumentieren Hartman et al., dass MDSD- und MDT-Technologien im Jahr 2007 ausreichend fortgeschritten sind, dass Modelle dem Quellcode als Artefakt der Softwareentwicklung gleichgestellt sind. Der Ablauf von MDT-Prozessen kann zwar im Detail variieren, folgt jedoch immer dem grundsätzlichen Prinzip, dass zunächst das Verhalten des SUT auf einer geeigneten Abstraktionsebene modelliert werden muss. Hierbei entsteht ein Testmodell des SUT, das im Anschluss zu einer Menge von Testfälle transformiert wird. Im Anschluss an diese Transformation steht die i. d. R. automatisierte Ausführung der Tests und die Bewertung der Ergebnisse.

Hartman et al. argumentieren weiterhin, dass in MDSD- und MDT-Prozessen Entwickler und Tester zwar beide auf die Verwendung von Modellen angewiesen sind, aber unterschiedliche Anforderungen an Modelle stellen, da sie das System aus unterschiedlichen Perspektiven beurteilen. Entwickler benötigen Plattformdetails und Richtlinien zur Implementierung, während Tester den Fokus vorrangig auf die Aspekte Benutzbarkeit und Systemgrenzen legen. Durch diese unterschiedlichen Perspektiven kann die Verwendung unterschiedlicher Modellierungssprachen gerechtfertigt sein, etwa die Verwendung generischer Modellierungssprachen wie die UML auf Seiten der Entwickler gegenüber einer DSL für die Modellierung seitens der Tester. Diese Diskrepanz hat das Potenzial, den Softwareprozess durch Einführung und Verwendung mehrere Modellierungssprachen und -Werkzeuge zu fragmentieren.

Es werden jedoch Modelle benötigt, die das Systemverhalten in geeigneter Weise abbilden. Diese Modelle haben das Ziel, eine Menge von Stimuli zu beschreiben, die auf ein System einwirken können und die vom System erwartete Reaktion. Modelle, die diese Reaktion nicht in vollem Umfang beinhalten können dennoch für die Generierung von Tests verwendet werden, machen es aber erforderlich, dass diese Informationen zu einem späteren Zeitpunkt integriert werden. Das in dieser Dissertation untersuchte Konzept zur Testautomatisierung verfolgt diesen Ansatz, indem ein zunächst nicht auf das Testen fokussierte Systemmodell einer Anwendung durch Anreicherung mit testspezifischen Informationen zu einem plattformunabhängigen Testmodell aufgewertet wird, bevor in einem weiteren Schritt plattform- und technologiespezifische Tests generiert werden.

Hartman et al. diskutieren weiterhin Vor- und Nachteile der Verwendung generischer Modellierungssprachen zur Modellierung von System und Tests im Vergleich zur separaten Modellierung dieser Artefakte mit unterschiedlichen Sprachen. Als Nachteil separater Modellierungssprachen kann angeführt werden, dass das zu testende System hier u. U. zweimal modelliert wird, wodurch nicht nur ein erhöhter Aufwand entsteht, sondern auch zusätzlicher Raum für Fehler geschaffen wird, wenn Systemmodell und Testmodell die Anforderungen unterschiedlich interpretieren und in der Konsequenz inhaltlich nicht kongruent sind.

Thummalapenta et al. [344] befassten sich 2012 ebenfalls mit der Automatisierung der Erstellung von Testfällen. Hintergrund ist die Tatsache, dass die Erstellung von Tests eine zeitaufwändige und damit teure Aufgabe ist und insbesondere Systemtests und Akzeptanztests zu Artefakten gehören, deren Erstellung nicht ohnehin zum Aufgabenspektrum des Programmierens gehört, wie es z. B. bei Unit-Tests der Fall ist. Systemtests und Akzeptanztests werden i. d. R. von Stakeholdern erstellt, die nicht zwingend selbst Entwickler sind und für die das Erstellen automatisierbarer Tests deshalb auch eine technische Hürde ist.

Thummalapenta et al. entwickeln deshalb eine Technologie, die automatisierbare Tests aus einer natürlichsprachlichen Testbeschreibung generieren. Ziel dieser Generierung ist hierbei nicht die Identifikation zu testender Komponenten oder Funktionen, sondern die Erzeugung von Testskripten in einer Programmiersprache, die von einer Testautomatisierungstechnologie ausgeführt werden kann. Die Autoren stützen sich hierbei auf die empirisch nachgewiesene Vermutung, dass natürlichsprachliche Testbeschreibungen i. d. R. einer vorhersehbaren Struktur folgen und ein eingeschränktes Vokabular verwenden. Einen vergleichbaren Ansatz folgt die in Abschnitt 2.5.2.6 diskutierte Automatisierungstechnologie Calabash, die Testfälle ebenfalls in einer an die natürliche Sprache angelehnten Syntax beschreibt. Das Ziel dieser Dissertation ist es, auch den Prozess der Testfallerstellung zu unterstützen, indem Entwurfsmodelle eines Softwaresystems zur Generierung von Tests wiederverwendet werden.

Zusammen mit der Generierung von Testfällen ist die Generierung von Testdaten ein wesentliches Problem der automatisierten Erstellung von Tests. Mirzaei et al. [252] untersuchen im Jahr 2012 einen Ansatz, wie durch symbolische Programmausführung sinnvolle Testdaten für das Testen von Android-Apps ermittelt werden können. Das Konzept der Autoren sieht vor, durch Verwendung einer angepassten Variante des Werkzeugs Symbolic PathFinder [272] den Aufrufgraphen des SUT zu ermitteln. Da Android Apps lose gekoppelte Aggregate aus Komponenten sind, stellt sich hier das Problem, dass kein zusammenhängender Graph ermittelt werden kann, sondern zu jeder Anwendungskomponente ein individueller Aufrufgraph erzeugt wird. Die Autoren behelfen sich durch Analyse des Quellcodes, um die einzelnen Graphen miteinander zu verbinden. Im Anschluss erzeugen die Autoren Sequenzen von Ereignissen oder Aufrufen, die einzelne Pfade durch den Aufrufgraphen repräsentieren. Diese enthalten ggf. parameterbehaftete Methodenaufrufe, für welche anstelle konkreter Werte jeweils Symbole bei der symbolischen Programmausführung verwendet werden. Die eigentliche Testdatengenerierung erfolgt durch Analyse des Byte-Codes auf Verzweigungen. Durch Verwendung eines Constraint-Lösers (engl. *Constraint Solver*) können dann für die symbolischen Werte konkrete Werte ermittelt werden, aus denen Testdaten erzeugt werden können.

Diese Dissertation verfolgt nicht das Ziel, aus Systemmodellen Testdaten zu erzeugen. Vielmehr steht hier die Generierung von Testfällen im Fokus. Die Relevanz der Arbeit von

Mirzaei et al. für diese Dissertation ergibt sich jedoch aus dem Ansatz der Autoren, Sequenzen von Ereignissen oder Aufrufen aus dem Code der Anwendung zu generieren. Nach Abschluss der Analyse verwenden die Autoren eine kontextfreie Grammatik, um solche Aufrufsequenzen zu erzeugen, die ein definiertes Abdeckungskriterium (z. B. Pfadabdeckung, Bedingungsabdeckung, Zweigabdeckung) erfüllen. Diese Überlegungen sind auch für das in dieser Dissertation untersuchte Konzept zur Testfallgenerierung interessant, da hier ebenfalls die Anforderung besteht, Tests zu generieren, die definierte Abdeckungskriterien erfüllen (vgl. Abschnitt 5.3.2).

2.5 Technologisches Umfeld der Testautomatisierung mobiler, kontextsensitiver Softwaresysteme

In der Softwaretechnik ist die Automatisierung von Tests seit langer Zeit ein Thema, dem sowohl akademische als auch industrielle Aufmerksamkeit gewidmet wird. Auch im Umfeld mobiler Plattformen hat sich eine Reihe von Technologien und Werkzeugen herausgebildet, Testaktivitäten durch Ablösung menschlicher Arbeitskraft wirtschaftlicher zu gestalten. Einige dieser Technologien wurden speziell für mobile Plattformen entwickelt, während andere ihren Ursprung in der traditionellen Softwareentwicklung haben und sie lediglich für die Verwendung im mobilen Umfeld angepasst wurden.

Allen im Jahr 2016 existierenden Werkzeugen ist gemeinsam, dass sie insbesondere die Anforderungen mobiler, kontextsensitiver Anwendungen nicht oder nur rudimentär erfüllen, so dass für das Testen dieser Anwendungen weiterhin ein hoher manueller Arbeitsaufwand notwendig ist.

In den folgenden Abschnitten werden existierende Technologien zur Unterstützung und Automatisierung von Testprozessen diskutiert. Diese sind grundsätzlich nach Teststrategie, Testtyp und Teststufe zu unterscheiden. Je nach Profilierung einer Testtechnologien hinsichtlich dieser Kriterien sind die Voraussetzungen zur Testautomatisierung für kontextsensitive mobile Anwendungen in unterschiedlichem Maß erfüllt.

2.5.1 Technologien für Komponententests, Stresstests und Crashtests

Unterschieden werden grundsätzlich Werkzeuge zur Durchführung von Komponententests, Stresstests und funktionalen Tests. Komponententests haben klar und eng abgegrenzte Softwareartefakte zum Gegenstand wie etwa einzelne Klassen oder deren Methoden. Stresstests, insbesondere ihre Sonderform Crashtests hingegen prüfen i. d. R. Gesamtsysteme auf Robustheit, können aber auch auf einzelne Komponenten angewendet werden.

2.5.1.1 JUnit / CUnit

Als Werkzeug des Komponententestens ist JUnit/*CUnit* [326, 34] bereits seit einigen Jahren ein etablierter Standard (vgl. Link [238]). Nah am Quellcode des SUT ausgerichtet werden Technologien der *x*Unit-Familie dazu verwendet, die Implementierung von Testfällen zusammen mit der Implementierung einer Software durchzuführen. Hierzu stellt JUnit (analog andere Vertreter der *x*Unit-Familie) ein Framework bereit, vorbereitende Aktivitäten des Testens,

die eigentliche Testdurchführung und nachbereitende Aktivitäten strukturiert und automatisiert durchzuführen. Es ist eine Frage der individuellen Testfallimplementierung inwieweit ein Testfall tatsächlich einen Komponententest repräsentiert oder bereits in Richtung Integrationstest abweicht. Die Technologie selbst setzt hier zunächst keine Grenzen, wenngleich die ursprüngliche Intention das Testen konkret abgrenzbarer Quellcodeabschnitte ist.

Hierzu bietet das JUnit-Framework die Möglichkeit, an bestimmten Stellen im Quellcode Methoden als Testmethoden auszuzeichnen. Dies kann innerhalb einer funktionalen Klasse des SUT geschehen oder in ausgelagerten Testklassen. Die Auszeichnung einer Methode erfolgt programmiersprachenspezifisch. In der Programmiersprache Java beispielsweise werden Annotationen der Form „*@Test*“ der Signatur einer Methode vorangestellt. An dieser Annotation erkennt das Framework, dass diese Methode bei der Testdurchführung auszuführen ist. Analog können Methoden annotiert werden, um Vor- bzw. Nachbedingungen herzustellen, um das SUT vor dem Ausführen eines Tests in einen definierten Zustand zu versetzen.

Quellcodefragment 2.1: Beispiel eines JUnit-Test [82]

```
1  import static org.junit.Assert.assertEquals;
2  import org.junit.Test;
3
4  public class CalculatorTest {
5    @Test
6    public void evaluatesExpression() {
7      Calculator calculator = new Calculator();
8      int sum = calculator.evaluate("1+2+3");
9      assertEquals(6, sum);
10   }
11 }
```

In Quellcodefragment 2.1 ist ein Beispiel eines JUnit-Tests abgebildet. Zu erkennen ist die Annotation „*@Test*“ in Zeile fünf, die die folgende Methode als Test kennzeichnet. Zeilen sieben und acht instantiieren die zu testende Klasse und führen auf dieser Instanz eine Methode aus. Das Ergebnis wird in einer Variablen gespeichert und in Zeile neun durch eine JUnit-Assertion mit einem Vorgabewert verglichen. Der Test ist erfolgreich, wenn diese Assertion positiv ausgewertet wird.

Es existieren JUnit-Derivate für mobile Plattformen. Für die Plattform Android kann die Standardvariante von JUnit verwendet werden, für Apple iOS existiert eine Variante die spezifisch auf die Programmiersprache Objective-C angepasst ist. Diese JUnit-Derivate haben allerdings ausschließlich zum Gegenstand, die Technologie prinzipiell für mobile Plattformen verfügbar zu machen. Die Besonderheiten des mobilen Testens und Aspekte der Mobilität und der Kontextsensitivität werden durch diese Technologien nicht berücksichtigt. So werden beispielsweise durch das Framework keine Methoden zur Verfügung gestellt, die Standortinformation oder Sensormesswerte zu Testzwecken generieren können.

Darüber hinaus arbeiten JUnit-artige Technologie immer innerhalb des SUT. Einige Aspekte des Testens mobiler Systeme erfordern jedoch Interaktionen außerhalb des SUT. Beispielsweise ist die Installation bzw. die Deinstallation oder auch das Zurücksetzen aller Daten ggf. Inhalt eines konkreten Testfalls. Solche Aktivitäten können regelmäßig nicht von SUT-lokalen Testfällen realisiert werden. Das heißt, eine mobile App kann sich nicht selbst auf einem mobilen Gerät installieren, da die Ausführung des für die Installation zuständigen Codes bereits eine funktionierende Installation voraussetzt. Um solche Testfälle abzudecken, bedarf es Methoden und Werkzeugen, die funktionale Testinhalte auch außerhalb des SUT erfüllen können.

Für diese Dissertation ist JUnit von Bedeutung, weil es die Basis für fortschrittlichere Testwerkzeuge wie beispielsweise Calabash (vgl. Abschnitt 2.5.2.6 und Abschnitt 5.5.3) bildet. Nahezu alle Testwerkzeuge auf höheren Teststufen führen die Testausführung auf Tests auf der Ebene von Komponententests zurück. Zur direkten Verwendung zum Testen kontextsensitiver, mobiler Anwendungen sind JUnit-artige Technologien nicht geeignet.

Einerseits sind Tests so nah am Quellcode, dass Parameter der Betriebsumgebung nicht durch den Test selbst beeinflusst werden können. Andererseits sind Komponententest zum Testen auf funktionaler Ebene nicht geeignet. Daher haben JUnit-artige Technologien zwar auch im mobilen Umfeld Anwendungsgebiete, die direkte Implementierung funktional orientierter Tests gehört jedoch nicht dazu.

2.5.1.2 UI/Application Exerciser Monkey

Zur Durchführung von Stresstests existieren ebenfalls Werkzeuge, deren Aufgabe es ist, das SUT so mit simulierten Interaktionen, Funktionsaufrufen oder anderen Stimuli zu versorgen, dass nichtfunktionale Anforderungen, wie etwa Robustheit, verletzt werden.

Hierzu gehört die Android-spezifische Technologie UI/Application Exerciser Monkey. Es handelt sich hierbei um ein Werkzeug, das in der Lage ist, einen pseudo-zufälligen Strom von Anwenderinteraktionen (z. B. Anklicken von Schaltflächen, Touch-Gesten) oder Interaktionen mit dem Gerät (z. B. Navigation auf den *Homescreen*) zu erzeugen. Es soll überprüft werden, ob eine Anwendung robust ist (d. h. nicht unerwartet terminiert) gegenüber außerplanmäßigen Interaktionen, d. h. solchen, die in keinem angenommenen Anwendungsfall vorgesehen sind.

Der UI/Application Exerciser Monkey kann auf einem Emulator oder auf einem Gerät ausgeführt werden. Der Tester spezifiziert die Länge des pseudo-zufälligen Interaktionsstroms und kann diesen zusätzlich profilieren. So kann beispielsweise vorgegeben werden, dass ein bestimmter prozentualen Anteil an Touch-Gesten enthalten ist. Ebenfalls ist es möglich, einen Interaktionsstrom zu reproduzieren. So kann überprüft werden, ob eine Interaktionsabfolge, die einen Programmabsturz verursacht hat, nach einer Anpassung des SUT fehlerfrei ausgeführt wird. Ein Beispiel ist in Quellcodefragment 2.2 dargestellt.

Quellcodefragment 2.2: Beispiel eines UI/Application Exerciser Monkey Test [137]

```
1 $ adb shell monkey -p your.package.name -v 500
```

Inhaltlich-funktionale Aspekte können durch solche Tests nicht abgedeckt werden. Es besteht keine Möglichkeit, den UI/Application Exerciser Monkey so zu konfigurieren, dass eine im Sinne eines potenziellen Anwendungsfalls sinnvolle Interaktionsabfolge generiert wird. Ebenso ist es nicht möglich, Eigenschaften der Betriebsumgebung zu berücksichtigen. Der UI/Application Exerciser Monkey ist nicht in der Lage, die Verfügbarkeit oder Qualität von Netzwerkverbindungen, Standort- oder Sensorinformationen zu berücksichtigen. So kann beispielsweise kein Stresstest konfiguriert werden, der die Robustheit einer Anwendung gegenüber repetetiven Statusänderungen von Netzwerkverbindungen oder Standortinformationen überprüft. Beide Szenarien sind valide Testinhalte, da diese Informationen gerade in mobilen Betriebsumgebungen volatil sind.

2.5.2 Technologien für funktionale Tests

Komponententests stehen funktionale Tests gegenüber. Sie haben nicht die korrekte Implementierung eines Quellcodefragments zum Gegenstand, sondern die Überprüfung der Konformität einer Implementierung einer Anforderung gegenüber ihrer Spezifikation. Auch hier haben sich zahlreiche Technologien etabliert, die auch im Umfeld mobiler Anwendungen einsetzbar sind. Einige Werkzeuge und Frameworks, die auch auf der Plattform Android angewendet werden können, werden in den folgenden Abschnitten erläutert.

2.5.2.1 Robotium

Robotium [300] ist eine Erweiterung des Android-Test-Frameworks mit dem Ziel der Vereinfachung von Tests, die Elemente des UI manipulieren (UI-Test). Die Technologie erweitert das JUnit-Konzept insofern, als dass sie Tests ermöglicht, die nicht mehr nur ein einzelnes Softwareartefakt zum Gegenstand haben. Vielmehr ist es mit Robotium möglich Tests zu implementieren, die sich über mehrere Komponenten einer Android-App erstreckten. Quellcodefragment 2.3 illustriert ein Beispiel.

Quellcodefragment 2.3: Beispiel eines Robotium Test [300]

```
1  public class EditorTest extends
2          ActivityInstrumentationTestCase2<EditorActivity> {
3
4    private Solo solo;
5
6    public EditorTest() {
7          super(EditorActivity.class);
8    }
9
10   public void setUp() throws Exception {
11     solo = new Solo(getInstrumentation(), getActivity());
12   }
13
14   public void testPreferenceIsSaved() throws Exception {
15
16         solo.clickOnText("More");
17         solo.clickOnText("Preferences");
18         solo.clickOnText("Edit␣File␣Extensions");
19         Assert.assertTrue(solo.searchText("rtf"));
20
21   }
22
23    @Override
24    public void tearDown() throws Exception {
25     solo.finishOpenedActivities();
26   }
27 }
```

In Abgrenzung zu standard JUnit ermöglich Robotium ein direktes Adressieren von UI-Elementen durch die Klasse *Solo*. Im Codebeispiel werden UI-Elemente anhand ihrer Beschriftung adressiert. Diese Funktionalität wird von Robotium bereitgestellt.

Robotium stellt hingegen keine Funktionen bereit, mit denen der Kontext der Anwendungsausführung manipuliert werden kann, so dass beispielsweise Tests für ortsbasierte Anwendungen nicht realisierbar sind. Hier bestünde zwar grundsätzlich die Option, eine Unterstützung für Tests ortsbasierter Apps zu integrieren, allerdings nur, weil die Plattform das Vortäuschen von Standortinformationen unter bestimmten Umständen im Rahmen der Testwerkzeuge der Plattform erlaubt. Für andere Kontextparameter wie etwa Sensordaten existiert diese Option jedoch nicht. Deshalb wird Robotium als untauglich zum Testen mobiler, kontextsensitiver Anwendungen eingeschätzt.

2.5.2.2 Espressso

Eine ähnliche Funktionalität wie das Framework Robotium wird durch das Framework Espresso [136] angeboten. Auch hier liegt der Fokus auf UI-Tests mit dem Anspruch, den Zugriff auf UI-Elemente im Quellcode des Tests besonders einfach und komfortabel und somit weniger anfällig für Fehler zu gestalten. In Quellcodefragment 2.4 ist ein Beispiel dargestellt.

Quellcodefragment 2.4: Beispiel eines Espresso Test [136]

```
1 @Test
2 public void greeterSaysHello() {
3   onView(withId(R.id.name_field))
4     .perform(typeText("Steve"));
5   onView(withId(R.id.greet_button))
6     .perform(click());
7   onView(withText("Hello␣Steve!"))
8     .check(matches(isDisplayed()));
9 }
```

Darüber hinaus bietet Espresso keine weiteren funktionalen Vorteile gegenüber JUnit für Android. Insbesondere definiert Espresso keine Methoden oder Schnittstellen zur Simulation von Kontextparametern und ist somit für im Hinblick auf die besonderen Anforderungen an das Testen mobiler, kontextsensitiver Anwendungen ungeeignet.

2.5.2.3 UIAutomator

Eine weiteres Framework zur Testautomatisierung aus dem Umfeld der JUnit-Familie ist die Technologie *UIAutomator*. Grundsätzlich gelten hier die selben Einschränkungen wie für andere JUnit-basierten Technologien. Insbesondere gilt auch hier, dass Tests innerhalb des Anwendungspakets des SUT ausgeführt werden (vgl. Abschnitt 4.3.3, Abbildung 4.8). Ein Test verfügt deshalb über keine anderen Berechtigungen als die zu testende Anwendung, wodurch einige Manipulationen des Geräts, des Emulators und insbesondere des Kontext unmöglich werden, beispielsweise eine Simulation unterschiedlicher Zustände der Netzwerkadapter (vgl.Abschnitt 3.1.2.2.2), von Sensordaten (vgl. Abschnitt 3.1.2.1.5) oder Datum und Uhrzeit des Systems (vgl. Abschnitt 3.1.2.1.2).

Der UIAutomator nimmt unter den JUnit-basierten Technologien dennoch eine Sonderrolle ein, da Tests hier bereits vor dem Einstiegspunkt in die zu testende App beginnen. Das heißt, sie haben vom Grundsatz her Zugriff auf Komponenten und UI-Elemente außerhalb des SUT. Während andere JUnit-Technologien lediglich auf Komponenten und UI-Elemente innerhalb des SUT zugreifen können, kann der UIAutomator-Test beispielsweise den Android-Home-Screen als Einstiegspunkt in die App definieren, einschließlich einer Navigation zur richtigen Seite des Home-Screen. Hierdurch ist es beispielsweise möglich zu testen, ob eine App das korrekte Launch-Icon präsentiert.

Darüber hinaus ermöglicht es die Technologie UIAutomator ebenfalls andere Apps in einen Test einzubeziehen. Ein UIAutomator-Test kann etwa die Systemeinstellungen aufsuchen, eine spezifische Einstellungsseite des SUT aufsuchen, um dort beispielsweise App-Einstellungen, die sogenannten *Preferences*, oder den Inhalt des Caches zu löschen. Diese Maßnahme kann in Tests zur Herstellung spezifischer Vorbedingungen zweckmäßig sein.

Dargestellt in Quellcodefragment 2.5 ist ein Beispiel eines UIAutomator-Test, der auf dem Home-Screen beginnt und von dort aus das SUT startet.

Quellcodefragment 2.5: Beispiel eines UIAutomator-Test [138]

```
@RunWith(AndroidJUnit4.class)
@SdkSuppress(minSdkVersion = 18)
public class ChangeTextBehaviorTest {

    private static final String BASIC_SAMPLE_PACKAGE
            = "com.example.android.testing.uiautomator.BasicSample";
    private static final int LAUNCH_TIMEOUT = 5000;
    private static final String STRING_TO_BE_TYPED = "UiAutomator";
    private UiDevice mDevice;

    @Before
    public void startMainActivityFromHomeScreen() {
        // Initialize UiDevice instance
        mDevice = UiDevice.getInstance(InstrumentationRegistry.↙
            ↳ getInstrumentation());

        // Start from the home screen
        mDevice.pressHome();

        // Wait for launcher
        final String launcherPackage = mDevice.getLauncherPackageName();
        assertThat(launcherPackage, notNullValue());
        mDevice.wait(Until.hasObject(By.pkg(launcherPackage).depth(0)),
                LAUNCH_TIMEOUT);

        // Launch the app
        Context context = InstrumentationRegistry.getContext();
        final Intent intent = context.getPackageManager()
                .getLaunchIntentForPackage(BASIC_SAMPLE_PACKAGE);
        // Clear out any previous instances
        intent.addFlags(Intent.FLAG_ACTIVITY_CLEAR_TASK);
        context.startActivity(intent);

        // Wait for the app to appear
        mDevice.wait(Until.hasObject(By.pkg(BASIC_SAMPLE_PACKAGE).depth(0)),
                LAUNCH_TIMEOUT);
    }
}
```

Zum Testen mobiler, kontextsensitiver Anwendungen ist die Technologie UIAutomator aufgrund mangelnder Schnittstellen zur Simulation von Kontext ungeeignet.

2.5.2.4 Monkeyrunner

Für die Plattform Android stellt der Betreiber das Werkzeug *Monkeyrunner* [141] zur Verfügung. Es handelt sich hierbei um eine Python-basierte Technologie, die eine API zur Erstellung von Tests und zur Steuerung eines Android Geräts und dem darauf laufenden SUT implementiert. Trotz der Namesnähnlichkeit hat der Monkeyrunner keine inhaltliche Relation zum UI/Application Exerciser Monkey. Der Monkeyrunner zeichnet sich gegenüber der Komponententestautomatisierungstechnologie JUnit für Android darin aus, dass Testfälle nicht ausschließlich aus dem Inneren des SUT gesteuert werden, sondern die Steuerung von außerhalb erfolgt. Hierdurch kann in der Testfallspezifikation eingeschränkt Einfluss auf die Betriebsumgebung des SUT genommen werden. Es ermöglicht beispielsweise die (De-)Installation des SUT auf einem Android Gerät oder einem Emulator. Dies ermöglicht die Spezifikation solcher Testfälle, die bereits vor der eigentlichen Inbetriebnahme des SUT ansetzen. Dies ist ein bedeutender Faktor der Spezifikation von Vorbedingungen, da so etwa die Erstinbetriebnahme einer App im Testfall berücksichtigt werden kann. In Quellcodefragment 2.6 ist beispielhaft ein Ausschnitt aus einem Monkeyrunner-Test abgebildet.

Darüber hinaus können mit dem Monkeyrunner funktionale Testfälle spezifiziert werden, die anzunehmende Szenarien der zukünftigen Verwendung der SUT aus Sicht des Anwenders repräsentieren. Inhalte von Testfällen können beispielsweise Installationsanweisungen, simulierte Interaktionen mit UI-Elementen oder das Anfertigen von Screenshots sein. Insbesondere

diese können außerhalb des SUT auf dem Emtwicklungscomputer gespeichert werden und bilden u. U. wichtige Artefakte der Beurteilung des SUT in semi-automatischen Tests.

Quellcodefragment 2.6: Beispiel eines Monkeyrunner-Test [141]

```
1  from com.android.monkeyrunner import MonkeyRunner, MonkeyDevice
2  device = MonkeyRunner.waitForConnection()
3  device.installPackage('myproject/bin/MyApplication.apk')
4  package = 'com.example.android.myapplication'
5  activity = 'com.example.android.myapplication.MainActivity'
6  runComponent = package + '/' + activity
7  device.startActivity(component=runComponent)
8  device.press('KEYCODE_MENU', MonkeyDevice.DOWN_AND_UP)
9  result = device.takeSnapshot()
10 result.writeToFile('myproject/shot1.png','png')
```

Zu den besonderen Fähigkeiten des Monkeyrunner gehört unter anderem, dass funktionale Tests auf mehreren Geräten zeitgleich ausgeführt werden können. Ebenso ist es möglich, unterschiedliche Konfigurationen des Android-Emulator in Rahmen der Testdurchführung automatisch zu starten und wieder zu beenden, so dass die Testdurchführung sowohl auf Emulatoren als auch auf Geräten vollständig ohne menschliche Interaktion erfolgen kann. Das macht den Monkeyrunner ideal zur Integration in CI-Systeme, bei denen das Bestehen von Testfällen Bedingung zur Einpflege von Quellcode ist.

Während der Monkeyrunner ein Werkzeug zur automatisierten Durchführung von Testfällen ist, das außerhalb des SUT steht und deshalb zumindest theoretisch weitreichende Möglichkeiten zur Kontrolle der Betriebsumgebung des SUT in simulierten Umgebungen hat, reicht die Standardimplementierung mit Ausnahme der (De-)Installation von Anwendungen, nicht wesentlich über die Möglichkeiten hinaus, die auch mit JUnit für Android erreicht werden können. Insbesondere existiert in der Standardimplementierung des Monkeyrunner keine Funktion, um Kontextparameter in Testfällen zu berücksichtigen.

Wünschenswert und von herausragender Bedeutung für das Testen kontextsensitiver, mobiler Systeme sind jedoch Funktionen, mit denen einem SUT zur Testlaufzeit bestimmte Umweltbedingungen vorgegeben werden können. Hierzu zählt unter anderem die Simulation von Standortinformationen, Sensormesswerten zur Unterstützung von ortsbasierten Anwendungsinhalten oder AR-Anwendungen. Diese Funktionalität kann im Monkeyrunner nur durch eine Erweiterung der Standard-API erreicht werden. Vom Grundsatz her ist Monkeyrunner hierzu auch geeignet, da es sich eben gerade nicht um ein in sich abgeschlossenes Werkzeug handelt, sondern um eine erweiterbare API, die je nach Bedarf angepasst werden kann. Für das in dieser Arbeit verfolgt das Ziel der Bereitstellung einer Methode zur automatischen Generierung und Durchführung von Testfällen ist Monkeyrunner deshalb grundsätzlich geeignet. Da hierzu allerdings ein signifikant hoher Aufwand notwendig ist, gilt es zunächst zu bewerten, ob sich Monkeyrunner gegen ähnliche Werkzeuge durchsetzen kann.

Aus der Perspektive eines Generators, der aus Testfallmodellen ausführbare Quellcodeartefakte erstellt, ist es unerheblich, welche konkrete Zieltechnologie bedient werden soll. Allerdings handelt es sich auch bei Testfällen um Artfakte, die einerseits Evolutionszyklen des Softwareprozesses unterliegen und deshalb trotz der initialen automatischen Generierung in späteren Phasen manuell modifiziert werden und andererseits von implementierungsfernen Bedarfsträger betreut werden. Da Monkeyrunner-Testfälle in Python erstellt werden, müssen Tester zwingend über Kenntnisse der Programmiersprache Python verfügen, um den fachlichen Inhalt von Testfällen zu beurteilen. In dieser Hinsicht ist Monkeyrunner anderen

Technologien unterlegen, die Testfälle in einem an die natürliche Spraches angelehnten Syntax beschreiben (z. B. Calabash), so dass spezielle Programmiersprachenkenntnisse nicht erforderlich sind.

2.5.2.5 Appium

Eine Technologie, die nicht nur native, mobile Anwendungen zum Gegenstand automatisierter Tests macht, sondern ebenfalls das Testen von mobilen Web-Anwendungen und hybriden Anwendungen erlaubt, ist das Framework *Appium* [14]. Die hier verfolgte Philosophie ist es einerseits, auf Modifikationen des SUT zu verzichten und so in einem Black-Box-Test-Ansatz das Testen eines auslieferbaren Softwareprodukts zu gewährleisten, das keine durch Testschnittstellen verursachten Angriffsvektoren exponiert. Anderseits sollen Entwickler und Tester nicht zur Verwendung einer spezifischen Programmiersprache zur Implementierung von Tests gezwungen sein. Aus diesen Grund implementiert Appium eine *Client*-Server-Architektur, die im Front-End eine REST-API anbietet, mit welcher Entwickler Tests und die zu testende Anwendung an Appium übermitteln können. Hierbei kann ebenfalls eine Reihe von Parametern übermittelt werden, die Anforderungen an die Zielplattform repräsentieren, u. a. Plattformarchitektur (z. B. Apple iOS, Android), Plattformversion, Spracheinstellungen und Orientierung (Hoch- oder Querformat). Zur eigentlichen Implementierung von Tests bietet Appium APIs für unterschiedliche Programmiersprachen an. Diese bieten Funktionen zur Manipulation des SUT und zur Auswertung der Ergebnisse. Appium bietet jedoch keine Funktionen zur Simulation von Kontext, wie etwa dem Standort des Anwenders oder von Sensorwerten.

2.5.2.6 Calabash

Neben einigen Testautomatisierungstechnologien, die sehr nah am Quellcode des SUT angesiedelt sind, entsteht auch und gerade im Umfeld mobiler Anwendungen das Bedürfnis, Testautomatisierung auch auf Teststufen jenseits von Komponententests durchzuführen. Eine der aktuell verfügbaren Technologien zur Befriedigung dieses Bedürfnisses ist das Technologierahmenwerk Calabash [369]. Es ermöglicht die Automatisierung der Durchführung von Akzeptanztests (primär UI-Akzeptanztests) für die mobilen Plattformen Android (Calabash-Android [121]) und Apple iOS (*Calabash-iOS* GitHub Community Project [122]).

Alleinstellendes Merkmal der Calabash-Technologie ist seine Fähigkeit, Beschreibungen von Testfällen in einer an die natürliche Sprache angelehnten Syntax in eine Abfolge von Instrumentierungsinstruktionen zu überführen, die eine vollständig automatische Testdurchführung gemäß der Kriterien i-vii in Abschnitt 4.3.1.3 erlaubt. Beide Varianten werden quelloffen in OSS-Projekten gepflegt und können deshalb von Anwendern und Entwicklern auf spezifische Anforderungen angepasst werden. Auch aus diesem Grund bildet Calabash die Technologiegrundlage zu dem in dieser Arbeit vorgestellten Lösungsvorschlag zur Testautomatisierung für mobile Anwendungen.

Calabash als Werkzeug zur Testautomatisierung ist zunächst plattformunabhängig. Allerdings handelt es sich um ein Framework, welches seinerseits auf einer Reihe plattformspezifischer Technologien basiert. Deshalb erfolgt in der Praxis die Unterscheidung in Calabash-Android und Calabash-iOS, wobei das Suffix jeweils die Zielplattform identifiziert. Testfallbe-

schreibungen können sowohl unter Verwendung von Calabash-Android als auch Calabash-iOS ausgeführt werden, erfordern aber je nach Plattform eine entsprechende spezifische Technologiebasis seitens des zu testenden Systems.

Implementierungen der jeweiligen Technologiebasis adaptieren die in *Gherkin* notierten Testfälle an plattformspezifische Werkzeuge und Technologien zur Interaktion mit dem SUT. Diese realisieren auf dem Zielsystem die dem Testfall einbeschriebenen Interaktionen mit der Anwendung durch gezielte und zweckorientierte Manipulation von Systemeigenschaften oder von Elementen der Benutzungsoberfläche der zu testenden Anwendung.

Funktionale Tests werden in Calabash unter Verwendung des BDD-Werkzeuges *Cucumber* aus in der DSL Gherkin notierten Testfallbeschreibungen durchgeführt. Bei Cucumber handelt es sich sich um eine Technologie des BDD, die es zulässt Verhaltensaspekte von Softwaresystemen in einer an die natürliche Sprache angelehnten DSL zu spezifizieren. Cucumber verwendet die DSL Gherkin, die aufgrund ihrer syntaktischen Eigenschaften sowohl menschenlesbar ist als auch maschinell verarbeitet werden kann.

2.5.2.6.1 Gherkin

Gherkin [123, 368] ist die vom Testwerkzeug Cucumber [83, 368] verwendete DSL zur Beschreibung funktionaler Testfälle. Sie erlaubt die verhaltensorientierte Notation von Testfällen in einer quasi-natürlichsprachlichen Syntax, ohne dass zur Testfallspezifikation Details der Implementierung bekannt sein müssen. Die quasi-natürlichsprache Syntax ist insofern ein herausstellendes Merkmal, als dass auch solche Bedarfsträger Testfälle erstellen können, die sich aus fachlich-inhaltlicher Perspektive und gerade nicht aus der technischen Perspektive mit dem SUT befassen. Um den Anspruch an die Lesbarkeit durch nicht-technische Bedarfsträger zu unterstreichen, implementiert Gherkins Grammatik den Wortschatz zur Testfallbeschreibung in mehreren (37, Stand Juli 2014) natürlichen Sprachen.

Gherkin ist eine Sprache, die Tests in Codeartefakte über mehrere Dateien mit der Dateiendung „.feature“ aufteilt. Jede dieser Dateien enthält ein sogenanntes Gherkin-Feature, eine Ordnungseinheit für Tests, die eine Menge inhaltlich-fachlich zusammenhängender Testfälle umfasst. Jedes Gherkin-Feature spezifiziert eine beliebige Anzahl von Gherkin-*Scenarios* (engl. Szenario, hypothetische Aufeinanderfolge von Ereignissen, die zur Betrachtung kausaler Zusammenhänge konstruiert wird), die sich orientierend an der Taxonomie in Abschnitt 4.1.1 jeweils auf das Artefakt Testfall abbilden lassen. Gherkin-Szenarien bestehen ihrerseits wiederum aus einer Menge von Gherkin-*Step-Definitions*, die jeweils einzelne Testschritte repräsentieren. Die Gesamtheit aller Gherkin-Features bildet eine Testsuite.

Ein Gherkin-Feature wird mit dem Schlüsselwort „Feature“ eingeleitet, auf welches frei wählbarer Text folgen darf, der den Inhalt des durch dieses Feature beschriebenen Tests natürlichsprachlich beschreibt. Darauffolgend leitet das Schlüsselwort „Scenario“ einen spezifischen Testfall ein. Einzelne Testschritte werden in den folgenden Zeilen spezifiziert. Einzelne Testschritte (die Gherkin Step Definitions) beschreiben hierbei wahlweise Vor-, Nachbedingungen oder konkrete durchzuführende Aktionen. Vorbedingungen werden hierbei durch das Schlüsselwort „Given“ (optional bei weiteren Vorbedingungen ersetzt durch „And“) eingeleitet, Aktionen durch das Schlüsselwort „When“. Nachbedingungen werden durch das Schlüsselwort „Then“ (optional bei weiteren Nachbedingungen ersetzt durch „And“) eingeleitet und können

durch das Schlüsselwort „And, but“ eingeschränkt werden. Dieses Schema lässt sich stets auf das natürlichsprachliche Konstrukt „Unter der Voraussetzung dass ‹*Vorbedingungen*› tut der Anwender ‹*Aktionen*› und dann kann ‹*Nachbedingungen*› beobachtet werden“ abbilden.

Quellcodefragment 2.7: Beispiel einer funktionalen Testfallspezifikation in Gherkin/Cucumber

```
1  Feature: Mobiler Taxiruf
2     Scenario: Benutzer lehnt die Verwendung des GPS ab, manuelle ↙
          ↳ Adresseingabe
3        Given my app is running
4        Given the view with id "dialog_locationmode" has appeared
5           And the view with id "button_manual" has appeared
6        When I press view with id "button_manual"
7        Then the view with id "dialog_address" has appeared
8        Given the view with id "editTextAdresse" has appeared
9           And the view with id "editTextStadt" has appeared
10          And the view with id "button_commitadress" has appeared
11       When I enter text "Gerlingstrasse 16" into field with id "↙
             ↳ editTextAdresse"
12          And I enter text "Essen" into field with id "editTextStadt"
13          And I press view with id "button_commitadress"
14       Then  the view with id "dialog_confirm" has appeared
```

In Quellcodefragment 2.7 ist exemplarisch ein Ausschnitt aus einer Calabash-Testfallspezifikation abgebildet. Spezifiziert wird ein einzelnes Feature, d. h. ein Testfall, einer fiktiven mobilen Anwendung, in welchem das Verhalten der Anwendung beschrieben ist wenn der Anwender einem bestimmten *Workflow* (engl. Arbeitsaublauf) folgt. Die Verwendung der Schlüsselwörter „Given“, „When“, und „Then“ sind aus der Perspektive der Testfalldurchführung optional und dienen lediglich der Aufrechterhaltung der natürlichsprachlichen Anmutung des Gherkin-Syntax.

Im abgebildeten Calabash-Beispiel in Quellcodefragment 2.7 sind in den Zeilen 3 bis 5 und 8 bis 10 Vorbedingungen deklariert, durch welche der Zustand der Anwendung insoweit eindeutig beschrieben ist, als dass von der Durchführung folgender Testschritte korrektes Verhalten erwartet werden kann. Im Anschluss an die Herstellung der Vorbedingungen wird in Zeile 7 eine Aktion ausgeführt, die eine Interaktion des Anwenders mit der Anwendung repräsentiert. Im abgebildeten Beispiel wird in Zeile 7 deklariert, dass ein Element mit der Benutzungsoberfläche durch Anklicken zu stimulieren ist. Erkennbar am Beispiel ist ebenfalls die Polymorphie von Vor- und Nachbedingungen, da hier das Sichtbarwerden eines UI-Elements sowohl als Vorbedingung als auch als Nachbedingung auftritt. Die Zeilen 8 bis 10 repräsentieren zum einen Vorbedingungen für die Durchführung weiterer Schritte, sind zugleich ebenfalls Nachbedingungen der Aktion in Zeile 7.

Das in Quellcodefragment 2.7 abgebildete Bespiel einer Calabash-Testfallbeschreibung verdeutlicht, wie Testfälle quasi-natürlichsprachlich deklariert werden. Sie spezifizieren das Verhalten des SUT. Abweichungen des tatsächlichen Verhaltens von der Spezifikation sind Fehlersituationen, die ihre Ursache sowohl in der Implementierung der zu testenden Anwendung haben können als auch in der Implementierung des Testfalls selbst (z. B. Tippfehler).

Für die in dieser Arbeit behandelte Methode zur modellbasierten Generierung von Testfällen aus Dokumenten des Systementwurfs ist Gherkin aus unterschiedlichen Gründen gut geeignet. Einerseits ist die natürlichsprachliche Anmutung der DSL geeignet, um Testfälle so zu repräsentieren, dass auch implementierungsferne Rollen im Softwareprozess in der Lage sind, den konkreten Inhalt und die fachliche Adäquatheit eines Testfalls zu beurteilen.

Insbesondere für Akzeptanztests ist das ein Kriterium von herausragender Bedeutung, da Akzeptanztests regelmäßig durch Bedarfsträger mit stark fachlicher Orientierung konzipiert und durchgeführt werden, denen sich Inhalte von Testfällen in quellcodenahem Syntax mangels spezifischer Programmiersprachenkenntnisse nicht erschließen.

Trotz der quasi-natürlichsprachlichen Syntax folgen in Gherkin geschriebene Testfälle einem definiert strukturierten Schema. Basierend auf dem in dieser Arbeit verfolgten Ansatz ausführbare Tests aus Modellen zu erzeugen, kann die maschinelle Generierung von Gherkin-Skripten deshalb besonders einfach durch Anwendung einer Modelltransformation, genauer einer M2T-Transformation, realisiert werden.

2.5.2.6.2 Cucumber

Cucumber [83, 368] ist ein Werkzeug zur Testautomatisierung aus dem Bereich des BDD. Wesentlicher Fokus liegt auf der Transition von der Dokumentation der Verhaltensspezifikation zu einer automatisierten Testdurchführung für eine Software, in welcher primär Verhaltensaspekte des SUT adressiert werden. Seitens der Testfallspezifikation bedient sich Cucumber der DSL Gherkin, die eine quasi-natürlichsprachliche Testspezifikation ermöglicht. Cucumber selbst stellt eine Ruby-basierte Technologie bereit, die in der Lage ist, Gherkin-Skripte zu analysieren und auf Fragmente ausführbaren Codes abzubilden, welcher den eigentlichen Inhalt des Tests auf implementierungstechnische Artefakte transferiert und auf das SUT anwendet. Die Funktionsweise des Cucumber Werkzeugs ist in Abbildung 2.1 dargestellt.

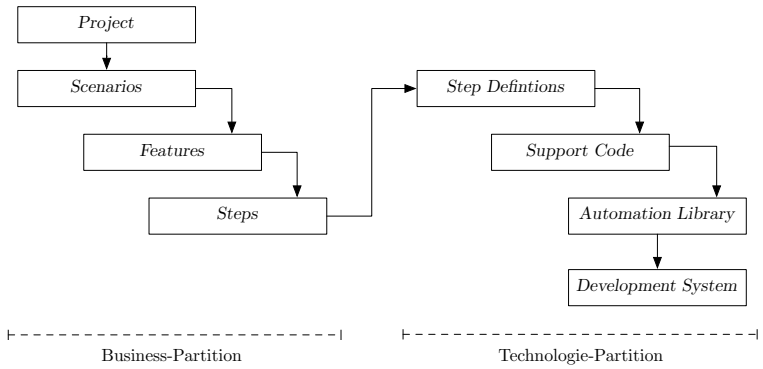


Abbildung 2.1: Schematische Abbildung der Funktionsweise und einzelner Funktionspartitionen des Werkzeugs Cucumber aus fachlicher Perspektive (Abbildung angelehnt an Wynne und Hellesoy [368]).

Cucumber unterscheidet zwei Partitionen des Werkzeugs. Die fachlich-inhaltliche Partition des Werkzeuges ist auf die Inhalte des Testfalls aus Geschäftsprozesssicht ausgerichtet. Sie spezifiziert *was* getestet werden soll. In ihr werden Testfälle mit Gherkin spezifiziert. Zur Durchführung des Tests werden Testschritte auf die jeweilige Technologiebasis des Zielsystems abgebildet. Hierzu kommt Instrumentierungstechnologie für das SUT zum Einsatz. Diese ist i. d. R. plattformspezifisch und kommt deshalb nicht ohne Vermittlungsschicht zwischen Cucumbers Ruby-Basis und der nativen Instrumentierungstechnologie des SUT aus. Das ist Aufgabe der technischen Partition des Cucumber-Werkzeugs – sie spezifiziert *wie* fachliches Verhalten getestet werden soll. Hierzu müssen Gherkin-Ausdrücke auf Methodenaufrufe des *Ruby-Automation-Code* abgebildet werden.

Hierzu müssen geeignete Abbildungsfunktionen in der technischen Partition implementiert werden. Diese Aufgabe erfordert weitreichende Kenntnisse sowohl der Zielplattform als auch des Cucumber-Werkzeugs. Diese Kenntnisse erstrecken sich dabei mindestens auf die Programmiersprache der Zielplattform und ihrer Instrumentierungstechnologie sowie auf die Programmiersprache Ruby. Das Erstellen des Ruby-Automation-Code ist deshalb eine Aufgabe, die im Normalfall im Auftrag eines fachlichen Bedarfsträgers durch einen technisch orientierten Bedarfsträger, i. d. R. einen qualifizierten Entwickler, in enger Zusammenarbeit mit Repräsentanten der Fachabteilung ausgeführt wird.

Zur Bereitstellung eines Gherkin-Ausdrucks durch welchen fachliche Testinhalte bedient werden können, muss zunächst eine geeignete Gherkin-Step-Definition erstellt werden. In Quellcodefragment 2.8 ist ein Beispiel einer solchen Step-Definition abgebildet. Sie stellt eine Schablone bereit, über welche in einer Testfallspezifikation ein graphisches UI-Element vom Typ Kontrollkästchen aktiviert bzw. deaktiviert werden kann. Zeile 1 in Quellcodefragment 2.8 leitet die Step-Definition ein und deklariert den Gherkin-Ausdruck:

„I toggle checkbox number ‹ganze zahl› *“*

Dieser kann in der späteren Verwendung (siehe Quellcodefragment 2.9) durch Angabe einer Zahl anstelle des Platzhalters parametrisiert werden.

Quellcodefragment 2.8: Beispiel einer Cucumber Step Definition

```
1 Then /^I toggle checkbox number (\d+)$/ do |checkboxNumber|
2   performAction('toggle_numbered_checkbox', checkboxNumber)
3 end
```

Aus der Sicht des fachlichen Testers manifestiert sich diese Step-Definition in der Verfügbarkeit des in Quellcodefragment 2.9 in Zeile 6 abgebildeten Gherkin-Ausdrucks. Zu erkennen ist in diesem Beispiel ebenfalls Austauschbarkeit der Schlüsselwörter *„Then“* und *„When“*, die durch Cucumber nicht unterschieden werden. In der Semantik des Testers hingegen wird zwischen der Spezifikation einer Vor- bzw. Nachbedingung und der Ausführung einer Interaktion mit dem SUT unterschieden.

Quellcodefragment 2.9: Cucumber-Testfallspezifikation zur Manipulation eines UI-Elements

```
1 Feature: Mobiler Taxiruf
2    Scenario: Benutzer passt die Einestellung zur Verwendung des GPS-↙
       ↳ Moduls an
3       Given my app is running
4       Given the view with id "dialog_locationmode" has appeared
5          And the view with id "button_manual" has appeared
6       When I toggle checkbox number 1
```

Die Bereitstellung der eigentlichen durch eine Gherkin-Step-Definition bereitgestellten Funktionalität, die sich im Regelfall in der Instrumentierung des SUT auf der Ebene von Komponententests manifestiert, entzieht sich dem Zugriff des Testers auf Akzeptanztestebene. Die Implementierung der in Zeile 2 Quellcodefragment 2.8 referenzierten Funktion ist plattformspezifisch und liegt außerhalb des im Rahmen dieses Abschnitts behandelten Diskussionsgegenstands. Die Bereitstellung solcher Gherkin-Step-Definitions, die Besonderheiten des Testens kontextsensitiver, mobiler Anwendungen implementieren, werden in Abschnitt 5.5.3 im Rahmen der Anpassungen Calabash im Detail diskutiert.

Bei der Ausführung von Tests werden die in der Testspezifikation enthaltenen Gherkin-Ausdrücke der Reihe nach abgearbeitet. Hierzu versucht Cucumber eine Step-Definition zu identifizieren, die auf den Gherkin-Ausdruck abbildbar ist. Existiert eine solche Step-Definition, wird der zugehörige Code – im Normalfall Zugriff auf Funktionen der plattformspezifischen Instrumentierungstechnologie – ausgeführt. Dieser kann dann sofern möglich die Gültigkeit von Vorbedingungen auf dem SUT erzwingen, Interaktionen mit dem SUT simulieren (z. B. durch Manipulation von UI-Elementen) oder die Gültigkeit von Nachbedingungen überprüfen. Jede Abweichung vom normalen Programmablauf, d.h. eine Vorbedingungen kann nicht hergestellt werden, eine Aktion kann nicht ausgeführt werden oder eine Nachbedingung ist verletzt, wird als Nichtbestehen des Testfalls gewertet.

2.5.2.6.3 Calabash Architektur und Funktionsweise

Apps für Smartphones und Tablets werden nicht auf Smartphones oder Tablets entwickelt, sondern die Entwicklung erfolgt i. d. R. auf einem Desktop-Computer. Ausnahmen hiervon werden gegenwärtig nur von wenigen Avantgarde-Drittanbietern gebildet, die Technologien anbieten, durch welche App-Entwicklung auf dem Zielgerät (d. h. Smartphone oder Tablet-Computer) ermöglicht wird. Hierbei handelt es sich gegenwärtig jedoch um Nischentechnologie, deren Marktrelevanz sich mutmaßlich erst mit der weiteren Konvergenz mobiler Geräte und Desktop-Computern einstellen wird.

Die Überwindung der Kluft zwischen Entwicklungsgerät (d. h. Desktop-Computer) und Zielgerät (z. B. Smartphone) ist von besonderem Interesse für Testaktivitäten. Plattformnative Instrumentierungstechnologie arbeitet direkt auf dem Zielgerät, denn nur so ist es aufgrund der System- und Anwendungsarchitektur mobiler Plattformen möglich, zur Laufzeit Zugriff auf Artefakte des SUT zu erlangen. Das ist einerseits in der Abwesenheit planmäßiger technischer Instrumentierungsschnittstellen (z. B. JTAG) begründet, andererseits aber auch in der Sicherheitsarchitektur mobiler Plattformen (z. B. Sandboxes, Prozessisolation, Virtualisierung), die Anwendungen zur Laufzeit gezielt gegen externe Zugriffe schützen.

Mit dem Ziel diese Kluft zwischen Entwicklungsgerät und Zielplattform für Testaktivitäten zu überwinden, implementiert Calabash einen Verbund des Werkzeugs Cucumber, des Automatisierungswerkzeugs Robotium und plattformnativer Instrumentierungstechnologie der es ermöglicht, Akzeptanztests für mobile Apps auf dem Entwicklungscomputer zu spezifizieren und durchzuführen während das SUT auf einer vom Entwicklungscomputer verschiedenen Zielplattform ausgeführt wird.

Zur Realisierung von Akzeptanztest auf mobilen Geräten implementiert Calabash deshalb eine Architektur, die eine Steuerung des Testablaufs auf dem Entwicklungscomputer auf der fachlichen Ebene von Akzeptanztests erlaubt, während der Zugriff auf Artefakte des SUT auf Ebenen unterhalb von Akzeptanztests direkt auf der Zielplattform erfolgt. Diese kann sich beispielsweise in einem Smartphone/Tablet-Computer oder in einem Geräteemulator manifestieren. Die Architektur des Calabash Werkzeugs ist in Abbildung 2.2 dargestellt.

In der Abbildung 2.2 ist links der Technologie-Stapel dargestellt, durch welchen die Testdurchführung auf dem Entwicklungscomputer erfolgt. Dieser Teil der Calabash-Architektur realisiert alle Cucumber-Aufgaben, d. h. die syntaktische Analyse von Gherkin-Features, die Abbildung von Gherkin-Ausdrücken auf Step-Definitionen und die Ausführung derjenigen

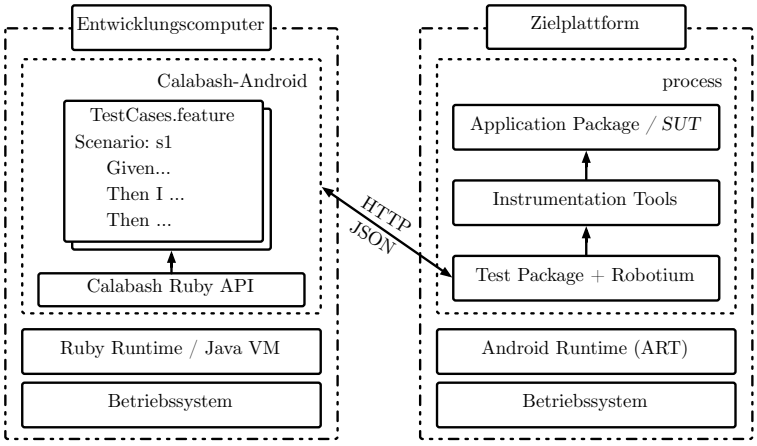


Abbildung 2.2: Schematischer Aufbau der Architektur des Calabash-Werkzeugs.

Codefragmente, durch welche die technische Implementierung einzelner Testschritte realisiert wird. Im besonderen Fall Calabash reduziert sich diese Implementierung auf Metaktivitäten des Testens (d. h. nicht-inhaltliche Testaktivitäten wie etwa (De-)Installation des SUT auf dem Zielgerät) und auf die Übertragung, den Empfang und die Analyse/Bewertung von Steuerbefehlen an das SUT. Dieses wird künstlich angereichert um Testschnittstellen auf der Zielplattform (rechts in Abbildung 2.2) ausgeführt.

Testfälle müssen so zum SUT positioniert werden, dass unter Berücksichtigung der Sicherheitsarchitektur der Zielplattform auf Codeartefakte des SUT zugegriffen werden kann. Hierzu kommen je nach Plattform unterschiedliche Konzepte zum Einsatz. Auf der Plattform Android beispielsweise wird diese Anforderung realisiert, indem neben dem SUT ein weiteres Anwendungspaket installiert wird. Dieses enthält einen generisches Test-Framework auf Robotium-Basis und ist mit demselben *Package-Identifier* (einem eindeutigen Identifizierungsmerkmal für Android Apps) versehen und mit demselben *Code-Signing-Key* (engl. Code-Signaturschlüssel, wesentliches Sicherheitsmerkmal der Android Plattform zur Sicherstellung der Authentizität der Urheberschaft einer App) unterzeichnet. In Androids Sicherheitsarchitektur werden das SUT und das Testpaket dann als organisatorische Einheit wahrgenommen. Das Testpaket selbst wird vom System als eigenständige App wahrgenommen, ihm wird aber Zugriff auf die Instrumentierungsschnittstellen des SUT gewährt.

Da der funktionale Cucumber-Testcode lokal auf dem Entwicklungsgerät ausgeführt wird, der eigentliche Testfall jedoch in Form eines Komponententest lokal auf der Zielplattform, muss ein Kommunikationskanal zwischen beiden Geräten existieren. Calabash realisiert diesen Kommunikationskanal in Form einer Netzwerkschnittstelle zwischen Entwicklungsgerät und Zielplattform, welche auf dem Protokoll *Hypertext Transfer Protocol* (HTTP) basiert. Das Testpaket enthält Methoden, mit denen auf Artefakte des SUT zugegriffen werden kann, z. B. Eingabe von Werten in Textfelder, ausführen von Operationen aus berührungsempfindlichen UI-Elementen, oder die UI-Elemente inspizieren können, z. B. lesen von Werten in Textanzeigefeldern.

Das Testpaket empfängt an einem HTTP-*Socket* (engl. Software-Modul der Netzwerkkommunkation) Steuerbefehle im Format *JavaScript Object Notation* (JSON), die von der

Calabash-spezifischen Cucumber-Implementierung während der Testdurchführung versendet werden. Diese Steuerbefehle werden auf spezifische Methodenaufrufe der Instrumentierung abgebildet. Das Resultat eines solchen Funktionsaufrufs wird über dieselbe HTTP-Schnittstelle an die auf dem Entwicklungsgerät ausgeführte Cucumber-Instanz zurückgesendet. Dort wird anhand der Rückmeldung das Bestehen oder Nichtbestehen eines Testschritts beurteilt.

Dieser Technologie-Stack basiert auf einer Reihe Kommandozeilenwerkzeuge und ist so konzipiert, dass er nicht nur von einem einzelnen Entwickler oder Tester unmittelbar verwendet werden kann, sondern er ermöglicht ebenfalls eine Verwendung innerhalb weiterer Automatisierungswerkzeuge der Softwareentwicklung. Die automatische Testdurchführung kann beispielsweise an SCM-Systeme angeschlossen werden, so dass ein Entwickler nur dann neuen Quellcode in den produktiven Zweig des SCM-Systems einpflegen darf, wenn alle Tests erfolgreich ausgeführt werden können. Ebenso denkbar ist die Integration in ein CI-System, so dass eine neue Produktversion nur in Abhängigkeit der Resultate einer automatisierten Testsuite veröffentlicht wird.

2.5.2.6.4 Beschränkungen

Vom Grundsatz her ist Calabash-Android eigens dazu entwickelt, die Durchführung von Akzeptanztests für mobile Anwendungen zu ermöglichen. Und obwohl Calabash die Möglichkeit eröffnet auch mobile Anwendungen mit einer Automatisierungstechnologie für Tests zu bedienen, ist der Funktionsumfang von Calabash aktuell auf Aspekte der Interaktion eines simulierten Anwenders mit dem SUT reduziert. Die wesentlichen Probleme des Testens kontextsensitiver, mobiler Anwendungen, nämlich die Auswirkungen von Mobilität und Kontextsensitivität auf den Softwareentwicklungsprozess (vgl. Abschnitt 3.2, Abschnitt 4.2) und insbesondere auf Testaktivitäten, werden durch Calabash bislang nicht adressiert.

Beispielsweise ist es bislang nur rudimentär möglich, Testfälle für Anwendungsfälle in ausreichender Detailtiefe zu realisieren, die den Standort des Anwenders verwenden. Die Calabash-Standardimplementierung ermöglicht lediglich die Simulation von WGS84-Koordinaten ohne weitere Metadaten, wie etwa Präzision oder Alter der Standortinformation. Gerade diese Metadaten sind für realitätsnahe Testfälle von Bedeutung, da sich die Qualität einer mobilen Anwendung zur Realisierung ortsbasierter Anwendungsfälle gerade in der Fähigkeit manifestiert, unpräzise oder widersprüchliche Ortsinformationen so zu verwenden, dass der fachliche Inhalt des Anwendungsfalls weitestgehend erhalten bleibt.

Weiterhin ist Calabash nicht in der Lage, Sensorinformationen (z. B. Magnetfeldsensor, Beschleunigungssensor) oder variierende Parameter der Netzwerkverbindung in Testfälle einzubeziehen. In mobilen Anwendungen sind diese Parameter jedoch häufig von Bedeutung, z. B. in solchen Anwendungen, die für Freizeitaktivitäten im Freien konzipiert sind und neben dem Standort des Anwenders zusätzlich eine Orientierung nach Himmelsrichtung oder Ähnliches darstellen (etwa bei *Geocaching*-Anwendungen), wobei die Verwendung von Sensordaten obligatorisch ist.

Um die im Rahmen dieser Arbeit vorgestellte Methode zur Generierung von Testfällen in der Praxis zu erproben, stellt Calabash trotz dieser technologischen Einschränkungen die unter den untersuchten Automatisierungstechnologien am besten geeignete dar. Um auch kontextsensitive Testfälle mit Testautomatisierung bedienen zu können, wurde die Calabash-

Standardimplementierung im Rahmen dieser Dissertation so angepasst, dass auch Standortangaben mit Metadaten, Sensordaten und sonstige relevante Kontextparameter (vgl. Abschnitt 3.1.2) in Testfällen verwendet werden können. Diese Anpassungen sind im Abschnitt 5.5.3 im Detail beschrieben.

2.5.2.7 Xamarin

Eine auf Calabash basierende Automatisierungstechnologie ist *Xamarin* [370]. Die Möglichkeiten der Simulation von Kontext gehen bei Xamarin nicht über die von Calabash hinaus, so dass Xamarin keine bessere Alternative zur Realisierung einer Testautomatisierungslösung darstellt. Bemerkenswert ist jedoch das Xamarin-Dienstleistungsangebot. Hier bietet der Betreiber eine Geräte-Cloud an, auf die durch Nutzer im Teilzeitnutzungsverfahren zugegriffen werden kann. Entwickler haben hier die Möglichkeit, eine App auf einer Menge unterschiedlicher Geräte zu testen, ohne hierfür ein eigenes Geräteportfolio betreiben zu müssen.

2.6 Zusammenfassung

In den Abschnitten 2.1 bis 2.5 wurde das wissenschaftliche und technische Umfeld des Testens mobiler, kontextsensitiver Apps detailliert beleuchtet. Nach dem gegenwärtigen Stand von Forschung und Technik im Jahr 2016 ist das Testen – und insbesondere die Automatisierung von Tests – für mobile Anwendungen ein Feld, dass zwar intensiv beforscht wird, das bislang aber keine praktisch nutzbaren Methoden oder Technologien hervorgebracht hat.

Die Untersuchung themenverwandter Arbeiten hat ergeben, dass die Verwendung der UML ein probates Mittel zur Modellierung von Tests und Testdaten ist, aber originär nicht über die notwendigen Sprachmittel für diese Aufgabe verfügt. Um die UML als Grundlage einer Technologie zur Generierung von Tests aus Verhaltensmodellen mobiler Anwendungen nutzbar zu machen, muss das Metamodell der UML um entsprechende Modellierungskonstrukte erweitert werden. In dieser Dissertation wird mit dem in Abschnitt 5.2.4 diskutierten UML-Profil zur Testfallmodellierung genau dieser Ansatz umgesetzt.

Von den in den vorherigen Abschnitten untersuchten Testautomatisierungstechnologien hat sich Calabash als geeignete Grundlage zur Implementierung eines Werkzeugs zur automatisierten Durchführung von Tests für mobile, kontextsensitive Anwendungen herauskristallisiert. In dieser Dissertation wird in Abschnitt 5.5.3 eine an diese besonderen Anforderungen angepasste Implementierung des Calabash-Frameworks vorgestellt.

Die im Vorangegangenen untersuchte wissenschaftliche Literatur hat ebenfalls aufgezeigt, dass mobile Plattformen im Jahr 2016 nicht über geeignete Schnittstellen verfügen, während der Testausführung Kontextparameter in einem ausreichenden Maß zu simulieren. Abschnitt 5.5.2 thematisiert deshalb eine angepasste Implementierung des Android-Betriebssystems, welches um Schnittstellen zur erweiterten Kontextsimulation ergänzt wurden.

Eine weitere Kernerkenntnis des Studiums themenverwandter Arbeiten ist, dass die Erstellung von Tests einer der wesentlichen Aufwandstreiber ist. Abschnitt 5.3 und Abschnitt 5.4 adressieren deshalb im Rahmen des Entwurfs einer Methode und eines Werkzeugs zur modellbasierten Testautomatisierung kontextsensitiver, mobiler Anwendungen eine modellbasierte Technologie, Tests aus Artefakten des Systementwurfs zu generieren.

Kapitel 3

Mobilität und Kontextsensitivität in mobilen Softwaresystemen

Die ersten PDAs haben viele Anwendungen von Desktop-Computern losgelöst und Smartphones haben die mobile Verwendung von Software selbstverständlich gemacht. Darüber hinaus ist Software nicht nur selbst mobil, sondern bezieht den Ort ihrer Ausführung und Eigenschaften ihrer Betriebsumgebung aktiv in die Steuerung von Kontroll- und Datenflüssen ein (vgl. Schilit et al. [311], Baldauf et al. [25]). Softwaremobilität und Kontextsensitivität werden durch den sich fortsetzenden Trend zur Miniaturisierung auch zukünftig weiter an Bedeutung gewinnen (Roman et al. [303], Picco et al. [282]). Nach Mobiltelefonen werden nun mit den sogenannten *Smartwatches* (z. B. Apple Watch, LG G Watch) weitere ständig am Körper geführte elektronische Geräte zum Ziel der Integration von Software in den Alltag.

Spezifische Softwareprodukte können über Gerätegrenzen hinweg miteinander und mit dem Anwender interagieren. Der Anwender ist beispielsweise in der Lage, den Zustand seiner körperlichen Fitness über Online-Plattformen wie etwa Google Fit / Google Health [149] oder Apple Health [16] zu beobachten. Softwareanbieter entwickeln eigens zu diesem Zweck Lösungen, die es dem Anwender ermöglichen, Aktivitäten jeder Lebenssituation durch Verwendung mobiler Geräte automatisiert zu protokollieren. Solche Anwendungen bieten dem Anwender ein ganzheitliches Nutzungserlebnis über die Grenzen verschiedener mobiler Geräte hinweg. So kann etwa eine Smartwatch mit integriertem Schrittzähler verwendet werden, um während des gesamten Tages die fußläufig zurückgelegte Wegstrecke zu ermitteln. Apps ermöglichen die Protokollierung sportlicher Aktivitäten (z. B. Ermittlung der zurückgelegten Strecke beim Lauftraining via GPS) und Web-Anwendungen dienen der Analyse und Statistik. Der Anwender hat hier den Eindruck, ein einziges monolithisches Softwareprodukt zu verwenden (vgl. Myers et al. [261]), welches sowohl mobile (Smartphone), nicht-mobile (Web-Anwendung) und ultramobile (Smartwatch, vgl. Abschnitt 3.1.1.2) Komponenten umfasst.

Abhängig von Art und Ort des *Deployment* (engl. Softwareverteilung auf Geräte) muss i. d. R. jedoch für jedes zu unterstützende Gerät ein individuelles Softwareartefakt entwickelt werden. Für jedes dieser Softwareartefakte muss Mobilität und Kontextsensitivität individuell während der Entwicklung und insbesondere während des Testens berücksichtigt werden. Die abstrakte Natur von Mobilität und Kontextsensitivität ist jedoch ein Problem, welches von der Informatik zwar in Reaktion auf den technischen Fortschritt mobiler Geräte intensiv

bearbeitet wird, bislang aber nicht abschließend erforscht wurde. Es ist im Jahr 2016 wissenschaftlicher Konsens, dass Mobilität und Kontextsensitivität beim Entwurf, bei der Implementierung und beim Testen von Software berücksichtigt werden müssen (vgl. Abschnitt 2.2). Die Frage, wie diese Faktoren jedoch zu berücksichtigen sind, kann bislang nicht allgemeingültig beantwortet werden. Es existieren gegenwärtig keine Werkzeuge der Softwaretechnik, wie etwa Modellierungssprachen, Diagrammtypen oder Softwareentwurfsmuster, die den Softwareprozess für die spezifischen Anforderungen von Mobilität und Kontextsensitivität unterstützen (vgl. Baumeister et al. [29], Strang und Linnhoff-Popien [336], Bettini et al. [42]). Solche Anforderungen müssen deshalb gegenwärtig informal natürlichsprachlich formuliert und dokumentiert werden. Sie entziehen sich aufgrund dessen weitgehend dem Zugriff zur Verwendung in MDA-/MDSD- und insbesondere MDT-Technologien wie dem in dieser Dissertation untersuchten Konzept der Testautomatisierung mobiler, kontextsensitiver Anwendungen, der eine Modellierung von Kontextparametern in Systemmodellen voraussetzt.

Um die Ausführungen zur Rolle von Mobilität und Kontextsensitivität im Lebenszyklus eines Softwareproduktes zu veranschaulichen, werden zwei Beispiele für mobile, kontextsensitive Anwendungen eingeführt, die im Fortgang dieser Dissertation immer wieder aufgegriffen werden. Sie werden exemplarisch verwendet, um zu verdeutlichen, welchen Einfluss Mobilität und Kontextsensitivität auf mobile Anwendungen haben und welche Anforderungen sich hierdurch an den Entwurf, die Implementierung, das Testen und den Betrieb stellen. Es werden zwei Beispielanwendungen eingeführt, um jeweils unterschiedliche Aspekte der Entwicklung und des Testens mobiler, kontextsensitiver Anwendungen hervorzuheben. Die erste Beispielanwendung soll einen ortsbasierten Taxiruf realisieren, die zweite Anwendung ist eine AR-Anwendung im Bereich Touristik. Es handelt sich jeweils um beispielhafte Anwendungen, die im Rahmen der Dissertation teilweise prototypisch realisiert wurden, um die Einflussfaktoren Mobilität und Kontextsensitivität exemplarisch zu diskutieren und die erarbeiteten Lösungsvorschläge zu validieren. Beide Anwendungen wurde ausgewählt, weil sie sich inhaltlich an existierenden Anwendungen orientieren, durch welche die beschriebene Funktionalität in ähnlicher Weise realisiert wird und die jeweils in den App Stores verfügbar sind.

Mobiler Taxiruf Diese Beispielanwendung ist eine mobile App, deren Funktion es ist, auf Wunsch des Anwenders ein Taxi zu dessen gegenwärtigen Aufenthaltsort zu rufen. Anwendungsinhalt ist ein realitätsnaher Geschäftsprozess, der tatsächlich von mehreren Anbietern in den App Stores der Plattformen Android, Apple iOS, Microsoft Windows Phone und BlackBerry angeboten wird. Er umfasst sowohl eine digitale Komponente (Auslösen der Bestellung einer Dienstleistung via App) als auch eine nicht-digitale Dienstleistung (Personenbeförderung). Beide Aspekte sind im Sinn der in dieser Dissertation verwendeten Definition kontextsensitiv (vgl. Abschnitt 3.1), da zur Leistungserbringung situative Faktoren des Anwenders – nämlich dessen Standort und das Datum der Leistungsanforderung – notwendig sind.

Zur isolierten Betrachtung relevanter Aspekte wird in dieser Dissertation gezielt von existierenden Produkten wie mytaxi™ [198] oder Uber™ [346] abstrahiert. Die Beispielanwendung Mobiler Taxiruf wird in dieser Dissertation kontinuierlich verwendet, um den Einfluss der konkreten Kontextfaktoren Zeit und Ort beispielhaft zu diskutieren. Hierzu

werden zunächst einige Anforderungen der Beispielanwendung spezifiziert. Im Folgenden werden die individuellen Anforderungen an die Beispielanwendung jeweils unter den Aspekten Modellierung, Implementierung, Betrieb und insbesondere Testen diskutiert.

Die Anwendung Mobiler Taxiruf ermöglicht es dem Anwender innerhalb Deutschlands zu jeder Tages- und Nachtzeit ein Taxi zu seinem gegenwärtigen Aufenthaltsort zu rufen.

Die Anwendung soll für die Plattform Android verfügbar sein, es sollen sowohl Smartphones als auch Tablet-Computer-Computer unterstützt werden.

Ein UI zeigt dem Anwender seinen aktuellen Standort auf einer Kartenansicht, die durch die Komponente Google Maps [152] auf Android Systemen bereitgestellt wird.

Die Ermittlung des Standorts erfolgt durch Verwendung des GPS-Moduls des Geräts. Verfügt das Gerät nicht über ein GPS-Modul oder kann in einem Zeitraum von 30 Sekunden der Standort des Anwenders nicht bestimmt werden, wird dem Anwender alternativ zur Kartenansicht ein Dialog zur manuellen Eingabe seines Standorts angezeigt.

Der Anwender kann manuell zur manuellen Standorteingabe wechseln.

Der Anwender erhält unmittelbar nach dem Öffnen der App eine Rückmeldung über seinen Aufenthaltsort. Hierzu werden neben dem GPS-Modul auch andere verfügbare Ressourcen zur Standortbestimmung verwendet. Der letzte bekannte Standort des Anwenders wird verwendet, wenn er nicht älter als 3 Stunden ist und sofort auf der Kartenansicht angezeigt.

Existiert kein letzter bekannter Standort des Anwenders oder die Information ist älter als 3 Stunden, wird die Kartenansicht auf den geographischen Mittelpunkt des Gebiets der Bundesrepublik Deutschland zentriert und bildschirmfüllend skaliert.

Die Kartenansicht wird während der Standortbestimmung angepasst, wenn Standortinformationen mit höherer Qualität (Genauigkeit, Alter der Information) bekannt werden.

Die Taxibestellung erfolgt über eine Schaltfläche auf dem UI.

Eine Bestellung kann ausgelöst werden, wenn eine Standortinformation vorliegt, die entweder manuell eingegeben wurde oder vom GPS-Modul erzeugt wurde, eine geschätzte Genauigkeit von höchstens 150 Meter ausweist und nicht älter als 30 Sekunden ist.

Die Bestellung erfolgt über einen Aufruf eines Web-Service. Die Entscheidung darüber, welche der zum Zeitpunkt des Funktionsaufrufs verfügbaren Internetverbindungen (Wi-Fi, UMTS, GPRS, etc.) verwendet wird, um mit dem Backend-System zu kommunizieren, wird dem Betriebssystem überlassen. Bei der Bestellung werden Standort, Uhrzeit und Telefonnummer des Anwenders an das Backend-System übertragen.

Nach erfolgreicher Bestellung wird dem Anwender ein Hinweisdialog angezeigt. Dieser gibt Auskunft über die zu erwartende Wartezeit bis zum Eintreffen des Taxis am Standort des Anwenders.

Steht keine Internetverbindung zur Verfügung oder wird die Internetverbindung nichtverfügbar, wird dem Anwender ein Hinweisdialog angezeigt.

Wird die App vom Betriebssystem unterbrochen, beispielsweise weil ein Anruf oder eine *Short Message Service* (SMS)-Nachricht eingeht, kann die App nach der Unterbrechung

fortgesetzt werden. Insbesondere werden bereits im UI getätigte Eingabe wiederhergestellt.

In Abschnitt 6.5.1 wird der in dieser Dissertation untersuchte Ansatz zur Testautomatisierung mobiler Anwendungen im Rahmen einer Fallstudie zur Validierung der Methode und der Technologie auf die App Mobiler Taxiruf angewendet.

Augmented Reality Tourist Information Diese Beispielanwendung implementiert einen digitalen Anwendungsfall der Tourismusbranche. Ein historisches Stadtarchiv hat eine umfangreiche Sammlung historischer Fotographien digitalisiert und möchte den Fundus Interessierten in einer App zugänglich machen[1].

Die App soll über den Vertriebsweg Google Play Store verteilt werden. Zusätzlich schafft die Betriebsgesellschaft der lokalen Touristeninformation eine Reihe von Leihgeräten an, die gegen ein Entgelt und eine Kaution an Interessierte verliehen werden. Hierbei handelt es sich ausschließlich um Tablets mit einer Displaydiagonale von 10 Zoll und der Android Version Lollipop.

Die App soll den Anwender auf einer historischen Entdeckungsreise durch die Stadt begleiten. Zulässige Standorte werden hierzu durch *Geofencing* auf das Stadtgebiet begrenzt. Der Anwender soll in der Lage sein, aus einer Anzahl vorgefertigter Routen zu unterschiedlichen stadtgeschichtlichen Themen zu wählen. Die App verwendet die Google Maps [152] Komponenten des Android-Systems, um auf einer Straßenkarte einen vorgesehenen Spazierweg und den aktuellen Standort des Anwenders anzuzeigen.

Bei Erreichen von *Points of Interest* (POIs, engl. Orte von Interesse, OVIs) entlang der ausgewählten Wegstrecke wird ohne Notwendigkeit manueller Interaktion ein informativer Text als Audio-Einspieler zur historischen Signifikanz des gegenwärtigen Standorts des Anwenders abgespielt.

Der Anwender hat die Möglichkeit, in einer Liste weitere Medien (Audio-Einspieler, Video-Beiträge und historische Fotos) mit Bezug zum gegenwärtigen Standort auszuwählen und abzuspielen bzw. anzuzeigen.

Die App bietet weiterhin eine *Augmented Reality* (AR)-Funktion an. Auf dem Display des Smartphones wird hierzu ein Live-Bild der rückwärtigen Gerätekamera angezeigt. Dieses Live-Bild wird in Abhängigkeit vom Standort des Anwenders und von der Ausrichtung des Geräts mit Zusatzinformationen überlagert. Wird das Gerät beispielsweise nach Norden ausgerichtet, wird das Live-Bild Kamera mit Kennzeichnungen zu den in nördlicher Richtung vom Standort des Anwenders liegenden POIs überlagert. Der Anwender kann dann durch Antippen eines POI direkten Zugriff auf zugehörige Medien nehmen. Weiterhin kann der Anwender den vorgesehenen Spazierweg dynamisch um die in der AR-Ansicht dargestellten POIs ergänzen.

[1] Die Idee zu dieser Beispielanwendung entstammt Anforderungen zu einem Projekt des Stadtarchivs Hagen, Fachbereich Kultur. Der Facebook-Auftritt des Stadtarchivs der Stadt Hagen [91] sollte im Jahr 2014 um eine App ergänzt werden. Im Rahmen der Bachelorarbeit von Andreas Kaiser [10] am Lehrstuhl für Software Engineering, insb. mobile Systeme an der Universität Duisburg-Essen, die im Rahmen der Bearbeitung dieser Dissertation betreut wurde, ist ein Prototyp angefertigt worden, der wesentliche Anteile der beschriebenen Funktionalität abdeckt, jedoch nicht im App Store veröffentlicht wurde.

Der Anwender hat weiterhin die Möglichkeit, an jedem Standort, zu welchem das Stadtarchiv eine historische Fotoaufnahme hinterlegt hat, das Live-Bild der Kamera mit der historischen Aufnahme zu überblenden.

Zur Finanzierung der App und des zugehörigen Backend-Systems zeigt die App weiterhin Hinweise auf das Angebot lokaler Gastronomie. Hierzu können Gastronomen Werbezeit buchen. Die Werbeanzeige selbst kann auf situative Faktoren des Anwenders abgestimmt werden, d. h. dessen Kontext. Neben dem Standort gastronomischer Angebote können Werbeanzeigen ebenfalls in Abhängigkeit von Tageszeit und Wetter konfiguriert werden. Der Betreiber mobiler Speiseeisverkaufswagen hat so beispielsweise die Möglichkeit, den Kontext des Anwenders in die Optimierung seiner Werbeanzeige auf lokale Nähe zu Eisverkaufswagen bei sommerlichem Wetter einzubeziehen.

Zur Leistungserbringung muss die App ständig den Aufenthaltsort des Anwenders bestimmen. Der Anwendungsfall erfordert hohe Genauigkeit der Ortsbestimmung. Auf Mobilfunktechnologie basierende Lokalisierungstechniken (vgl. Abschnitt 3.1.2.1.3) sind deshalb ungeeignet und es muss GPS verwendet werden.

Die Orientierung und Ausrichtung des verwendeten mobilen Geräts im Verhältnis zur Erdoberfläche ist zur Realisierung der AR-Funktion notwendig. Hierzu verwendet die App den Beschleunigungssensor und den Magnetfeldsensor des Geräts.

Die App benötigt zudem eine sporadisch verfügbare Internetverbindung, um Anwendern neu hinzukommende Inhalte schnell verfügbar zu machen. Darüber hinaus ist keine permanent verfügbare Internetverbindung erforderlich, da Inhalte vor Antritt des historischen Themenspaziergangs unter Verwendung einer Wi-Fi-Verbindung auf das Gerät geladen werden.

Wird die App vom Betriebssystem unterbrochen, beispielsweise weil ein Anruf oder eine SMS eingeht, wird die App nach der Unterbrechung fortgesetzt.

Die von dieser fiktiven Beispielanwendung beschriebene Funktionalität wird von einer Reihe real verfügbarer Apps in ähnlicher Weise implementiert (z. B. Wikitude™ [362], eTips London Travel Guide™ [111], timetraveler berlin wall™ [298]). Um die Eigenschaften mobiler, kontextsensitiver Anwendungen am Beispiel zu diskutieren, ist es jedoch zweckmäßig von der Komplexität existierender Produkte zu abstrahieren und stellvertretend eine Beispielanwendung zu verwenden, deren Funktionsumfang vollständig bekannt ist und über deren Implementierung sichere Annahmen getroffen werden können.

Die Funktion dieses Kapitels ist die Einbettung der in dieser Dissertation untersuchten Methode zur Testfallgenerierung und deren automatisierte Ausführung in den Gesamtkontext mobiler Anwendungen. Für das Verständnis der Methode und des Werkzeugs zur Testautomatisierung mobiler, kontextsensitiver Anwendungen ist es notwendig, Eigenschaften und Auswirkungen von Kontext auf Softwaresysteme im Detail zu diskutieren und ingesamt in Bezug zur Thematik Testen von Software zu setzen.

In den folgenden Abschnitten werden die besonderen Herausforderungen von Mobilität und Kontextsensitivität für die Entwicklung und den Betrieb von Software adressiert. Abschnitt 3.1.1 diskutiert die in dieser Arbeit verwendete Terminologie. In Abschnitt 3.1.2 wer-

den relevante Kontextparameter betrachtet, die mobile Geräte der Smartphone-Generation in der Lage sind durch Sensoren zu ermitteln. Abschnitt 3.2 thematisiert die Auswirkungen von Mobilität und Kontextsensitivität auf den Betrieb von kontextsensitiven Anwendungen.

3.1 Mobilität und Kontextsensitivität

Der Begriff *Kontext* wurde in der Softwaretechnik bereits in unterschiedlichen Zusammenhängen verwendet. Beispielsweise wird der Wechsel des aktuell bearbeiteten Programms (oder *Threads* in der Parallelverarbeitung in Prozessoren) als Kontextwechsel bezeichnet. Während diese Bedeutung weiterhin gültig ist und in diesem Zusammenhang Verwendung findet, hat sich der Begriff *Kontext* auch in anderen Zweigen der Softwaretechnik etabliert – insbesondere im Umfeld mobiler Anwendungen. Um Verwechslungen vorzubeugen und die Verwendung des Begriffs in dieser Arbeit klar abzugrenzen, werden zentrale Begriffe im Abschnitt 3.1.1 in ihrer Bedeutung definiert.

Kontextsensitivität beschreibt die Eigenschaft von Software, Parameter ihrer Betriebsumgebung in Software nutzbar zu machen. Hierzu kommen alle Parameter in Frage, die eine Software in der Lage ist festzustellen – beispielsweise durch Messung mit Hilfe von Sensoren oder durch Abfrage von Schnittstellen des Betriebssystems, Middleware oder Backend-Systemen. Zwar ist in gegenwärtigen Smartphone-Generationen bereits ein umfangreiches Sortiment Sensoren verfügbar, dennoch sind mobile Geräte in der spezifischen Ausstattung eingeschränkt. Eine erschöpfende Messung aller Parameter der Betriebsumgebung einer Software oder eines Geräts ist deshalb nicht möglich. Zudem sind auch nicht alle Parameter dieser Betriebsumgebung für eine spezifische Software relevant. Eine nicht erschöpfende Auswahl möglicher Kontextparameter wird in Abschnitt 3.1.2 diskutiert. Diese Auswahl der für eine mobile, kontextsensitive Anwendung nutzbaren Kontextparameter wird einerseits durch verfügbare Sensoren begrenzt, die physikalische Parameter der Betriebsumgebung erfassen können und andererseits durch logische Parameter, die aus mehreren physikalischen Parametern bestimmt werden oder der Software durch externe Schnittstellen zugeführt werden.

3.1.1 Begriffe

Die Begriffe Mobilität und Kontextsensitivität sind in dieser Arbeit von zentraler Bedeutung. Eine einheitliche Definition dieser Begriffe existiert jedoch bislang nicht im Kontext des Software Engineering (vgl. Jung et al. [206]). Die Gültigkeit der im Rahmen dieser Dissertation erarbeiteten Technologien zur Testautomatisierung für mobile Anwendungen basiert jedoch auf einer spezifischen Taxonomie dieser und weiterer verwandter Begriffe. In den folgenden Abschnitten werden diese Begriffe deshalb für den Geltungsbereich dieser Arbeit festgelegt und in das technische Umfeld eingebettet.

3.1.1.1 Mobilität

Der Begriff *Mobilität* (lat. mobilitas) ist das Abstraktum des Begriffs *mobil* (lat. mobilis) und bezeichnet die Eigenschaft von Dingen oder Menschen standortunabhängig, beweglich (auch im Sinne der Geisteshaltung) und transportierbar zu sein [100]. In der deutschen Sprache

entstammt der Begriff *mobil* aus dem frz. *mobile* und fand zunächst in militärischer Bedeutung Gebrauch (vgl. Mobilmachung, mobilisieren) [227], hat sich aber später im allgemeinen Sprachgebrauch auch außerhalb dieser Konnotation etabliert. Der Begriff *Mobilität* wird vom Begriff *Beweglichkeit* insofern abgegrenzt, als dass er sich neben physischen und geografischen Räumen ebenfalls auf soziale und virtuelle Räume erstreckt.

In der Softwaretechnik findet der Begriff *Mobilität* Anwendung auf alle Artefakte in Softwaresystemen. Sowohl Geräte, Personen in allen Stakeholder-Positionen als auch Programm- und Maschinencode können mobil, d.h. standortunabhängig, beweglich und transportierbar sein. Angewendet auf Geräte und Personen ist Mobilität in der Softwaretechnik eindeutig im Sinne der Beweglichkeit zu verstehen. Personen bewegen sich von einem Ort zu einem anderen Ort und führen dabei gegebenenfalls Geräte (z. B. Smartphones) mit sich. Mit der Evolution von *Mainframe*-Computertechnologie zu Desktop-Computern zu Smartphones und Tablets sind Geräte ihrerseits ebenfalls mobil geworden. Insbesondere für die letztgenannte Klasse von Geräten hat sich die Verfügbarkeit von Rechenleistung und Netzwerkkonnektivität seit der Markteinführung des iPhone im Jahr 2007 aus der Anwenderperspektive zu einer Selbstverständlichkeit entwickelt.

Für Softwareartefakte ist der Begriff *Mobilität* hingegen differenzierter zu betrachten. Ziel mobiler Software ist es, dem Anwender eine Technologie anzubieten, die es ihm ermöglicht, standortungebunden Softwarefunktionalität verwenden zu können. Solang die Funktionalität bedient werden kann, ist es zunächst einmal unerheblich, ob diese Funktionalität lokal auf dem Gerät des Anwenders erbracht wird oder über eine Netzwerkverbindung auf einem entfernten Computersystem ausgeführt wird. Aus der Perspektive des Anwenders ist eine Software also dann mobil, wenn sie sich ungeachtet der Mobilitätssituation des Anwenders standortunabhängig verwenden lässt. Sie stellt dem Anwender ihre Funktionalität an seinem jeweiligen Standort zur Verfügung.

Aus der Perspektive der Softwaretechnik kann sich Mobilität einer Software jedoch vielgestaltig manifestieren. Eine Software kann beispielsweise komponentenvollständig auf einem mobilen Gerät installiert werden und sich zur Laufzeit immobil gegenüber ihrer Laufzeitumgebung verhalten. Es ist dann eine Frage des Anwendungsfalls, ob Mobilität des Anwenders oder des Geräts Einfluss auf das Laufzeitverhalten der Software hat. Dieses Modell von Softwaremobilität wird als physikalische Mobilität bezeichnet (vgl. Carzaniga et al. [72], Picco et al. [283], Roman et al. [303], Murphy et al. [259]). Die Anwendung selbst ist immobil gegenüber ihrer Ausführungsplattform, ist aber zusammen mit ihr mobil. In dieser Hinsicht greift der Begriff *Mobilität* gemäß Definition im Universalwörterbuch der deutschen Sprache [100] zu kurz und ist nicht unverändert anwendbar. Ein Referenzsystem zur Quantifizierung der Mobilität wird durch diese Definition nicht gefordert, ist aber im Kontext der Softwaretechnik zwingend. Es ist von entscheidender Bedeutung, ob sich Software relativ zu ihrer Ausführungsplattform bewegt oder stationär, d. h. immobil, ist. Im zweiten Fall dürfen nach der Installation über bestimmte Parameter der Ausführungsplattform sichere Annahmen getroffen werden (z. B. kann eine Anwendung davon ausgehen, dass sich die Plattformarchitektur nach dem Installationszeitpunkt nicht mehr in wesentlichen Eigenschaften verändert). Die Software ist aber dennoch als mobil zu bezeichnen, weil über Schnittstellen der Ausführungsplattform zur Betriebsumgebung keine sicheren Annahmen getroffen werden dürfen. Mobilität des Geräts verursacht

Variabilität der Betriebsumgebung, wie etwa Verfügbarkeit von Netzwerkkonnektivität oder durch das Betriebsökosystem bereitgestellte Dienste (z. B. der Spracherkennungsdienst *Speech Interpretation and Recognition Interface* (Siri) in Apples Betriebsinfrastruktur oder die Google Play-Dienste [153]).

Book et al. [54, 53, 55] unterscheiden aus diesem Grund mehrere Modi der physikalischen Mobilität. Einen starken Fokus auf Interaktionen des Anwenders mit Geschäftsprozessen legt das Konzept der *Anwendermobilität*, in welchem insbesondere zwischen einem lokalen Anwender (*Local User*), der sich stets am Ort der Anwendung befinden muss, einem mobilen Anwender (*Mobile User*) und einem sich in Bewegung befindlichen Anwender (*In-motion User*) unterschieden. Zweiterer zeichnet sich dadurch aus, dass er einen Geschäftsprozess von einem entfernten Standort (relativ zur Ausführungsplattform) aus ausführen kann, während sich der in Bewegung befindliche Anwender bei der Ausführung des Geschäftsprozesses in Bewegung befinden darf. Diese Kategorien können ebenfalls auf die zur Ausführung des Anwendungsfalls verwendeten Geräte angewendet werden. Ein Gerät kann einen Anwendungsfall entfernt von der Ausführungsplattform ausführen (*Mobile Device*, [54, 53, 55]) oder es darf sich währenddessen zusätzlich in Bewegung befinden (*In-motion Device*, [54, 53, 55]). Eine Standortdifferenz zwischen ausführendem Gerät und der Ausführungsplattform kommt in der Praxis insbesondere bei Web-Anwendungen vor, bei denen das UI lokal auf dem Gerät angezeigt wird, die Anwendungslogik jedoch auf einem Server implementiert ist.

Neben der physikalischen Mobilität existiert weiterhin das Konzept der logischen Mobilität. Carzaniga et al. [72], Picco et al. [283], Roman et al. [303], Murphy et al. [259] unterscheiden in ihren Arbeiten explizit zwischen physikalischer und logischer Mobilität. Logische Mobilität bezieht sich hierbei auf nicht-physikalische Softwareentitäten wie beispielsweise Agenten (*Mobile Agent*, [283]), Programmcode oder Daten. Carzaniga et al. [72] sowie Fuggetta et al. [118] führen den Begriff *Code Mobility* ein und bezeichnen hiermit Szenarien, in denen der Ort der Ausführung von Programmcode zur Laufzeit dynamisch wechseln kann, beispielsweise zwischen unterschiedlichen Computern innerhalb eines Netzwerks (vgl. Hülder [190]). Innerhalb des Konzepts der Codemobilität unterscheiden Carzaniga et al. [72] weiterhin zwischen starker Mobilität (*Strong Mobility*) und schwacher Mobilität (*Weak Mobility*). *Strong Mobility* ermöglicht es, einen in Ausführung befindlichen Programmabschnitt zu pausieren, zu serialisieren, an einen anderen Ausführungsort zu transportieren und dort unter Wiederherstellung des Zustands fortzusetzen. *Weak Mobility* bestimmt zur Laufzeit einen Ausführungsort für einen Programmabschnitt und führt den Code dort aus. Eine Übertragung von Programmcode zur Laufzeit ist nicht vorgesehen. Stattdessen ist Voraussetzung, dass am Ort der Codeausführung alle notwendigen Ressourcen verfügbar sind.

Roman et al. [303] erkennen in der Mobilität von Anwendern und Geräten neue Anforderungen an die Robustheit von Softwaresystemen. Traditionelle Annahmen über die Erreichbarkeit eines Kommunikationspartners – der ggf. über die Ressourcen verfügt Programmcode auszuführen – sind in Mobilitätssituationen obsolet. Logische Mobilität von Programmcode ist deshalb zwingende Konsequenz aus Anwendermobilität. Die Bedürfnisse eines sich physikalisch in Bewegung befindlichen Anwenders (*In-motion User*, [54, 53, 55]) können nur befriedigt werden, wenn Anwendungsfälle durch logische Mobilität von Softwareartefakten unterstützt werden. Um von den technischen Herausforderungen logischer Mobilität zu abstrahieren, kann

Middleware zum Einsatz kommen, die technische Details für den Softwareentwickler abstrahiert (z. B. *Remote Services for Open Services Gateway Initiative* (OSGi) (rOSGi) [104]). Eine solche Middleware die speziell auf die Anforderungen physikalischer und logischer Mobilität zugeschnitten ist wurde bereits im Jahr 2001 von Murphy et al. [259] vorgestellt. Hier wurde das Konzept der mobilen Agenten (*Mobile Agent*) aufgegriffen, bei denen es sich um aktive Softwarekomponenten handelt, die Anwendungsfälle im Namen des Anwenders ausführen. Mobile Geräte werden in diesem Konzept auf Container reduziert, deren Aufgabe lediglich die Bereitstellung von Ressourcen ist.

Logische Mobilität und mobile Agenten werden in dieser Arbeit nicht explizit betrachtet. Die in dieser Arbeit untersuchte Technologie zur Testautomatisierung hat Akzeptanztests zum Gegenstand, also nur solche Artefakte eines Softwaresystems, die sich dem Anwender bei der Ausführung eines Anwendungsfalls offenbaren. Insofern abstrahiert diese Arbeit von den Konzepten physikalischer und logischer Mobilität, als dass der Standort des Programmcodes für die Testdurchführung unerheblich ist, solang gewährleistet ist, dass Faktoren der Betriebsumgebung (d. h. der Kontext, Abschnitt 3.1.1.4) berücksichtigt werden.

In dieser Arbeit haben die Begriffe *Mobilität* und *mobile Anwendung* zentrale Bedeutung. Verwendet werden sie stets für ein physikalisches Mobilitätskonzept, in welchem die Standorte von Anwendern und mindestens der Benutzungsschnittstelle der Anwendung identisch sind (*Local User*, [54]), sich Gerät und Anwender darüber hinaus aber in Bewegung befinden dürfen (*In-motion User, In-motion Device*, [54, 53, 55]). Weiterhin gilt der Begriff *mobile Anwendung* im Kontext dieser Arbeit für native Anwendungen für die jeweilig spezifische Plattform. Die Begriffe *mobile Anwendung* und App werden synonym verwendet. Web-Anwendungen, die auf mobilen Geräten verwendet werden, sind nicht Gegenstand dieser Arbeit.

3.1.1.2 Mobiles Gerät

Im Sprachgebrauch der Softwaretechnik gelten Geräte als *mobile Geräte* oder *Mobilgeräte*, wenn sie ortsunabhängige Verfügbarkeit von persönlichen Informationen, die Nutzung drahtloser Netzwerkverbindungen und Telekommunikationsdienste sowie eine geringe Baugröße ineinander vereinen. An die Baugröße wird die Anforderung gestellt, dass sie ausreichend klein ist, um das Gerät ohne körperliche Anstrengung ständig in Jacken-, Hemd-, oder Hosentasche mitzuführen (Ichikawa et al. [191], Godwin-Jones [125], Wiese et al. [361]). Smartphones und Tablets werden regelmäßig den mobilen Geräten zugeordnet, Laptops hingegen nicht, obgleich sie einige der genannten Kriterien erfüllen.

Die Kategorie der mobilen Geräte wird ergänzt durch *ultramobile Geräte*. Hierunter fallen solche Geräte, die zwar einerseits mobile Geräte sind, aber aufgrund ihrer besonderen Gestaltung in Form und Größe gezielt von Smartphones abgegrenzt werden sollen (vgl. Godwin-Jones [125]), wie etwa Smartwatches (vgl. Schmidt et al. [315]). Ziel dieser expliziten Abgrenzung ist die Alleinstellung dieser Geräte im Hinblick auf technische Eigenschaften (gegenüber Smartphones anders priorisierte, reduzierte Interaktionsmodalitäten, außerhalb der gemeinsamen Verwendung mit dem Smartphone i. d. R. stark eingeschränkte Funktion, besonders gering dimensionierte Bauform) und Vermarktungseigenschaften (*ultramobile Geräte* sollen als Ergänzung zum Smartphone wahrgenommen werden, nicht als Ersatz).

3.1.1.3 Mobile Anwendung

Der Begriff *mobile Anwendung* bezeichnet eine Anwendung, die unter Verwendung nativer Plattformtechnologien implementiert und auf einem mobilen Gerät installiert und ausgeführt wird (vgl. Muccini et al. [258]). Sie können von Anwendungen für Desktop-Systeme durch eine Reihe von Eigenschaften abgegrenzt werden. Mobile Anwendungen sind i. d. R. auf die Erfüllung einer bestimmten Aufgabe zugeschnitten und haben einen klar abgegrenzten Funktionsumfang. Besonders zeichnen sie sich durch ihre ortsunabhängige Verfügbarkeit und ihre Einbettung in die Infrastruktur einer mobilen Plattform aus und haben i. d. R. vergleichsweise uneingeschränkten Zugriff auf die Hardwarekomponenten des mobilen Geräts. Insbesondere hierdurch unterscheiden sie sich von Web-Anwendungen, die für die Verwendung auf mobilen Geräten optimiert sind. Die Begriffe *mobile Anwendung* und *App* werden synonym verwendet.

In dieser Dissertation schließt die Definition des Begriffs *mobile Anwendung* mobilisierte Web-Anwendungen ausdrücklich aus. Zwar überschneiden sich die Eigenschaften mobiler Anwendungen und mobilisierter Web-Anwendungen in einigen Aspekten. Dennoch sind sie in ihren grundsätzlichen technologischen Grundlagen und in ihrer Architektur so verschieden, dass die in dieser Dissertation untersuchten Konzepte zur Testautomatisierung mobiler Anwendungen nicht auf mobilisierte Web-Anwendungen übertragbar sind.

Alternativ zur Verwendung nativer Plattformtechnologien können mobile Anwendungen auch durch Verwendung von *Cross-Platform*-Technologie implementiert werden. Die hier verwendete Begriffsdefinition *mobile Anwendung* schließt solche Anwendungen ein, die von der verwendeten Cross-Platform-Technologie zu Code kompiliert werden, der in der nativen Laufzeitumgebung ausgeführt wird, nicht jedoch solche, die durch andere Laufzeitumgebungen von der nativen Laufzeitumgebung entkoppelt sind.

3.1.1.4 Kontext

Der Begriff *Kontext* findet in der Softwaretechnik vielfältige Verwendung und hat insbesondere im Spannungsfeld der Entwicklung mobiler Anwendungen einen divergenten Deutungsumfang ausgebildet. Das Universalwörterbuch der deutschen Sprache [100] definiert den Begriff *Kontext* (aus dem lat. *contextus*) als eine enge Verknüpfung bzw. einen engen Zusammenhang. Diese Definition zielt zunächst primär auf den *Kontext der Rede* ab, also auf die inhaltlich sinnhafte Verknüpfung von Wörtern und Sätzen. Das Etymologische Wörterbuch der Deutschen Sprache [227] führt den Begriff *Kontext* als neoklassische Bildung aus dem Begriff *Text* und der lat. Präposition *con-* (lat. zusammen, gemeinsam, verstärkend). Der Begriff *Text* wiederum entstammt dem lat. *textus* mit der eigentlichen Wortbedeutung *Gewebe (der Rede)*, mit zugehörigem Verb *textere*, lat. weben, flechten, kunstvoll zusammenfügen, verwandt mit dem altgr. τέχνη (*téchnē*), Handwerk, Kunst, Fertigkeit, Wissenschaft [100, 227]. Diese ursprüngliche Wortbedeutung des *sprachlichen Kontext* hat weitere Formen der Bedeutung erhalten, wie etwa den *persönlichen* und den *sozialen Kontext*, der Beziehungen zwischen Akteuren und ihrem Wissen bzw. wechselseitigen Wissensannahmen einbezieht (Bußmann [66], Lexikon der Sprachwissenschaft), also konkrete Bezugsentitäten benennt.

Im Umfeld Softwaretechnik ist von dieser Wortbedeutung der Aspekt des engen Zusammenhangs bzw. einer engen Verknüpfung zwischen Entitäten erhalten geblieben. Bezugsenti-

täten sind hier nun nicht mehr nur Elemente der Sprache oder menschliche Akteure, sondern Entitäten der Softwaretechnik oder Entitäten der Betriebsumgebung, die auf ein Softwaresystem einwirken. Der *technische Kontext* beschreibt also jede Art der Information, die Interaktionen von Softwareentitäten mit ihrer Umgebung charakterisieren (vgl. Abowd et al. [3]). Diese Entitäten können Personen, Orte, physikalische Objekte oder Softwareobjekte sein. Im Umfeld Software in mobilen Geräten schließt der Begriff *Kontext* insbesondere alle Umgebungsfaktoren ein, die durch ein Gerät unter Verwendung von Sensoren messbar sind oder einem Gerät bzw. einer Software durch andere Mittel verfügbar werden (vgl. Schilit et al. [311], Brown et al. [61], Schmidt et al. [315]).

Das Umfeld mobiler Geräte stellt sich stark heterogen dar (vgl. Grønli et al. [158]). Mobile Geräte verfügen i. d. R. über ein Repertoire unterschiedlicher Sensoren. Es existiert in der Industrie jedoch kein Konsens über konkrete Mindestausstattung mobiler Geräte mit spezifischen Sensoren. Zudem ist die Betriebsumgebung für ein mobiles Gerät auch über Technologien wahrnehmbar, die nicht unmittelbar Bauteil des Geräts sind (z. B. fitbit™-Schrittzähler [115], eine als Armband ausgeführte Perpherietechnologie, die ihre Funktion nur in Symbiose mit einem Smartphone vollständig erfüllen kann). Eine erschöpfende Aufzählung von Kontextcharakteristika ist deshalb weder möglich noch sinnvoll.

Um Kontext in der Softwaretechnik greifbar zu machen, bedarf es dennoch einer anwendbaren Begriffsdefinition, die Entscheidungen darüber zulässt, ob eine Entität einer Betriebsumgebung einer Software zu deren Kontext gehört. Schilit et al. [311] definieren drei wesentliche Aspekte des Begriffs *Kontext* als:

> „*where you are, who you are with, and what resources are nearby*“ [311]

Wesentliche Kontextmerkmale sind also der Standort des Anwenders, dessen soziales Umfeld sowie nahegelegene verfügbare Ressourcen, bei welchen es sich jedoch gerade nicht ausschließlich um Ressourcen im softwaretechnischen Sinn (z. B. Speicher, Taktgeschwindigkeit eines Prozessors) handelt. Im Anschluss wird diese Definition durch die Aufzählung konkretisiert:

> „*context includes lighting, noise level, network connectivity, communication costs, communication bandwidth, and even the social situation*“ [311]

Mit Ausnahme der sozialen Situation, handelt es sich hierbei um technische Parameter, die durch Verwendung geeigneter Sensoren messbar gemacht werden können.

Schmidt et al. [315] argumentieren, dass aufgrund der unterschiedlichen Natur dieser Faktoren eine weitere Klassifikation dem besseren Begriffsverständnis dienlich sei. Sie entwerfen deshalb ein Kontextmodell in dem explizit zwischen menschlichen Faktoren und physikalischen Eigenschaften der Umgebung unterschieden wird. Hierdurch wird insbesondere eine weitere Begriffsdifferenzierung möglich, die Aussagen über konkret betrachtete und relevante Kontextparameter zulässt.

Dey [92] hält die Definition von Schilit et al. [311] ebenfalls für zu kurz gegriffen und stellt seinerseits eine erweiterte Definition vor:

> „*Context is any information that can be used to characterise the situation of an entity. An entity is a person, place, or object that is considered relevant to the*

> *interaction between a user and an application, including the user and applications themselves.*“ [92]

In dieser Definition sind die drei Aspekte der Definition von Schilit et al. [311] immernoch enthalten, allerdings wird sie auf allgemein für Software relevante Objekte erweitert. Diese Erweiterung ist insofern sinnvoll, als dass nun auch nicht-technische Parameter der Betriebsumgebung eingeschlossen werden, die nicht lokal auf einem mobilen Geräte messbar sind, andererseits jedoch für einen Anwender und insbesondere Interaktion mit anderen Anwendern relevant sein können.

Wird der Begriff *Kontext* in dieser Arbeit verwendet, ist damit die Gesamtheit aller mittelbar oder unmittelbar zur Steuerung des Kontroll- oder Datenflusses einer mobilen Anwendung verfügbaren Informationen gemeint. Dies schließt situative Faktoren des Anwenders ebenso mit ein, wie lokal auf dem Gerät erhobene Sensormesswerte, die Ausführungsplattform oder Informationen, die der Anwendung von außerhalb zugeführt werden.

Für das Testen mobiler Anwendungen kann der Kontext einer mobilen Anwendung zwar in seinem vollen Umfang relevant sein. Durch Einschränkungen der technischen Testwerkzeuge können jedoch nicht alle Kontextfaktoren beim Testen berücksichtigt werden. Diese Arbeit hat eine Testautomatisierungstechnologie für kontextsensitive, mobile Anwendungen zum Ziel. Die für dieses Vorhaben relevanten Kontextparameter fokussieren deshalb primär auf die Kategorie der physikalischen Betriebsumgebung nach Schmidt et al. [315], ergänzt um einige logische Kontextparameter, und werden in Abschnitt 3.1.2 detailliert diskutiert.

3.1.1.5 Kontextsensitivität

Im vorangegangen Abschnitt 3.1.1.4 wurde der Begriff *Kontext* in seiner Bedeutung für diese Arbeit erläutert. Software kann auf Veränderungen ihres Kontexts reagieren und Kontextvariablen so zur Steuerung ihres Kontroll- bzw. Datenflusses verwenden. Hierzu muss Software Eigenschaften ihrer Betriebsumgebung, d. h. ihren Kontext, analysieren und geeignete Rückschlüsse zur Steuerung des Programmablaufs ziehen. Diese Analyse der Betriebsumgebung ist ein Prozess, bei dem Umgebungsparameter durch aktive oder passive lokale Sensoren oder durch externe Quellen überwacht werden. Aus der Art der Veränderung der Messwerte über den Zeitraum der Überwachung hinweg können dann durch geeignete Techniken (z. B. Mustererkennung, algorithmische Schlussfolgerung) Kontroll- und Datenflüsse entsprechend gesteuert werden. Software, die ihre Betriebsumgebung in einer solchen Weise beobachten und neben Aktionen des Anwenders Stimuli des Kontexts als zusätzliche Eingabemodalität verwendet, wird als *kontextsensitive* (bzw. *Context-aware Software* im angloamerikanischen Sprachraum) Software bezeichnet.

Der Begriff *Context-aware Computing* (d. h. kontextsensitive in Software) wurde in der wissenschaftlichen Literatur erstmals von Schilit und Theimer [312] im Jahr 1994 in diesem Zusammenhang eingeführt. Er beschreibt Software, die in der Lage ist, Veränderungen in der Umgebung des Anwenders wahrzunehmen und darauf zu reagieren. Im Jahr 1995 führte Schilit im Rahmen einer Dissertation [313] diese Überlegungen weiter zu einem Architekturvorschlag für kontextsensitive Softwaresysteme. Hier wird Kontextsensitivität in Softwaresystemen charakterisiert als:

> „*[. . .] application adaptation triggered by such things as the location of use, the collection of nearby people, the presence of accessible devices and other kinds of objects, as well as changes to all these things over time.*“ [313]

Entscheidendes Kriterium kontextsensitiver Software ist also, dass der Kontext einer Anwendung Einfluss auf die Steuerung von Kontroll- und Datenflüssen in Anwendungen hat. Kontext wird also zu einem Eingabevektor neben Interaktionen des Anwenders mit der Software (vgl. Broens und van Halteren [59]).

Aufgrund des Standes der technischen Entwicklung zum Ende der 1990er Jahre waren kontextsensitive Anwendung in ihrer Leistungsfähigkeit stark eingeschränkt. Mobile Geräte hatten bei weitem nicht die Leistungsfähigkeit und Vielseitigkeit der Smartphone-Generationen der iPhone-Ära. Die Verfügbarkeit von GPS-Technologien war ein Begeisterungsfaktor (d. h. ein Nutzen stiftendes Merkmal nach Kano-Modell, Kano et al. [211]) und keineswegs selbstverständlich. In der Folge konzentrierte sich die wissenschaftliche Betrachtung kontextsensitiver Anwendungen zunächst primär auf den Standort des Anwenders. Dies veranlasste Schmidt et al. [315] im Jahr 1999 zu einer differenzierteren Betrachtung von Kontextsensitivität. Insbesondere argumentieren Schmidt et al. [315], dass der Standort des Anwenders allein noch kein sinnvoll zu verwendender Kontextparameter ist, sondern erst durch Einbettung in einen Anwendungsfall eine spezifische Bedeutung erhält. Hierzu müssen die Standortinformationen um weitere Kontextparameter ergänzt werden. Dieses Modell bezeichnen Schmidt et al. [315] als *Sensor Fusion*, also als Aggregation unterschiedlicher Sensoren und sonstiger Kontextquellen, um eine algorithmisch verwertbare Situationsbeschreibung zu berechnen.

Dieses Modell der Kontextsensitivität wurde in den folgenden Jahren von unterschiedlichen Wissenschaften unter weiteren Perspektiven untersucht (z. B. Cheverst et al. [77], Roman et al. [303], Dey [92], Beigl et al. [37], Judd und Steenkiste [205], Becker und Dürr [35], Henricksen und Indulska [177]), ohne dass wesentlichen Ergänzungen oder neue Perspektiven aufgetan wurden. Stattdessen fokussierten sich Wissenschaftler auf die Nutzbarmachung von Kontext in (mobilen) Anwendungen durch unterschiedliche Middleware-Konzepte (z. B. Hülder [190]).

Indes unterliegt der Markt mobiler Kommunikationstechnologie einem ständigen Wandel. Insbesondere ist die Technologielandschaft gekennzeichnet durch stark ausgeprägte technologische Alleinstellungsmerkmale weniger dominanter Akteure (z. B. Apple, Google). Es konnte sich deshalb keines der vorgeschlagenen Frameworks durchsetzen, so dass die Implementierung von Kontextsensivität in Anwendungen immernoch ein für jede Anwendung individuell zu lösendes Problem ist (vgl. Ayed et al. [21]).

Der Begriff *Kontextsensitivität* wird in dieser Arbeit verwendet, um solche Softwaresysteme zu beschreiben, die Kontextparameter verwenden, um das Verhalten der Anwendung auf die jeweilige Situation des Anwenders anzupassen. Hierzu werden Kontextparameter als gleichrangige Eingabevektoren neben der direkten Manipulation eines UI durch einen Anwender behandelt. Eine mobile Anwendung heißt *kontextsensitiv*, wenn sie durch Beobachtung ihrer mittelbaren und unmittelbaren Umgebung, beispielsweise durch Verwendung von Sensoren oder anderer geeigneter technischer Maßnahmen in der Lage ist, wesentliche situative Faktoren – also ihren *Kontext* – algorithmisch zu interpretieren und zur Steuerung des Kontroll- und Datenflusses nutzbar zu machen.

3.1.2 Kontextparameter

Die Betriebsumgebung einer mobilen Anwendung, d. h. ihr Kontext, ist eine Aggregation einzelner Parameter, die in ihrer Gesamtheit auf eine kontextsensitive Anwendung einwirken. Hierzu gehören sowohl physikalische Merkmale der Betriebsumgebung wie etwa Temperatur, Lichtintensität oder Magnetfelder, physikalische Merkmale des Geräts wie z. B. lineare Beschleunigung oder Winkelgeschwindigkeit, als auch situative und soziale Faktoren des Anwenders. Die Menge potenziell relevanter Kontextparameter ist nicht beschränkt, eine erschöpfende Auflistung ist daher nicht möglich und für die in dieser Dissertation untersuchte Methode zur modellbasierten Generierung von Testfällen nicht sinnvoll.

In der Softwaretechnik ergeben sich hierdurch eine Reihe von Herausforderungen bei der Entwicklung kontextsensitiver Anwendungen. Zum einen ist bei der Spezifikation von Softwaresystemen die Fragestellung zu lösen, wie Kontextsensitivität in einem Spezifikationsdokument zu fixieren ist. Bereits im einfachen Fall einer ortsbasierten Anwendung – als Sonderfall kontextsensitiver Softwaresysteme – ist für den konkreten Anwendungsfall zu definieren, was mit dem Standort des Anwenders geschehen soll, wie der Standort des Anwenders ermittelt werden soll (z. B. via GPS oder basierend auf der Topologie zellulärer Kommunikationsnetzwerke), in welcher Granularität die Information benötigt wird (WGS84-Koordinaten oder in Form einer Wohnanschrift), wie sie softwaretechnisch durch Datenstrukturen abzubilden ist, in welcher Häufigkeit die Information abgefragt werden soll und wie in Fehlersituationen mit einer ungültigen Positionsbestimmung verfahren werden soll. Für andere messbare Kontexteigenschaften ergeben sich ähnliche Fragestellungen (z. B. Maßeinheit, Referenzsystem usw.). Diese Fragen werden bei der Implementierung von Software erneut aufgegriffen, wenn ein Programmierer vor der Aufgabe steht, kontextsensitive Funktionalität einer Anwendungsspezifikation in Code zu realisieren. Es ist beispielsweise festzulegen, welche API oder welches Rahmenwerk verwendet werden soll oder wie die Transformation technischer Fehler zu fachlichen Fehlern erfolgen soll. Beim Testen ergibt sich anschließend die Aufgabe, ein SUT reproduzierbar mit Testdaten zu versorgen, die situative Faktoren, d. h. den Kontext, adäquat repräsentieren (vgl. Broens und van Halteren [59]).

Um diese Fragestellungen zu adressieren ist es notwendig, Kontextparameter zu klassifizieren und zu charakterisieren. Abowd et al. [3], Dey [92, 93] klassifizieren in ihren Arbeiten Kontextparameter. Sie unterscheiden hierbei drei Entitäten: Orte (Koordinaten, Adressen, Gebäude, Räume), Menschen (Individuen, Gruppen) und Dinge (physikalische Objekte, technische Komponenten). Nach Baldauf et al. [25] in Anlehnung an Abowd et al. [3] und Dey et al. [92, 93] kann jede dieser Entitäten durch eine Reihe von Attributen aus vier Kategorien beschrieben werden: Identität, Ort, Status und Zeit. Dabei liegt die Annahme zugrunde, dass jede Kontextentität eine eindeutige Identität besitzt, sie also widerspruchsfrei adressiert werden kann. Weiterhin kann jeder Kontextentität ein Ort zugeordnet werden. Diese Zuordnung kann absolut oder relativ zu anderen Entitäten erfolgen. Das heißt, einer Kontextentität kann ein absoluter Standort in einem Referenzsystem zugeordnet werden (z. B. WGS84-Koordinaten) oder sie kann einer anderen nahegelegen sein. Der Status einer Kontextentität bezieht sich auf intrinsische Eigenschaften des betrachteten Kontextparameters. Hierdurch werden die wesensbestimmenden Eigenschaften einer Kontextentitäten definiert,

wie etwa ausgeübte Aktivität (Menschen) oder die gemessene Temperatur (physikalische Objekte oder technische Komponenten). Das vierte Attribut ist Zeit, welches eine chronologische Ordnung für Kontextentitäten definiert. Die Klassifikation nach Baldauf et al. [25] lässt also beispielsweise einen Kontext zu, der sich aus einer Person definiert, die sich zu einer spezifischen Zeit an einem spezifischen Ort befindet und in Bezug steht zu einer physikalischen Entität, beispielsweise einem mobilen Gerät, dessen Sensoren spezifische Messwerte bestimmter Messgrößen erzeugen. Ändert sich eines dieser Attribute, ändert sich der Kontext. Diese Klassifikation lässt beliebig granulare Kontextbeschreibungen zu. Eine vollständige Kontextbeschreibungen definiert für jedes Attribut einer Kontextentität Werte, die in ihrer Gesamtheit den Kontext eindeutig definieren. Durch unvollständige Wertebestimmung können so ebenfalls gröbere Kontexte definiert werden, beispielsweise durch Weglassen einer Spezifikation der Zeit oder eines Ortes. Die Kontextdeklaration einer Person ohne nähere Spezifikation weiterer Attribute definiert beispielsweise die Existenz dieser Person als Kontext. Durch hinzufügen einer Ortsangabe kann dieser Kontext so erweitert werden, dass er im Rahmen einer ortsbasierten Anwendung verwendet werden kann. Baldauf et al. [25] verwenden hierzu die Terminologie des *Context Atoms* (auch Korpipää und Mäntyjärvi [229]), um solche Kontextattribute zu bezeichnen, die nicht feiner strukturiert werden können. Durch Aggregation bzw. Kompositionen werden Kontextatome zu komplexem Kontext kombiniert.

Das von Baldauf et al. [25] diskutierte Modell der strukturellen Hierarchie von Kontextparametern wurde bereits von Schmidt et al. [315] adressiert. Hier wird Kontext als Situation und Umgebung eines Akteurs oder Geräts definiert, für die eine diskrete Menge individueller Kontextparameter relevant ist. Für jeden Parameter wird implizit oder explizit ein Wertebereich festgelegt, der für den jeweiligen Kontext wesensbestimmend ist. Schmidt et al. [315] klassifizieren Kontextparameter auf der ersten Ebene zwischen menschlichen Faktoren und der physikalischen Umgebung. Innerhalb dieser beiden Klassen können relevante Parameter bestimmt werden, deren Ausprägung den Kontext bestimmen (vgl. Abbildung 3.1).

Kontextparameter der Kategorie menschliche Faktoren werden wiederum in drei Klassen unterschieden: Informationen über den Akteur (z. B. Gewohnheiten, Emotionen, biologische Faktoren), Informationen über die soziale Einbettung des Akteurs (z. B. Gruppenzugehörigkeit, räumliche Nähe zu anderen Akteuren) und Aktivitäten des Akteurs (ausgeübte Tätigkeit und deren Ziele und Zusammenhang mit anderen Aktivitäten). Die physikalische Umgebung wird ebenfalls in drei Kategorien unterteilt: Ort (absoluter Standort bezüglich eines Referenzsystems, relativer Standort bezüglich einer anderen Entität), Infrastruktur (verfügbare Ressourcen, Kommunikationsnetzwerke, usw.) und physikalische Beschaffenheit der Umgebung (Licht, Magnetfelder, Geräusche, usw.). Um den Kontext für praktische Zwecke nutzbar zu machen, gilt es relevante Parameter zu identifizieren und solche Wertebereiche zu bestimmen, die eine Abgrenzung unterschiedlicher Kontextsituationen ermöglicht. Schmidt et al. [315] beziehen hierbei insbesondere die historische Entwicklung von Kontextparametern mit ein, d. h. die Identifikation des zukünftigen Kontext ist abhängig vom vorhergehenden Kontext, also der Veränderung der Werte individueller Parameter über die Zeit.

Ein weiteres Klassifikationsschema von Kontextentitäten wird von Prekop und Burnett [287] vorgeschlagen. In ihrem Ansatz gehen Prekop und Burnett von einem aktivitätszentrierten Kontextmodell aus, in welchem kontextbestimmende Faktoren nicht (physikalische) Eigen-

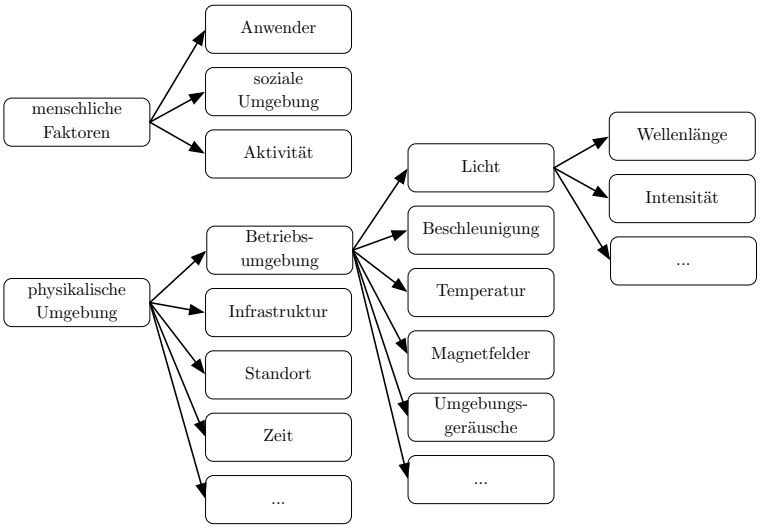


Abbildung 3.1: Hierarchisches Modell von Kontextparametern nach Schmidt et al. [315]

schaften der Betriebsumgebung sind, sondern Aktivitäten, die von Menschen oder Software-Agenten ausgeführt werden. Es ist hier die Verbindung von Agent und Aktivität, durch welche der Kontext verursacht wird. Das heißt, der Kontext existiert erst, wenn ein Akteur eine Tätigkeit ausübt. Erst in diesem Moment werden auch Parameter der Betriebsumgebung zu relevanten Kontextparametern. Aktivitäten sind ggf. nicht atomar und lassen sich in weitere Aktivitäten zerlegen. In diesem Fall wird die übergeordnete Aktivität selbst zum Kontext untergeordneter Aktivitäten. Kontext wird somit nach Prekop und Burnett zu einem hierarchischen Konstrukt, in dem die Bedeutung einzelner Parameter durch die Einbettung in einen übergeordneten Kontext festlegt wird.

Wiederum ein parameterzentriertes Kontextmodell diskutieren Hofer et al. [182]. Hier wird der Kontext einer Anwendung im Wesentlichen durch physikalische Umgebungsparameter bestimmt, wenngleich das vorgeschlagene Modell um andere Parameter erweiterbar ist. Hofer et al. stellen hierbei den Verwendungszweck von Kontextinformationen in den Vordergrund. Ziel ist es, durch eine geeignete Architektur die Voraussetzungen für Software zu schaffen, die Softwaredienste nicht nur zu jeder Zeit, an jedem Ort und mit jedem geeigneten Medium bereitstellen, sondern insbesondere „*das Richtige* zur *richtigen Zeit* auf die *richtige Art*“ [182] bereitzustellen. Weil hierbei davon auszugehen ist, dass *das Richtige* nicht ausschließlich lokal auf einem mobilen Geräte bestimmbar ist, sondern hierzu der Kontext anderer Entitäten berücksichtigt werden muss, unterscheiden Hofer et al. physikalischen und logischen Kontext. Physikalische Kontextparameter umfassen hierbei solche, die lokal und direkt durch Sensoren ermittelt werden können. Sie entsprechen in dieser Hinsicht den Kontextatomen der Klassifikation nach Baldauf et al. [25] sowie Korpipää und Mäntyjärvi [229]. Weiterhin unterscheiden Hofer et al. zwischen lokalem und entferntem Kontext. Ausschlaggebend hierfür ist, ob Kontextinformationen lokal auf einem mobilen Gerät ermittelt werden können oder ob zu einer eindeutigen Interpretation auch Informationen aus anderen Quellen notwendig sind. Beispielsweise könnte ein mobiles Gerät, das selbst nicht über einen Temperatursensor verfügt aufgrund der Temperaturinformation eines sich in der Nähe befindlichen Geräts Rückschlüs-

se auf die eigene Betriebsumgebung ziehen. Bei diesem Vorgehen können spezifische Kontextinformationen durch andere Kontextinformationen substituiert werden. Voraussetzung ist hierfür jedoch eine funktionierende Netzwerkverbindung, die eine Kommunikation von Kontextinformationen zwischen beteiligten Entitäten erlaubt sowie eine geeignete Middleware, die es beteiligten Akteuren erlaubt, solche Informationen miteinander zu teilen.

Chen et al. [75] bewerten die bis zum Jahr 2003 vorgestellten Konzepte zur Berücksichtigung von Kontext in Software als nicht ausreichend auf Kollaboration mehrerer Entitäten ausgerichtet. Deshalb entwerfen die Autoren in ihrem Ansatz eine Kontextontologie mit dem Ziel, den Kontextbegriff über unterschiedliche Anwendungen hinweg zu homogenisieren. Die Autoren bewerten eine Ontologie als geeignetes Mittel, da sie Kontextparameter mit gegenseitigen Abhängigkeiten losgelöst von spezifischen Technologien modelliert. Ähnlich der von Hofer et al. [182] vorgestellten Middleware verwenden Chen et al. eine zentrale Komponente zur Aggregation von Kontextinformationen. Diese wird jedoch nicht lokal auf einem Gerät gepflegt, sondern auf einem dedizierten Server, der über Netzwerkschnittstellen erreichbar ist. Hierdurch ergibt sich der Vorteil, dass diese sogenannte *Context Knowledge Base* (engl. Kontextwissensbasis) universell über Gerätegrenzen hinweg auch in heterogenen Umgebungen einsetzbar ist. Nachteilig ist hingegen die Notwendigkeit einer Netzwerkverbindung, bei deren Ausfall Kontextinformationen insgesamt nichtverfügbar werden. Dieser Ansatz aus dem Jahr 2003 trug der geringen Leistungsfähigkeit der zu dieser Zeit verfügbaren mobilen Geräte Rechnung. Hintergrund war eine Verlagerung des rechenintensiven Schlussfolgerns auf der Kontextontologie auf ein leistungsfähiges Gerät. Als Besonderheit des von Chen et al. vorgestellten Ansatzes ist hervorzuheben, dass neben dem Standort des Geräts oder des Anwenders keine weiteren physikalischen Kontextparameter berücksichtigt wurden, wohl aber ein Aktivitätskontext ähnlich dem von Prekop und Burnett [287] vorgestellten Konzept.

Korpipää und Mäntyjärvi [228, 229] halten die Praxis, den Kontext einer Softwareanwendung auf einzelne messbare physikalische Eigenschaften der Betriebsumgebung einer Software zu reduzieren ebenfalls für nicht ausreichend. Informationen, die in primitiven Datenformaten aus einzelnen Quellen entnommen werden können und nicht weiter in kleinere Einheiten zerlegt werden können, werden auch bei Korpipää und Mäntyjärvi als *Context Atoms* (engl. Kontextatome) bezeichnet (vgl. Baldauf et al. [25]). Jedes einzelne Kontextatom kann je nach Anwendungsfall bereits eine nützliche Information sein oder erst die Kombination mehrerer Kontextatome. Das Zusammenwirken mehrerer Kontextatome zu einem spezifischen Zeitpunkt formt den Kontext, welcher sich beispielsweise als Eintritt eines Kontextereignisses manifestiert. Durch hierarchische Beziehungen von Kontextatomen können abstrakte Kontextbeschreibungen erzeugt werden. Korpipää und Mäntyjärvi entwerfen in ihrer Arbeit eine Kontextontologie, die die Verwendung von Sensordaten in mobilen Anwendungen zur algorithmischen Interpretation des Kontext nutzbar machen soll. Zu diesem Zweck definieren die Autoren eine Reihe Attribute, die ein Kontextatom charakterisieren (z. B. Kontexttyp, numerischen Wert, Konfidenz, Quelle, usw.). Insbesondere unterscheiden die Autoren zwischen zwei Kategorien des Attributs Kontexttyp. In die erste Kategorie fallen alle Kontextatome, die zu jedem Zeitpunkt gültig quantisierbar sind. Zu diesen gehören z. B. Temperatur oder Beleuchtung. Zu diesen Kontextatomen ist – vorbehaltlich Defekten des Messgeräts – stets ein eindeutiger, gültiger Wert zuordenbar. Jeder mögliche Wert eines solchen Attributs defi-

niert dann jeweils ein Kontextatom. In die zweite Kategorie von Kontexttypen fallen solche Parameter, die zu diskreten Zeitpunkten ungültig oder nichtexistent sein können.

Das in dieser Dissertation erarbeitete Konzept zum automatisierten Testen mobiler, kontextsensitiver Anwendungen setzt ein geeignetes Kontextmodell voraus. Um die Anforderungen an ein Metamodell zur Modellierung von Kontextinformationen in mobilen Anwendungen zu entwerfen, gilt es zunächst zu analysieren, welche Kontextparameter im Rahmen des Testens relevant sind. Zum einen ergibt sich die Relevanz spezifischer Kontextparameter aus dem Anwendungsfall. In dieser Hinsicht ist die Menge und Art relevanter Kontextparameter nicht beschränkt. Zum anderen müssen allerdings die technischen Möglichkeiten mobiler Geräte berücksichtigt werden. Hierdurch wird die Menge und Art von Kontextparametern auf solche Eigenschaften der Betriebsumgebung beschränkt, die von einem mobilen Gerät direkt oder indirekt ermittelt werden können.

In dieser Dissertation wird ein parameterzentriertes Kontextmodell vorausgesetzt, in welchem sich der (komplexe) Kontext einer mobilen Anwendung aus einer Menge atomarer physikalischer oder logischer Kontextparametern zusammensetzt. Physikalische Parameter umfassen alle Eigenschaften der Betriebsumgebung, die durch Sensoren direkt oder indirekt messbar sind. Fachliche Parameter beziehen sonstiges Wissen über die Situation des Anwenders mit ein. In den folgenden Abschnitten werden wesentliche Kontextparameter, ihre Eigenschaften und mögliche Einsatzszenarien diskutiert. Eine Diskussion ihrer von Auswirkungen auf den Betrieb von Software erfolgt in Abschnitt 3.2. Die Auswahl der betrachteten Kontextparameter ist nicht erschöpfend, sondern orientiert sich an den technischen Möglichkeiten zeitgenössischer Smartphone-Generationen.

3.1.2.1 Physikalische Kontextparameter

Physikalische Kontextparameter repräsentieren im Vergleich zu logischen Kontextparametern den einfachen Fall der Datenerhebung und Interpretation. Sie umfassen Informationen über die Betriebsumgebung einer Software, die durch direkte oder indirekte Messung erfasst werden (z. B. lineare Beschleunigung) oder axiomatisch als gegeben vorausgesetzt werden dürfen (z. B. Zeit). Wenn ein Gerät die technische Möglichkeit bietet, einen physikalischen Parameter der Betriebsumgebung zu ermitteln, so sind mögliche Zweifel an der Gültigkeit dieser Information auf bauartbedingte Eigenschaften der Messeinrichtung zurückzuführen. Im Vergleich hierzu besteht bei logischen Kontextparametern die Möglichkeit, dass Zweifel an der Gültigkeit einer Information durch Spielraum bei der Interpretation der Daten möglich ist.

Ein einzelner Sensor ist entweder defekt oder er liefert Daten mit einer bekannten, bauartbedingten Präzision. Fehler bei der Interpretation dieser Daten sind auf unsachgemäße Verwendung zurückzuführen. Ein Magnetfeldsensor beispielsweise misst magnetische Flussdichte im Umfeld des Sensors. Mutmaßungen über die Quelle des Magnetfelds können auf Grundlage der Messdaten nicht angestellt werden. Anderes verhält es sich, wenn zusätzlich der Standort des Anwenders bekannt ist und sind über diesen Standort Informationen zu möglichen Quellen von Magnetfeldern bekannt sind. Ebensowenig ist es möglich, allein aus den Messdaten eine Semantik zur Verwendung dieser Daten in einer mobilen Anwendung abzuleiten. Diese Aufgabe ist in der Anwendungslogik zu lösen und repräsentiert eine Kernanforderung der Softwaretechnik für kontextsensitive Systeme.

3.1.2.1.1 Plattform und Gerät

Voranging technische aber dennoch signifikante Kontextparameter, die maßgeblichen Einfluss auf eine mobile Anwendung haben, sind Plattform und Gerät. Die verschiedenen mobilen Plattformen diktieren Entwicklern weitreichend die zur Entwicklung zu verwendenden Technologien, wie etwa Programmiersprachen und Frameworks, die über Plattformgrenzen hinweg nur in Ausnahmefällen identisch sind. In der Konsequenz entstehen in der Entwicklung einer App für mehrere Plattformen im Normalfall mehrere Softwareartefakte, d. h. jeweils ein für das Deployment geeignetes Kompilat pro Plattform. Das gilt selbst dann, wenn zur Entwicklung Cross-Platform-Technologien verwendet werden. Aus fachlicher Perspektive hingegen wird eine App als monolithisch im Bezug zu Plattformgrenzen aufgefasst, d. h. Gegenstand eines Entwicklungsprojekts ist eine App, die mehrere Plattformen unterstützt, wenngleich das in der Praxis die Implementierung unterschiedlicher Kompilate bedeutet.

Zur Laufzeit ist es für eine Anwendung hingegen durchaus von Bedeutung auf welcher konkreten Plattform, in welcher Betriebssystemversion und auf welchem spezifischen Gerät sie ausgeführt wird. Die APIs moderner mobiler Plattformen abstrahieren einige spezifische Geräteeigenschaften zwar i. d. R. (z. B. bereichsweise Klassifikation der Pixeldichte eines spezifischen Displays und automatisierte Auswahl einer zugehörigen Grafikressource). In einigen Fällen muss der Entwickler dennoch Unterscheidungen im Quellcode treffen, um beispielsweise eine bestimmte Funktionalität auf eine spezifische Geräteklasse zu limitieren oder in ihrer Realisierung den konkreten technischen Möglichkeiten des Geräts anzupassen.

Aus der Perspektive einer mobilen Anwendung sind Plattform und Gerät also als Kontextparameter zu betrachten, da dieser Kontext die Verfügbarkeit einiger Anwendungsfunktionen (z. B. Art und Anzahl vorhandener Kameras, Anzahl verschiedene Sensoren) u. U. einschränkt. Die Anwendung Google Kamera [151] beispielsweise enthält die Funktion *PhotoSphere* zum Aufnehmen von 360°-Panoramaaufnahmen. Diese Funktion ist jedoch nur auf Geräten verfügbar, die über ein Gyroskop verfügen. Die Plattform Android fordert jedoch nicht von Geräteanbietern, alle Geräte mit diesem Sensor auszustatten. In der Konsequenz muss im Quellcode unterschieden werden, ob ein Gyroskop verfügbar ist und der Kontrollfluss der Anwendung muss entsprechend gesteuert werden. Die App passt ihr spezifisches Laufzeitverhalten im Kontext Gyroskopverfügbarkeit an.

Die exemplarisch in dieser Dissertation verwendeten Beispielanwendungen Mobiler Taxiruf und AR Tourist Information sind jeweils sensitiv gegenüber den Kontextparametern Plattform und Gerät. Beide Anwendungen sind nur für die Plattform Android konzipiert, so dass die Unterscheidung der Plattform bereits durch die Infrastruktur des Plattformbetreibers realisiert wird. Beide Anwendungen verwenden das GPS-Modul. Für AR Tourist Information ist die Verwendung von GPS obligatorisch, es sollte deshalb unterbunden werden, dass die App auf Geräten, die nicht über ein GPS-Modul verfügen, installiert werden kann. Dies kann beispielsweise durch eine entsprechende statische Konfiguration des sogenannten Android-App-Manifests erfolgen und repräsentiert eine Selektion möglicher Zielgeräte anhand der Kontextparameter Gerät und Plattform zum Entwicklungszeitpunkt. Mobiler Taxiruf hingegen bietet alternativ zur Verwendung von GPS eine manuelle Adresseingabe an. Die App soll deshalb auch auf Geräten installiert werden können, die nicht über ein GPS-Modul verfügen.

Die Unterscheidung, ob GPS verfügbar ist, geschieht deshalb zur Laufzeit. Für AR Tourist Information ist zudem neben dem Vorhandensein einer rückwärtigen Kamera die Größe und Auflösung des Displays zu berücksichtigen, da die Positionierung der Kennzeichnung von POIs als Überlagerung des Live-Bildes der Kamera auf Koordinaten des Displays abgebildet werden.

Die Kontextparameter Gerät und Plattform sind im Umfeld mobiler Anwendungen insofern von besonderer Bedeutung, als dass sie in jeder mobilen Anwendung Einfluss haben, wenngleich dieser Einfluss sich dem Anwender nicht offensichtlich darstellt. Mobile Plattformen bieten im Normalfall ein heterogenes Spektrum an Geräten mit unterschiedlichen Eigenschaften (vgl. Grønli et al. [158]), so dass einige Aspekte mobiler Anwendungen fast immer implizit kontextsensitiv sind. Hiervon ist primär das UI einer Anwendung betroffen, von dem vorausgesetzt wird, dass es auf allen Geräten der jeweiligen Plattform spezifikationskonform dargestellt wird. Da zum Entwurfszeitpunkt die tatsächlichen Displayeigenschaften jedoch unbekannt sind, muss ein UI-Entwurf flexibel genug sein, um auch unter variierenden Displaygrößen eine befriedigende *User Experience* (UX, engl. Anwendererfahrung, Anwendererlebnis) zu gewährleisten. Konkret bedeutet das, es müssen für unterschiedliche Klassen von Displaygrößen unterschiedliche graphische Ressourcen mit der App ausgeliefert werden, von denen dann in Abhängigkeit vom Kontext, also in diesem Fall einem konkreten Gerät, eine bestimmte Ressource ausgewählt wird. Analog bestimmen die Kontextparameter Plattform und Gerät über die Verfügbarkeit und konkrete Ausprägung anderer Ressourcen, wie etwa Sensoren oder anderer Hardware- und Softwarekomponenten.

Da hierdurch Funktionalität und Aussehen einer App geprägt wird, sind die Kontextparameter Plattform und Gerät auch Gegenstand des Testens. Es ergibt sich hier die Anforderung, einen spezifischen Gerätekontext für die Testdurchführung reproduzieren zu können. Ohne adäquate Testtechnologie, die in der Lage ist einen solchen Gerätekontext zu simulieren, kann eine App nur wirksam auf einer repräsentativen Auswahl von Geräten getestet werden, die den in der Testfallbeschreibung spezifizierten Gerätekontext bestmöglich abbildet. Hierdurch werden in Testaktivitäten Ressourcen gebunden, etwa in Form manuellen Arbeitsaufwands oder der Bereitstellung und Pflege eines repräsentativen Geräteportfolios und dem Erwerb von Kompetenz zur Beurteilung kontextspezifischer Anforderungen an das Testen im Hinblick auf die Kontextparameter Plattform und Gerät.

3.1.2.1.2 Datum und Uhrzeit

Apps verwenden häufig Datum- und Uhrzeit, ohne dass sie explizit als kontextsensitive Anwendungen wahrgenommen werden. Hofer et al. [182] und Baldauf et al. [25] führen in ihren Überlegungen zu Kontextsensitivität hierzu den Begriff *Time-awareness* (engl. Zeitsensitivität) in Computersystemen ein, um Systeme zu beschreiben, in denen der Ausführungszeitpunkt einer Funktion eine semantische Bedeutung für das Ergebnis hat (vgl. Begriffsdefinition *Kontextsensitivität*, Abschnitt 3.1.1.5). Das heißt, dass konkret von einer Funktionsausführung zu erwartende Ergebnis ist neben anderen Parametern von der Zeit der Ausführung abhängig.

Zeit ist ein Kontextparameter, der auf allen Computersystemen verfügbar ist. Wenngleich die Korrektheit der durch API-Aufrufe erworbenen Informationen fraglich sein kann (z. B. falsch eingestellte Uhr), kann vorausgesetzt werden, dass zu jedem Zeitpunkt eine bestimmte Datums- und Uhrzeitangabe gültig ist.

Zeit ist ein Kontextparameter, der auch in der wissenschaftlichen Literatur bereits identifiziert und diskutiert wurde (vgl. Schmidt et al. [315], Hofer et al. [182], Broens und van Halteren [59], Baldauf et al. [25]). Mögliche Anwendungen für die Kontextparameter Datum und Uhrzeit sind vielfältig. Anwendungen zur Arbeitszeiterfassung beispielsweise verwenden Datum und Uhrzeit als Kontextparameter, der durch ein Abrechnungssystem schließlich direkt auf geldwerte Kosten abgebildet wird (z. B. automatisierte Leistungsabrechnung). Eine weiteres Beispiel ist die mobile Anwendung DB Navigator [88] der Deutschen Bahn, die es dem Anwender ermöglicht, basierend auf seinem gegenwärtigen Standort und Datum und Uhrzeit Verkehrsverbindungen zu ermitteln.

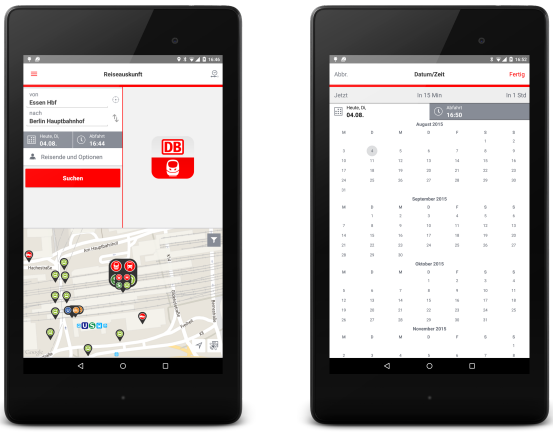


Abbildung 3.2: Bildschirmabdruck der App DB Navigator als Beispiel einer kontextsensitiven Anwendung mit besonderem Fokus auf den Parameter Zeit.

Der Charakter der Kontextsensitivität mit besonderem Fokus auf den Kontextparameter Zeit wird am Beispiel der DB Navigator App an der Funktion „Jetzt“ deutlich. Diese erlaubt es, eine Zugverbindung in zeitlicher Relation zum Zeitpunkt des Funktionsaufrufs zu ermitteln. Beim Testens einer solchen Anwendung ist diese Kontextabhängigkeit insofern zu berücksichtigen, als dass der Testfall selbst in diesem Augenblick kontextsensitiv wird. Um eine Testdurchführung unabhängig von Datum und Uhrzeit zu ermöglichen, muss ein Orakel ebenfalls abhängig von Datum und Uhrzeit der Testdurchführung Erwartungswerte bereitstellen. Konsequent müssen Testfälle entsprechend parametrisierbar sein.

Mobile Geräte bieten neben der manuellen Einstellung von Datum und Uhrzeit ebenfalls die Option, Datum und Uhrzeit vom *Internet Service Provider* (ISP) zu beziehen. Es ergibt sich für den Anwender der Vorteil, dass die Uhr des Smartphones auch nach Überschreiten einer Zeitzonengrenze die am Ort gültige Uhrzeit anzeigt, ohne dass eine manuelle Intervention nötig ist. Für Fahrzeugführer in Logistik- oder Personenverkehrsunternehmen (z. B. Zugführer, Piloten) ist das Passieren von Datums- oder Zeitzonengrenzen nicht ausgeschlossen (z. B. Zugfahrt durch den Eurotunnel von Coquelles, FR nach Folkstone, GB). Anwendungen, die eine automatisierte Arbeitszeiterfassung auf mobilen Geräten ermöglichen (z. B. das Produkt SPEDION App™ [327]) müssen solche Übergänge berücksichtigen, um Korrektheit der Zeiterfassung zu gewährleisten. Hinzu kommen Besonderheiten wie unterschiedliche Zeitpunkte der Umstellung von Normalzeit auf Sommerzeit in verschiedenen Ländern.

Datum, Uhrzeit und Standort treten häufig gemeinsam auf. Die Anzeige von Uhrzeit und Datum auf einem UI erfolgt i. d. R. nicht in koordinierter Weltzeit (UTC) sondern in einem auf die Präferenz oder Zeitzone des gegenwärtigen Aufenthaltsort des Anwenders angepassten Format (vorbehaltlich Sonderfällen, wie z. B. Fliegerei). Das Darstellungsformat von Datum und Uhrzeit ist deshalb eine kontextsensitive Funktion mit den Parametern Zeit und Standort. Es existieren praktische Anwendungsfälle, in denen eine Datums- und Zeitangabe durch eine Standortangabe substituiert wird (vgl. Vieira et al. [352]). Beispielhaft kann hier eine Erinnerungs- oder Weckerfunktion angeführt werden, die einen Reisenden beim Eintreffen an einem bestimmten Ort alarmieren soll, auch wenn der konkrete Zeitpunkt des Eintreffens an diesem Ort nicht sicher bekannt ist (z. B. Verwendung von Google Voice: „Ok Google, Wecker auf Ankunft in Berlin einstellen!“ bei einer Zugreise mit dem Ziel Berlin).

Es können demnach mehrere Kategorien von Zeitangaben unterschieden werden: (1) absolute, (2) relative und (3) symbolische Datums- und Zeitangaben. Die Kategorie der symbolischen Datums- und Zeitangaben schließt substituierende ein. Diese Klassifikation ist für die Modellierung von Kontext zum Zwecke der Anwendungs- und Testspezifikation relevant und wird in Abschnitt 5.2.2.1 erneut aufgegriffen.

In den beiden Beispielanwendungen Mobiler Taxiruf und AR Tourist Information treten Datum und Uhrzeit als relevante Kontextparameter auf. Mobiler Taxiruf stellt Datum und Uhrzeit zusammen mit den Standort fest und übermittelt die Information an einen Backend-Service. Die Bereitstellung der Dienstleistung ist also vom Zeitpunkt der Anwendungsausführung abhängig. Zwar ist das Verhalten der App nicht direkt abhängig von Datum und Uhrzeit, wohl aber die Erbringung der Dienstleistung. Der Anwender hat eine klare Erwartung an die App, so dass eine explizite Unterscheidung zwischen digitalem und nicht-digitalem Anteil der Dienstleistung aus Anwenderperspektive nicht sinnvoll ist. Ein Defekt in der Verarbeitung der Kontextparameter führt dazu, dass die Dienstleistung nicht wie erwartet erbracht werden kann. Bei Verwendung der App in Grenzgebieten zwischen Ländern, die keiner gemeinsamen Sommerzeitregelung unterliegen, könnte beispielweise das mobile Gerät des Anwenders in einem Mobilfunktnetz des einen Landes eingebucht sein, so dass sich eine Uhrzeitdifferenz zum Land der Dienstleistungserbringung ergibt, weil sich das mobile Gerät an der vom Netzbetreiber vorgegeben Uhrzeit orientiert. Für AR Tourist Information ergibt sich ein ähnliches Problem, allerdings führt fehlerhafte Behandlung von Kontextparametern nicht zwangsweise zum einem Defekt in der Leistungserbringung.

Wenngleich der Umgang mit Datums- und Uhrzeitangaben in Software durch koordinierte Weltzeit (UTC) verhältnismäßig einfach zu implementieren ist, ergeben sich hierdurch jedoch Fehlerquellen, die in Tests zu berücksichtigen sind. Zur natürlichen Dynamik der Zeit kommt im Umfeld mobiler Anwendungen zusätzlich hinzu, dass der Anwender sich während der Ausführung der Software über Zeitzonengrenzen hinweg bewegen kann, ein Wechsel zwischen Sommer- und Winterzeit stattfinden kann oder beide Störungen zusammen auftreten. Insbesondere in Anwendungsfällen, die die Differenz zwischen zwei Datumsinformationen verwerten, muss hier sorgfältig evaluiert werden, ob oben genannte Störungen zu Verletzung der Spezifikation führen können und deshalb beim Testen gesondert berücksichtig werden müssen.

Im Vergleich zu anderen Kontextparametern ist Zeit insofern von besonderer Bedeutung, als dass ihre Verwendung in einer App direkt auf den Prozess der Softwareentwicklung ein-

wirkt. Dies gilt insbesondere für Anwendungen, die in Zusammenarbeit mit einem Backend-System einen Anwendungsfall realisieren, da mehrere Systeme, ggf. über Plattformgrenzen hinweg, synchron zusammenarbeiten müssen. Gerade beim Testen stellt sich die Herausforderung, Tests zu entwerfen, durch welche die Abhängigkeit vom Kontextparameter Zeit effektiv und effizient geprüft werden kann. Ultimativ könnte die Verwendung von Zeit als Kontextparameter dazu führen, dass die Ausführung von Tests auf bestimmte Zeitintervalle festgelegt wird, wodurch beispielweise lange Wartezeiten zwischen individuellen Tests liegen, in denen auf ein geeignetes Zeitfenster gewartet werden muss. Anforderung an eine Testautomatisierungstechnologie ist also, dass sie in der Lage sein muss, Datum und Uhrzeit auf den beteiligten Systemen zu manipulieren. Für den Sonderfall der Verwendung der App in der Nähe von Zeitzonengrenzen muss die Testautomatisierungstechnologie weiterhin in der Lage sein, einen Wechsel des Standorts des Geräts bzw. des Anwenders zu simulieren, um die Reaktion der App auf einen Zeitzonenwechsel zu überprüfen (vgl. Vieira et al. [352]).

3.1.2.1.3 Standortinformationen

Mobilität ermöglicht die Realisierung einer Vielzahl von Anwendungen, die Menschen in Alltagssituationen – die häufig Mobilitätssituation sind – begleiten und insbesondere Informationen am Aufenthaltsort des Anwenders verfügbar machen, die vor der Smartphone-Ära i. d. R. nur ortsgebunden, beispielsweise am Desktop-Computer im Büro oder daheim verfügbar waren. Mobilität des Anwenders und insbesondere die Kenntnis über den Aufenthaltsort des Anwenders ist ein wesentlicher Innovationstreiber (vgl. Roman et al. [303]). Tatsächlich verhält es sich sogar so, dass im Umfeld mobiler Anwendungen die Klasse der ortsbasierten Anwendungen noch vor der Klasse der kontextsensitiven Anwendungen wissenschaftlich diskutiert wurde und die Wende zu einem umfassenderen Kontextbegriff erst um das Jahr 2000 vollzogen wurde (vgl. Schmidt et al. [314, 315], Beadle et al. [31], Roman et al. [303]).

Das Wissen um den aktuellen Aufenthaltsort des Anwenders kann wesentlich zur Anpassung und zur Individualisierung von Anwendungsfällen beitragen. Neben fachlichen Eingabeparametern kann der aktuelle Standort des Anwenders als Parameter zur Durchführung von Berechnungen oder zur Parametrisierung des Aufrufs externer Dienste herangezogen werden. Mögliche Anwendungsfälle hierfür sind unter anderem der Abruf aktueller Straßenverkehrsinformationen (z. B. Ort-zu-Ort Navigation, Staumeldungen oder Fahrpläne öffentlicher Verkehrsmittel) im geographischen Umfeld des Anwenders.

In den Anfängen mobiler Softwaresysteme war die direkte Erfassung des Aufenthaltsorts eines Anwenders ein technisches Problem, da die Hardware der Prä-Smartphone Generationen mobiler Telekommunikationsgeräte nicht über die erforderlichen Komponenten zur autonomen Standortbestimmung verfügten. Aus diesem Grund kamen zunächst Technologien zum Einsatz, durch welche der Standort des Anwenders im Bezug zu künstlichen lokalen Referenzsystemen (z. B. Beacons, vgl. Want et al. [358]) angenähert werden konnte. Hierdurch konnten erstmals standortbezogene Dienste (wie etwa die Identifikation im Bezug zum Anwender nahgelegener Peripherie, z. B. Drucker, Lokalisierung innerhalb von Gebäuden) implementiert werden. Ein solches Lokalisierungssystem innerhalb eines Bürokomplexes wurde bereits im Jahr 1992 von Want et al. [358] diskutiert. Die Verwendung künstlicher lokaler Referenzsysteme hat allerdings Nachteile, die deren Eignung für den flächendeckenden Ein-

satz in Mobilfunktelefonen im Consumer-Segment in Frage stellen. Einerseits erfordert der Betrieb lokaler künstlicher Referenzsysteme den großflächigen Betrieb sogenannter Beacons (z. B. iBeacon der Firma Apple oder NFC-Tags), durch welche der Standort eines mobilen Geräts durch Triangulation ermittelt werden kann. Andererseits wäre für diese Technologie ebenfalls der Einsatz speziell für diesen Zweck angepasster und standardisierter Hardware in mobilen Geräten erforderlich (im Jahr 2016 steht diese Technologie durch BLE-Beacons zur Verfügung). Die Bereitstellung zusätzlicher künstlicher lokaler Referenzsysteme zur Verbesserung der Qualtität der Positionsbestimmung bedeutet einen technischen Aufwand, der aus ökonomischen Gründen nicht in größerem Maßstab sinnvoll realisiert werden kann.

Mit dem Betrieb terrestrischer Mobilfunknetze ist der Aufwand der Schaffung eines zusätzlichen künstlichen Referenzsystems obsolet geworden. Die Bereitstellung der Kommunikationsinfrastruktur bedingt aus technischen Gründen zugleich den Betrieb eines solchen künstlichen lokalen Referenzsystems. Durch die zelluläre Struktur von Mobilfunknetzen ist die wechselseitige Zuordnung von Mobiltelefonen zu mindestens einem *Base Station Subsystem* (BSS) für den Betrieb erforderlich. Hierdurch ist zu jedem Zeitpunkt der Standort eines Mobilfunktelefons mit dem Standort der zugeordneten BSS implizit mit deren geographischen Standort gegeben. Durch variierende geographische Ausdehnungen individueller Mobilfunkzellen kann die geographische Position einzelner Mobilfunktelefone allerdings nur ungenau bestimmt werden, wodurch die Verwendung dieser Technologie für Anwendungen mit hohen Präzisionsanforderungen ausscheidet.

Die Weiterentwicklung der technischen Gegebenheiten hat die Notwendigkeit künstlicher lokaler Referenzsysteme zugunsten eines globalen geographischen Referenzsystems weitgehend obsolet gemacht, wodurch Entwicklungshürden für ortsbasierte mobile Anwendungen wesentlich herabgesetzt wurden. Insbesondere das GPS ermöglicht die weitgehend autonome Standortbestimmung mobiler Telekommunikationsgeräte mit hoher Genauigkeit. Bei aktuellen Smartphone-Generationen gehört die Unterstützung der Positionsbestimmung mittels GPS bereits zum Standardfunktionsumfang, wobei sich der Standort des Geräts unter optimalen Bedingungen mit einer Genauigkeit von wenigen Metern Abweichung von der tatsächlichen Position bestimmen lässt. Mit der flächendeckenden Verfügbarkeit genauer Standortinformationen ohne die Notwendigkeit künstlicher lokaler Referenzsysteme können vielfältige Anwendungszenarien unter Einbeziehung von Standortinformationen realisiert werden, wobei sich der Themenkomplex ortsbasierter Anwendungen als besonders ergiebig für aktuelle und zukünftige Anwendungs- und Geschäftsmodelle erwiesen hat.

Der aktuelle Aufenthaltsort des Anwenders ist dazu geeignet, die zur Durchführung von Berechnungen oder zur Parametrisierung des Aufrufs externer Dienste erforderlichen Parameter implizit und ohne direkte Interaktion mit dem Anwender zu erheben. Zum einen wird hiermit durch Entlastung des Anwenders der Bedienkomfort einer mobilen Anwendung gesteigert, zum anderen können Parameter einbezogen werden, die sich dem bewussten Wissen des Anwenders entziehen. Beispielsweise der Abruf von aktuellen Fahrplaninformationen unter Berücksichtigung von Echtzeitinformationen (z. B. Zugverspätungen) oder der Taxiruf zu einem nur vage bekannten Ort (z. B. das Ermitteln einer Straßenadresse aus WGS84-Koordinaten via *Reverse Geocoding*) werden durch präzise Standortinformationen erst ermöglicht oder mindestens stark vereinfacht.

Standortinformationen geben Auskunft über die Position von Entitäten im Bezug zu einem definierten Referenzsystem. Referenzsysteme können geographische Referenzsysteme (z. B. WGS84) oder symbolische Referenzsysteme sein. Zu berücksichtigen ist ebenfalls, dass sich ein Referenzsystem relativ zu einem weiteren Referenzsystem in Bewegung befinden kann. In geodätischen Referenzsystemen werden Standortinformationen explizit durch Angabe von Koordinaten (z. B. in kartesischen oder geodätischen Koordinatensystemen) angegeben (vgl. Bettini et al. [42]). Hierzu kommt beispielsweise das WGS84-Referenzsystem in Frage. Standortinformationen, die zu jedem Zeitpunkt den Standort einer Entität relativ zur Erdoberfläche bestimmen, werden als *geodätische Standortinformationen* bezeichnet. Standortinformationen, die durch geodätische Koordinaten gegeben sind definieren durch das Referenzsystem implizit Entfernungsmaße, die es erlauben die Entfernung zwischen Entitäten zu quantifizieren und somit angrenzende oder überlappende Regionen zu charakterisieren. Beispiele geodätischer Standortinformationen sind:

- „N52°30′2″ E13°23′56″“ (WGS84-Koordinaten der Stadt Berlin)
- „Unternehmensstandort Dortmund“ (kann auf geodätische Koordinaten einer Straßenadresse abgebildet werden)
- „Spreewaldregion“ (durch WGS84-Koordinaten polygonal begrenzt)

Bedingt durch die gemeinsame technologische Basis GPS werden geodätische Standortinformationen gegenwärtig von den APIs aller mobilen Plattformen als WGS84-Koordinaten dargestellt. Das trifft selbst dann zu, wenn die Erhebung der Standortinformation nicht direkt durch Verwendung eines GPS-Empfängers, sondern durch die Verwendung einer Ersatztechnologie ermittelt wurde. Als Ersatztechnologie kommt beispielsweise eine Anfrage beim Mobilfunkbetreiber in Betracht, welcher aus den Betriebsdaten (insbesondere aktuell zugeordnete BSSs) des Mobilfunkgeräts dessen ungefähre Position ermitteln kann. Ebenfalls in Frage kommt die Positionsermittlung durch Auswertung der sich gegenwärtig in Empfangsreichweite befindlichen Wi-Fi-Hotspot[2].

Unabhängig von der Art der Erhebung der Standortdaten kommen jedoch mehrere Möglichkeiten in Betracht, wie sich das Resultat einer Standortbestimmung manifestieren kann. Neben dem Erfolgsfall kommen eine Reihe unterschiedlicher Fehler in Betracht: Nichtverfügbarkeit, Mehrdeutigkeit, Ungenauigkeit (vgl. Henricksen und Indulska [176]). Im Erfolgsfall ist nach dem Aufruf der entsprechenden API-Funktion der aktuelle Standort des Smartphones bekannt und für die geographische Länge, für die geographische Breite und ggf. auch für die Höhe über dem mittleren Meeresspiegel liegen gültige Werte vor. Zusätzlich kann über API-Funktionen i. d. R. ebenfalls die Genauigkeit der Standortbestimmung ermittelt werden. In diesem Fall wird durch geographische Länge und geographische Breite der Mittelpunkt eines Kreises bestimmt, dessen Radius (üblicherweise in Meter, abhängig von Plattform und SDK) durch die Messgenauigkeit gegeben ist. Geographische Länge beschreibt die Position östlich oder westlich des Nullmeridian. Ihr Wert ist der Winkel mit dem Erdmittelpunkt als Scheitelpunkt zwischen dem Nullmeridian und einem Punkt auf der Erdoberfläche der Werte

[2]Das Unternehmen Google kartographiert seit 2010 im Rahmen des Projekts Street View Wi-Fi-Netzwerke. Diese Informationen stehen für die Lokalisierung mobiler Anwender zur Verfügung.

zwischen -180° und 180° annehmen kann. Das Vorzeichen bestimmt, ob der Winkel in östlicher (+) oder in westlicher (-) Richtung gemessen wird. Geographische Breite beschreibt die Entfernung eines Punktes vom Äquator, gemessen im Winkelmaß. Sie kann Werte zwischen 0° (am Äquator) und 90° (an den Polen) annehmen.

Sollte die Positionsbestimmung nicht erfolgreich sein, liegen für keinen dieser Parameter gültige Werte vor. Um diese Information in mobilen Softwaresystemen nutzbar zu machen, müssen entsprechende Modelle in der Lage sein, diese möglichen Zustände abzubilden. Unter anderem ergeben sich hieraus die in Abschnitt 5.2.2.1 diskutierten Anforderungen an Modelle zur Spezifikation von Standortabhängigkeit. Ungültige Standortinformationen können beispielsweise erzeugt werden, wenn die Positionsbestimmung via GPS keine aussagekräftigen Ergebnisse produziert. Dieser Fall tritt ein, wenn in urbanem Gebiet der Empfang der zur Positionsbestimmung notwendigen Satellitensignale durch hohe Gebäude eingeschränkt ist (Effekt bekannt als sogenanntes Urban Canyoning, vgl. Vicek et al. [350], Peterson et al. [278], Cui und Ge [84], Jabbour et al. [202]) oder innerhalb von Gebäuden unmöglich ist. Fehlermeldungen treten für gewöhnlich dann auf, wenn der Anwender der Verwendung von Standortinformation innerhalb einer Anwendung nicht zugestimmt hat. Beide Fälle unterscheiden sich insofern, als dass Ersterer ggf. Positionsdaten mit starker Abweichung vom tatsächlichen Standort erzeugt, während der Zweite eine Positionsbestimmung vollständig unmöglich macht. Nichtverfügbarkeit und Ungültigkeit von Standortinformationen müssen adäquat abgebildet werden.

In der praktischen Verwendung von Standortinformationen ist häufig der Fall anzutreffen, dass die präzise Standortbestimmung durch WGS84-Koordinaten (Abbildung 3.3a) dem Anwendungsfall nicht angemessen ist. Insbesondere für Anwendungsfälle die Entitäten des Fachkonzepts (d. h. Anwendungsinhalte, verfügbare Funktionen usw.) auf geographische Regionen anstelle individueller Punkte des geodätischen Referenzsystems abbilden. Geographische Regionen können auf unterschiedliche Arten definiert werden. Eine Möglichkeit ist Definition einer Region durch Spezifikation eines Kreises, gegeben durch einen Mittelpunkt in WGS84-Koordinaten und den Radius (Abbildung 3.3b). Hier ergibt sich zwar der Vorteil, dass die Region mit vergleichsweise geringem Aufwand zu modellieren ist. Allerdings können von kreisrunder Form abweichende Regionen mit dieser Methode bestenfalls durch mehrere angrenzende oder überlappende kreisrunde Regionen angenähert werden. Alternativ kann die Bestimmung einer Region durch Spezifikation einer polygonal begrenzten Fläche (Abbildung 3.3c) erfolgen, deren Eckpunkte wiederum durch WGS84-Koordinaten bestimmt sind. Dies erlaubt einerseits die präzise Spezifikation komplexer Regionen, führt aber zu erhöhtem Modellierungsaufwand. Die Möglichkeit zusammenhängende Regionen aus einzelnen kreisrunden Regionen, aus Polygonen oder Polygonzügen zu erzeugen ist insofern vorteilhaft, als dass mit geringem Aufwand Entfernungen zwischen Regionen oder überlappende Teilregionen bestimmt werden können.

Ebenfalls denkbar sind Anforderungen, Standortinformationen relativ zu abstrakten symbolischen und ggf. mehrdeutigen Referenzsystemen zu spezifizieren (vgl. Bettini et al. [42]). Hier kommt beispielsweise die Anforderung in Frage, an einem Unternehmensstandort einen Druckauftrag für ein Dokument an einen nahegelegen Drucker zu versenden, dessen genauer Standort im Bezug zu einem geodätischen Referenzsystem für den aktuellen Anwendungsfall nicht bekannt und nicht relevant ist. Diese Art von Standortinformationen werden als

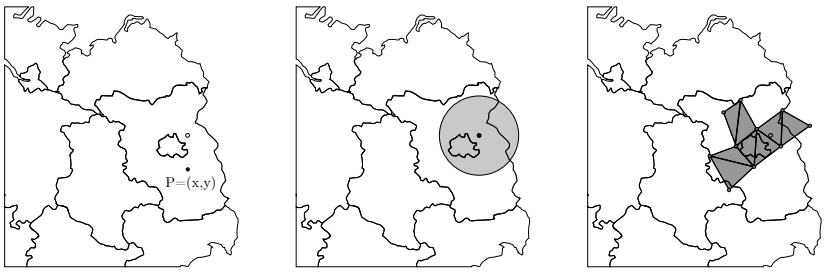
symbolische Standortinformationen bezeichnet, da sie konkrete Standorte durch Symbole referenzieren. Die korrekte Interpretation der Symbole ist abhängig vom aktuellen Kommunikationskontext und kann mehrdeutig sein. In Abhängigkeit des Kommunikationskontextes kann eine Abbildung symbolischer Standortinformationen auf geodätische Standortinformationen möglich sein. Symbolische Referenzsysteme implizieren i. d. R. keine Abstandsmaße. Beispiele symbolischer Standortentitäten sind:

- „An der Rezeption eines unserer Autohäuser“
- „An allen Unternehmensstandorten in Deutschland“

Symbolische und geodätische Standortinformationen können zu *statischen Standortinformationen* zusammengefasst werden, sofern sich die jeweilig verwendeten Referenzsysteme nicht relativ zueinander in Bewegung befinden oder aber für die weitere Verarbeitung eine Momentaufnahme des Standortes ausreichend ist. Ebenfalls in Frage kommen Anwendungen, die Standorte relativ zu sich in Bewegung befindlichen, dynamischen Referenzsystemen spezifizieren. Der Standort einer Entität kann bei *dynamischen Standortinformationen* im günstigsten Fall anhand einer Trajektorie, d. h. des Bewegungspfads eines Objektes, bestimmt werden. Ebenfalls ist denkbar, dass keines der verwendeten Referenzsysteme auf ein geodätisches Referenzsystem abgebildet werden kann. Beispiele für dynamische Standortentitäten sind:

- Anwender befindet sich in einem Zug
- Anwender befindet sich auf der Autobahn A3 zwischen Köln und Essen
- weniger als 200m vom Anwender entfernt

Eine besondere Herausforderung bei der Verwendung dynamischer Standortinformationen ist zusätzlich, dass sie sich häufig auf symbolische Standortinformationen beziehen. In den obigen Beispielen trifft dies auf die Entitäten „Zug“, „Autobahn“ und „Anwender“ zu. Den ersten beiden Beispielen ist gemeinsam, dass davon ausgegangen werden kann, dass sich der Anwender in Bewegung befindet. Ohne jedoch die Entität „Zug“ näher zu charakterisieren (Zugnummer, Zielbahnhof, Fahrplan usw.), kann der tatsächliche Standort des Anwenders



(a) Punkt definiert durch WGS84-Koordinatenpaar

(b) Kreisrunde Region definert durch WGS84-Koordinatenpaar und Radius

(c) Ploygonal begrenzte Region

Abbildung 3.3: Aspekte der Darstellung von Standortinformationen, wie sie in typischen Anwendungsfällen auftreten.

nicht eindeutig bestimmt werden. Es hängt vom Anwendungsfall ab, ob die Abbildung auf ein geodätisches Referenzsystem relevant ist.

Eine weitere Klassifikation von Standortinformationen in atomare und zusammengesetzte (*Atomic vs. Compound Location* [75]) wird von Chen et al. [75] vorgeschlagen. Zugrunde liegt die Erkenntnis, dass sich eine physische Entität zu einem Zeitpunkt nur an genau einem physikalischen Standort befinden kann. Insbesondere die Berücksichtigung symbolischer Standortbestimmungen ermöglicht jedoch die Definition einer Enthaltenseinseigenschaft. Das heißt, ein symbolischer Standort kann andere symbolische Standorte enthalten (z. B. wird ein Konferenzraum einer bestimmten Etage eines Gebäudes zugeordnet, dieses ist einer Straßenadresse zuordenbar, die wiederum auf geodätische Koordinaten abbildbar ist). Dieses Konzept kann auf polygonal begrenzte Flächen übertragen werden (z. B. befindet sich ein Universitätscampus innerhalb der Stadtgrenzen und enthält selbst eine Anzahl an Gebäuden, usw.).

Die Klassifikation von Standortinformationen in geometrische, geodätische, symbolische, statische, dynamische, atomare und zusammengesetzte Standortinformationen wurde bereits von Becker und Dürr [35] und Chen et al. [75] thematisiert und die Notwendigkeit dieser Klassifikation mit den unterschiedlichen Anwendungsfällen von Standortinformationen in Softwaresystemen und deren Relevanz für Verhaltensaspekte von Softwaresystemen begründet.

Insgesamt ist der Standort des Anwenders ein besonders signifikanter Kontextparameter. Zum einen gehört die Standortbestimmung zum Standardfunktionsumfang zeitgenössischer SDKs mobiler Plattformen. Zum anderen werden mobile Geräte vorwiegend in mobilen Szenarien verwendet, in denen sich der Anwender in Bewegung befindet und einerseits einen gesteigerten Informationsbedarf hat und der Standort andererseits zur Schlüsselinformation für digitale Dienste wird. Während in einigen mobilen Anwendungen die Verwendung des Standortes des Anwenders lediglich ein Begeisterungskriterium ist (vgl. Kano-Modell nach Kano et al. [211]), ist die Verwendung von Standortinformationen in anderen Anwendungen obligatorisch für den Anwendungsfall. Die in Abbildung 3.4 abgedruckten Apps wären ohne die Integration von Standortinformationen nicht in der Lage, ihren Bestimmungszweck zu erfüllen. Die Anwendungen Runtastic Laufen & Fitness [307] sowie Zombies, Run! [322] aus dem Fitness Sektor beispielsweise erfassen den Standort des Anwenders kontinuierlich während des Laufens mit dem Ziel, über einen längeren Zeitraum hinweg Statistiken über die Laufleistung des Anwenders zu führen (Runtastic Laufen & Fitness) oder das Laufen durch Integration spielerischer Inhalte aufzuwerten (Zombies, Run!). Die App Golf GPS & Scorecard [338] integriert darüber hinaus ortsbasierte Informationen der unmittelbaren Umgebung des Anwenders, um dessen Golfspiel durch digitale Unterstützung aufzuwerten.

Die beiden Beispielanwendungen Mobiler Taxiruf und AR Tourist Information sind kontextsensitiv gegenüber dem Standort des Anwenders. Für beide Anwendungen ist der Standort von wesentlicher Bedeutung, da keine der beiden Anwendungen ohne Standortinformationen in der Lage ist, ihren Bestimmungszweck zu erfüllen. Die Anwendung Mobiler Taxiruf bietet jedoch die Möglichkeit an, den Standort manuell in die Anwendung einzugeben. Die Eingabe erfolgt in als Straßenadresse, während bei der Verwendung des GPS-Modus die Standortinformation zunächst als WGS84-Koordinaten vorliegen. Im Fall der Anwendung Mobiler Taxiruf ist der Anwender ein *Mobile User* (vgl. Abschnitt 3.1.2.1.3, Book et al. [54, 53, 55]), eine einmalige Feststellung des Standorts ist daher ausreichend. AR Tourist Information implementiert

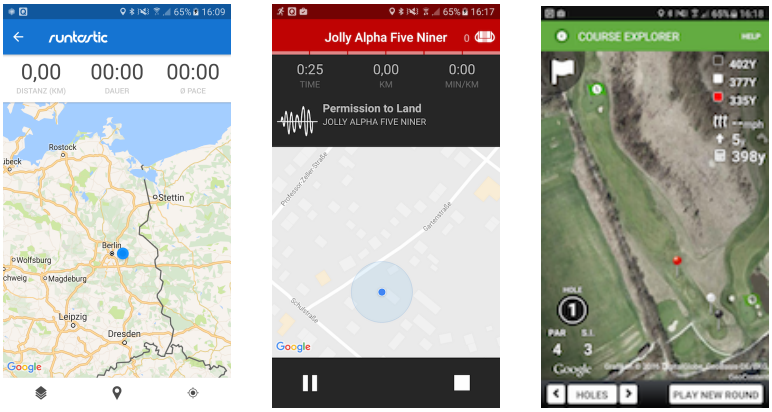


Abbildung 3.4: Bildschirmabdrucke der Apps Runtastic Laufen & Fitness [307], Zombies, Run! [322] und Golf GPS & Scorecard [338]. Diese Apps verwenden den Standort des Anwenders als primären Eingabevektor.

hingegen ein *In-motion User*-Konzept (vgl. Abschnitt 3.1.2.1.3, Book et al. [54, 53, 55]). Der Standort des Anwenders muss deshalb während des Betriebs der App kontinuierlich ermittelt werden. Aufgrund der technischen Eigenschaften des GPS ist ständig mit Signalverlust oder Verschlechterung der Qualität der Standortinformation zu rechnen. Bei der Implementierung der Anwendung muss weiterhin berücksichtigt werden, dass eine ständige Standortermittlung unter Verwendung des GPS-Moduls ebenfalls hohe Anforderungen an die Energieversorgung stellt. Die Qualität dieser Anwendungen wird deshalb wesentlich durch ihre Toleranz gegenüber temporärer Nichtverfügbarkeit des Kontextparameters Standort beeinflusst.

Beim Testen muss der Standort des Geräts bzw. des Anwenders berücksichtigt werden, wenn das Ergebnis eines Funktionsaufrufes in der Anwendung vom Standort abhängt. Um kostenintensive Vor-Ort-Testverfahren zu vermeiden oder zu minimieren, ist die Möglichkeit Standortinformationen und Standortwechsel zu simulieren für Testautomatisierungswerkzeuge obligatorisch. Verfügbare Technologien unterstützen diese Anforderung jedoch nur in beschränktem Umfang.

3.1.2.1.4 Magnetfeld

Mobile Geräte der Smartphone-Generation verfügen i. d. R. über Magentfeldsensoren (synonym Magnetometer). Aufgabe dieser Sensoren ist es, die Flussdichte von sich in der Umgebung des Geräts befindlichen Magnetfeldern zu messen. Primäre Verwendung in der Praxis ist die Vermessung des Erdmagnetfelds. Aus der lokal messbaren Stärke und Richtung der magnetischen Flussdichte können je nach Anwendungsgebiet unterschiedlichste Informationen gewonnen werden. Störungen des Magnetfelds der Erde können beispielsweise auf die Gegenwart metallischer Objekte hinweisen.

Während es in der Praxis außerhalb der Softwareentwicklung eine Reihe von Verwendungen für Magnetfeldsensoren gibt, so ist die primäre Verwendung des Magnetometers in der Anwendungsentwicklung für mobile Systeme die Ermittlung der Lage eines mobilen Geräts im Verhältnis zur Erdoberfläche. Hierbei handelt es sich um eine Schlüsselfunktion für viele mobile Anwendungen, wie etwa Navigation, Anzeige eines virtuellen Kompass oder AR-

Anwendungen (vlg. Kähäri und Murphy [210]), die in Abhängigkeit von der Ausrichtung des Geräts den Inhalt des UI anpassen (Abbildung 3.5).

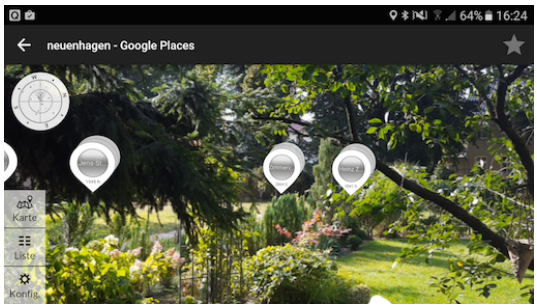
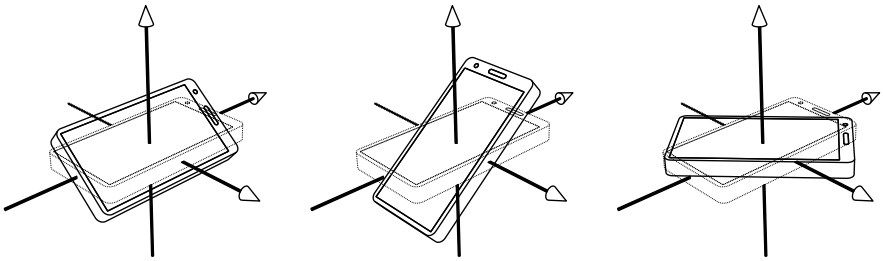


Abbildung 3.5: Bildschirmabdruck der AR-Anwendung wikitude™, die ein Live-Bild der Kamera je nach Standort und Blickrichtung des Anwenders mit Zusatzinformationen anreichert.

Hierzu wird die Stärke des Erdmagnetfelds in den Achsen des lokalen Referenzsystems des mobilen Geräts gemessen. Unter Hinzuziehung weiterer Informationen wie etwa lokale Stärke und Richtung von Beschleunigungskräften (insbesondere die Erdgravitation) und unter Einbeziehung des Standorts des Nutzers kann für jede der drei Achsen des Referenzsystems des mobilen Geräts der Winkel zwischen den Normalen des Koordinatensystems und den magnetischen Polen des Erdmagnetfelds, also Roll-, Gier- und Nickwinkel, bestimmt werden (Abbildung 3.6).



(a) Rollen (b) Nicken (c) Gieren

Abbildung 3.6: Roll-, Gier- und Nickwinkel eines mobilen Geräts. Bestimmung der jeweiligen Winkel erfolgt durch Auswertung von Magnetfeld- und Beschleunigungssensor.

Roll-, Gier- und Nickwinkel bzw. deren Änderung finden in mobilen Anwendungen nicht nur als Parameter für die Steuerung von Anwendungsinhalten (wie es in AR-Anwendungen der Fall ist) Verwendung. Physikalische Bewegung eines mobilen Geräts kann ebenfalls als zusätzlicher Eingabekanal verwendet werden, wobei der Magnetfeldsensor neben dem Gyroskop und dem Beschleunigungssensor eine Schlüsselrolle bei der Lagebestimmung spielt. Bereits im Jahr 1996 wurde von Rekimoto [294] ein sensorbasierte Adaption eines UI vorgestellt, in dem es dem Anwender möglich war, durch Neigen und Kippen des Geräts durch ein Optionsmenü zu navigieren. Diese Funktion wird im Jahr 2016 von ultramobilen Geräten wie etwa *Android Wear Smartwatches* (Android-basierte Computer in Kleinformat zum Tragen am Handgelenk) wieder aufgegriffen. Diese ermöglichen es beispielsweise durch Rotation des Handgelenks durch einzelne Seiten des UI zu blättern (vgl. Abbildung 3.7). Ebenfalls in Fra-

ge kommt diese Technologie um 3D-UIs zu steuern. Katzakis und Hori [214] diskutieren in ihrer Arbeit einen Ansatz, wie der Magnetfeldsensor (in Zusammenarbeit mit dem Beschleunigungssensor) eines mobilen Geräts verwendet wird, um virtuelle Objekte in einem 3D-UI zu manipulieren.

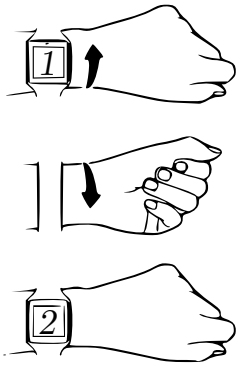


Abbildung 3.7: UI Navigation durch Neigen eines Android Wear Geräts (Abbildung entlehnt aus [324])

Neben der Lagebestimmung kann der Magnetfeldsensor eines mobilen Geräts auch für andere Anwendungen eingesetzt werden. Beispielsweise präsentieren Rohs et al. [302], Essl und Rohs [110] und Tanaka [339, 340] bereits in den Jahren 2004-2007 sowie Ketabdar et al. [217] (2011) Technologien, die mobile Geräte als Musikinstrumente verwendet. Unterschiedliche Töne werden hier erzeugt, indem Roll-, Gier- und Nickwinkel des Geräts verändert werden. Hierzu wird zur Messung dieser Winkel neben dem Beschleunigungssensor ebenfalls auf den Magentfeldsensor zugegriffen.

Einen weiteren Anwendungsfall der Verwendung des Magnetfeldsensors wird von Ketabdar et al. [219, 218] (auch Harrison und Hudson [169]) vorgestellt. Hier wird eine lokale Manipulation des natürlichen Magnetfelds der Erde durch Überlagerung mit einem in einen Stift oder Fingerring integrierten Permanentmagneten als Eingabekanal in eine mobile Anwendung nutzbar gemacht. Die Autoren schlagen eine Methode zur Authentifizierung an Diensten durch eine in der Luft in der Nähe des Gerätes ausgeführte Signatur (d. h. eine Bewegungsgeste) vor. Im Gegensatz zur Verwendung des Erdmagnetfelds zur Bestimmung von Roll-, Gier- und Nickwinkel werden hier allerdings nicht ausschließlich naturgegebene Umgebungsparameter verwendet, sondern eine künstliche Magnetfeldquelle ist notwendig. Somit ist dieser Anwendungsfall deutlich an den Grenzen des hier verwendeten Kontextbegriffs gelagert. Für das Testen mobiler Anwendungen ist er aber dennoch relevant, da dieser spezielle Anwendungsfall unter Verwendung zeitgenössischer Technologie nicht automatisiert getestet werden kann. Das in dieser Dissertation vorgestellte Konzept zur Testautomatisierung mobiler Anwendungen durch Simulation von Kontextparametern (vgl. Abschnitt 5.5) ist aber geeignet, auch solche Anwendungsfälle zu bedienen.

Navigationsanwendungen auf mobilen Geräten sind ohne digitalen Kompass kaum praktisch funktionsfähig. Grundlage zu deren Realisierung ist die lokale Messung des Erdmagnetfelds. In den meisten Anwendungsfällen wird das Magnetfeld der Erde passiv als Kontextparameter verwendet. Das es sich hierbei jedoch um eine naturgegebene physikalische Größe handelt, die je nach Standort unterschiedlich wirkt und zudem unregelmäßig ist, müs-

sen Anwendungen, die diesen Parameter verwenden, mit besonderer Aufmerksamkeit getestet werden (z. B. Deklination[3]). Neben dem natürlichen Rauschen des Sensors kommt hierbei insbesondere zum Tragen, dass das Magnetfeld der Erde lokal durch unterschiedlichste Störquellen überlagert werden kann. Solche Störungen sind ihrerseits dynamisch, so dass es ohne Zugriff auf spezialisierte Labortechnik kaum möglich ist, reproduzierbare Testfälle für Anwendungen zu erstellen, die den Kontextparameter Magnetfeldstärke verwenden. Bereits das verwendete mobile Gerät hat Einfluss auf die vom Magnetfeldsensor erzeugten Daten. Darüber hinaus haben magnetische Gegenstände in der Nähe des Geräts Einfluss auf die Sensordaten. Eine mögliche Störquelle ist beispielsweise der Magnetverschluss eines Schutzetuis.

Während das Testen von Anwendungen, die Messdaten des Magnetfeldsensors als Kontextparameter verwenden, mit besonderen Schwierigkeiten hinsichtlich der Reproduzierbarkeit der Testumgebung verbunden ist, sind die Qualitätsansprüche an solche Anwendungen gleichzeitig hoch. Eine Navigationsanwendung, die den Nutzer aufgrund einer falschen Interpretation von Sensordaten fehlleitet, erfüllt ihren Bestimmungszweck nicht. Es ist deshalb Anforderung an eine Testautomatisierungstechnologie, reproduzierbare Surrogate für Sensormessdaten bereitzustellen.

3.1.2.1.5 Sonstige physikalische Kontextparameter

Neben dem Magnetfeldsensor verfügen Smartphones i. d. R. über eine Reihe weiterer Sensoren. Hierzu gehören unter anderem Sensoren zur Messung von Beschleunigung, Gravitation (als Spezialfall der Beschleunigung), Rotationsgeschwindigkeit, Umgebungstemperatur, Lichtintensität, Annäherung an Objekte, Luftdruck oder Luftfeuchtigkeit. Die tatsächliche Ausstattung mit spezifischen Sensoren ist geräteabhängig. Analog zur Diskussion im vorherigen Abschnitt 3.1.2.1.4 können diese Sensoren für unterschiedliche Anwendungen verwendet werden. Der Sensor zur Messung der Lichtintensität beispielsweise wird vom Betriebssystem zur Steuerung der Displayhelligkeit verwendet.

In den SDKs mobiler Plattformen ist die Verwendung von Sensoren i. d. R. nach dem *Subscriber*-Entwurfsmuster realisiert, d. h. eine Komponente registriert einen Beobachter an einem Sensor und dieser wird in Intervallen mit neuen Messwerten versorgt. Vor der Verwendung eines spezifischen Sensors muss jedoch sichergestellt werden, dass das konkrete Geräte über diesen Sensor verfügt, d. h. im Kontext dieses Geräts dieser Sensor verwendet werden kann. Durch die Variabilität der in mobilen Geräten je nach konkretem Modell verfügbaren Sensoren wird auch die Vielfalt der mit einem spezifischen Gerät verfügbaren kontextsensitiven Anwendungen bestimmt. Hier manifestieren sich die Kontextparameter Plattform und Gerät im Zusammenhang mit der Verwendung anderer Kontextparameter.

Die Umgebungstemperatur ist beispielsweise ein Kontextparameter, der im Labor verhältnismäßig gut reproduziert werden kann. In der Praxis ist jedoch fraglich, ob dieser Kontextparameter sinnvoll verwendbar ist. Eine Thermometer-App könnte diesen Kontextparameter verwenden, um dem Anwender die Umgebungstemperatur anzuzeigen (z. B. Smart Thermometer [81]). Allerdings stellt sich die Frage, inwiefern ein Temperatursensor innerhalb eines Geräts, dessen elektronische Bauelemente Wärme erzeugen, zuverlässig zwischen Umgebung-

[3] Winkel zwischen der magnetischen und der geographischen Nordrichtung.

stemperatur und der selbst erzeugten Wärme differenzieren kann. Hierdurch werden bereits Anforderungen an Tests für eine solche Anwendungen induziert, deren Aufwand bei manuellem Testen kaum zu beherrschen ist. Neben der Vielzahl unterschiedlicher Geräte die beim Testen verwendet werden müssen, um sicherzustellen, dass die durch die App angezeigte Temperatur auf der überwiegenden Mehrzahl der in der Zielgruppe der App zu erwartenden Geräte korrekt ist, muss ebenfalls die Vielfalt der möglichen Anwendungsszenarien berücksichtigt werden. Beispielsweise wiesen Ichikawa et al. [191], Wiese et al. [361] nach, dass Smartphones i. d. R. körpernah (z. B. in Hemd- oder Hosentasche) transportiert werden, so dass anzunehmen ist, dass die durch den Temperatursensor ermittelte Temperatur durch die Körpertemperatur des Anwenders verfälscht ist und die Umgebungstemperatur nicht korrekt abbildet.

Analog gilt für die übrigen durch zeitgenössische Smartphones ermittelbaren Kontextparameter Gravitation, Beschleunigung, Rotationsgeschwindigkeit, Lichtintensität, Annäherung an Objekte, Luftdruck oder Feuchtigkeit, dass diese nur bedingt in Testumgebungen reproduzierbar sind. Durch manuelles Testen kann hier nur in Ausnahmefällen Reproduzierbarkeit von Tests gewährleistet werden, weil Kontextparameter i. d. R. nicht zuverlässig ohne die Verwendung spezialisierter Technologie in Laborbedingungen für eine große Gerätevielfalt hergestellt werden können.

3.1.2.2 Logische Kontextparameter

Logische Kontextparameter (in der Literatur gelegentlich auch als virtuelle Kontextparameter bezeichnet, z. B. Vieira et al. [352], Korpipää und Mäntyjärvi [229, 228]) repräsentieren im Vergleich zu atomaren physikalischen Kontextparametern den nicht-trivialen Fall der Bestimmung des Kontext. Sie werden durch Komposition atomarer, physikalischer, lokal auf einem Gerät messbarer Daten ermittelt.

Die Definitionen der Begriffe *Kontext* (Abschnitt 3.1.1.4) und *Kontextsensitivität* (Abschnitt 3.1.1.5) im Umfeld Software für mobile Geräte zielen i. d. R. darauf ab, situative Faktoren des Anwenders und seiner Umgebung in einer Anwendung zu berücksichtigen, die über Werte einzelner Sensoren hinausgehen. Die mögliche Vielfalt lokal auf einem Gerät ermittelbarer logischer Kontextparameter wird durch das Kreuzprodukt aller auf dem Gerät verfügbaren Kontextquellen bestimmt und zusätzlich durch andere Kontextquellen erweitert. Eine Diskussion möglicher Kombinationen und deren Verwendung in einer App ist jedoch nicht zielführend, da der Verwendungszweck konkreter logischer Kontextparameter stark durch den jeweiligen Anwendungsfall geprägt ist und nicht sinnvoll generalisiert werden kann.

3.1.2.2.1 Bewegung und Orientierung

Mobile Geräte gemäß der Definition in Abschnitt 3.1.1.2 verfügen i. d. R. über Sensoren, die es Anwendungen erlauben, Parameter der Dynamik der Geräteverwendung als Eingabeparameter zu verwenden. Mögliche Anwendungsgebiete sind etwa Navigationsaufgaben (in Zusammenarbeit mit dem Magnetfeldsensor), in denen das Gerät vom Anwender bewegt wird, weil sich dessen Standort verändert. Andere Anwendungsgebiete zielen darauf ab, unabhängig vom Standort des Anwenders gezielte Bewegungen des Geräts im Sinne einer Geste als Interaktionskanal mit der Anwendung nutzbar zu machen. Patel et al. [273] untersuchen beispielsweise

die Möglichkeit, Schütteln des Geräts als Methode zu digitalen Authentifizierung gegenüber Dienstanbietern zu verwenden. Williamson et al. [364] und Ruiz et al. [305] untersuchen die Möglichkeit, Schütteln des Geräts zur Interaktion mit Anwendungen zu nutzen. Diesen Anwendungsfällen ist gemeinsam, dass sie i. d. R. mehrere Sensoren gleichzeitig zur Bestimmung des Kontext verwenden.

Mobile Geräte unterstützen diese Art der Interaktion durch eine Reihen unterschiedlicher Sensoren, wie etwa Beschleunigungssensor, Rotationssensor, Gyroskop oder Schrittzähler. Hierbei kann die Anzahl der in Hardware realisierten Sensoren unter umständen geringer sein als die Anzahl Sensoren, die einem Entwickler durch API-Schnittstellen angeboten werden. Zusätzliche (virtuelle) Sensoren werden in Software durch Aggregation von Daten mehrerer Hardware-Sensoren bereitgestellt.

Eine der prominentesten Anwendungen des Beschleunigungssensors in Zusammenarbeit mit dem Magnetfeldsensor ist die Klassifikation der aktuellen Orientierung eines mobilen Geräts in Hochformat und Querformat (vgl. Abbildung 3.8). Der Entwurf von UIs ist im Umfeld mobiler Anwendungen stark von der Orientierung des Geräts geprägt. UI-Designer stehen vor der Herausforderung, für jede Orientierung des Geräts eine im Sinn des Anwendungsfalls zweckmäßige Benutzungsoberfläche zu entwerfen. Die Darstellung tabellarischer Ansichten beispielsweise profitiert von einer Orientierung im Hochformat, weil dann mehr vertikale Darstellungsfläche für Tabellenzeilen verfügbar ist als im Querformat. Die Unterscheidung in Hochformat und Querformat ist deshalb zweckmäßig, da mobile Geräte i. d. R. über nichtquadratische Bildschirme verfügen und die unterschiedlichen Ausrichtungen Hochformat und Querformat je nach Anwendungsfall unterschiedlich gut geeignet sind, um Inhalte auf dem UI einer Anwendung darzustellen.

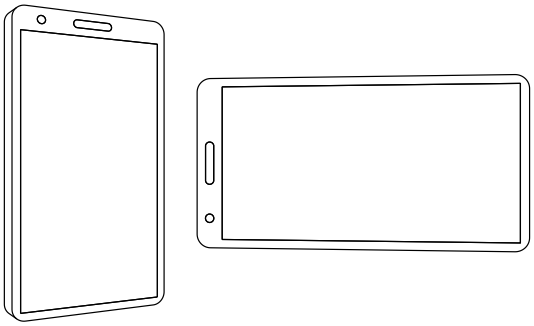


Abbildung 3.8: Klassifikation der Orientierung eines mobilen Geräts im Hochformat (links) und im Querformat (rechts)

Entwickler stehen hingegen vor der Aufgabe, einerseits die Vorgaben des UI-Designers umzusetzen und andererseits zu gewährleisten, dass sich eine Anwendung beim Übergang von Querformat ins Hochformat (und anders herum) spezifikationskonform verhält. Mobile Plattformen verhalten sich hier unterschiedlich. Die Plattform Apple iOS unterbricht eine Anwendung zur Anpassung des UI nicht (d.h. eine Korrektur des Seitenverhältnisses der virtuellen Darstellungsfläche) und blendet lediglich eine Animation der Oberfläche ein. Die Plattform Android hingegen terminiert die aktuelle *Activity*[4] (engl. Aktivität, hier als Bezeichnung einer

[4]Komponente der Android Anwendungsarchitektur die Logik und UI von in sich abgeschlossenen Ausschnitten von Anwendungsfällen ineinander vereint.

Android Anwendungsarchitekturkomponente) und startet sie neu im jeweils gültigen Oberflächenformat. Es ist in diesem Fall Aufgabe des Entwicklers dafür Sorge zu tragen, dass ein Anwender die Interaktion mit der Anwendungen trotz des Orientierungswechsels möglichst störungsfrei fortsetzen kann. Hierzu müssen ggf. Inhalte des UI wieder hergestellt werden.

Während für die Gestaltung von UIs normalerweise nur eine Klassifikation in Hochformat und Querformat ausschlaggebend ist, müssen diese beiden Formate technisch zusätzlich in Hochformat normal und auf dem Kopf stehend und Querformat links und rechts unterschieden werden. Diese Unterscheidung ist insbesondere für solche Anwendungen relevant, die auf die Gerätekamera(s) zugreifen. Während der Inhalt des UI in den meisten Fällen automatisch durch das Betriebssystem an die jeweils gültige Orientierung angepasst wird, muss die Ausrichtung der Gerätekamera (bspw. im Fall einer AR-Anwendung) durch den Entwickler manuell an die gültige Geräteorientierung angepasst werden, um zu verhindern, dass ein auf dem UI angezeigtes Live-Bild der Kamera auf dem Kopf stehend angezeigt wird.

Neben eines wahrnehmbaren Darstellungsdefekts aus der Perspektive des Anwenders kann dies außerdem zur Folge haben, dass Softwarekomponenten, die Annahmen über die Ausrichtung eines Kamerabilds machen, nicht ordnungsgemäß funktionieren. Eine App, die *OCR*-Technologie (engl. Optical Character Recognition, optische Zeichenerkennung) verwendet (z. B. Raiffeisen ELBA [290][5]), um Informationen aus einem mit der Gerätekamera aufgezeichneten Bild zu extrahieren, macht beispielweise die Annahme, dass die Eingabedaten in Form von Bildinformationen Text in der üblichen Ausrichtung, d. h. korrekt herum in landesspezifischer Schriftrichtung (links nach rechts, rechts nach links) vorliegt, um erwartungskonforme Resultate zu erzeugen.

Viele Anwendungen verwenden Informationen von Beschleunigungs- oder Rotationssensoren auch zu anderen Zwecken. Nicht für alle Anwendungen ist die Orientierung des Geräts in Hochformat und Querformat sinnvoll. Entwickler können die Entscheidung treffen, die Orientierung des UI unabhängig von der Orientierung des Geräts festzusetzen. Dies ist beispielsweise bei Spielen der Fall, in denen ohnehin auf die Verwendung von Standard-UI-Komponenten verzichtet wird. Beschleunigungs- oder Rotationssensoren können dann verwendet werden, um Anwendungsinhalte abhängig von der Orientierung des Geräts zu steuern.

Die Anwendungen Doodle Jump [237] (Abbildung 3.9a) ist ein Beispiel für eine App, in der Anwendungsinhalt durch direkte physikalische Manipulation der Orientierung des Geräts gesteuert wird. Die Anwendung Runtastic Squats [308] ist ein Beispiel, in dem Anwendungsinhalte durch Manipulation der vertikalen Position des Geräts gesteuert wird. Im Gegensatz zum Kontextparameter Standort (vgl. Abschnitt 3.1.2.1.3) ist hier jedoch nicht die Position des Anwenders im Sinne einer Standortbestimmung relevant, sondern eine Veränderung der vertikalen Position des Anwenders geringer Größenordnung.

Doodle Jump [237] implementiert die Steuerung des Spielgeschehens durch neigen des Geräts um die Längsachse, also Veränderung des Roll-Winkels. Nick- und Gier-Winkel sind für die Steuerung jedoch nicht bedeutsam. Die Orientierung des UI ist bei dieser Anwendung auf das Hochformat festgelegt, d.h. die Orientierung des UI wird nicht automatisch an die Orientierungsklassen Hoch- bzw. Querformat des Geräts angepasst. Die App Real Racing 3 [106]

[5]Die App Raiffeisen ELBA [290] verwendet die Gerätekamera um Zahlungsinformationen aus einem abfotografierten Überweisungsbeleg zu extrahieren.

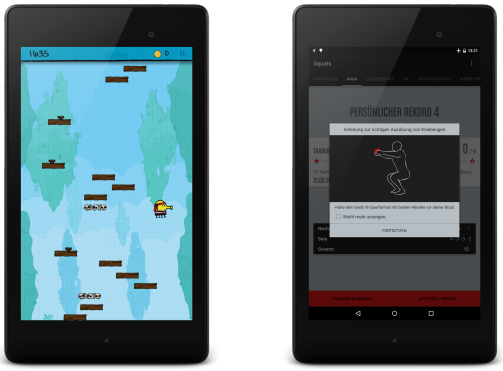


Abbildung 3.9: Bildschirmabdrucke der Apps Doodle Jump [237], Real Racing 3 [106] und Runtastic Squats [308]. Diese Apps verwenden die Manipulation der Orientierung oder der Position des Geräts als Eingabevektor.

ermöglicht dem Anwender die Steuerung eines virtuellen Rennwagens durch Rotation des Geräte um dessen Hochachse, d.h. Veränderung des Gier-Winkels. Hier ist die Anwendung auf das Querformat festgelegt. Runtastic Squats [308] ist eine App aus dem Fitness-Sektor und ermöglicht es dem Anwender automatisch die Anzahl der absolvierten Kniebeuge mitzuzählen, wenn das Smartphone am Körper getragen wird. Hierzu wird der Beschleunigungssensor verwendet, der mit ausreichender Zuverlässigkeit die Auf- und Abbewegung des Anwenders beim Kniebeugen feststellen kann.

Beschleunigung und Orientierung sind Parameter, die sich nicht zwingend auf den Kontext des Anwenders im Sinne situativer Faktoren beziehen, da sie i. d. R. willentlich vom Anwender verursacht werden. Beschleunigung und Orientierung des Geräts sind allerdings Kontextparameter, die Einfluss auf den Kontrollfluss der App nehmen können. Es sind Faktoren der Betriebsumgebung, deren Dynamik durch das Verhalten des Anwenders gesteuert wird. Die App hingegen ist selbst passiv. Sie kann keinen Einfluss auf Werte dieser Kontextparameter nehmen, soll aber in der Lage sein, in intendierter Weise auf sie zu reagieren.

Eine andere Domäne von Anwendungen, die Informationen von Beschleunigungs- oder Gyroskopsensoren verwenden, sind sogenannte Activity Tracker (vgl. Abschnitt 2.2.1). Prominentes und zugleich aus Anwendungssicht, nicht aber aus technischer Sicht, einfaches Beispiel ist der Schrittzähler, der die Anzahl der in einem Beobachtungszeitraum zurückgelegten Schritte aus einem Strom von Daten aus dem Beschleunigungssensor ermittelt. Unter der Annahme von Durchschnittswerten kann aus der Anzahl zurückgelegter Schritte auf die zurückgelegte Distanz und den Kalorienverbrauch geschlossen werden. Ein Beispiel für solche Anwendungen ist die App Google Fit [150] für Android Smartphones und Smartwatches (Abbildung 3.10).

Für Anwendungen dieser Domäne treten Bewegung, Beschleunigung und Änderung der Geräteorientierung unwillkürlich auf, d. h. sie werden nicht absichtlich zum Zweck der Anwendungssteuerung vom Anwender herbeigeführt. Sensrodaten treten als Kontextparameter im Sinne der Definition in Abschnitt 3.1.1.5 auf, d. h. durch Beobachtung der mittelbaren und unmittelbaren Betriebsumgebung der Anwendung werden situative Faktoren des Geräts, der Anwendung und des Anwenders algorithmisch oder heuristisch interpretiert und zur Steuerung des Kontroll- und Datenflusses nutzbar gemacht.

Abbildung 3.10: Die App Google Fit [150] auf einer Android Smartwatch. Sie verwendet die Kontextparameter Bewegung und Beschleunigung passiv, um Rückschlüsse auf körperliche Aktivitäten des Anwenders zu ziehen (z. B. Laufen, Rennen, Radfahren).

Aus der Perspektive der Softwareentwicklung ergeben sich durch die Verwendung dieser Interaktionskanäle neben funktionalen auch nicht-funktionale Anforderungen. Beschleunigung, Bewegung und Orientierung des mobilen Geräts können nur unter Zuhilfenahme entsprechender Sensoren in Apps verwendet werden. Die Verwendung dieser Sensoren wirkt sich allerdings auf den Energieverbrauch des Geräts aus, da zusätzliche Hardware-Bausteine aktiviert werden müssen, die im Normalbetrieb deaktiviert sind. Unachtsamer Umgang mit Sensoren kann zu vorzeitiger Erschöpfung des Geräteakkus führen. Dies gilt insbesondere im Zusammenhang mit dem Lebenszyklus der App, der auf mobilen Plattformen vom Betriebssystem gesteuert wird. Das Betriebssystem kann einer Anwendung jederzeit und ohne Konsens des Anwenders Ressourcen entziehen und anderen Anwendungen zuteilen. Es liegt in der Verantwortung des Entwicklers, die verwendeten Sensoren bei geeigneten Zustandswechseln des Lebenszyklus zu (de-)aktivieren, um den Energieverbrauch der Anwendung zu reduzieren. Sensoren sollten deaktiviert werden, wenn eine App durch eine andere App vom Bildschirm verdrängt wird, etwa wenn eine App durch einen eingehenden Telefonanruf unterbrochen wird.

Gegenwärtig stellen mobile Plattformen keine Schnittstellen bereit, um Sensoren in einem Testmodus zu betreiben, der es ermöglicht, ein SUT mit präzise im Testfall definierten Sensorinformationen zu versorgen. Anwendungen, die auf Sensoren zugreifen sind deshalb gegenwärtig vom Testen mit Automatisierungswerkzeugen ausgeschlossen, da diese nicht in der Lage sind, Sensorwerte innerhalb des SUT entsprechend den Vorgaben der Testfallspezifikation zu simulieren. Tests müssen deshalb manuell durchgeführt werden, wobei es menschlichen Testern unmöglich ist, die zum Zeitpunkt der Testfalldurchführung gültigen Kontextbedingungen ein zweites Mal exakt wiederherzustellen. Einerseits haben hier Faktoren Einfluss, die aus physikalischen Gründen durch menschliche Tester nicht steuerbar sind (z. B. Flussdichte von Magnetfeldern), andererseits aber auch Faktoren die menschliche Tester aus biologisch-motorischen Gründen nur begrenzt reproduzieren können.

Verwendet werden die Sensoren mobiler Plattformen i. d. R. durch Aufruf plattformspezifischer API-Funktionen, die nach dem *Observer Pattern*[6] implementiert werden. D. h. Anwendungskomponenten, die Sensordaten verwenden, registrieren sich als Abonnenten dieser Daten bei zuständigen Verteilerkomponenten der jeweiligen Plattform. Tritt eine Änderung der Sensordaten auf, ruft das System *Callback*-Methoden der Abonnenten auf. Je nach Sensor

[6] Softwareentwurfsmuster, bei dem Änderungen eines Datenobjekts an abhängige Objekte und Strukturen weitergegeben werden. Ein Objekt abonniert hierzu Änderungsinformationen, die von der Datenquelle an alle Abonnenten veröffentlicht wird.

kann das in festen Intervallen oder nur bei signifikanten Veränderungen von Werten der Fall sein. Übermittelt wird neben einem Zeitstempel pro Sensor ein Datenfeld sensorspezifischer Länge. Jede Komponente des Datenfelds enthält dann die Messgröße des jeweiligen Sensors zum Messzeitpunkt entlang der jeweiligen Achse des gerätedefinierten Koordinatensystems.

Die Beispielanwendung Mobiler Taxiruf ist agnostisch gegenüber der Orientierung des Geräts. Die Anwendung AR Tourist Information hingegen ist nur in der Lage, die AR-basierten Aspekte ihrer Funktionalität zu realisieren, wenn die Orientierung des Geräts im Verhältnis zur Erdoberfläche und im Verhältnis zum magnetischen Nordpol der Erde bekannt ist. Auf Geräten, die nicht über die erforderlichen Sensoren zur Ermittlung dieser Kontextparameter verfügen (d. h. Magnetfeldsensor, Beschleunigungssensor oder alternative Sensoren), kann die Anwendung deshalb nicht sinnvoll ausgeführt werden.

Diese Dissertation stellt in Kapitel 5 einen Lösungsansatz vor Gerätesensoren in einem Testmodus zu betreiben, der es erlaubt, Sensordaten zuverlässig für die Testdurchführung zu reproduzieren.

3.1.2.2.2 Netzwerkverfügbarkeit

Die Verbindung zu Netzwerken, insbesondere dem Internet, ist eine der wichtigsten und zugleich wesensbestimmenden Ressourcen für mobile Gräte. Die Verfügbarkeit von Netzwerkverbindungen ist darüber hinaus ebenfalls ein Kontextparameter im Sinn der in Abschnitt 3.1.1.4 und Abschnitt 3.1.1.5 diskutierten Definition. Die Präsenz bestimmter Netzwerkverbindungen wird durch die Betriebssysteme und individuelle Anwendungen mobiler Plattformen ständig überwacht. Hierdurch kann sichergestellt werden, für jede Netzwerkkommunikation das jeweils bestmögliche Netzwerk zu verwenden (z. B. im Hinblick auf die Latenzzeit und Übertragungsrate) und einzelne Aktivitäten auf die Verwendung bestimmter Netzwerktypen zu beschränken, z. B. im Hinblick auf Kosten der Netzwerkkommunikation.

Neben der bestimmungsgemäßen Verwendung von Netzwerkverbindungen können aus der Präsenz von Netzwerkverbindungen Rückschlüsse auf die Situation und die Aktivität des Anwenders gezogen werden. Wi-Fi-Netzwerke beispielsweise haben i. d. R. nur eine geringe Reichweite. Befindet sich ein mobiles Gerät innerhalb der Reichweite eines Wi-Fi-Netzwerkes, welches dem Gerät bekannt ist (d. h. der Anwender hat Zugangsdaten zu diesem Netzwerk bereitgestellt), ist mit dieser Information i. d. R. weiteres Wissen assoziiert. Das Wi-Fi des Bürokomplexes beispielsweise ist mit der Arbeitsstelle des Anwenders assoziiert. Ist es in Reichweite, kann gemutmaßt werden, dass sich der Anwender bei der Arbeit befindet. Ist hingegen ein Hotspot der Deutschen Bahn oder der Deutschen Telekom in Reichweite und ist zugleich der Verbindungsstatus der *Global System for Mobile Communication* (GSM), UMTS oder LTE Netzwerks durch typische Verbindungsabbrüche geprägt, kann gemutmaßt werden, dass sich der Anwender auf einer Bahnfahrt befindet. Ist wiederum das Wi-Fi-Netzwerk in der Wohnung des Anwenders in Reichweite, kann eine mobile Anwendung mutmaßen, dass sich der Anwender zuhause aufhält und daraufhin beispielsweise eine Heimautomatisierungslösung entsprechend parametrisieren.

Netzwerkverbindungen unterliegen in mobilen Szenarien jedoch Schwankungen hinsichtlich Latenzzeit, Übertragungsrate und Verfügbarkeit, so dass von einer mobilen Anwendung gefordert wird, tolerant gegenüber spontanen Veränderung aller Parameter der Netzwerk-

verbindung zu sein. Für praktische Zwecke werden Parameter von Netzwerkverbindungen i. d. R. auf wenige Klassen reduziert, etwa Wi-Fi, 3G/4G Mobilfunknetz, kostenpflichtig, kostenfrei, nicht-verfügbar. Ob diese Kontextinformation für eine mobile Anwendung relevant ist und wie eine mobile Anwendung auf Veränderungen des Kontextparameters Netzwerk reagiert wird durch den konkreten Anwendungsfall bestimmt.

3.1.2.2.3 Sonstige logische Kontextparameter

Im Normalfall werden Informationen aus der Kombination mehrerer Sensoren vom Betriebssystem verwendet, um Aspekte des praktischen Alltagsgebrauchs im normalen Betrieb zu steuern. Der Annäherungssensor in Kombination mit dem Helligkeitssensor beispielsweise wird verwendet um festzustellen, ob der Anwender das Gerät während eines Telefonats ans Ohr hält (Kouhi [231], Dietz et al. [96], Ballendat et al. [26]). So kann verhindert werden, dass durch Kontakt des berührungsempfindlichen Displays mit dem Ohr oder der Wange unbeabsichtigt eine Interaktion mit dem Gerät ausgelöst wird, wie etwa versehentliches Starten einer App oder Beendigung des Telefonats. Diese Funktion wird vom Anwender als selbstverständlich hingenommen. Aus der Perspektive des Software Engineering hingegen implementiert dieser Fall allerdings eine kontextsensitive Anwendung, in welcher konkret die Telefonieanwendung (oder auch das Betriebssystem) den Kontext der Nutzung, in diesem Fall ein Telefonat, erkennt und das berührungsempfindliche Display für das Zeitintervall für Interaktionen sperrt. Der Kontext Telefonat kann vom Gerät algorithmisch durch Aggregation unterschiedlicher Informationen bestimmt werden. Zunächst ist dem Betriebssystem bekannt, ob zu einem Zeitpunkt ein Telefonat geführt wird, da in diesem Fall charakteristische Hardware-Komponenten aktiv sind. Stellt der Annäherungssensor zusätzlich einen für ein Telefonat typischen Abstand zu einem Objekt fest (Ohr oder Wange des Anwenders) und stellt zeitgleich der Helligkeitssensor eine Abnahme der Lichtintensität fest, kann das System daraus ableiten, dass der Anwender das Gerät zum Telefonieren ans Ohr geführt hat.

Der Annäherungssensor jedoch nicht auf genau diese Verwendung festgelegt, sondern kann von Programmierern beliebig zur Realisierung anderer Anwendungsfälle verwendet werden, wobei sich dann die für diese Dissertation zentralen Fragestellungen des effektiven und effizienten Testens solcher Anwendungen stellen, da mobile Plattformen keine Schnittstellen zur Manipulation von Sensordaten im Kontext des Testens anbieten.

Eine ähnliche Anwendung unter Hinzuziehung von Daten aus Bewegungssensoren wurde von Wiese et al. [361] (vgl. Abschnitt 2.2.1) untersucht. Die Autoren untersuchen das Phänomen *Accidental Pocket Dialing* (engl. unbeabsichtigte Wahl einer Telefonnummer während sich das Mobiltelefon in der Hosen- oder Handtasche befindet, ausgelöst durch Kontakt des berührungsempfindlichen Displays mit Gegenständen oder der Hand in der Hosentasche) und entwerfen eine kontextbasierte Lösung zur Verhinderung des Problems. Als Kontextparameter werden unter anderem Daten des Beschleunigungssensors ausgewertet. Der Charakter der Kontextsensitivität wird hier insbesondere dadurch deutlich, dass Messwerte der verwendeten Sensoren hier als Werkzeug zur Realisierung eines übergeordneten Anwendungsfalls, nämlich der Verhinderung unbeabsichtigter Telefonanrufe, verwendet werden.

Ein weiteres Beispiel für einen Kontextsensor ist das Mikrofon eines mobilen Geräts, welches neben seiner eigentlichen Aufgabe – der Aufzeichnung von Audiodaten – ebenfalls die

Funktion erfüllt, das Niveau von Umgebungsgeräuschen zu erfassen. Diese Information kann verwendet werden, um das Verhalten des Geräts oder spezifischer Anwendungen gezielt zu beeinflussen. Als Beispiel für eine Beeinflussung des Geräteverhaltens kann etwa die Klingeltonlautstärke automatisch an das Niveau der Umgebungsgeräusche oder situative Faktoren des Anwenders angepasst werden (vgl. Tsai et al. [345]). Als Beispiel für eine gezielte Verwendung in einer Anwendung kann ein zweites Mikrofon verwendet werden, um die Sprachqualität einer Audioaufzeichnung (z. B. beim Telefonieren) durch herausfiltern von Umgebungsgeräuschen zu verbessern (vgl. Vaseghi [349], Jeub et al. [203]).

Ähnlich zu den Sensoren, mit denen die Bewegung oder Orientierung des Geräts festgestellt werden kann, gilt auch für die übrigen Sensoren, dass die Verwendung als primärer Eingabekanal und Verwendung als sekundärer Kontextparameter nicht klar voneinander abgrenzbar sind. Vielmehr ist es eine Frage der Umstände des konkret implementierten Anwendungsfalls, ob Sensordaten den Status eines primären Eingabekanals haben oder zur passiven Beobachtung der Umgebung von Anwender, Gerät und Anwendung verwendet werden. Auf der Ebene der Implementierung verliert diese Unterscheidung ihre Bedeutung.

In jedem Fall besteht die Anforderung, die unter Verwendung von Kontextparametern realisierten Funktionen reproduzierbar zu testen. Hier stellt sich die Frage, inwiefern Reproduzierbarkeit von Testdaten durch menschliche Akteure garantiert werden kann. In Fällen, in denen dies fraglich ist, verbleibt die Verwendung einer geeigneten Automatisierungstechnologie als einzige Option zur reproduzierbaren Testdurchführung, insbesondere in Fällen, in denen mehrere Kontextparameter gleichzeitig auf eine Anwendung einwirken und für die Realisierung des Anwendungsfalls relevant sind.

3.1.2.3 Fachliche Kontextparameter

Je nach Anwendungsfall kann es zur Begründung von Annahmen über situative Faktoren des Anwenders und dessen Umgebung notwendig sein, Informationen über andere Anwender oder deren Geräte einzubeziehen. Beispielsweise kann der Kontext des Anwenders als in einer Besprechung befindlich interpretiert werden. Aus einer positiven Interpretation dieser Situation könnte beispielsweise abgeleitet werden, dass das Mobiltelefon des Anwenders für die Dauer des Besprechungstermin stumm geschaltet wird, um Störungen zu vermeiden. Dieses zunächst trivial anmutende Beispiel ist technisch jedoch nur mit hohem Aufwand zu realisieren.

Die Kontextinformation könnte beispielweise durch eine Analyse der Übereinstimmung des Aufenthaltsortes des Anwenders mit dem in dessen Kalender verzeichneten Ort der Besprechung innerhalb eines ebenfalls aus dem Kalender extrahierten Zeitintervalls abgeleitet werden. Eine exakte Ortsbestimmung innerhalb von Gebäuden kann im Jahr 2016 allerdings nicht mit ausreichender Sicherheit umgesetzt werden, da die primär zur Ortsbestimmung eingesetzte Technologie GPS innerhalb von Gebäuden keine ausreichend Signalstärke erreicht und alternative Technologien, etwa Beacons, weder flächendeckend verfügbar noch standardisiert sind. Andere Ortungstechnologien (etwa Anmeldeinformationen an BSS der Mobilfunkinfrastruktur) kommen aufgrund ihrer geringen Genauigkeit von vornherein nicht in Betracht. Auch ist nicht sichergestellt, dass der Ort eines Besprechungstermins im Kalender des Anwenders geeignet hinterlegt ist. Beispielsweise könnte es sich um eine symbolische Standortangabe handeln, die nur unter Hinzuziehung weiteren Wissens auf eine geodätische Standortangabe

abbildbar ist. Eine Analyse des Standorts des Anwenders allein ist also nicht geeignet, um die Situation mit ausreichender Sicherheit zu interpretieren. Mobile Geräte verfügen aber neben Sensoren auch über weitere technische Eigenschaften und Technologien, die bei der Interpretation der Situation des Anwenders verwendet werden können. Beispielsweise kann die Präsenz bekannter Wi-Fi-Netzwerke einen Anhaltspunkt liefern. Im Sinn der Kontextinterpretation könnten allerdings auch Informationen über andere Entitäten hinzugezogen werden. Ein mobiles Gerät könnte beispielsweise über Netzwerktechnologien wie Wi-Fi oder Bluetooth die Anwesenheit weiterer mit diesem Besprechungstermin assoziierten Personen feststellen.

Zur ganzheitlichen Betrachtung des Kontextes der Ausführung von Software auf mobilen Geräten sind also neben den physikalischen Parametern, die direkt über die Sensoren des Geräts erfasst werden können auch andere Parameter von Bedeutung. Allerdings sind diese Parameter individuell komplex und in ihrer Vielfalt nicht beschränkt. Die im Rahmen dieser Dissertation untersuchte Technologie zur Automatisierung von Tests für kontextsensitive Anwendungen ist auf solche Kontextparameter fokussiert, die lokal auf dem Gerät unter Verwendung von Sensoren des Geräts ermittelt werden können. Eine Erweiterung auf fachliche Kontextparameter ist zwar im konkreten Einzelfall denkbar, aber in ihrer Gesamtheit nicht generalisierbar. Deshalb wird das Testen von Anwendungen unter Berücksichtigung fachlicher Kontextparameter in dieser Dissertation nicht berücksichtigt.

3.1.3 Zusammenfassung

In den vorangegangenen Abschnitten wurden einige prominente Kontextparameter und Möglichkeiten zu deren Verwendung diskutiert. Die Verwendung von Sensoren zur Erfassung physikalischer Kontextparameter ist auf den relevanten mobilen Plattformen der Smartphone-Ära in ähnlicher Weise realisiert. Eine Softwarekomponente registriert nach dem Subscriber-Entwurfsmuster an Schnittstellen des Plattform-API Komponenten zur Beobachtung spezifischer Kontextparameter und empfängt anschließend Aktualisierungen von Messwerten. Diese bilden die Grundlage einer fachlichen Interpretation, beispielsweise der Bestimmung des Standorts des Anwenders, der Orientierung des Geräts oder anderer situativer Faktoren. Die Interpretation von Kontextparametern ist je nach konkretem Parameter unterschiedlich aufwändig. Standortdaten beispielsweise werden i. d. R. vom den Schnittstellen des API der Zielplattform in einer direkt verwendbaren Form (i. d. R. WGS84-Koordinaten) bereitgestellt oder können mit geringem Aufwand durch Reverse Geocoding in eine intuitive Darstellung überführt werden. Andere Kontextparameter müssen aus verschiedene Kontextquellen, z. B. Sensoren, aggregiert werden, um in einer Anwendung nutzbar zu sein. Hierzu zählt beispielsweise die Orientierung eines mobilen Geräts im Verhältnis zu einem geodätischen Referenzsystem, in dem Roll-, Gier- und Nickwinkel relativ zum magnetischen Nordpol der Erde und der Erdoberfläche abgebildet werden. Die fachliche Interpretation von Kontextparametern ist hierbei eine Kernleistung einer kontextsensitiven Anwendung bei der Realisierung von Anwendungsfällen, durch welche die Qualität einer solchen Anwendung wesentlich bestimmt wird. Deshalb ist das Testen kontextsensitiver Anwendung eine Aktivität mit besonders hohem Ressourcenbedarf, weil einerseits der Heterogenität der Landschaft mobiler Geräte begegnet werden muss und andererseits die Komplexität im Umgang mit Kontext besondere Herausforderungen an das Testen stellt.

3.2 Auswirkungen von Mobilität und Kontextsensitivität auf den Betrieb von Softwaresystemen

Kontextsensitivität und Mobilität haben prägenden Einfluss auf die fachliche und technische Gestaltung mobiler Anwendungen. Eine Vielzahl mobiler Anwendungsfälle ist ohne Einbeziehung von Kontextparametern nicht realisierbar (z. B. ortsbasierte Anwendungen). Die Verarbeitung von Kontextparametern ist jedoch ein nicht-triviales Problem und erfordert deshalb eine detaillierte Spezifikation, eine sorgfältige Implementierung und gründliches Testen.

Die Spezifikation einer Software fixiert die zu realisierende Funktionalität. Durch die Integration von Mobilität und Kontextsensitivität wird der Raum für Defekte in Spezifikation und Implementierung stark erweitert. Der tatsächliche Grad der Spezifikationskonformität einer Software offenbart sich aufgrund inadäquaten Testens oftmals erst während des Betriebs. In den folgenden Abschnitten werden Auswirkungen von Mobilität und Kontextsensitivität auf den Betrieb von Softwaresystemen diskutiert, um zu verdeutlichen, dass Mobilität und Kontextsensitivität besondere Herausforderungen an Softwareprodukte stellen.

Die hier verwendeten Definitionen der Begriffe *Mobilität* (Abschnitt 3.1.1.1) und *Kontext* (Abschnitt 3.1.1.4) setzen den Ort der Ausführung desjenigen Teils einer mobilen Anwendung, mit dem der Anwender direkt interagieren kann, mit dem Standort des Anwenders gleich. Das bedeutet auf der einen Seite, dass viele Kontextparameter, die eine Anwendung durch Sensoren erfassen kann, mit großer Wahrscheinlichkeit auch auf den Nutzer zutreffen. Das gilt insbesondere für physikalische Eigenschaften der Betriebsumgebung, wie etwa Temperatur, Magnetfelder, Beschleunigung, etc. Der Anwender führt sein mobiles Gerät i. d. R. ständig mit sich (vgl. Ichikawa et al. [191], Wiese et al. [361]). Hierdurch ändern sich die äußeren Parameter der Betriebsumgebung ständig: Der Anwender bewegt sich von einem Ort zum anderen, verstaut das Gerät in der Tasche, holt es hervor, verändert dabei die Orientierung des Geräts, usw. Auf der anderen Seite heißt das aber auch, dass Aktivitäten des Nutzers unmittelbar auf die Verfügbarkeit von Ressourcen der Anwendung einwirken und somit eine vollständige Funktionsfähigkeit der Anwendung durch das Verhalten des Anwenders eingeschränkt wird. Der Anwender könnte sich beispielsweise mit seinem Gerät an einen Ort begeben, den GPS-Signale nicht erreichen können (z. B. Tiefgarage) oder der Anwender kann die Verwendung des GPS-Moduls verbieten, indem er die Systemeinstellungen entsprechend konfiguriert. Hierdurch entstehen Anforderungen an Fehlertoleranz und Robustheit mobiler Anwendungen, die über Anforderungen an Desktop- und Serveranwendungen hinausgehen (vgl. Schilit [313], Abowd et al. [2], Raento et al. [289], Baldauf et al. [25]). Während beispielsweise der Verlust einer Netzwerkverbindung bei Desktop- und Serveranwendungen i. d. R. eine Ausnahmesituation ist, gehört er im Umfeld mobiler Anwendung zum regelmäßigen und bestimmungsgemäßen Betrieb des mobilen Geräts.

Auf einige Aspekte des Betriebs mobiler Anwendungen haben Mobilität und Kontextsensitivität signifikanten Einfluss. Einer dieser Aspekte ist die Netzwerkverbindung, die in vielen mobilen Anwendungen Voraussetzung für eine korrekte Funktion ist. Mobilität des Anwenders induziert eine Vielfalt möglicher Umstände, durch welche die Qualität einer Netzwerkverbindung eingeschränkt wird. Betritt der Anwender beispielsweise Areale, die nicht oder nur schlecht von den Funksignalen der Mobilfunktechnologien durchdrungen werden

können (z. B. unterirdische U-Bahnhaltestellen, Tiefgarage) oder durchfährt einen Tunnel, so ist ein Verbindungsabriss nur durch besondere technische Einrichtungen zur Weiterleitung von Mobilfunksignalen zu vermeiden (beispielsweise durch GSM-*Repeater* in Tiefgaragen). Eine weitere mögliche Ursache für abbrechende Netzwerkverbindungen könnte auch eine Reise mit hoher Geschwindigkeit sein, wie etwa eine Zugfahrt in einem ICE der Deutschen Bahn, der regelmäßig Geschwindigkeiten von mehr als 250 $\frac{\text{km}}{\text{h}}$ erreicht. Verbindungsprobleme entstehen hier aus dem Umstand, dass Prä-LTE[7] Technologien bei hohen Geschwindigkeiten aufgrund des Doppler Effekts und häufigen *Handover* zwischen BSSs entlang der Bahnstrecke einen großen Anteil der verfügbaren Bandbreite auf Fehlerkorrektur und erneute Übertragung verlorener oder verstümmelter Datenpakete aufwenden müssen (vgl. Kaaranen [208], Sesia et al. [319]). GSM spezifiziert beispielsweise eine maximale Reisegeschwindigkeit von 250 $\frac{\text{km}}{\text{h}}$. Die UMTS-Spezifikation erlaubt Reisegeschwindigkeiten bis zu 500 $\frac{\text{km}}{\text{h}}$, wobei die Datenrate in Makrozellen dann allerdings auf 144 $\frac{\text{kbit}}{\text{s}}$ absinkt. In Mikrozellen wird eine maximale Datenrate 384 $\frac{\text{kbit}}{\text{s}}$ spezifiziert, die maximale Reisegeschwindigkeit von Teilnehmern jedoch auf 120 $\frac{\text{km}}{\text{h}}$ begrenzt (vgl. 3rd Generation Partnership Project (3GPP) [1]). Der limitierende Faktor ist dann die Häufigkeit des Handover zwischen verschiedenen BSSs. Zudem handelt es sich bei UMTS um ein sogenanntes *Shared Access Medium* (engl. Begriff für Übertragungsmedien, in denen sich Teilnehmer das Medium teilen), in dem die maximal erreichbare Datenrate von der Anzahl zeitgleich bei derselben BSS eingebuchten Teilnehmer abhängig ist (vgl. Kaaranen [208]). Im Jahr 2016 werden Geschwindigkeiten wesentlich über 300 $\frac{\text{km}}{\text{h}}$ von Zügen im Linienbetrieb i. d. R. nicht erreicht. Allerdings spielt neben der Reisegeschwindigkeit des Teilnehmers auch die Richtung der Bewegung relativ zur BSS eine Rolle. Der Anwender erlebt das als Herabsetzung der Netzwerkqualität, die sich durch geringe Datenübertragungsraten bei gleichzeitig hohen Latenzzeiten, bis hin zur völligen Unbenutzbarkeit, bemerkbar macht.

Ähnliches gilt auch für andere Kontextparameter. Die Verfügbarkeit eines stabilen GPS-Signals etwa wird durch den Aufenthaltsort des Anwenders wesentlich beeinflusst. Auch hier gilt, dass an Orten, die nicht von Satellitensignalen erreicht werden können, Ortsinformationen durch GPS nicht verfügbar sind. Allerdings stellt sich eine eventuelle Nichtverfügbarkeit technologiebedingt unter Umständen erst mit Zeitverzögerung ein, so dass im Zeitintervall zwischen störungsfreiem Betrieb und Eintritt der Störung (etwa betreten eines Gebäudes) Daten minderer Qualität vorliegen. Das kann sich beispielsweise durch eine Erhöhung des Präzisionsradius[8] bemerkbar machen. Für den Entwickler ergeben sich hierbei Gefahren im Umgang mit potenziell ungültigen, veralteten oder unpräzisen GPS-Informationen. Insbesondere besteht die Gefahr, dem Anwender auf dem UI der Anwendung Ortsinformationen anzuzeigen, die über einen Abruf der letzen bekannten Standortinformationen durch Aufruf einer entsprechenden API-Funktion ermittelt wurden. Diese Information kann ggf. stark veraltet sein, so dass der tatsächliche Standort des Anwenders mit dem Ergebnis des API-Aufrufs nicht übereinstimmt.

[7] LTE unterliegt ähnlichen Effekten, allerdings kompensiert die Technologie Fehler besser als Vorgängertechnologien, so dass LTE auch im Vergleich zu GSM und UMTS hohen Reisegeschwindigkeiten nutzbar ist.

[8] GPS stellt neben einem Koordinatenpaar ebenfalls Metainformationen zur Signalqualität bereit. Üblich ist es in den APIs mobiler Plattformen die Präzision einer GPS-Ortung in der Einheit Meter anzugeben. Der Wert beschreibt einen Radius um einen durch die WGS84-Koordinaten gegebenen Punkt. Mit einer plattformspezifischen Wahrscheinlichkeit wird der tatsächliche Standort des Anwenders innerhalb des Kreises angenommen, der durch das Koordinatenpaar und den Radius definiert wird.

Die Kontextparameter Standort und Netzwerkverbindung erzeugen durch ihre Kombination Effekte, durch welche die Präzision einer Ortsbestimmung wesentlich herabgesetzt werden kann. Mobile Geräte bedienen sich i. d. R. des sogenannten *Assisted*-GPS, bei welchem die Zeit bis zum ersten GPS-Fix[9] durch Auswertung von Mobilfunknetzen verkürzt wird (vgl. Djuknic und Richton [99], Zandbergen [374]). Wird ein mobiles Gerät nun für einen Zeitintervall im sogenannten Flugmodus (d. h. Deaktivierung aller Funksendeeinrichtungen des Geräts) betrieben und erfährt währenddessen eine signifikante Ortsänderung, liegen im GPS-Empfänger nach Zeitpunkt der Wiederinbetriebnahme veraltete GPS-Almanach[10] Daten vor, mit der Konsequenz, dass Standortinformationen erheblich vom tatsächlichen Standort abweichen können bis das GPS-Modul den Standort neu berechnet hat.

Für weitere physikalische Kontextparameter gilt, dass diese im Betrieb der mobilen Anwendungen Störungen unterliegen können, die durch Entitäten in der Umgebung des Anwenders bestimmt werden. So kann beispielsweise der Magnetfeldsensor durch magnetische Objekte im direkten Betriebsumfeld (etwa elektrische Antriebsmotore in Automobilen, Straßenbahnen etc., vgl. Skvortzov et al. [323], Mayor et al. [244]) in solcher Art beeinflusst werden, dass der intendierte Verwendungszweck des Sensors nicht mehr erfüllt werden kann. Der Sensor misst dann nicht mehr das Magnetfeld der Erde, sondern ein mit Störfaktoren überlagertes Magnetfeld. Der Sensor ist jedoch nur in Ausnahmesituation in der Lage, eine solche Störung zu erkennen, z. B. durch außergewöhnlich große Messwerte. Zudem ist es möglich, dass einige Kontextparameter nur unter Hinzuziehung anderer Parameter sinnvoll verwendet werden können, selbst wenn der Anwendungsfall das fachlich nicht erfordert. Ein digitaler Kompass etwa kann nur unter Einbeziehung des Standortes des Nutzers korrekt implementiert werden, um die ortsabhängige Deklination des Erdmagnetfelds zu berücksichtigen.

Im Umfeld mobiler Anwendung handelt es sich hierbei jedoch um Situationen des Alltagsgebrauchs. Spontaner Verlust der Netzwerkverbindung, ungültige Ortungsdaten oder von der Erwartung abweichende Sensordaten begründen keine Anwendungsdefekte im eigentlichen Sinn, sondern entsprechend der bestimmungsgemäßen Verwendung mobiler Geräte (vgl. Kleinrock [226]). Von mobilen Anwendung erwartet der Anwender deshalb, dass sie trotz kontextbedingter Schwankungen in der Verfügbarkeit und der Güte von Ressourcen ihre Aufgabe erfüllen oder zumindest auf die Wahrscheinlichkeit ungültiger Kontextparameter hinweisen.

Die Auswirkungen von Mobilität und Kontextsensitivität auf den Betrieb von Softwaresystemen ist also, dass zu keinem Zeitpunkt die Verfügbarkeit oder Gültigkeit einer Kontextressource vorausgesetzt werden darf. Spontane Veränderung in Verfügbarkeit und Qualität von Kontextparametern dürfen nicht als Ausnahmesituation begriffen werden, sondern sind vielmehr zu erwartende Konkretisierungen des Standardbetriebs. Die Qualität einer mobilen Anwendung bemisst sich deshalb nicht lediglich an der Qualität der Umsetzung des Standardanwendunginhaltes, sondern insbesondere durch die Qualität des Umgangs mit mobilitäts- und kontextinduzierten Veränderungen der Betriebsumgebung.

Im Fall der Beispielanwendung Mobiler Taxiruf äußern sich die Auswirkungen von Kontextsensitivität auf den Betrieb in der potenziellen Nichtverfügbarkeit der Standortinformation. Nachdem der Anwender sich für einen Zeitintervall innerhalb eines Gebäudes aufgehalten

[9] GPS-Fix bezeichnet eine durch Auswertung externer Quellen (hier Satellitensignale) berechnete Position.
[10] GPS-Almanach bezeichnet Daten zur Satellitennavigation geringer Genauigkeit mit langer Lebensdauer.

hat, das Gebäude anschließend verlässt und die Anwendung verwendet, um ein Taxi an seinen Standort zu rufen, ist technologiebedingt damit zu rechnen, dass die Lokalisierung via GPS mit einer Zeitverzögerung erfolgt. Um zu verhindern, dass der Anwender diese Wartezeit unangenehm empfindet, implementiert die Anwendung Mobiler Taxiruf parallel zur Lokalisierung via GPS zusätzlich die Lokalisierung unter Verwendung alternativer Technologien. Eine Standortbestimmung unter Ausnutzung der Netzwerktopologie ist i. d. R. zwar ungenau, aber schnell. Auf dem UI der Anwendung kann deshalb rasch eine Kartenansicht mit dem ungefähren Standort des Anwenders angezeigt werden. Bei Verfügbarwerden von Lokalisierungsinformationen via GPS wird die Kartenansicht dann mit der nun in höherer Genauigkeit vorliegenden Standortinformation angepasst. Zweck dieses Vorgehens ist die subjektive Verkürzung der Wartezeit, ähnlich eines Fortschrittsbalkens (vgl. Myers [260], Harrison et al. [168]). Es ist jedoch nicht gewährleistet, dass der Standort des Anwenders überhaupt bestimmt werden kann. Kann der Standort nicht technisch bestimmt werden, hat der Anwender hier die Möglichkeit, die Kontextinformation manuell einzugeben. Bei Abwesenheit einer Netzwerkverbindung hingegen wird die Anwendung nutzlos, da das Backend-System nicht erreichbar ist. Hierin wird der Einfluss von Mobilität und Kontextsensitivität auf eine Anwendung besonders deutlich. Je nach implementierten Anwendungsfall kann Nichtverfügbarkeit von Kontextinformationen eine Anwendung vollständig unbrauchbar machen.

Im Fall der Beispielanwendung AR Tourist Information wird der Einfluss von Kontextsensitivität auf den Betrieb von Anwendungen besonders deutlich. Die Einschränkungen bezüglich des Standortes des Anwenders gelten für diese App noch wesentlich restriktiver als für die App Mobiler Taxiruf. Denn die AR-Anwendung ist zwingend auf eine präzise Ortung des Anwenders angewiesen, ohne Option Kontextparameter manuell zu spezifizieren. Auf einem Stadtrundgang ist jedoch damit zu rechnen, dass sich der Anwender an Orte begibt, an denen kein GPS-Signal verfügbar ist. Für den Betrieb der Beispielanwendung AR Tourist Information ergeben sich hieraus besondere Anforderungen an den Fehlertoleranz und Robustheit.

Zusammenfassend sind die Auswirkungen von Mobilität und Kontextsensitivität auf den Betrieb von Software, dass kaum sichere Annahmen über Verfügbarkeit und Qualität von Ressourcen gemacht werden dürfen, selbst dann nicht, wenn diese Ressourcen zur Erbringung der fachlichen Funktion obligatorisch sind. Hier ist es wesentlich, die Qualität einer Ressource algorithmisch oder heuristisch zu bewerten, um rechtzeitig darauf reagieren zu können, wenn Ressourcen nicht-verfügbar werden. Dem Anwender darf zu keinem Zeitpunkt der Eindruck entstehen, dass eine Software bestimmungsgemäß funktioniert, obgleich das aufgrund herabgesetzter Qualität von Kontextparametern nicht der Fall sein kann.

3.3 Zusammenfassung

In den vorangegangenen Abschnitten wurden die Auswirkungen der Kontextfaktoren Plattform und Gerät, Datum und Uhrzeit, Mobilität des Anwenders sowie physikalischer Kontextparameter diskutiert. Diesen ist gemeinsam, dass durch sie die Anwendungskomplexität und damit die Kosten für Aktivitäten der Qualitätssicherung signifikant erhöht werden.

Architekturen mobiler Plattformen führen fremdgesteuerte Lebenszyklusmodelle in Anwendungen ein, durch welche außerplanmäßige Abbruch- und Eintrittspunkte in Anwendun-

gen entstehen, die durch Tests abgedeckt werden müssen. Die je nach Plattform variierende Geräteheterogenität ist ein weiterer Komplexitätstreiber, der den Testaufwand mit der Anzahl potenzieller Zielgeräte überproportional skaliert. Es muss eine repräsentative Anzahl von Geräten identifiziert, beschafft und gepflegt werden, um beim Testen von Anwendungen verwendet zu werden. Aufgrund herstellerspezifischer Eigenschaften ist es i. d. R. nicht ausreichend, einen Anwendung auf nur einem Gerät oder gar nur auf dem Emulator zu testen.

Der Kontextparameter Zeit ist ein Komplexitätstreiber, dessen Auswirkungen sich je nach Umfang eines Entwicklungsprojekts bis in das Backend-System einer mobilen Anwendung hinein erstrecken können. Verwendet eine App diesen Kontextparameter, kann das bedeuten, dass Tests für diese Anwendungen ebenfalls durch temporale Abhängigkeiten charakterisiert werden. Je nachdem, wie viel Kontrolle über das Backend-System ausgeübt werden kann, bedeutet das u. U., dass bestimmte Tests nur zu bestimmten Zeiten ausgeführt werden können, wodurch die Testplanung vor zusätzliche organisatorische Herausforderungen gestellt wird.

Anwendungen für mobile Systeme werden i. d. R. in Mobilitätsszenarien verwendet, die selbst dann zusätzlichen Testaufwand generieren, wenn die Anwendung keinen expliziten Gebrauch vom Standort des Anwenders macht. Der Standort des Anwenders beeinflusst jedoch die Verfügbarkeit weiterer Ressourcen, wie etwa Netzwerk, deren Verfügbarkeit deshalb in mobilen Anwendungen nicht vorausgesetzt werden darf. Abhängigkeiten zu solchen Ressourcen müssen in Tests adressiert werden, wodurch die Anzahl durchzuführender Tests weiter erhöht wird. Verwendet eine Anwendung den Standort des Anwenders als Eingabevektor, ergibt sich hieraus die Notwendigkeit die Anwendung unter der Annahme fachlich und technisch gültiger und ungültiger Standorte zu testen. Ultimativ kann das bedeuten, dass bei Nicht-verfügbarkeit adäquater Simulationstechnologie repräsentativ selektierte Standorte aufgesucht werden müssen, um Tests durchzuführen (vgl. Agarwal et al. [5]). Dies gilt insbesondere im Zusammenhang mit sekundären Kontextparametern, die in Abhängigkeit vom Standort des Anwenders Schwankungen unterliegen (z. B. Netzwerkverfügbarkeit, Magnetfeld) oder sich der Anwender beim Verwenden der Anwendung in Bewegung befinden darf (*In-motion User*, *In-motion Device* [53, 55, 54], z. B. Zugfahrt, Autofahrt).

Herausragende Bedeutung nehmen sensorbasierte Kontextfaktoren beim Testen mobiler Anwendungen ein. Es existieren in mobilen Geräten i. d. R. keine Schnittstellen zur Versorgung eines SUT mit Testdaten für Sensoren, so dass zunächst einmal nur manuelle Tests in Frage kommen. Aus physiologisch-motorischen Gründen ist es einem menschlichen Akteur jedoch kaum möglich, Testdaten bei der Testdurchführung exakt zu reproduzieren. Es kommen zwar prinzipiell Unit-Tests in Frage, um auswertende Algorithmen zu testen. Tests auf darüberliegenden Ebenen können gegenwärtig jedoch ausschließlich manuell mit Näherungswerten für Testorakel durchgeführt werden. Das heißt, die Bewertung, ob ein Test einer sensorbasierten Anwendung erfolgreich bestanden ist, unterliegt der subjektiven Beurteilung des Testers.

Eine Verbesserung von Simulationstechnologie und insbesondere deren Integration in Automatisierungstechnologien kann wesentlich dazu beitragen, den Aufwand für das Testen mobiler, kontextsensitiver Anwendungen herabzusetzen. Diese Dissertation untersucht eine modellbasierte Automatisierungstechnologie gemäß dem Paradigma *Design for Testability* [281], mit dem Ziel die Simulation von Kontext mit der Automatisierung der Testdurchführung zu integrieren.

Kapitel 4

Testen und Testautomatisierung kontextsensitiver mobiler Softwaresysteme

Softwareprodukte sind Resultate der Implementierung einer Menge von Anforderungen. Diese werden während der Anforderungsanalyse ermittelt, dokumentiert und in einer Spezifikation vertraglich belastbar fixiert. Die Spezifikation enthält sowohl funktionale als auch nicht-funktionale Anforderungen. In die Kategorie funktionaler Anforderungen fallen alle Aspekte eines zu implementierenden Systems, die Funktionalität aus Anwendersicht bereitstellen. Nicht-funktionale Anforderungen beschreiben Rahmenbedingungen, unter denen funktionale Anforderungen zu realisieren sind, wie etwa Zuverlässigkeit, Benutzbarkeit, Flexibilität, Wartbarkeit, Änderbarkeit, Prüfbarkeit, Korrektheit etc. (ISO/IEC [199]).

Insbesondere das Qualitätsmerkmal Korrektheit ist Gegenstand der in dieser Dissertation untersuchten modellbasierten Methode zur Erstellung und automatisierten Durchführung von Softwaretests für mobile, kontextsensitive Anwendungen. Software verhält sich korrekt bzw. konsistent gegenüber einer Spezifikation, wenn sie alle funktionalen und nicht-funktionalen Anforderungen erfüllt. Die Feststellung der Konsistenz eines Softwaresystems gegenüber seiner Spezifikation wird als Testen bezeichnet. Beim Testen kommen problemadäquate Methoden und Werkzeuge zum Einsatz, um zu überprüfen, ob ein Softwaresystem strukturell und dynamisch den Anforderungen entspricht, also genau die Eigenschaften aufweist, die in der Spezifikation fixiert sind.

Aufgrund der Komplexität von Software ist es technisch und wirtschaftlich weder möglich noch sinnvoll, vollständig und unter Berücksichtigung aller möglichen äußeren Umstände zu testen (Patton [275], Liggesmeyer [236], Myers et al. [261]). Dies gilt insbesondere für die Klasse der mobilen, kontextsensitiven Anwendungen (vgl. Zhang und Adipat [376], Dehlinger und Dixon [90], Haller [164], Joorabchi et al. [204], Muccini et al. [258], Kapitel 2). Durch die hohe Komplexität des Zusammenhangs zwischen Anwendungsverhalten und Kontext ist es nicht möglich, eine vollständige Testabdeckung unter Berücksichtigung aller möglichen Werte individueller Kontextparameter zu erreichen. In der Praxis werden deshalb funktionale und nicht-funktionale Anforderungen priorisiert und anhand der sich hieraus ergebenden Rangfolge einzelner Aspekte unterschiedlich intensiv gegenüber einer Spezifikation überprüft.

Dieses Kapitel bettet die besonderen Eigenschaften mobiler, kontextsensitiver Anwendungen (vgl. Kapitel 3) in das Umfeld des Testens ein und bereitet damit die Grundlage für die in Kapitel 5 vorgestellte Methode zur Testautomatisierung mobiler Anwendungen. Hierzu werden im Abschnitt 4.1 zunächst grundlegende Methoden, Eigenschaften und Begriffe des Testens von Software im Bezug zum hier untersuchten Konzept der automatisierten Testfallerstellung und Durchführung diskutiert. Insbesondere erfolgt eine Klassifizierung von Defekten mit besonderem Fokus auf mobile Anwendungen, welche als Grundlage für die in Abschnitt 5.5.1 diskutierten Anforderungen an ein Werkzeug zur Testautomatisierung und Simulation von Kontextparametern dienen. Die Auswirkungen von Mobilität und Kontextsensitivität auf das Testen von Software werden in Abschnitt 4.2 betrachtet. Abschnitt 4.3 thematisiert grundlegende Methoden, Technologien und Werkzeuge der Testautomatisierung.

4.1 Testen von Software

Das Testen von Software ist eine Tätigkeit, die in unterschiedlichen Phasen des Softwareprozesses durchgeführt wird. Je nach Phase und beteiligten Bedarfsträgern wird eine Software unter unterschiedlichen Perspektiven betrachtet. Entwickler haben beispielsweise eine andere Sicht auf eine Software als ihr zukünftiger Anwender. Um diese Unterschiede im Testen zu adressieren, werden Softwaretests unter unterschiedlichen Perspektiven auf das zu testende System klassifiziert. Zu etablierten Klassifikationskriterien gehören beispielsweise Abstraktionsebene des Tests, durchführender Akteur, Zeitpunkt der Testdurchführung, fachlicher Inhalt oder die technische Orientierung (Patton [275], Liggesmeyer [236], Myers et al. [261]).

Üblich sind beispielsweise Klassifikationen von Testaktivitäten nach Art der Prüftechnik, welche beschreibt, wie die Qualität von Software gemessen wird. In Frage kommen hier z. B. statische Tests, die eine Software prüfen, ohne sie auszuführen. Dies kann z. B. durch Quellcodeaudits oder Analysetechniken auf dem fertigen Kompilat erfolgen. Im Fokus statischer Tests stehen strukturelle Eigenschaften. Beispielsweise kann ermittelt werden, welche Klassen oder Funktionen im System mit welcher Häufigkeit verwendet werden. Den statischen Tests stehen dynamische Tests gegenüber. Diese untersuchen das Laufzeitverhalten eines zu testenden Systems. Beispielsweise simulieren dynamische Tests das Verhalten des zukünftigen Anwenders funktionsorientiert, um Softwaredefekte vor der Auslieferung zu ermitteln. Dynamische Tests können ebenfalls eine Analyse struktureller Eigenschaften zum Gegenstand haben.

Ebenfalls üblich ist eine Klassifikation nach dem Testkriterium. Sie beschreiben, was durch Tests ermittelt werden soll, wobei Tests jeweils auf einen spezifischen Aspekt des Softwareprodukts ausgerichtet sind. Adressiert werden etwa konkrete funktionale und nicht-funktionale Anforderungen wie etwa Systemsicherheit, Performanz oder das Verhalten eines Systems unter großer Last. Bei der Klassifikation nach Testkriterium werden spezifische Aspekte aus der Anforderungsdokumentation isoliert und Testaktivitäten werden auf diesen Aspekt fokussiert.

Weiterhin kann eine Klassifikation nach dem gewählten methodischen Ansatz oder dem Wissensstand über das zu testende System erfolgen. Sind beispielsweise Interna der zu testenden Software bekannt und haben Einfluss auf den Inhalt und die technische Durchführung von Tests, so werden diese Tests als White-Box-Tests bezeichnet. Als Black-Box-Tests hingegen werden Tests bezeichnet, die ohne Kenntnisse der Implementierung durchgeführt werden.

Ebenfalls geläufig ist eine Klassifikation anhand konkreter Qualitätseigenschaften von Software, wie etwa Zuverlässigkeit, Funktionalität, Benutzbarkeit, Wartbarkeit (vgl. ISO/IEC 9126 [199]). Tests orientieren sich hierbei nicht an konkreten funktionalen Inhalten, sondern insgesamt an Metriken messbarer Qualitätseigenschaften, die sich über den gesamten Funktionsumfang erstrecken können.

Obwohl die wissenschaftliche Literatur zum Testen von Software eine Vielzahl an Klassifikationen kennt, gibt es keinen Konsens bezüglich einer Taxonomie, die Tests jeweils genau einer Kategorie zuordnet. Vielmehr ist eine Frage der konkreten inhaltlichen, technischen und organisatorischen Ausrichtung eines bestimmten Tests, wie dieser zu klassifizieren ist.

Im Fokus dieser Dissertation stehen Akzeptanztests. Diese werden aus der Perspektive des Anwenders ausgeführt und sind stets dynamische Tests, d. h. sie werden durch Ausführen des SUT unter gesicherten Bedingungen durchgeführt. Sie können sowohl funktionale als auch nicht-funktionale Eigenschaften von Software zum Gegenstand haben. Dem gegenüber stehen die hier nicht weiter betrachteten statischen Tests, zu deren Durchführung das SUT nicht ausgeführt wird, sondern der Code analysiert wird.

Die folgenden Abschnitte erläutern die in dieser Dissertation verwendeten Begriffe aus dem Umfeld des Testens von Software (Abschnitt 4.1.1) im besonderen Hinblick auf die hier untersuchten Forschungshypothesen (Abschnitt 1.2.4). Ziel ist eine Einordnung der verwendeten Begriffe in das Umfeld des Testens. Im Anschluss erfolgt eine Diskussion der Klassifikation von Softwaredefekten insbesondere im Umfeld mobiler, kontextsensitiver Anwendungen.

4.1.1 Begriffe und Taxonomie

Die im Kapitel 5 vorgestellte Methode und Automatisierungstechnologie für das Testen mobiler Systeme bedient sich einer umfangreichen Taxonomie domänenspezifischer Begriffe. Einige dieser Begriffe sind im Kontext dieser Arbeit besonders relevant, weil sie in der vorgestellten Methode zur automatisierten Generierung und Durchführung von Tests für mobile Anwendungen zentrale Artefakte repräsentieren. Die spezifische Bedeutung dieser Begriffe wird in den folgenden Abschnitten erläutert.

4.1.1.1 Testen

Aktivitäten des Softwareprozesses, deren Fokus auf der Sicherstellung der Qualität eines Softwareprodukts liegt, werden unter dem Begriff *Testen* zusammengefasst. In einer weiten Begriffsdefinitionen umfassen diese sowohl planerische Aktivitäten wie beispielsweise die Entwicklung einer Teststrategie und die Erstellung von Testplänen. Engere Definitionen (z. B. die des *Standards Coordinating Committee of the Computer Society of the* IEEE [192, 193]) beschreiben Testen als solche Aktivitäten, die ein Softwaresystem unter spezifizierten Bedingungen ausführen und dabei Resultate beobachten, aufzeichnen und bewerten, um Rückschlüsse auf die Spezifikationskonformität des getesteten Systems zu ziehen. Ein besonderer Fokus liegt dabei auf der Überprüfbarkeit und dem Nachweis der Korrektheit dieser Aktivitäten.

Eine ebenfalls etablierte Begriffsdefinition stammt von Myers et al. [261] und beschreibt Testen als:

> „*[. . .] process of executing a program with the intent of finding errors.*“ [261]

Testen ist also ein Vorgang, der gezielt versucht Defekte in Software aufzudecken. Der Intention nach ist diese Definition nach Myers et al. insofern von der des Standardglossars der IEEE abzugrenzen, als dass hier keine wertneutrale Beurteilung des getesteten Systems gemeint ist. Myers et al. argumentieren, dass menschliche Akteure zielorientiert agieren und das Erreichen des Ziels einen außerordentlich hohen Stellenwert im menschlichen Wertemodell einnimmt. Dieser Effekt tritt auch beim Testen von Software auf, indem unterbewusst Testfälle oder Testdaten gewählt oder erstellt werden, von denen eine geringe Wahrscheinlichkeit vermutet wird einen Defekt in einer Software zu entdecken [261].

> „*Human beings tend to be highly goal-oriented [. . .]. If our goal is to demonstrate that a program has no errors, then we will be steered subconsciously toward this goal; that is, we tend to select test data that have a low probability of causing the program to fail.*“ [261, Kap. 2, S. 6]

Myers et al. unterstellen hier ein psychologisches Paradoxon, weil die wertschöpfende Tätigkeit der Softwareentwicklung die Erstellung eines korrekten Systems ist und nicht das Erstellen von Tests, die keine Defekte auffinden. Wertschöpfendes Testen äußert sich aber gerade darin, zu versuchen Defekte früh, d. h. vor der Auslieferung des Produkts, zu entdecken und nicht etwa durch Fokussierung auf solche Tests, die das sogenannte *Happy-Path*-Kriterium (Kontrollflusspfad durch eine Anwendung, der den typischen, störungsfreien Anwendungsfall repräsentiert, vgl. Vieira et al. [351], Hartmann et al. [172]) befriedigen und unter Vernachlässigung kritischer Kontrollflusspfade versuchen, zu vermeintlich positiven Testergebnissen zu gelangen. Die Definition des Begriffs *Testen* nach Myers et al. ist deshalb intentional destruktiv ausgelegt, um den Einfluss psychologischer Effekte dieser Art zu verringern.

Patton [275] legt eine ähnlich gelagerte Definition des Begriffs *Testen* zugrunde und postuliert in diesem Zusammenhang Qualitätskriterien an einen Softwaretester:

> „*The goal of a software tester is to find bugs, find them as early as possible, and make sure they get fixed.*“ [275]

Das zielgerichtete frühe Auffinden von Defekten ist auch hier ein prominenteres Ziel als das bloße neutrale Beobachten und Bewerten der Spezifikationskonformität von Software.

Im Rahmen dieser Arbeit wird der Begriff *Testen* als Baustein des Softwareprozesses verwendet, der alle notwendigen Aktivitäten beinhaltet, um im Rahmen einer definierten Teststrategie den wiederholbaren Nachweis der Konformität eines Softwareproduktes gegenüber seiner Spezifikation zu erbringen. Diese Aktivitäten umfassen die Festlegung der Teststrategie, das Erstellen von Testplänen, die Erstellung, Zusammenstellung und Ausführung von Testfällen, die Auswahl und Anwendung geeigneter Werkzeuge zur Durchführung von Tests sowie die Analyse und Bewertung von Testergebnissen. Insbesondere fällt die Erstellung von Modellen zur automatisierten Ableitung von Testfällen unter diese Definition. Hierdurch wird das Modellieren von Softwaresystemen von einer Aktivität, die üblicherweise in den frühen Phasen von Softwareprozessen stattfindet zu einer Aktivität, die sich über den gesamten Softwareprozess – und hier insbesondere auf Phasen der Qualitätssicherung – erstreckt. Diese Erweiterung ist notwendig, um auch für solche Softwaresysteme modellbasierte Tests zu erstellen, zu denen keine Entwurfsmodelle vorliegen (vgl. Beziehung zwischen Modell und Code nach Kelly

und Tolvanen [216], Abschnitt 5.2.1). In Abgrenzung zur Definition nach Pol et al. [285] wird als Messgröße nicht der erforderliche Zustand eines Softwareproduktes herangezogen, sondern ausschließlich dessen Spezifikation definiert. Hierbei handelt es sich um unterschiedliche Anforderungen, da die Spezifikation einer Software nicht zwingend mit dem der Intention nach erforderlichen Zustand der Software (aus Anwenderperspektive) identisch ist, nämlich im Fall einer unvollständigen oder defekten Spezifikation. Die in dieser Dissertation betrachtete Methode zur modellbasierten Testfallgenerierung und automatisierten Ausführung hat ausschließlich Softwareartefakte zum Gegenstand. Fehlerhafte Spezifikationen werden hiervon explizit nicht berührt. Die Definition des Begriffs Testen umfasst im Rahmen dieser Dissertation ebenfalls das Erstellen von Testdaten sowie die Bereitstellung und Parametrisierung einer Testumgebung und insbesondere die Simulation eines Kontext für die Testdurchführung.

4.1.1.2 Defekt

Der Standardglossar der IEEE [192] definiert einen *Defekt* als Abweichung eines errechneten, beobachteten oder gemessenen Wertes oder eines Programmzustands vom spezifikationsgemäß zu erwartenden Wert oder Zustand. Liggesmeyer [236] beschreibt einen Defekt als statisch im Programmcode vorhandene Ursache von Fehlverhalten oder des Ausfalls von Software. Diese Definition weist jedoch keinen Bezug zur Spezifikation der Software auf, es bleibt also unklar, anhand welcher Anhaltspunkte ein Fehlverhalten einer Software diagnostiziert wird.

Patton [275] elaboriert hier wesentlich detaillierter und beschreibt einen Defekt als einen Zustand einer Software, die (1) Eigenschaften nicht aufweist, die gemäß Spezifikation gefordert sind, (2) Eigenschaften aufweist, die gemäß Spezifikation unzulässig sind, (3) Eigenschaften aufweist, die in der Spezifikation nicht explizit adressiert werden, (4) Eigenschaften aufweist, die in der Spezifikation nicht adressiert werden, aber adressiert werden sollten und (5) die Software Eigenschaften aufweist, die in den Augen des zukünftigen Anwenders als zu kompliziert zu verstehen, zu kompliziert zu verwenden, für den Verwendungszweck inadäquat oder inperformant ist. Die Kriterien 4 und 5 treten hier insofern besonders hervor, da sie primär der Spezifikation eines Softwaresystems entspringen als dem Kompilat mit der Konsequenz, dass nicht nur die Software selbst Gegenstand des Testens ist, sondern auch ihre Spezifikation.

Diese Dissertation stützt sich auf die Definition des Standardglossars der IEEE und bezeichnet mit einem Defekt jede Abweichung von der Systemspezifikation, ergänzt um die Kriterien 1 und 2 nach Patton. Im hier untersuchten Konzept der modellbasierten Testautomatisierung wird davon ausgegangen, dass Modelle der Systemspezifikation als Eingabeartefakte für die Generierung von Testfällen verwendet werden. Der Algorithmus zur Generierung von Testfällen (vgl. Abschnitt 5.3.3) verfügt nur über Informationen, die in der Systemspezifikation enthalten sind. In der Konsequenz ist der hier untersuchte Ansatz nicht geeignet, Fehler in der Systemspezifikation zu entdecken. Eine Analyse der Systemspezifikation auf Fehler, Unvollständigkeit oder Unplausibilität ist nicht Gegenstand dieser Dissertation.

4.1.1.3 Teststrategie

Pol et al. [285] definieren eine *Teststrategie* als planmäßiges Vorgehen zur Festlegung, welche Teile einer Software mit welchem Aufwand und unter Zuhilfenahme welcher Technologien und

Werkzeuge zu testen sind. Zu testende Aspekte einer Software sollen so ausgewählt werden, dass anzunehmende Einsatzszenarien bestmöglich abgebildet und entsprechend ihres jeweiligen Risikos priorisiert werden. Deshalb sind Testaktivitäten so zu planen, dass im Rahmen des Budgets häufige Anwendungsszenarien oder Anwendungsszenarien mit großem Schadenpotenzial intensiv getestet werden, während seltene Anwendungsfälle oder unkritische Anwendungsfälle vergleichsweise gering priorisiert werden. Die Teststrategie bestimmt ebenfalls die Auswahl der zum Testen zu verwendenden Werkzeuge und testbegleitende Rahmenbedingungen wie etwa Testzyklen, Testphase, Testarten.

Für mobile, kontextsensitive Anwendungen gilt es eine Auswahl von Testfällen so zu treffen, dass einerseits den besonderen Anforderungen mobiler, kontextsensitiver Anwendungen Rechnung getragen wird und andererseits das Testen der Software mit einem angemessenen ökonomischen Aufwand durchgeführt werden kann. Hierzu müssen zusätzlich zu üblichen Abdeckungsmetriken des Testens weitere, neue Abdeckungsmetriken wie z. B. Geräteabdeckung oder Kontextabdeckung berücksichtigt werden. Geräteabdeckung ist eine Metrik, die beschreibt welcher Anteil der zielgruppenrelevanten Geräte durch Tests abgedeckt wird. Kontextabdeckung beschreibt, welcher Anteil der für eine Anwendung relevanten Kontextparameter durch Tests abgedeckt wird.

Im Rahmen dieser Arbeit wird der Begriff *Teststrategie* so erweitert, dass auch solche Rahmenbedingungen, die für mobile, kontextsensitive Softwaresysteme spezifisch sind einbezogen werden. Für den Fall manueller Testaktivitäten standortbezogener Funktionen in mobilen Anwendungen legt die Teststrategie beispielsweise fest, zu welchem Anteil solche Funktionen an anzunehmenden zukünftigen Einsatzorten zu testen sind (Feldtest) und für welche Funktionsbereiche Labortests ausreichend sind.

4.1.1.4 Testplan

Ein *Testplan* ist die Spezifikation und Dokumentation von Testaktivitäten. Während die Teststrategie die Einbettung von Testaktivitäten in den Softwareprozess zum Gegenstand hat, wird in einem Testplan festgehalten, welche Teile eines Softwareprodukts unter Verwendung welcher Ressourcen, unter Einsatz welcher Testarten auf welchen Teststufen (vgl. Abschnitt 4.1.1.6) in welchen Testphasen, unter Verwendung welcher Werkzeuge mit welcher Intensität getestet werden. Ebenfalls wird im Testplan fixiert, wie häufig individuelle Testfälle unter Verwendung welcher Testdaten zu wiederholden sind und zu welchen Zeitpunkten im Softwareprozess Testaktivitäten erfolgen (IEEE [193, 192]).

In dieser Dissertation wird der Begriff *Testplan* in genau dieser Definition verwendet. In Ergänzung umfasst die Testplanung die Bestimmung von Zielplattformen in definierten Versionen konkreter mobiler Geräte und Emulatorkonfigurationen.

4.1.1.5 Testart

Eine *Testart* beschreibt, in welcher Art und Weise der Testprozess mit welchen konkreten Zielen durchgeführt wird. Hierbei werden vor allem die Aspekte fachliche und technische Korrektheit des zu testenden Systems unterschieden, welche nicht zwingend deckungsgleich sind. Von besonderem Interesse ist hierbei die Tiefe der Kenntnis über das zu testende System.

Tests, bei denen strukturelle Eigenschaften des Systems bis hin zum Quellcode bekannt sind werden als White-Box-Test (auch *Glass-Box-Tests*) bezeichnet. Tests, bei denen strukturelle Eigenschaften des Systems nicht bekannt sind bzw. fachliche Aspekte des zu testenden Systems ungeachtet der strukturellen Systemeigenschaften im Fokus stehen, werden als Black-Box-Tests (auch funktionale Tests) bezeichnet. Ebenfalls existieren Mischmodelle, diese werden als *Grey-Box-Tests* bezeichnet.

Eine Unterscheidung von *Testarten* ist ebenfalls nach anderen Kriterien möglich. Beispielsweise nach ausführendem Akteur oder nachzuweisender Qualitätseigenschaft. In dieser Dissertation werden Akzeptanztests untersucht, deshalb ist eine weitere Klassifikation über White-Box-Tests, Grey-Box-Tests und Black-Box-Tests hinaus nicht weiter relevant.

4.1.1.5.1 White-Box-Tests

Softwaretests, die unter Kenntnis und Zugriff auf den Quellcode von Software durchgeführt werden, sind in der Taxonomie der Softwaretechnik als White-Box-Tests (auch strukturelle Tests, Glass-Box-Test) verankert. Sie werden unter expliziter Kenntnis der Struktur und des Verhaltens des zu testenden Softwaresystems erstellt. Der Entwickler des Tests kann hierbei auf den Quellcode zugreifen und verfügt deshalb über dasselbe Wissen wie der Programmierer der Software. Gegenüber anderen Testarten, in denen dieses Wissen nicht oder nur teilweise vorliegt, besteht der Vorteil, dass Testfälle Interna des zu testenden Systems direkt adressieren können. White-Box-Tests können beispielsweise Werte von Variablen oder Inhalte von Speicheradressen zur Bewertung der Konformität einer Software gegenüber ihrer Spezifikation heranziehen (vgl. Patton [275], Liggesmeyer [236], Myers et al. [261], IEEE [192]).

Die Erstellung von White-Box-Tests ist in der Praxis häufig Aufgabe von Programmierern, beispielsweise jener Entwickler, die das zu testende System implementieren. White-Box-Tests zeichnen sich zudem dadurch aus, dass sie im Rahmen des *Test Driven Development* (TDD) häufig noch vor der Implementierung des zu testenden Quellcode[s] erstellt werden.

Ein weiterer Vorteil von White-Box-Tests gegenüber anderen Testarten besteht insbesondere in der Erstellung von Testfällen. Hier kann der Entwickler des Testfalls gezielt strukturelle Eigenschaften der Software berücksichtigen. Beispielsweise können Testfälle gezielt so konstruiert werden, dass während der Testausführung Verzweigungspunkte im Quellcode der Software mit Variablenbelegungen entsprechend den Verzweigungsbedingungen adressiert werden. So ist eine präzise Kontrolle der Testabdeckung (z. B. Verzweigungsabdeckung, Pfadabdeckung, Anweisungsabdeckung) gewährleistet. Diese Möglichkeit bleibt alternativen Testarten i. d. R. aufgrund mangelnder Kenntnisse der internen Funktionsweise der Software verschlossen.

Aufgrund der Quellcode[nahen] Natur von White-Box-Tests können i. d. R. nicht alle Teststufen (vgl. Abschnitt 4.1.1.6) bedient werden. Insbesondere Akzeptanztests werden i. d. R. nicht durch White-Box-Tests realisiert. Sie dienen einer reproduzierbaren, funktionalen Überprüfung einer Software nach den Anforderungen eines Bedarfsträgers. Strukturelle Eigenschaften der Software sind für die funktionale Leistungserbringung jedoch i. d. R. nicht relevant. Andere Quellcode[nahe] Teststufen werden hingegen primär durch White-Box-Tests realisiert. Hierzu gehören u. a. Unit-Tests und Integrationstests, da beide nahe an der Implementierung eines Softwaresystems ausgelegt sind.

4.1.1.5.2 Black-Box-Tests

Softwaretests, deren Erstellung und Durchführung ohne Kenntnis der inneren Struktur und ohne den Zugriff auf den Quellcode erfolgt und die funktionale Aspekte einer Software zum Gegenstand haben, werden als Black-Box-Tests (oder funktionale Tests) bezeichnet. Gegenstand eines Black-Box-Tests ist die Überprüfung der Spezifikationskonformität eines Softwareartefakts hinsichtlich seiner Funktion. Im Gegensatz zum White-Box-Test steht hier nicht korrekte Implementierung einzelner Quellcode[artefakte] im Fokus, sondern die Funktionsfähigkeit eines Softwaresystems aus der Perspektive des zukünftigen Anwenders (vgl. Patton [275], Liggesmeyer [236], Myers et al. [261], IEEE [192]).

Entgegen anderen etablierten Definitionen (Patton [275], Myers et al. [261], IEEE [192]) argumentiert Liggesmeyer [236], dass Black-Box-Tests und funktionale Tests keine Synonyme sind und unterstellt, dass mit Black-Box-Test-Techniken auch nicht funktionsorientiert getestet werden kann. Liggesmeyer führt hierzu die Testtechnik *Back-to-Back-Testing* an, bei welcher unterschiedliche Implementierungen derselben Spezifikation unter Verwendung identischer Testdaten miteinander verglichen werden. Erzeugen diese unterschiedliche Resultate, ist mindestens eine Implementierung defekt. Zwar kann dieses Prinzip gut mit automatisierten Testwerkzeugen umgesetzt werden, ist aber nur dann effizient, wenn unterschiedliche Implementierungen ebenfalls maschinell erzeugt werden können oder die Kritikalität des Anwendungsfalls ein manuelles Erstellen unterschiedlicher Implementierungen rechtfertigt. Da unterschiedliche Implementierungen mobiler Anwendungen im Jahr 2016 nicht maschinell erstellt werden können, spielt *Back-to-Back-Testing* in dieser Dissertation keine weitere Rolle. Da mobile Anwendungen häufig für mehr als nur eine Plattform realisiert werden, bestünde hier jedoch die Möglichkeit, Back-to-Back-Testing auf die Kompilate für verschiedene Plattformen anzuwenden. Weicht das Verhalten auf einer Plattform vom Verhalten auf einer anderen Plattform ab, könnte ein Defekt vorliegen, es sei denn, die Spezifikation sieht für unterschiedliche Plattformen unterschiedliches Anwendungsverhalten vor.

Während White-Box-Tests überprüfen, ob eine Software *richtig* (im Sinne technischer Korrektheit) implementiert wurde, überprüfen Black-Box-Tests, ob *das Richtige* (im Sinne der Spezifikation) implementiert wurde (Wynne und Hellesoy [368]). Beide Testarten sind nur begrenzt voneinander abhängig, d. h. eine Software kann technisch korrekt, aber nicht spezifikationskonform sein. Der umgekehrte Fall, also Spezifikationskonformität bei nichtkorrekter technischer Implementierung, ist hingegen nur in Sonderfällen möglich. Dieser Fall kann eintreten, wenn Funktionalität implementiert wurde, die gemäß der Spezifikation nicht vorgesehen ist. Die Software ist dann definitionsgemäß defekt (vgl. Abschnitt 4.1.1.2), wenngleich Black-Box-Tests nicht in der Lage sind diesen Defekt zu erkennen. Ein Testplan wird deshalb i. d. R. sowohl White-Box-Tests als auch Black-Box-Tests für eine Anwendung definieren.

Black-Box-Tests werden durchgeführt, indem alle für den Anwender direkt verfügbaren Schnittstellen mit Testdaten versorgt werden und anschließend die dem Anwender verfügbaren Schnittstellen auf erwartungskonforme Ausgaben überprüft werden. Es obliegt dem Verantwortlichen der Testfallerstellung beim Entwurf von Black-Box-Tests deshalb eine besondere Sorgfaltspflicht bei der Auswahl der Testdaten und der Auswahl kritischer Programmabläufe, die den Arbeitsfluss eines typischen Anwenders möglichst realitätsgetreu abbilden.

Black-Box-Tests sind primär fachlich orientiert. Deshalb muss ihre Erstellung und Durchführung im Kontrast zu White-Box-Tests nicht zwingend durch Akteure mit Programmierkenntnissen erfolgen. In der Praxis ist es zweckmäßig, diese Aufgabe durch einen Bedarfsträger aus der Fachabteilung und gerade nicht aus der IT-Abteilung durchführen zu lassen. So kann sichergestellt werden, dass die Erstellung und Durchführung von Testfällen den notwendigen Bezug zu fachlichen Inhalten des zu testenden Systems aufweisen und technischen Details im Vergleich keine überproportionale Aufmerksam zuteil wird.

Das in dieser Arbeit untersuchte Konzept zur automatisierten modellbasierten Generierung und Durchführung von Tests hat Akzeptanztests zum Gegenstand. Diese gehören zur Familie der Black-Box-Tests, sie basieren deshalb ausschließlich auf der Systemspezifikation oder auf zum Zweck des Testens erstellten Modellen der zu testenden Software.

4.1.1.5.3 Grey-Box-Tests

In der Praxis des Testens von Softwaresystemen kann es vorkommen, dass sich ein funktionaler Testfall nicht vollständig im Rahmen eines Black-Box-Tests abbilden lässt. Das kann beispielsweise der Fall sein, wenn die Realisierung eines funktionalen Tests eine strukturelle Metainformation benötigt, die selbst aber keinen direkten Einfluss auf die zu testende Funktion hat. Bei der manuellen Durchführung von Tests ist dieser Fall ungewöhnlich, bei der Verwendung von Automatisierungstechnologie hingegen repräsentiert er den Normalfall.

Im Umfeld von Web-Anwendungen oder mobilen Anwendungen für die Plattform Android kann dieser Fall beispielsweise eintreten, wenn im Rahmen eines Akzeptanztests ein Element der graphischen Benutzungsoberfläche explizit adressiert werden soll, um einen funktionalen Aspekt einer Anwendung zu testen (z. B. Anmeldung mit Benutzernamen und Passwort an einem Dienst oder einer Anwendung). Bei manueller Durchführung kann ein spezifisches UI-Element durch einen menschlichen Akteur unschwer identifiziert werden. Eine Automatisierungstechnologie verfügt hingegen nicht über Fähigkeit einer intuitiven Beurteilung der Frage, welches UI-Element adressiert werden soll. Insbesondere dann nicht, wenn ein UI mehrere gleichartige Elemente (z. B. mehrere Texteingabefelder mit unterschiedlicher Bedeutung, etwa Benutzername und Passwort) anbietet.

Dem Charakter nach handelt es sich dann zwar um einen Black-Box-Test, da ausschließlich die Überprüfung der Spezifikationskonformität des SUT Gegenstand der Bewertung der Testergebnisse ist. Zur technischen Realisierung der automatisierten Testdurchführung muss jedoch eine strukturelle Eigenschaft des SUT, nämlich die Identität des UI-Elements (z. B. Texteingabefeld, Schaltfläche), bekannt sein, um der Automatisierungstechnologie eine maschinelle Stimulation (z. B. Eingabe von Text in ein Textfeld, Simulation einer Interaktion des Anwenders) des fraglichen UI-Elements zu ermöglichen.

In diesem Fall können Grey-Box-Tests verwendet werden, um zwar einerseits den fachlichen Fokus der Black-Box-Test-Strategie zu erhalten, andererseits aber dennoch in ausreichendem Umfang strukturelle Eigenschaften des zu testenden Systems zu berücksichtigen. Grey-Box-Tests sind demnach ein Mischmodell aus White-Box-Test und Black-Box-Test (vgl. Linzhang et al. [239], Liggesmeyer [236], Yang et al. [373]).

Diese Dissertation untersucht eine modellbasierte Methode zur Generierung von Testfällen und deren automatisierte Ausführung für mobile, kontextsensitive Anwendungen am konkre-

ten Beispiel der Plattform Android. Informationen über strukturelle Eigenschaften des UI des SUT sind hier Voraussetzung zur Identifikation spezifischer UI-Elemente. Deshalb erzeugt die hier untersuchte Methode zur modellbasierten Testfallgenerierung Grey-Box-Tests, die ihrer Intention nach jedoch Black-Box-Tests entsprechen, da die dem Test bekannte Strukturinformation bei der Bewertung der funktionalen Spezifikationskonformität nicht relevant ist.

4.1.1.6 Teststufe

Tests für Softwaresysteme können auf unterschiedlichen Stufen durchgeführt werden. Zum Gegenstand haben sie jeweils unterschiedliche Artefakte des zu testenden Systems und sie orientieren sich i. d. R. grob an den Phasen der Softwareentwicklung. Kennzeichnendes Merkmal ist vorrangig jedoch das bei der Testdurchführung zu untersuchende Artefakt. Definiert werden *Teststufen* i. d. R. durch ihre Nähe zum Quellcode des zu testenden Systems bzw. zum konkreten Anwendungsfall. Die Spanne der Teststufen erstreckt sich hierbei von Komponententests über Integrationstests, Systemtests, Akzeptanztests bis hin zu Regressionstests. Eine Teststufe bezeichnet eine Menge gemeinsam durchgeführter Testaktivitäten, die bestimmten Abstraktionsebenen eines Softwaresystems zugeordnet werden können. Indes sind Tests unterschiedlicher Teststufen nicht disjunkt. Eine Teststufe ist eine Charakterisierung eines individuellen Tests oder einer Testsuite, keine hierarchische Abgrenzung.

4.1.1.6.1 Komponententest

Komponententest (auch Unit-Test) bezeichnet einen Test für Software, der dem Quellcode sehr nahe ist. In der Regel werden Komponententests zusammen mit dem Quellcode programmiert, häufig in derselben Programmiersprache, und haben konkret identifizierbare Quellcode[artefakte] zum Gegenstand, wie beispielsweise einzelne Klassen in objektorientierten Programmiersprachen oder einzelne Methoden (IEEE [192, 193]). In Abgrenzung zum Akzeptanztest konzentriert sich der Fokus von Komponententests auf „*building the thing right*“ (Wynne und Hellesoy [368]). Zur Realisierung von Komponententests haben sich einige Technologien (z. B. JUnit oder CUnit, vgl. Abschnitt 2.5.1.1) etabliert, deshalb bezeichnet insbesondere die englische Begriffsvariante *Unit-Test* sowohl das Konzept als auch spezifische Realisierungsvarianten unter Verwendung spezifischer Technologie. Insbesondere Werkzeuge für die Erstellung von JUnit/CUnit Tests sind Bestandteil vieler IDEs. Im Rahmen des TDD sind Komponententests zum Kernkonzept der Softwareentwicklung auf Quellcode[ebene] avanciert.

Im TDD wird der Entwicklungsprozess dahingehend umgekehrt, dass die Entwicklung von Tests vor der Entwicklung der Funktionalität erfolgt. Iterativ wird dann die Komplexität von Testfällen erhöht, während die zu testende Softwarefunktion den Testfällen hinterher entwickelt wird (Beck [33, 32], Dehlinger und Dixon [90]). Insgesamt ist die Werkzeugunterstützung für Komponententests im Vergleich zu anderen Teststufen vergleichsweise fortgeschritten.

Komponententests sind für diese Dissertation insofern von Bedeutung, als dass in der Praxis der Werkzeugverwendung, insbesondere auf der Plattform Android, Komponententests und Tests auf anderen Stufen nicht mehr klar voneinander abgegrenzt werden können. Die Technologien Robotium (Abschnitt 2.5.2.1) und UIAutomator (Abschnitt 2.5.2.3) beispielsweise realisieren Tests so nah am Quellcode, dass objektiv nicht mehr zwischen einzelnen Teststu-

fen unterschieden werden kann. Insbesondere basiert das in dieser Dissertation als Basistechnologie verwendete Automatisierungswerkzeug für Quellcode[ferne] UI- bzw. Akzeptanztests Calabash (vgl. Abschnitt 2.5.2.6) intern auf dem Framework Robotium, das seinerseits auf JUnit basiert und deshalb klar im Spektrum der Komponententest-Werkzeuge angesiedelt ist. Die Technologie Calabash hingegen ist klar auf der Ebene funktionaler Tests verortet, womit auf das in dieser Dissertation erstellte Testautomatisierungswerkzeug (vgl. Kapitel 5) dieser Stufe zuzuordnen ist.

4.1.1.6.2 Integrationstest

Ein Integrationstest ist ein Softwaretest, dessen unmittelbarer Gegenstand die Interoperabilität mindestens zweier Komponenten miteinander ist (IEEE [193, 192], Patton [275]). Von den beteiligten Komponenten wird angenommen, dass diese ihre jeweiligen Komponententests erfolgreich absolviert haben. Ein Integrationstest kann allerdings auch zwischen Komponenten erfolgen, die nicht auf derselben Abstaktionsebene angesiedelt sind. So können Integrationstests beispielsweise einzelne Klassen in objektorientierten Entwicklungsprojekten oder einzelne Funktionen oder Quellcode[abschnitte] im Verbund mit komplexen Teilsystemen testen. Ziel von Integrationstests ist es zu überprüfen, ob einzelne Komponenten spezifikationskonform miteinander zusammenarbeiten.

Im Umfeld mobiler Anwendungen für die Plattformen Apple iOS, Android und Microsoft Windows Phone haben Integrationstests eine besonders hervorgehobene Bedeutung, da die Architekturen dieser Plattformen Entwurfsmuster vorgeben, die einen funktional orientieren Komponentenschnitt erzwingen, in denen Anwendungen aus maximal lose gekoppelten Komponenten (z. B. Android *Activities*, *Services*, *ContentProvider* [128, 132, 144] oder Apple iOS *UIViewController* [18]) gebildet werden. Funktionsorientiertes Testen einer mobilen Anwendung für die Plattformen Apple iOS, Android und Microsoft Windows Phone ist deshalb zwangsläufig in die Klasse der Integrationstests einzuordnen.

4.1.1.6.3 Systemtest

Systemtests sind Softwaretests, bei denen das in seine Betriebsumgebung integrierte System oder Produkt in seiner Gesamtheit überprüft wird (IEEE [193, 192]). In der Regel werden Systemtests als Simulation der tatsächlichen Anwendung unter Verwendung von Testdaten durchgeführt. In Abgrenzung zum Akzeptanztest können Systemtests nicht nur funktionale Inhalte haben, sondern ebenfalls strukturelle, statische Aspekte des Systems testen. Hierbei können mehrere Testarten zum Einsatz kommen.

Im Umfeld mobiler Anwendungen stellt sich die Erstellung und Durchführung von Systemtests besonders herausfordernd dar, da eine mobile Anwendung ihre Funktionalität häufig nur in Verbindung mit einem Backend-System erbringt. Hierdurch wird neben der mobilen Anwendung auch das Backend-System zum Gegenstand von Systemtests, wodurch mehrere Softwareartefakte, die sich in ihrer Funktionalität ggf. nur partiell überschneiden, auf unterschiedlicher Technologie basieren und u. U. unterschiedlichem organisatorischen Hoheitsgebiet zugeordnet sind einem gemeinsamen Test unterzogen werden.

4.1.1.6.4 Akzeptanztest

Ein Akzeptanztest ist eine Art von Softwaretest, bei welchem ein System in seiner Gesamtheit hinsichtlich seiner Vollständigkeit und Spezifikationskonformität aus einer funktionalen Perspektive überprüft wird. Ein Akzeptanztest resultiert bei positivem Ausgang in der Akzeptanz des Nutzers (IEEE [192, 193]). In Abgrenzung zu anderen Tests werden Akzeptanztests i. d. R. aus der Sicht des zukünftigen Anwenders durchgeführt.

Akzeptanztests umfassen sowohl fachlich-inhaltliche Prüfungen als auch Tests der Anwendungsoberfläche, sogenannte UI-Tests. Inhalt ist die Prüfung der Korrektheit einer Software aus Sicht des zukünftigen Anwenders. Aus diesem Grund werden Akzeptanztests häufig nicht durch Entwickler, sondern durch Vertreter des Auftraggebers durchgeführt (Myers et al. [261]).

In Fokus dieser Arbeit steht die automatisierte Generierung und Durchführung von Testfällen auf der Teststufe Akzeptanztests für kontextsensitive, mobile Systeme. Aufgrund der Besonderheiten mobiler Anwendungen in Abgrenzung zu nicht-mobilen Softwaresystemen (vgl. Abschnitt 3.2, Abschnitt 4.2) ist es notwendig, die konkreten fachlichen Inhalte von Akzeptanztests auf die Anforderungen mobiler Systeme anzupassen. Deshalb umfasst die hier angewendete Definition des Begriffs Akzeptanztest nicht nur solche Aktivitäten, die Software auf Korrektheit aus fachlich-funktionaler Sicht des Anwenders prüfen, sondern ebenfalls alle Aktivitäten zur Herstellung eines adäquaten Zustands der Testumgebung. Dies schließt ganz konkret die Simulation relevanter Kontextparameter (vgl. Abschnitt 3.1.2) während der Testdurchführung mit ein.

4.1.1.6.5 Regressionstest

Der Lebenszyklus eines Softwareprodukts ist nach der initialen Entwicklung und Inbetriebnahme i. d. R. nicht abgeschlossen. Zwischen Erstinbetriebnahme und Außerdienststellung erfolgen Zyklen der Wartung und Weiterentwicklung, in denen sowohl Defekte korrigiert werden als auch inhaltlich-funktionale Anpassungen vorgenommen werden. Es muss sichergestellt werden, dass durch diese Veränderungen keine neuen Defekte, eine sogenannte Regression (Rückgang oder Rückschritt, lat. *regredi*: umkehren, zurückgehen), in das System eingeführt wird. Dies wird durch Regressionstests sichergestellt. In Abgrenzung zu den übrigen Teststufen zeichnet sich ein Regressionstest nicht durch seine technische oder inhaltliche Orientierung, sondern durch den Zeitpunkt seiner Durchführung aus. Die Aktivität des Regressionstestens beinhaltet das wiederholte Ausführen einer Menge von Testfällen unterschiedlicher Testarten, Teststufen und Testtypen nachdem Änderungen an einer Software vorgenommen wurden. Das kann bereits während der initialen Entwicklung in inkrementellen Softwareprozessen oder nach Instandhaltungsaktivitäten erfolgen. Bei der prakitschen Verwendung von CI-Systemen oder SCM-Systemen ist das Bestehen einer Regressionstestsuite häufig Voraussetzung, um neuen Quellcode in den aktiven Entwicklungszweig einer Software einführen zu dürfen.

Die hier untersuchte Methode zur automatisierten Testfallerstellung und Durchführung ist für Regressionstests besonders vorteilhaft, da sich Regressionstests durch häufige Testdurchführung auszeichnen. Diese Teststufe ist daher ein Umfeld, in dem sich die Investition in die Einführung einer Automatisierungslösung besonders schnell amortisiert.

4.1.1.7 Testfall

Die IEEE definiert einen Testfall im Standardglossar der Software Engineering Terminologie [192] und im IEEE Standard für Software und Systemtest Dokumentation [193] als Menge von Testeingabedaten, Ausführungsbedingungen und erwarteten Ergebnissen (sogenannte Sollergebnisse), die mit dem konkreten Ziel erstellt werden, eine Software oder einen Anwendungspfad durch eine Software auf Konformität zur Anforderungsspezifikation zu überprüfen.

Testfälle stellen Kernartefakte dieser Dissertation dar. Ziel der hier vorgestellten Methode zur Testautomatisierung ist die Generierung von Testfällen aus Modellen des Systementwurfs. Deshalb treten Testfälle einerseits als abstrakte Artefakte des Testens gemäß der Definition der IEEE auf, andererseits aber auch als technische Artefakte. Diese manifestieren sich zum einen in Form von Modellelemente eines Metamodells zur Testfallmodellierung (Abschnitt 5.2.3.2) und zum anderen in Quellcode[fragmenten] der Testautomatisierungstechnologie Calabash (Abschnitt 2.5.2.6, Abschnitt 5.4).

Im Fokus stehen Testfälle für kontextsensitive, mobile Anwendungen. Aufgrund deren besonderer Eigenschaften (vgl. Abschnitt 3.2, Abschnitt 4.2) werden Testeingabedaten im Vergleich zu anderen Softwaresystemen um solche Daten erweitert, die sich nicht nur unmittelbar aus der Interaktion des Anwenders mit der Software ergeben (z. B. Eingabe von Werten in Felder des UI). Aus diesem Grund umfasst der Begriff Testfall in der Erweiterung der Definitionen der IEEE [194, 192, 193] neben den Testeingabedaten, Ausführungsbedingungen und Sollergebnissen zusätzlich solche Parameter, durch welche die Ausführungsbedingungen um kontextspezifische situative Faktoren des SUT und des Anwenders (vgl. Abschnitt 3.1.2) ergänzt werden.

4.1.1.8 Testschritt

Testfälle beschreiben definitionsgemäß Pfade durch eine zu testende Anwendung (dem SUT). Solche Pfade repräsentieren Anwendungsfälle und umfassen eine Abfolge individueller Aktionen, die Interaktionen des zukünftigen Anwenders mit der Anwendung repräsentieren.

Im Kontext dieser Arbeit werden diese individuellen Aktionen als *Testschritte* bezeichnet. Ein Testschritt repräsentiert eine (atomare) Aktion eines Anwenders im typischen Nutzungskontext der zu testenden Anwendung, die nicht sinnvoll in weitere Detailstufen zerlegt werden kann. Liggesmeyer [236] führt den Begriff Testschritt ebenfalls mit einer nahezu identischen Semantik als einen Schritt eines Testfalls ein, konkret als Eintritt eines Ereignisses in einem bestimmten Zustand, dem Auslösen von Aktionen und ggf. einem Zustandswechsel [236].

Typische Testschritte sind beispielsweise die Eingabe von Werten in UI-Elemente oder die Manipulation von UI-Elementen wie etwa das Anklicken einer Schaltfläche. Es ist eine Frage des konkret zu untersuchenden Softwareartefakts, wann eine Aktion als atomar gilt (d. h. nicht sinnvoll in detailliertere Aktionen zerlegt werden kann). Bei der Durchführung eines funktionalen Tests einer UI-Maske im Rahmen eines Akzeptanztests (beispielsweise eine Plausibilisierung von Eingabewerten) kann die Eingabe einer Zeichenkette als atomare Aktion gelten. Im Rahmen eines UI-Test, in dem die Korrektheit eines bestimmten UI-Elements geprüft wird (z. B. ein UI-Element zur automatischen Vervollständigung von Benutzereingaben), ist es hingegen sinnvoll die Eingabe einer Zeichenkette in weitere Aktionen zu zerlegen, die

eine sukzessive Eingabe einzelner Zeichen repräsentieren. Das in Abschnitt 5.2.3.2 vorgestellte Metamodell zur Testfallmodellierung ist dergestalt, dass Aktionen auf unterschiedlichen Abstraktionsebenen definiert werden können, um solche Fälle abzubilden.

Der in dieser Dissertation verwendete Algorithmus zur Generierung von Tests aus Systemmodellen erzeugt individuelle Testschritte, die in übergeordneten Testfällen aggregiert werden. Hierzu definiert das in Abschnitt 5.2.3.2 vorgestellte Metamodell ein entsprechendes Modellierungselement *TestStep*, welches einen Testschritt inhaltlich abbildet. Die Testautomatisierungstechnologie und insbesondere die Kontextsimulationstechnologie (vgl. Abschnitt 5.5.1) verwenden Testschritte als atomare Eingabeobjekte.

4.1.1.9 Testsuite

Software bietet Anwendern i. d. R. Unterstützung für mehr als nur einen konkreten Anwendungsfall. Anwendungsfälle können in mehreren Varianten auftreten. Beispielsweise können innerhalb eines Anwendungsfalls Alternativen zulässig sein oder einzelne Aktionen oder Gruppen von Aktionen können optional wiederholt werden. Deshalb ist die Korrektheit eines Softwaresystems im Normalfall mit einer Menge von Testfällen sicherzustellen.

Eine *Testsuite* ist eine Zusammenstellung inhaltlich miteinander verwandter Testfälle. Beispielsweise können in einer Testsuite alle Tests eines funktionalen Aspekts einer Software zusammengefasst werden. In der Terminologie von Testautomatisierungswerkzeugen beschreibt der Begriff *Testsuite* ein technisches Artefakt, das eine Menge von Testfällen singulär adressierbar macht, beispielsweise bei JUnit (Abschnitt 2.5.1.1) und verwandten Technologien, wie etwa dem hier als Grundlage verwendeten Werkzeugs Calabash. Der Begriff *Testsuite* ist hier auf das Calabash-Artefakt *Feature* (Abschnitt 2.5.2.6) übertragbar.

4.1.1.10 Vorbedingung

Die Bewertung der Ergebnisse eines Tests kann nur unter der Annahme erfolgen, dass diese Ergebnisse das Resultat einer Durchführung von Testfällen unter definierten Bedingungen sind. Testergebnisse sind nicht verwertbar, wenn Umgebungsbedingungen, die Einfluss auf das Testergebnis haben, zum Zeitpunkt der Testdurchführung nicht vollständig bekannt waren. Es ist dann nicht auszuschließen, dass ein SUT sich tatsächlich korrekt verhält, aber dennoch ein vermeintlicher Softwaredefekt beobachtet wurde, dessen Ursache weder in der Spezifikation des Testfalls noch in der Spezifikation der Software berücksichtigt ist. Denkbar ist ebenfalls, dass sich das SUT vermeintlich korrekt verhält, dieses Verhalten aber nur zufällig durch eine unbeobachtete Eigenschaft der Testumgebung zustande gekommen ist. Die Ursache hierfür könnte im SUT oder aber in der Unvollständigkeit der Spezifikation zu suchen sein.

Deshalb ist es für das Testen von wesentlicher Bedeutung, dass Eigenschaften der Testumgebung in allen relevanten Parametern bekannt sind. Die Gesamtheit dieser relevanten Parameter werden als *Vorbedingungen* bezeichnet. Sie können für unterschiedliche Artefakte des Testens gelten: Testsuite, Testfall und Testschritt. Auf der Ebene individueller Testschritte definieren Vorbedingungen den konkreten Ausführungskontext einer bestimmten Aktion.

Insbesondere beim Testen kontextsensitiver, mobiler Anwendungen schließen Vorbedingungen neben den fachlich-inhalten Parametern des einzelnen Testschritts und unmittelbaren

technischen Parametern der Betriebsumgebung (z. B. Plattform, Betriebssystem, Version des Betriebssystem, Bauform und Formfaktor des verwendeten Geräts) auch mittelbare technische und situative Kontextparameter (z. B. Ort, Zeit, Sensormesswerte usw., vgl. Abschnitt 3.1.2) mit ein. Aussagefähige Tests setzen daher voraus, dass auch diese Parameter bei der Testdurchführung bekannt bzw. kontrollierbar sind.

Im Kontext dieser Arbeit umfassen Vorbedingungen für Testschritte, Testfälle und Testsuites neben fachlich-inhalten und unmittelbaren Umgebungsparametern auch mittelbare physikalisch messbare und nicht physikalisch messbare Eigenschaften der Testumgebung, durch welche die Testdurchführung situativ charakterisiert wird. Wesentliches Augenmerk der in dieser Arbeit vorgestellten Methode zur automatisierten Generierung und Durchführung von Testfällen liegt auf Maßnahmen zur Herstellung von Vorbedingungen im Rahmen des technisch Möglichen.

4.1.1.11 Nachbedingung

Tests überprüfen die Spezifikationskonformität einer Software in Abhängigkeit von der gewählten Teststufe anhand unterschiedlicher Artefakte. Die eigentliche Überprüfung erfolgt anhand eines Vergleichs von Sollergebnissen mit tatsächlich beobachteten Ergebnissen. Dieser Vergleich basiert auf einer Menge von Bedingungen, die nach der Testausführung gelten müssen. Nachbedingungen sind Teil der Testspezifikation und werden bei der Erstellung von Testfällen dokumentiert. Je nach Teststufe können Nachbedingungen technisch oder fachlich-inhaltich ausgerichtet sein. In ihrer Gesamtheit beschreiben sie den Zustand einer Software und ihrer Betriebsumgebung nach dem Ausführen von Funktionen oder nach der Interaktion eines Anwenders mit der Software. Zugrunde liegt die Annahme, dass der Zustand eines Softwaresystems durch eine Menge Variablen beschrieben wird. Durch Operationen des Systems, z. B. verursacht durch Interaktionen des Anwenders oder durch den Einfluss von Kontextparametern, wird der Systemzustand und damit die Belegung der Variablen verändert (vgl. Liggesmeyer [236]). Die konkrete Belegung der Variablen nach einer Operation auf einem System ist dann die Nachbedingung.

4.1.1.12 Testautomatisierung

Ein Automatisierungswerkzeug bezeichnet in Anlehnung an die Definition der IEEE [192] eine Software, die als Eingabe eine Software und eine aufbereitete Repräsentierung der Spezifikation der zu testenden Software erwartet und – ggf. mit manueller Assistenz – eine Beurteilung der Spezifikationskonformität der zu testenden Software erzeugt. Automatisierung kann sich auf den gesamten Testprozess oder auf ausgewählte Teile des Testprozesses erstrecken.

In dieser Dissertation werden manuell aufbereitete UML-Aktivitätsdiagramme als Eingabe für ein Automatisierungswerkzeug verwendet. Automatisierte Teile des Testprozesses sind Generierung plattformspezifischer Testfälle und deren Ausführung auf der Zielplattform unter Simulation bestimmter Aspekte des Ausführungskontext.

4.1.2 Klassifizierung von Defekten in mobilen Softwaresystemen

Um Aktivitäten des Testens zu fokussieren ist es sinnvoll, Defekte zu klassifizieren und für jede Klasse zu bestimmen, in welcher Art sie sich auf das Softwaresystem auswirken. Weiterhin ist eine Klassifikation von Softwaredefekten hilfreich bei der Formulierung von Anforderungen an eine Testautomatisierungstechnologie, wie sie in Abschnitt 5.5.1 diskutiert wird.

Im Umfeld mobiler Anwendungen ist der Raum für Defekte gegenüber traditionellen Anwendungen stark erweitert. Zusätzlich zu Defekten, die in mobilen Anwendungen und traditionellen Anwendungen gleichermaßen auftreten können, treten bei mobilen Anwendungen zahlreiche andere Faktoren, insbesondere der Kontext der Anwendung und des Anwenders, als zusätzliche Quellen von Defekten in Erscheinung. Mögliche Klassifikationen können hierbei auf unterschiedliche Aspekte der Software, des Softwareprozesses oder Benutzung der Software ausgerichtet sein. Im Umfeld mobiler Anwendungen werden aufgrund der Eigenschaften mobiler Anwendungen angepasste Defektklassifikationen benötigt, um Prozesse der Qualitätssicherung zu unterstützen. Beispielsweise können für solche Defekte, die auf Mobilität der Software (z. B. bei ortsbasierten Anwendungen oder Netzwerkverlust in allgemeinen Mobilitätsszenarien) zurückgeführt werden können, Maßnahmen zur Fehlerkorrektur (z. B. Zuordnung eines geeigneten Mitarbeiters zu Korrektur des Defekts) effizienter realisiert werden.

Eine Möglichkeit Softwaredefekte zu klassifizieren ist die Einordnung von Defekten anhand ihres Schweregrads (vgl. Patton [275]), durch welchen beschrieben wird, wie stark ein Auftreten des Defekts das Softwaresystem beeinträchtigt. Übliche Klassen der Einordnung nach Schweregrad sind beispielsweise (1) Stabilität (Systemcrash, Datenverlust, Datenkorruption), (2) inhaltliches Fehlverhalten (z. B. Berechnung falscher Resultate, unzureichende Funktionalität), (3) Interaktionsdefekte (z. B. Defekte des UI, etwa Rechtschreibfehler, defektes UI-Layout) und (4) Vorschläge für Verbesserungen [275]. Eine Kategorisierung nach diesem Muster erlaubt eine oberflächliche Einschätzung von Softwaredefekten hinsichtlich ihres Schadpotenzials, gibt aber zunächst keine Hilfestellung bei der Bewertung der Dringlichkeit einer Defektkorrektur und insbesondere nicht bei der Bestimmung eines konkreten Vorgehens bei der Defektbehebung. Wünschenswert wäre jedoch eine Klassifikation, die diese Aufgabe unterstützt, beispielsweise durch Hinweise auf zu verwendende Prüftechniken zur Reproduktion des Defekts, welche Grundlage einer Fehlersuche und Fehlerbehebung ist.

Eine weitere etablierte Klassifikationsstrategie ist die Zuordnung von Defekten zu einer Prioritätsklasse. Eine mögliche Klassifikation wäre beispielsweise eine Klassifikation in (1) höchste Priorität (muss sofort korrigiert werden, blockiert die weitere Entwicklung bis die Korrektur abgeschlossen ist), (2) hohe Priorität (muss korrigiert werden, bevor die nächste Produktiteration an den Kunden übergeben wird), (3) mittlere Priorität (wird korrigiert, wenn es die Zeit bis zur Übergabe an den Kunden zulässt) und (4) geringe Priorität (sollte korrigiert werden, beeinträchtigt die Benutzbarkeit des Produkts aber nicht) [275]. Diese Klassifikation ist im Vergleich mit der vorherigen primär auf wirtschaftliche Aspekte und Risiken des Softwareprozesses ausgerichtet.

Problematisch an diesen beiden Klassifikationsschemata ist jedoch, dass der Inhalt des Defekts durch die Klassifikation nicht hinreichend beschrieben wird, um Qualitätssicherungsaktivitäten wirksam zu steuern. Die erste Klassifikation lässt keine Schlussfolgerung über einen

geeigneten Korrekturzeitpunkt zu, die zweite Klassifikation gibt keinen Aufschluss über den Schweregrad des Defekts. Darüber hinaus können beide Klassifikationen bei einer Klassifikation in die Klassen 1-4 zu einer widersprüchlichen Einschätzung des Schadpotenzials eines Defekts führen. Es ist denkbar, dass ein Defekt ein System zwar stark beeinträchtigt (Klasse 1 nach erster Klassifikation), die Defektursache jedoch beherrschbar ist und der Defekt zudem nur selten auftritt, eine Klassifikation nach dem zweiten Schema in Klasse 3 deshalb ausreichend wäre. Keine beider Klassifikationen ermöglicht im Softwareprozess eine Entscheidung über eine adäquate Allokation von Ressourcen zur Korrektur des Defekts.

Ein weiteres Klassifikationsschema ordnet Defekte einem Artefakt des Entwicklungsprozesses zu. Beispielsweise kann ein Defekt in der Spezifikation des Systems, in der Architektur, im Quellcode oder im Test selbst vorliegen. Dieses Schema lässt ebenfalls gültige Aussagen über Defekte und ihre Ursachen zu. Aufgrund des Zusammenhangs zwischen Korrekturaufwand und Entdeckungszeitpunkt eines Defekts (vgl. Basili und Perricone [28], Boehm et al. [51, 49] sowie Shull et al. [321]) lässt diese Klassifikation Rückschlüsse auf den Schweregrad des Defekts zu. Defekte, deren Ursachen bereits in den frühen Phasen des Softwareprozesses liegen (z. B. Anforderungsanalyse, Spezifikation), die jedoch erst spät (z. B. in späten Phasen der Implementierung oder beim *Acceptance Testing*) entdeckt werden, verursachen einen hohen Korrekturaufwand, da u. U. Modifikationen an vielen Teilsystemen oder Komponenten notwendig werden (vgl. Bowen [56]).

Möglich ist ebenfalls eine Klassifikation nach Symptom des Defekts, d. h. nach Eigenschaften des Problems, die im Zusammenhang mit dem Defekt beobachtet wurden. Ein solches Klassifikationsschema kann dabei helfen, die Effektivität einer problemklassenbezogenen Fehlerbehebungsstrategie zur messen. Allerdings ist das Klassifikationsschema nicht beliebig zwischen Softwareprojekten übertragbar, da die Vielfalt der beobachtbaren Symptome mit der fachlichen und technischen Ausrichtung des jeweiligen Projekts variiert. Beispielsweise sind für mobile Anwendungen grundlegend andere Symptome zu beobachten als für traditionelle Systeme. Insbesondere die Vielfalt von Kontextparametern würde hier zu einer Vielzahl von Defektklassen führen, nicht mehr effektiv generalisiert und in einer gemeinsamen Fehlerbehebungsstrategie behandelt werden könnten.

Dieses Problem wurde bereits im Jahr 1980 von Bowen [56] für Defektklassifikationen außerhalb des Umfelds mobiler, kontextsensitiver Software diskutiert. Der Autor schlägt ein kausatives Klassifikationsschema vor, in welchem Defekte anhand ihrer Ursache kategorisiert werden. Als mögliche Defektklassen schlägt Bowen *Design*, *Interface*, *Data Definition*, *Logic*, *Data Handling*, *Computational* und *Other* vor. In die Klasse Design fallen Softwaredefekte, die aus einer ungeeigneten oder unsachgemäßen Überführung von Anforderungen in den Systementwurf hervorgehen (der Begriff Design bezieht sich hier nicht auf die Gestaltung des UI, sondern auf den Entwurf der Systemarchitektur). Fehler dieser Kategorie werden i. d. R. erst spät entdeckt und können nur noch mit großem Aufwand korrigiert werden. In die Klasse Interface (engl. Schnittstelle, hier im Sinn einer technischen Schnittstelle, nicht graphische Benutzerschnittstelle) fallen Defekte, die auf einer fehlerhaften Kommunikation von Systemkomponenten untereinander beruhen. In die Klasse Data Definition fallen Defekte, die durch einen fehlerhaften Umgang mit Daten und Variablen verursacht werden, insbesondere in Systemen, in denen solche Daten global von unterschiedlichen Komponenten verwendet

werden. Die Klasse Logic beinhaltet Defekte, die auf eine fehlerhafte Implementierung der Geschäftslogik zurückzuführen sind. In die Klasse Data Handling fallen alle Defekte, die auf den Umgang mit persistent gespeicherten Daten zurückzuführen sind, z. B. Datenverlust oder Dateninkonsistenz. In die Klasse Computational fallen Fehler, die auf der Implementierung von Berechnungsroutinen beruhen.

Die Arbeit von Bowen aufgreifend untersuchten Collofello und Balcom [80] im Jahr 1984, ebenfalls ohne besonderen Fokus auf mobile Systeme Klassifikationsschemata für Softwaredefekte. Getrieben von der Erkenntnis, dass das Auffinden und Beseitigen von Defekten häufig einen signifikanten Anteil der für ein Projekt verfügbaren Ressourcen in Anspruch nimmt, untersuchten die Autoren Strategien, wie das über einen Zeitraum angesammelte Wissen über Softwaredefekte in zukünftigen Projekten zur optimierten Fehlerkorrektur und zur effizienten Allokation von Ressourcen nutzbar gemacht werden kann. Collofello und Balcom verwenden hierzu ein an Bowen angelehntes kausatives Klassifikationsschema. Die Klassifikation von Fehlern gemäß dieses Schemas ermöglicht es, eine Analyse darüber anzustellen, welche Ursachen in der Vergangenheit welche Fehler bzw. welche Art von Fehlern verursacht haben. Hieraus können Handlungsanleitungen zur zukünftigen Vermeidung gleichartiger Fehler abgeleitet werden. Wenngleich im Jahr 1984 an mobile Geräte und Apps im Sinne der Smartphone-Generation kaum zu denken war, ist die vorgeschlagen Klassifikation von Softwaredefekten ebenfalls auf mobile, kontextsensitive Anwendungen übertragbar. Solche Fehler, von denen sicher bekannt ist, dass sie durch einen fehlerhaften Umgang mit Kontextparametern oder Eigenschaften von Geräte- oder Anwendermobilität verursacht werden, geben Aufschluss über Optionen zur zukünftigen Vermeidung dieser Fehler.

Die Klassifikationschemata von Bowen und Collofello und Balcom sind grundsätzlich auch in der Smartphone-Ära unverändert gültig. Allerdings adressieren diese Klassifikationsschemata nur Defekte, die in den Softwaresystemen der 1980er Jahren typisch waren, also eine Ära der Informationstechnologie, in der die Verbreitung von Computern in privaten Haushalten eben erst in den Anfängen lag. Aus diesem Grund sind in diesen Klassifikationsschemata einige Defektklassen, beispielsweise Defekte des *User-Interface* (UI), nicht präsent. In der Smartphone-Ära gewinnen jedoch zusätzlich weitere Defektklassen an Bedeutung, wie etwa mobile Netzwerkkonnektivität oder Standortinformationen, also allgemein Defekte, die auf die Verwendung von Kontextparametern zurückzuführen sind. Auch hier besteht die Möglichkeit, das Wissen über typische Fehlerursachen auch bei der Erstellung und Durchführung von Testfällen für mobile Anwendungen zu berücksichtigen. Insbesondere bei der Testdurchführung können typische Fehlerursachen mobiler, kontextsensitiver Anwendungen gezielt durch die Verwendung einer adäquaten Kontextsimulationstechnologie herbeigeführt werden, wie etwa spontaner Verlust einer Netzwerkverbindung, sich spontan verändernde Qualitätseigenschaften von Netzwerkverbindungen oder Standort- oder Sensorinformationen.

Holl und Elberzhager [185] untersuchten in ihrer Forschungsarbeit aus dem Jahr 2014 spezifische Defektklassifikationen für mobile Anwendungen. Die Autoren erkennen, dass für mobile Anwendungen die bereits von Bowen und Collofello und Balcom erstellen Klassifikationsschemata weiterhin gültig sind, aber um weitere Klassen ergänzt werden müssen, um die besonderen Merkmale mobiler, kontextsensitiver Software abzubilden. Diese Klassen sind Konnektivität, Energie, Geräteeinstellungen, User Interface, außerplanmäßige Programmun-

terbrechung (nicht Programmabsturz, sondern Suspendierung zugunsten einer anderen Anwendung), Schnittstellen und Ressourcen. Defekte, die in diese Klassen einzuordnen sind, haben i. d. R. Ursachen, die für mobile Anwendungen charakteristisch sind.

In die Klasse Konnektivität fallen alle Defekte einer mobilen Anwendung, die durch spontane Veränderung qualitativer Eigenschaften der Netzwerkschnittstelle verursacht werden können. Da davon auszugehen ist, dass sich ein Anwender in Bewegung befindet, muss ebenfalls damit gerechnet werden, dass sich während der Verwendung einer Anwendung die Qualität einer Netzwerkverbindung verändert. Wenn der Anwender Bereiche betritt oder verlässt, in denen eine Wi-Fi-Verbindung verfügbar ist, kann sich das von einer Anwendung zur Kommunikation verwendete Netzwerk ändern. Die API-Funktionen mobiler Plattformen sind i. d. R. darauf ausgerichtet, immer die günstigste Netzwerkverbindung zur Kommunikation zu verwenden. Ob eine Netzwerkverbindung die günstigste ist, hängt jedoch von mehreren Parametern wie etwa Bandbreite, Latenzzeit oder Kommunikationskosten ab. Im Normalfall präferieren mobile Plattformen eine Wi-Fi-Verbindung vor 3G/4G Verbindungen (z. B. GPRS, UMTS, LTE). Es ist also möglich, dass während einer Netzwerkkommunikation ein Wechsel der verwendeten Netzwerktechnologie erfolgt. Von einer Anwendung wird erwartet, diesen Wechsel ohne Nebeneffekte zu tolerieren. Von einem Testautomatisierungswerkzeug wird deshalb gefordert, den Wechsel der Verfügbarkeit von Netzwerktechnologien simulieren zu können, um die Anwendung auf Defekte der Klasse Konnektivität zu prüfen.

In die Klasse Energie fallen alle Defekte einer Software, die in Relation zur Energieversorgung des Geräts stehen, auf dem eine Anwendung ausgeführt wird. In einigen Fällen reagiert eine App aktiv auf den Zustand der Energieversorgung eines mobilen Geräts, beispielsweise bei kritischen Operationen, deren außerplanmäßiger Abbruch zu Datenverlust führen könnte. Von weitaus größerer Bedeutung ist jedoch der Energiekonsum einer Anwendung. Der Energievorrat ist die wesentliche Ressource mobiler Geräte, von deren Verfügbarkeit die Funktion des Geräts insgesamt abhängig ist (vgl. Balasubramanian et al. [24], Carroll und Heiser [71], Perrucci et al. [276], Pathak et al. [274]). Energieeffizienz ist deshalb ein Qualitätsmerkmal mobiler Anwendungen. Ist ein hoher Energieverbrauch einer App oder einer ihrer Komponenten für den Anwender nicht nachvollziehbar, wird das i. d. R. als Mangel wahrgenommen und kann zu einer erheblichen Frustration beim Anwender führen [274]. Folglich müssen im Rahmen der Qualitätssicherung einer mobilen Anwendung Anstrengungen unternommen werden, den Energieverbrauch einer Anwendung zu messen und zu optimieren. Ein mögliches Fehlerszenario ist beispielsweise die Weiterverwendung energieintensiver Ressourcen (z. B. Sensoren, Kamera oder der 3D-Hardwarebeschleunigung), wenn eine Anwendung pausiert oder vom Betriebssystem in den Hintergrund verlagert wird, also für den Anwender nicht mehr sichtbar ist, weil eine andere Anwendung den Bildschirm okkupiert. Hieraus ergibt sich ebenfalls die Anforderung an ein Testautomatisierungswerkzeug, einerseits den Energiebedarf einer Komponente zu messen und andererseits im Rahmen der Simulation von Kontextparametern den Energievorrat entsprechend einem Test zu simulieren [274].

In die Klasse Geräteeinstellungen fallen Defekte, die auf Konfigurationsparameter eines mobilen Geräts zurückgeführt werden können, wie etwa Klingeltonlautstärke, Spracheinstellungen oder Zeitzone. Um eine App auf Defekte dieser Kategorie zu überprüfen, muss sie auf Geräten oder Emulatoren mit unterschiedlichen Konfigurationseinstellungen getestet wer-

den. Anforderung an ein Testautomatisierungswerkzeug ist deshalb die Fähigkeit, bestimmte Konfigurationsparameter des Geräts während der Testdurchführung anpassen zu können.

In die Klasse User Interface fallen alle Defekte, die mit der Darstellung von Inhalten auf dem Bildschirm oder sonstigen Benutzerschnittstellen in Zusammenhang stehen. Mobile Geräte können vom Anwender beliebig orientiert werden (Hochformat, Querformat). Vom UI erwartet der Anwender in beiden Fällen eine korrekte Darstellung von Inhalten. Folglich müssen für diese Defektklasse entsprechende Tests erstellt werden und ein Testwerkzeug muss in Lage sein, unterschiedliche Displaygrößen und Orientierungen abzubilden.

Der Klasse außerplanmäßige Programmunterbrechung werden Defekte zugeordnet, die auf die betriebssystembestimmte Ressourcenallokation in mobilen Plattformen zurückgeführt werden können. Das Betriebssystem bestimmt hoheitlich über alle Ressourcen und kann einer App jederzeit Ressourcen entziehen, wenn diese für andere ggf. höher priorisierte Anwendungen benötigt werden (z. B. eingehender Telefonanruf oder Textnachricht). Der Anwender ist sich der Möglichkeit bewusst, dass eine App zu jedem Zeitpunkt durch eine andere vom Bildschirm verdrängt werden kann, hat aber die Erwartung, mit der Verwendung der App nach der Störung im Zustand vor der Unterbrechung fortfahren zu können. Eine mobile Anwendung muss also gezielt auf Defekte der Klasse außerplanmäßige Programmunterbrechung getestet werden, woraus sich eine Anforderung an ein Testautomatisierungswerkzeug ergibt, eine solche außerplanmäßige Programmunterbrechung herbeiführen zu können (z. B. durch Simulation eines eingehenden Telefonanrufs).

Eine Klasse von Defekten, die von Holl und Elberzhager nicht diskutiert wurde, sind solche, die im Zusammenhang mit der Verwendung von Kontextparametern stehen (Abschnitt 3.1.2). Diese Dissertation ist auf diese Klasse von Defekten fokussiert und aggregiert und erweitert hierzu die von Bowen, Collofello und Balcom sowie Holl und Elberzhager definierten Defektklassen. Hierzu wird von einem Testautomatisierungswerkzeug gefordert, auch sonstige physikalische Kontextparameter bei der Testdurchführung simulieren zu können, insbesondere durch Bereitstellung künstlicher Sensordaten und Standortinformationen.

Im Jahr 2011 untersuchten Hu und Neamtiu [188] einen Ansatz zur Erkennung von Defekten des UI von Android-Anwendungen (Abschnitt 2.3.3). Teil der Forschungsarbeit von Hu und Neamtiu war die Android-spezifische Klassifikation von Defekten. Zu den diskutierten Defektklassen gehören *Activity*-Defekte, *Event*-Defekte, dynamische Typisierungsdefekte, unbehandelte *Exceptions*, API-Defekte, *I/O*-Defekte und *Concurrency*-Defekte.

Die Plattform Android gibt für die Anwendungsentwicklung in einigen Aspekten verbindliche Entwurfmuster vor. Hierzu gehört die Implementierung mindestens einer Activity als Einstiegspunkt für solche Anwendungen, die eine grafische Benutzungsoberfläche anbieten[1]. An die Benutzung der Komponente Activity ist die Implementierung eines durch die Plattform vorgegebenen Protokolls gekoppelt, das nicht in allen Aspekten vom *Compiler* erzwungen wird. In die Klasse der Activity-Defekte fallen solche Defekte, die auf der Verletzung dieses Activity-Protokolls beruhen (z. B. Störung der Wiederherstellung des UI bei Wechseln des Zustands des Activity-Lebenszyklus). Die Klasse der Activity-Defekte ist vom Grundsatz her auch auf andere Plattformen übertragbar, da beispielsweise auch Apple iOS, Microsoft Win-

[1] Android-Anwendungen sind lose gekoppelte Aggregate aus unterschiedlichen Architekturkomponenten, von denen nicht alle eigene UIs anbieten. Android-Anwendungen können vollständig auf UIs verzichten.

dows Phone und BlackBerry jeweils plattformspezifische Entwurfsmuster erzwingen. Lediglich die Terminologie ist nicht auf andere Plattformen übertragbar. Die in dieser Dissertation untersuchte Methode zur Testfallgenerierung und automatisierte Ausführung von Tests unter Berücksichtigung von Kontextparametern wird in Kapitel 6 am Beispiel der Plattform Android validiert. Deshalb gehört die Klasse der Activity-Defekte ebenfalls in das Spektrum der für diese Dissertation relevanten Defektklassen.

Die Klasse der *Event*-Defekte umfasst solche Anwendungsdefekte, die durch nicht spezifikationskonforme Reaktion einer Anwendung auf Interaktionsereignisse hervorgerufen werden. Denkbar ist beispielsweise, dass eine Benutzerinteraktion auf dem UI nicht das gewünschte Verhalten der Anwendung erzeugt (z. B. Anklicken einer Schaltfläche öffnet wider Erwarten nicht die nächste Activity).

In die Klasse der dynamischen Typisierungsdefekte fallen Laufzeitfehler, die auf fehlerhaften Umgang mit Datentypen, insbesondere bei der Übergabe von Variablenwerten verursacht werden. Da Android-Anwendungen aus lose gekoppelten Komponenten bestehen, müssen Werte von Variablen zwischen Komponenten transportiert werden. Um die Flexibilität der losen Komponentenkopplung zu erhalten, soll eine Komponente deshalb nur über minimales Wissen über andere Komponenten verfügen, wodurch ein Informationsaustausch über typsichere Methodenaufrufe praktisch ausgeschlossen ist[2]. Stattdessen werden Variablenwerte über ein systemweites Nachrichtensystem zwischen Komponenten ausgetauscht, wodurch Datentypinformationen nicht immer erhalten bleiben. In der Konsequenz drohen hier Crashes (d. h. Programmabstürze) durch zulässiges *Type-Casting*. Diese Dissertation fokussiert auf Akzeptanztests auf der Ebene von Black-Box-Tests. Hier machen sich Typisierungsdefekte i. d. R. durch Crashes des SUT bemerkbar, ohne dass die konkrete Fehlerursache für den Anwender erkennbar wird. Deshalb impliziert diese Defektklasse keine besonderen Anforderungen.

Für die Defektklassen unbehandelte *Exceptions* (d. h. fehlerbedingte Ausnahmesituationen) sowie *I/O*-Defekte und *Concurrency*-Defekte (Fehler verursacht durch Effekte der Prozessnebenläufigkeit) gilt dieselbe Einschränkung. Die konkrete Fehlerursache ist für den Anwender i. d. R. nicht zu beobachten, lediglich unerwartetes Verhalten des SUT.

Für mobile Plattformen im Allgemeinen signifikant ist jedoch die von Hu und Neamtiu beschriebene Defektklasse API-Defekte. Mobile Plattformen unterliegen kurzen Iterationszyklen, bei denen im Durchschnitt alle sechs Monate eine neue Plattformversion ausgegeben wird (vgl. Oh und Hong [265]). Zum Zeitpunkt der Aktualisierung werden nicht alle mobilen Geräte ebenfalls aktualisiert. Im Normalfall erfolgt eine Angleichung der Betriebssystemversion nur mit einer Verzögerung, häufig von mehreren Wochen, und einige Geräte erhalten überhaupt keine Aktualisierungen. Deshalb können Entwickler nur eingeschränkt vorhersehen, auf welcher Plattformversion eine Anwendung vom Anwender installiert wird. Zwar können Maßnahmen getroffen werden, das Spektrum der für eine Anwendung zulässigen Betriebssystemversionen einzuschränken. Ein zuverlässiges Ausschließen von Defekten, die durch Versionskonflikte verursacht werden, ist dennoch nicht möglich. Von besonderem Interesse für das Testen ist hierbei, dass Versionskonflikte eine Installation einer Anwendung auf einem

[2]Eine Ausnahme ist die Verwendung des Architekturmusters *Bound Services* [131]. Hier sind typsichere Methodenaufrufe möglich, allerdings sind die beteiligen Komponenten dann nicht mehr lose gekoppelt. Die aufrufende Komponente muss dann Kenntnisse über strukturelle Eigenschaften der aufgerufenen Komponente verfügen.

Gerät nicht notwendigerweise unmöglich machen, die Anwendung jedoch zur Laufzeit aufgrund eines Versionskonflikts abstürzt. An ein Testautomatisierungswerkzeug wird deshalb die Anforderung gestellt, mehrere Betriebssystemversionen zu unterstützen.

Mobile Anwendungen sind auf den im Jahr 2016 verfügbaren mobilen Plattformen i. d. R. als lose gekoppelte, asynchrone Komponenten realisiert, die gemeinsam eine übergeordnete Funktionalität erbringen. Bei der Zusammenarbeit dieser Komponenten kann es zu zeitlichen Verzögerungen kommen, die durch den Anwender durchaus wahrnehmbar sind, etwa als verzögerte Reaktion der App auf eine Interaktion. Ravindranath et al. [292] ergänzen die Menge der Defektklassen mobiler Software deshalb um die Klassen geduldiger und ungeduldiger Anwender. Dem ungeduldigen Anwender wird unterstellt, eine Reaktion einer App auf eine Interaktion grundsätzlich ohne Zeitverlust zu erwarten. In einigen Fällen, z. B. bei Abruf von Daten aus einer Netzwerkquelle, ist diese Erwartungshaltung unrealistisch, da technologiebedingt mit Latenzzeiten zu rechnen ist. An ein Testautomatisierungswerkzeug wird deshalb die Anforderung gestellt, sowohl einen geduldigen als auch einen ungeduldigen Anwender simulieren zu können. Dies manifestiert sich in der maximalen Geschwindigkeit, mit welcher Sequenzen von UI-Interaktionen und anschließender Überprüfung von Nachbedingungen erfolgen.

Das hier entwickelte Konzept der Testautomatisierung für mobile Anwendungen berücksichtig viele dieser Defektklassen, einige explizit, andere implizit. Explizit werden solche Defektklassen adressiert, die unmittelbar auf Kontextparameter bzw. deren Verarbeitung abgebildet werden können. Darüber hinaus adressiert die hier untersuchte Methode zur Generierung von Testfällen und deren Durchführung ausschließlich funktionale Defekte, deren Ursache im Quellcode der Software zu verorten sind. Defekte in der Systemspezifikation oder in anderen Artefakten werden nicht berücksichtigt.

4.2 Auswirkungen von Mobilität und Kontextsensitivität auf das Testen von Softwaresystemen

Aufwandstreiber beim Testen mobiler Anwendungen können in zwei Kategorien unterteilt werden. In die erste Kategorie fallen technische Eigenschaften mobiler Plattformen (z. B. Geräteheterogenität, Steuerung des App-Lebenszyklus), Entwicklungswerkzeuge (z. B. Emulatoren, Cross-Compiler) und Unkenntnis konkreter Eigenschaften des spezifischen Geräts, auf dem die Anwendung ausgeführt wird (vgl. Dehlinger und Dixon [90], Muccini et al. [258], Joorabchi et al. [204], Haller [164]). In die zweite Kategorie fallen Mobilität und Kontextsensitivität (vgl. Morla und Davies [256], Zhang und Adipat [376], Broens und van Halteren [59]).

Die besonderen Architekturen mobiler Plattformen erschweren die Definition einer Menge von Testfällen selbst für nicht-kontextsensitive Anwendungen, in denen Mobilität keinen direkten Einfluss auf den durch die Software realisierten Anwendungsfall hat. Ursache hierfür ist z. B. das von Betriebsystem gesteuerte Lebenszyklusmodell, in dem Anwendungen oder ihre Komponenten zu jedem Zeitpunkt vom System terminiert oder suspendiert werden können (vgl. Hu und Lee [189]) sowie die Flexibilität, mit welcher mobile Betriebssysteme die zur Ausführung einer Funktionalität verwendete Implementierung einer Komponente auswählen.

Mobilität und Kontextsensitivität erweitern darüber hinaus den Raum für mögliche Fehler (vgl. Vieira et al. [352]). Der Standort des Anwenders und andere Kontextparameter beein-

flussen Kontroll- und Datenflüsse kontextsensitiver Anwendungen. Hierbei muss sowohl in der Implementierung als auch beim Testen berücksichtigt werden, dass Ressourcen, die Kontextinformationen erzeugen unzuverlässig sind. Aufgrund der drahtlosen Netzwerkverbindungen, auf die mobile Geräte i. d. R. angewiesen sind, ist beispielsweise während des gesamten Lebenszyklus einer mobilen Anwendung damit zu rechnen, dass sich die Verbindungsqualität verändert oder die Verbindung abbricht. Zentrale Kontextquellen, die über Netzwerkverbindungen abgefragt werden, sind deshalb per se unzuverlässig. Für lokale Kontextressourcen, wie etwa Sensoren oder ein GPS-Modul, gilt dass davon ausgegangen werden muss, dass externe Störquellen die Kontextinformationen verfälschen könnten. Diese Fälle müssen bei der Erstellung von Testfällen einkalkuliert werden, wodurch die Auswahl repräsentativer Testdaten erschwert wird, die Durchführung von Tests fehleranfällig ist und einen hohen manuellen Aufwand erzeugt, der nur in Ausnahmefällen wirksam durch Automatisierungstechnologie unterstützt werden kann.

Die folgenden Abschnitte diskutieren die Auswirkungen ausgewählter Kontextparameter auf das Testen mobiler Anwendungen. Die Auswahl beruht hier auf der typischen Verfügbarkeit von Sensoren, die in mobilen Anwendungen als Kontextquellen verwendet werden und orientiert sich an den in Abschnitt 3.1.2 exemplarisch diskutierten Kontextparametern.

4.2.1 Kontextparameter Geräteheterogenität

Die Komplexität mobiler Anwendungen wird neben der eigentlichen Funktionalität der Anwendung wesentlich durch die Heterogenität der Ausführungsplattform bestimmt (vgl. Joorabchi et al. [204], Vieira et al. [352]). Insbesondere für die mobile Plattform Android existiert eine Vielzahl unterschiedlicher Geräte. Das Unternehmen OpenSignal Inc. stellte für das Jahr 2015 bereits mehr als 24.000 unterschiedliche Android-basierte Geräte fest[3].

Durch Ausnutzung der durch die Plattformbetreiber bereitgestellten Vertriebskanäle kann ein Softwareprodukt mit geringem Vertriebsaufwand global verfügbar gemacht werden und steht dann für alle Geräte bereit, die Zugang zum jeweiligen App Store haben. Aus der Perspektive des Vertriebs ist eine solche globale Infrastruktur zwar ein Vorteil, hat aber den Nachteil, dass eine potenziell defekte Anwendung ohne Einschränkung sofort für alle Anwender verfügbar ist. So wiesen Möller et al. [254] im Jahr 2012 in einer Studie nach, dass mehr als die Hälfte aller Anwender eine Aktualisierung für Apps innerhalb der ersten sieben Tage nach deren Veröffentlichung installieren.

Um eine rasche Penetration des Marktes mit einer defektbehafteten Anwendung zu verhindern, gilt es mobile Anwendungen auf einer repräsentativen Menge unterschiedlicher mobiler Geräte zu testen, um sicherzustellen, dass sich eine Anwendung mit Berücksichtigung der im Zielmarkt zu erwartenden Heterogenität von Geräten und deren Eigenschaften spezifikationskonform verhält (vgl. Haller [164], Khalid et al. [222]).

Während der Testdurchführung auf Geräten mit unterschiedlichen Parametern kann die Situation eintreten, dass auf einem bestimmten Gerät oder auf einer Teilmenge der Geräteauswahl Fehler auftreten, die auf den übrigen Geräten nicht zu beobachten sind. Eine im Jahr 2012

[3] Die Anzahl der unterschiedlichen Geräte wurde durch die Auswertung von Daten berechnet, die über die OpenSignal™-App erhoben wurden. Es handelt sich also nicht um eine Schätzung, sondern es haben bis zum Jahr 2015 mehr als 24.000 unterschiedliche Geräte die OpenSignal Inc.-App heruntergeladen und ausgeführt.

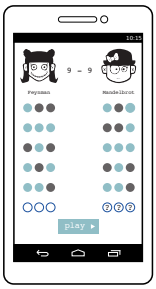
von Han et al. [166] durchgeführte Studie erbrachte beispielsweise den Nachweis, dass Geräte der Hersteller HTC und Motorola herstellerspezifische Anwendungsfehler hervorbrachten, die auf anderen Geräten nicht zu beobachten waren. Zudem bietet die Landschaft mobiler Geräte auch unabhängig von spezifischen Herstellen das Potenzial heterogenitätsbedingter Anwendungsdefekte, z. B. verursacht durch Bauform, Speicherkapazität, Prozessorgeschwindigkeit, Displaydiagonale, Auflösung, Pixeldichte, usw.

Je nach Art des Fehlers kann ein solches Ereignis dazu führen, dass die Testergebnisse für alle Geräte ungültig werden und ein fehlgeschlagener Test nach Korrektur der App auf allen Geräten wiederholt werden muss. Wiederholung des Tests auf nur genau dem Gerät, auf dem ein Fehler aufgetreten, reicht nicht aus, da nicht sicher ausgeschlossen werden kann, dass die Fehlerkorrektur nicht zu Regressionen (vgl. Regressionstest, Abschnitt 4.1.1.6.5) auf anderen Geräten geführt hat (vgl. Joorabchi et al. [204]).

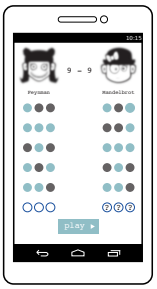
Beispielsweise muss durch Tests untersucht werden, dass das UI einer App auch auf Geräten mit kleinem Display spezifikationskonform angezeigt wird. Mögliche Fehler könnten hier auftreten, wenn aufgrund der geringen Displayfläche eines solchen Geräts ein Teil des UI abgeschnitten dargestellt wird und Bedienelemente entweder verzerrt dargestellt werden oder gar nicht mehr erreichbar sind, weil sie außerhalb der darstellbaren Displayfläche liegen. Eine weitere Besonderheit ergibt sich, weil Displaygröße und Auflösung Eigenschaften mobiler Geräte sind, die sich beliebig überschneiden können: Es existieren Geräte mit hoher Auflösung und geringer Displaydiagonale und Geräte mit großer Displaydiagonale mit geringer Auflösung. Hier muss durch Tests sichergestellt werden, dass alle UI-Elemente auf allen relevanten Konfigurationen in angemessener Qualität dargestellt werden und nicht etwa auffällige Kompressionsartefakte oder Unschärfe aufweisen.

Abbildung 4.1 illustriert diesen Fall am Beispiel einer Anwendung, die auf zwei Geräten mit identischer Displaydiagonale, aber unterschiedlicher Auflösung installiert wurde. Mobile Anwendungen werden mit Paketen graphischer Ressourcen ausgeliefert, die jeweils geeignete Ressourcen für unterschiedliche Geräteklassen enthalten. Die Plattform Android (analog Apple iOS) definiert beispielsweise sechs Klassen der Pixeldichte für Displays mobiler Geräte (*low*, *medium*, *high*, *extra-high*, *extra-extra-high*, *extra-extra-extra-high*) [135]. Eine typische Fehlerursache ist es zu versäumen, Ressourcen für alle sechs Klassen bereitzustellen. Die Plattform greift dann auf Ressourcen einer niederen Klasse zurück. Eine Skalierung auf die Pixel-Abmessungen des Geräts resultiert dann in unscharfer Darstellung. Entdeckt werden kann dieser Fehler nur dann, wenn die Anwendung tatsächlich auf Geräten aller sechs Klassen ausgeführt wird. Die Einschätzung, ob die Darstellung einer graphischen Ressource den gesetzten Qualitätskriterien genügt, kann allerdings nur mit hohem Aufwand automatisiert werden. Instrumentierungstechnologie kann aber verwendet werden, um zu bewerten, ob prinzipiell eine Grafikressource an der korrekten Stelle auf dem Display angezeigt wird.

Ebenfalls wäre es möglich, dass eine Anwendung auf Geräten mit geringer Prozessorleistung oder Speicherkapazität eine schlechte Performanz aufweist, während dieses Problem bei anderen Geräten nicht auftritt. Anwendungen, die ein aufwendig animiertes UI verwendenden, könnten beispielsweise auf einigen Geräten von Hardware-Beschleunigung profitieren, während ein entsprechendes Bauelement auf anderen Geräten nicht oder nur in einer geringeren Ausbaustufe vorhanden ist. Die Auswahl der repräsentativen Geräte kann sich noch während der



(a) Darstellung auf Gerät der Klasse hdpi mit 480x854 Bildpunkten



(b) Darstellung auf Gerät der Klasse xx-hdpi mit 1080x1920 Bildpunkten

Abbildung 4.1: Auswirkungen von Geräteheterogenität auf graphische Ressourcen. Der Entwickler hat es versäumt, graphische Ressourcen für die Geräteklasse *extra-extra-high* (xxhdpi) bereitzustellen. Während die Grafik auf einem Gerät der Klasse *high* (hdpi) korrekt dargestellt wird, ist die Darstellung auf einem Gerät der xxhdpi-Klasse unscharf.

Projektlaufzeit verändern, so dass neben dem Aufwand für das eigentliche Testen zusätzlicher Aufwand zur Beobachtung und Anpassung des Geräteportfolios entsteht.

4.2.1.1 Auswirkungen auf das Testen

Als konkrete Auswirkung der Kontextparameter Plattform und Gerät auf das Testen mobiler Anwendungen ergibt sich ein erhöhter Aufwand für alle Testaktivitäten, da die Heterogenität der Gerätelandschaft als Multiplikator wirkt. Eine Teststrategie muss folglich die Frage beantworten, welche Auswahl von Geräten den anvisierten Zielmarkt repräsentiert. Im Einzelfall ist hierzu eine Marktstudie zur Verbreitung von Plattformen, Geräten und Betriebssystemversionen pro Plattform im jeweiligen Zielmarkt durchzuführen. Nachdem eine solche Menge repräsentativer Geräte ausgewählt wurde, müssen diese Geräte beschafft, inventarisiert und über ihren gesamten Lebenszyklus hinweg betreut werden.

Es ist zu erwarten, dass für unterschiedliche App-Projekte die Menge der für den Zielmarkt repräsentativen Geräte ebenfalls unterschiedlich ist. Die Pflege eines entsprechenden Geräteportfolios generiert deshalb neben den finanziellen Aufwendungen zur Beschaffung auch personellen Aufwand, um das Geräteportfolio zu steuern, zu pflegen und Verfügbarkeit und Einsatzbereitschaft individueller Geräte zu gewährleisten (vgl. Haller [164]).

Die Auswahl repräsentativer Geräte gehört zudem zu denjenigen Aktivitäten des Testens, die nur schwer automatisiert werden können (vgl. Abschnitt 4.3.1) und Ergebnisse erzeugen, die schnell altern und ggf. bei langen Projektlaufzeiten zum Ende des Projekts bereits obsolet sind. Deshalb ist die Beobachtung der Marktentwicklung im repräsentativen Geräteportfolio eine Aktivität, der während der gesamten Projektlaufzeit Aufmerksamkeit gewidmet werden muss (vgl. Sanaei et al. [310], Vieira et al. [352]). Eine Verschiebung von Geräteeigenschaften verändert den Betriebskontext des Softwareproduktes und zieht deshalb u. U. Veränderungen der Anforderungen an das Softwareprodukt nach sich (z. B. Unterstützung einer völlig neuen Geräteklasse, wie etwa Smartwatches). Diese Probleme existieren im Umfeld nicht-mobiler Anwendungen zwar ebenfalls, dort aber in wesentlich abgeschwächter Form, da Lebenszyklen von Computern und Plattformen in der Desktop- und Serverlandschaft um ein vielfaches

länger sind als im Umfeld mobiler Geräte, wo in etwa im Intervall von sechs Monaten mit einer neuen Plattformversion zu rechnen ist (vgl. Oh und Hong [265]).

Dieser Aufwand setzt sich bei der eigentlichen Testdurchführung fort. Jeder Testfall muss auf allen Geräten des für die zu testende Anwendung bestimmten Geräteportfolios durchgeführt werden. Der hierzu erforderliche Aufwand wächst überproportional zur Anzahl der Geräte (vgl. Ham und Park [165]). Zusätzlich zur direkt von der Anzahl der Testgeräte abhängigen benötigen Zeit für die Testdurchführung müssen Testgeräte entsprechend den Anforderungen des jeweiligen Testfälles vorbereitet werden. Diese Maßnahmen umfassen beispielsweise die Installation einer bestimmten Betriebssystemversion auf dem Testgerät, die Installation anderer Anwendungen deren Ressourcen für die zu testende Anwendung benötigt werden (z. B. Adobe AIR Laufzeitumgebung [4], Google Play-Dienste [153]) in der jeweilig benötigten Version, Einrichten von Benutzerkonten auf dem Gerät (z. B. Google-Konto, Apple iTunes Konto) oder Aufladen des Geräteakkus auf den im Testfall spezifizierten Stand.

Am Beispiel der Anwendung Mobiler Taxiruf ist im Rahmen der Teststrategie zunächst festzulegen, welche Geräteauswahl den Zielmarkt adäquat abdeckt. Da die App Vertriebskanal einer auf das Gebiet der Bundesrepublik Deutschland beschränkten Dienstleistung ist, muss im Rahmen einer Marktforschung erhoben werden, welche Geräteklassen und Hersteller im deutschen Markt in der anvisierten Zielgruppe signifikante Marktanteile erreichen. Diese Marktforschung erzeugt eine Menge individueller Geräte, für die es jeweils festzustellen gilt, welche Betriebssystemversionen für diese Geräte verfügbar sind. Nach unten wird diese Auswahl durch diejenige Betriebssystemversion begrenzt, mit welcher das jeweilige Gerät zum Auslieferungszeitpunkt ausgerüstet war. Im Ergebnis dieser Marktanalyse wird eine Anzahl von Geräten als zielgruppenrelevant identifiziert. Die mindestens zu unterstützende Betriebssystemversion wird durch die Geräteauswahl technisch vorgegeben, könnte aber aus anderen Gründen (z. B. Marketing) noch weiter herabgesetzt werden. Aus der Menge der als relevant identifizierten Geräte muss nun eine Auswahl der zum Testen der App zu verwendenden Geräte getroffen werden. Sofern das Testen der App *in-house*, also unter Aufwendung unternehmenseigener Ressourcen, erfolgen soll, müssen diese Geräte beschafft und den Entwicklern und Testern zugänglich gemacht werden.

4.2.1.2 Grenzen existierender Lösungskonzepte und Technologien

Es ist selbst großen Unternehmen kaum möglich, eine Android App auf allen verfügbaren Geräten zu testen. Alternativ zur Gerätebeschaffung besteht die Möglichkeit, Tests bei einem *Device-Cloud*-Anbieter[4], z. B. Xamarin Inc. [370] oder AppThwack.com [19], durchzuführen.

In der Regel werden Gerätefarmen im Teilzeignutzungsverfahren betrieben, so dass ein spezifischer Ausführungszeitpunkt für Tests nicht garantiert ist. Hierdurch können z. B. Tests, die die Kontextparameter Datum und Uhrzeit verwenden nicht von der Gerätevielfalt kommerzieller Anbieter von Gerätefarmen profitieren. Weiterhin könnte es aus Datenschutzgründen ausgeschlossen sein, eine App vor ihrer Veröffentlichung einem Drittanbieter zur Verfügung zu stellen, da dieser nur eingeschränkt eine Geheimhaltung garantieren kann.

[4]Kommerzielle Betreiber bieten Zugang zu Gerätefarmen an, auf denen Tests ausgeführt werden können. Hier kann nicht nur von einer heterogenen Gerätevielfalt profitiert werden, sondern auch von einer diversen geographischen Verteilung.

Weiterhin geben die Betreiber spezifische Technologien zur Implementierung von Testfällen vor. Die Verwendung besonderer System-Builds als Alternative für die Standardimplementierung der Zielplattform, beispielsweise ergänzt um spezielle Testschnittstellen, ist dann nicht möglich. Es ist dann ausgeschlossen, solche Tests durchzuführen, die eine Plattformimplementierung *Designed for Testability* erfordern, also per se alle Tests, die mit simulierten Kontextparametern durchgeführt werden.

Eine Kernaktivität des Testens, nämlich die Auswahl der relevanten Geräte, kann durch die Verwendung von Gerätefarmen ebenfalls nicht entfallen.

4.2.2 Kontextparameter Datum und Uhrzeit

In Abschnitt 3.1.2.1.2 wurde die Bedeutung von Datum und Uhrzeit als Kontextparameter diskutiert. Sofern eine mobile Anwendung den Kontextparameter Zeit verwendet, entstehen hierdurch Anforderungen an Testfälle, diese Abhängigkeit von Datum und Zeit abzubilden. Eine besondere Schwierigkeit ergibt sich hierbei, wenn die durch die zu testende App implementierte zeitabhängige Funktion u. U. nicht vollständig auf dem mobilen Gerät realisiert ist, sondern ihrerseits kontextsensitive Funktionen auf einem Backend-System aufruft. In diesem Fall wird der Testfall selbst kontextsensitiv hinsichtlich Datum und Uhrzeit der Testausführung (vgl. Hofer et al. [182], Baldauf et al. [25]).

Zu unterscheiden sind für die Kontextparameter Datum und Uhrzeit zwei Fälle: (1) Die Kontextabhängigkeit existiert ausschließlich lokal auf dem mobilen Gerät oder (2) Kontextabhängigkeit existiert im Backend-System. Im zweiten Fall muss zusätzlich unterschieden werden, ob für das Testen ein dediziertes Backend-System verfügbar ist (2a), dass während des Testens beliebig manipuliert und nach Bedarf in einen spezifischen Zustand versetzt werden kann oder ob lediglich ein Live-System verfügbar ist (2b), welches nicht beeinflussbar ist.

4.2.2.1 Auswirkungen auf das Testen

Kontextsensitivität führt Anforderungen an Testplanung und Testdurchführung ein, die nicht nur zu einer Erhöhung der Anzahl von Testfällen, sondern ebenfalls zu einer erhöhten organisatorischen Komplexität von Testaktivitäten führt. Insbesondere wenn der Kontextfaktor Zeit in einer Anwendung verwendet wird, kann sich der Aufwand signifikant erhöhen. Tests werden dann ihrerseits kontextsensitiv, mögliche Zeitpunkte für die Durchführung können zu außergewöhnlichen Tageszeiten liegen, an denen Mitarbeiter nicht verfügbar sind.

In Softwareprojekten, bei denen Datum und Uhrzeit nur auf dem mobilen Geräte als Kontextparameter verwendet werden, müssen zunächst solche Werte für Datum und Uhrzeit identifiziert werden, die einen relevanten Testfall repräsentieren. Die Ausführung dieser Tests muss dann genau zu diesen spezifizierten Uhrzeiten durchgeführt werden. Alternativ können auf dem Gerät oder dem Emulator entsprechende Werte für Datum und Uhrzeit eingestellt werden. In der Regel kann dies in den Geräteeinstellungen erfolgen, bedeutet aber, dass der eigentliche funktionale Testfall um zusätzliche manuelle Schritte ergänzt werden muss, wodurch nicht nur der Aufwand ansteigt, sondern auch weiteres Fehlerpotenzial entsteht.

In Softwareprojekten, in denen zur Realisierung eines Anwendungsfalls sowohl eine mobile Anwendung als auch ein Backend-System entwickelt werden, ist eine ähnliche Vorgehensweise

denkbar. Allerdings müssen Datum und Uhrzeit dann nicht nur auf dem mobilen Gerät, sondern auch im Backend-System an jeden individuellen Testfall angepasst werden, wodurch weiterer Aufwand und eine weitere potenzielle Fehlerquelle hinzukommt.

Wenn eine mobile Anwendung zu entwickeln ist, die mit einem bereits existierenden Backend-System kommunizieren soll, besteht i. d. R. nicht die Option, das sich bereits im Betrieb befindliche Backend-System zum Zweck des Testens der App zu manipulieren. Dieser Fall könnte z. B. eintreten, wenn das Web-basierte Selbstbedienungsportal einer Bank, einer Versicherung oder ein Informationsportal eines Verkehrsbetriebs durch eine App ergänzt werden soll. Die Gründe für einen solchen Ausschluss können vielfältig sein. Testen auf einem Live-System kann gegen Sicherheitsrichtlinien verstoßen, etwa im Umgang mit personenbezogenen Daten. Ebenfalls könnte das Testen der mobilen Anwendung im Verbund mit Schnittstellen des Backend-Systems dieses in einen inkonsistenten Zustand überführen, der mit dem regelmäßigen Betrieb kollidiert. Neben diesen technischen Gründen sind Datenschutzbestimmungen zu beachten, die Testen auf einem Live-System ausschließen könnten.

In Fällen, in denen ein existierendes Backend-System im Live-Betrieb zum Testen verwendet werden muss, ist es u. U. notwendig, die Durchführung des Tests auf einen bestimmten Zeitpunkt festzulegen. Denkbar sind beispielsweise Nachrichten-Apps, die in Abhängigkeit von der Tageszeit jeweils unterschiedliche Medien-Streaming-Angebote ausliefern. Als Beispiel kann hier die App MDR Nachrichten[5] [253] zitiert werden. Diese implementiert eine Video-Streaming-Funktion, die in Abhängigkeit vom Kontextparameter Zeit ein unterschiedliches Programm anbietet. Das UI der App ist kontextsensitiv, weil der Bereich zur Darstellung des Streaming-Angebots je nach Tageszeit unterschiedliche Inhalte anzeigt. Zusätzlich sind im Backend-System bestimmte Medien-Streaming-Angebote nur zu bestimmten Tageszeiten verfügbar. Die Auswirkung auf das Testen ist, dass die Durchführung bestimmter Testfälle nur wirksam in den Randbereichen des Wechsels des Streaming-Programms erfolgen kann.

Die App Quizduell [113] (abgelöst durch die App Quizduell im Ersten [201]) implementiert in ähnlicher Weise eine Funktion, in welcher App-Anwender in das Geschehen der zugehörigen Fernseh-Show Quizduell des ARD [232] zur Sendezeit durch Teilnahme via App eingreifen können. Hier existiert eine direkte Abhängigkeit der App-Funktionalität von Datum und Uhrzeit, da Inhalte synchron mit der Sendezeit einer Fernseh-Show angeboten werden. In der Konsequenz müssen Tests mit den Sendezeiten des Live-Streaming-Dienstes synchronisiert werden, was bei der Planung des Softwareprozesses zu berücksichtigen ist. Hier kann der Fall eintreten, dass Testfälle nur zu Zeiten durchgeführt werden können, die nicht nur außerhalb der Arbeitszeiten der Mitarbeiter des Softwareherstellers liegen, sondern durch Verbindung zu einem Live-Fernsehprogramm zusätzlich das Risiko bergen, dass der Testbetrieb nicht nur ein Live-System stört, sondern dass diese Störung unmittelbar öffentlich wahrnehmbar ist.

Die Beispielanwendung AR Tourist Information ist kontextsensitiv gegenüber Datum und Uhrzeit zur Realisierung der Werbungsfunktion. Unter der Voraussetzung, dass diese Funktion lokal auf dem Gerät hinterlegte Werbedaten auswertet, kann ein Test dieser Funktion erfolgen, indem Datum und Uhrzeit auf entsprechende Werte eingestellt werden und überprüft wird,

[5] Interna der App MDR Nachrichten [253] sind dem Autor dieser Dissertation bekannt, weil diese App im Rahmen eines Industrieprojekts im Zeitraum 2010-2011 im Unterauftrag am Lehrstuhl für Software Engineering, insb. mobile Systeme der Universität Duisburg-Essen bearbeitet wurde.

ob auf dem Display die korrekte Werbeanzeige eingeblendet wird. Die Spezifikation der Anwendung AR Tourist Information könnte definieren, dass in einem definierten Datumsbereich, z. B. zwischen Anfang Juni und Ende September an jedem Tag zwischen 11:00 Uhr und 17:00 Uhr, die Werbeanzeige einer dem Erlebnisrundgang nahegelegenen Eisdiele angezeigt wird, wenn bei Sonnenschein eine Außentemperatur von 20 °C erreicht wird.

Da ein Werbender für die Darstellung seiner Werbeanzeige in der App eine Zahlung an Betreiber der App leistet, hat er nicht nur ein besonderes Interesse an einer im Sinne der Spezifikation der App korrekten Implementierung dieser Funktion, sondern sogar die Legitimation, den Nachweis der Korrektheit der Implementierung einzufordern. Ein Defekt könnte juristische Folgen haben, etwa Regresszahlungen an den Werbenden. Deshalb könnte diese Funktion im Rahmen eines Akzeptanztest zum Abnahmekriterium für die App werden.

Die konkreten Auswirkungen auf das Testen manifestieren sich erneut in einem erhöhten Aufwand zur Manipulation des Standorts des Geräts auf einen der Testspezifikation entsprechenden Ort sowie der Manipulation des Sensors für die Umgebungstemperatur (ersatzweise eine Web-basierte Schnittstelle zur Ermittlung von Wetterinformationen). Zusätzlich müssen Datum und Uhrzeit des Geräts auf geeignete Werte eingestellt und die korrekte Darstellung der Werbeanzeige auf dem UI der App überprüft werden.

Welche Datums- und Zeitwerte geeignet sind wird durch die Testspezifikation festgelegt. In Frage kommen hier beispielsweise Randbereichstests, die Werte nahe den Definitionsgrenzen als Testdatum verwenden. Im konkreten Beispiel könnte etwa für jeden Tag innerhalb des Werbezeitraumes überprüft werden, ob für den Zeitwert 10:59 Uhr (17:01 Uhr) die Werbeanzeige spezifikationskonform nicht eingeblendet wird, für den Zeitwert 11:00 Uhr (17:00 Uhr) aber auf dem UI sichtbar ist. Zusätzlich sollte getestet werden, ob die Anzeige der Werbeanzeige außerhalb des definierten Datumsbereichs tatsächlich unterbleibt.

Manuelles Testen limitiert die Durchführung dieser Tests aus ökonomischen Gründen auf Randbereichstests gültiger Datums- und Zeitwerte. Selbst hier wird dem Tester bei der Durchführung vieler Tests (z. B. für alle werberelevanten Standorte) ein enormes Maß an Aufmerksamkeit und Akkuratesse abverlangt: Einstellen von Datum und Uhrzeit, Verwendung von Werkzeugen des SDK zur Manipulation von Standort und Temperatursensor (nur Emulator, nicht Gerät) und Durchführen des eigentlichen Tests. Die Verwendung von Automatisierungstechnologie ermöglicht das Testen der App mit allen möglichen diskreten Datums- und Zeitwerten zwischen 1. Januar und 31. Dezember eines gegebenen Jahres einschließlich Berücksichtigung von Schaltjahren. Hierdurch kann eine vollständige Abdeckung aller zulässigen Datumswerte im Sinne der Anwendungsspezifikation erreicht werden. Voraussetzung ist allerdings eine Testautomatisierungstechnologie, die es beherrscht, Datum und Uhrzeit des Geräts zu manipulieren, auf der die Anwendung ausgeführt wird.

4.2.2.2 Grenzen existierender Lösungskonzepte und Technologien

Im Fall der ausschließlich lokalen Abhängigkeit von Datum und Zeit sind die Auswirkungen der Kontextsensitivität verhältnismäßig gut beherrschbar. Beispielsweise können Datum und Uhrzeit auf dem Testsystem manuell bei der Testausführung entsprechend der Testspezifikation eingestellt werden. Sowohl auf Emulatoren als auch auf Geräten ist das i. d. R. ohne technische Hürden möglich, erfordert aber manuelle Intervention.

Auf der Plattform Android ist es einer regelmäßigen App nicht möglich, die Datums- und Uhrzeiteinstellungen zu verändern. Testwerkzeuge, die auf Code basieren, wie etwa JUnit-Familie, können deshalb in diesem Aspekt nicht in Automatisierungstechnologien verwendet werden. Sie werden vom System stets als regelmäßige App eingestuft und verfügen nicht über die zur Anpassung von Datum und Zeit notwendigen Berechtigungen. Insofern wird von einer Automatisierungstechnologie zur Realisierung kontextsensitiver Tests gefordert, eine entsprechende Schnittstelle nach dem Paradigma *Design for Testability* [281] zu implementieren.

Im zweiten Fall, Kontextsensitivität gegenüber Datum und Uhrzeit im Backend-System, ist es nicht ausreichend Datum und Uhrzeit lokal auf dem Gerät gemäß der Testspezifikation anzupassen. Sofern das Backend-System über eine Testschnittstelle verfügt, die es zulässt, Daten eigens für Testzwecke zu Erstellen und zu manipulieren (Fall 2a), können Tests so implementiert werden, dass für die Durchführung nicht nur die Zeit des Geräts dem Test entsprechend eingestellt wird, sondern auch Datum und Uhrzeit auf dem Backend-System (vgl. Haller [164]). Eine solche Manipulation von mobiler Anwendung, mobilem Gerät und Backend-System kann entweder manuell oder unter Verwendung von Automatisierungstechnolgie erfolgen. Im Jahr 2016 existiert allerdings keine Testautomatisierungstechnologie, die Datum und Uhrzeit auf zwei physikalisch getrennten Systemen synchron manipulieren kann.

4.2.3 Kontextparameter Mobilität

Anwendungen, die Standortinformationen verwenden, müssen Tests unterzogen werden, die diese Kontextinformation explizit berücksichtigen. Das bedeutet, es muss nicht nur die spezifikationskonforme Funktionalität einer App überprüft werden, sondern es muss überprüft werden, ob die App-Funktionalität im Bezug auf den sich möglicherweise verändernden Standort des Anwenders erfüllt wird. Hierzu ist es erforderlich, dass im Rahmen der Testdurchführung der Standort und abhängige Kontextparameter – wie z. B. Verfügbarkeit weiterer Ressourcen, etwa Netzwerkonnektivität – entsprechend der Testfallspezifikation simuliert werden, d. h. Bereitstellung von Standortinformationen und sonstiger assoziierter Kontextparameter.

Standorte werden in der Informationsverarbeitung durch eine Menge von Variablen beschrieben (vgl. Abschnitt 3.1.2.1.3). Die SDKs mobiler Plattformen verwenden hier den WGS84-Standard, der eine Standortinformation in drei Variablen, geographische Breite, geographische Länge und die Höhe über dem Meeresspiegel, im Bezug zu einem Referenzellipsoid der Erde abbildet. Geographische Länge wird in Grad bestimmt und kann reelle Werte zwischen -180 und 180 (jeweils im Bezug zur willkürlichen Festlegung des Nullmeridians) annehmen, geographische Breite nimmt reelle Werte zwischen -90 und 90 an. Aus der Überabzählbarkeit der der reellen Zahlen im Intervall (0,1) (zweites Diagonalargument nach Cantor Cantor [69], Meschkowski [245]) folgt aufgrund der bijektiven Abbildung $f : (0,1) \to L : x \mapsto -180 + 360x$ unmittelbar die Überabzählbarkeit der möglichen Werte für den Längengrad (analog Breitengrad, Höhe). Hieraus folgt, dass unter Abstraktion technischer und praktischer Eigenschaften von Computersystemen auch in unbegrenzter Zeit nicht alle möglichen Testdaten für eine ortsbasierte Anwendung durch Probieren getestet werden können.

Diese theoretische Überlegung wird durch technische Eigenschaften digitaler Computersysteme abgeschwächt. Die zur Darstellung von Standortinformationen verwendeten Datentypen begrenzen je nach Architektur der mobilen Plattform die Auflösung und Präzision der

Testdaten. Es besteht jedoch weiterhin das Problem, dass die Anzahl der Testfälle, die zur Abbildung aller technisch möglichen Standortinformationen benötigt würden, in der Praxis zu groß ist. Aufgrund der minimalen Auflösung der GPS-Technologie ist eine Teststrategie, die alle technisch darstellbaren Standortinformationen als Testdaten verwendet darüber hinaus nicht sinnvoll (z. B. Testdaten die nur wenige Millimeter voneinander entfernt liegen). Sinnvoll ist es hingegen, Testdaten für den Standort so auszuwählen, dass kritische Randbereiche durch Testfälle abgedeckt werden (vgl. Bereichstest oder *Domain Test*, Whittaker et al. [360], Ammann und Offutt [7], Liggesmeyer [236], Vieira et al. [352]).

Die projektspezifische Teststrategie macht Vorgaben, welche Geräte und welche Emulatorkonfigurationen mit welchen konkreten Standortinformationen zu verwenden sind, um die App oder einzelne funktionale Aspekte der App zu testen. Im Anschluss müssen auf dem Gerät bzw. auf dem Emulator statische Kontexteigenschaften eingestellt werden, die während der Testdurchführung invariant sind. Hierzu zählt beispielsweise die Auswahl eines ISP, der bei Verwendung einer Internetverbindung via 3G/4G Mobilfunktechnologie den Internetzugang bereitstellt. Die folgende eigentliche Testdurchführung umfasst alle notwendigen Schritte, die in individuellen Tests definiert sind: Navigieren im UI der SUT zur Funktion, die Gegenstand des Tests ist, ggf. Eingabe von Daten in UI-Masken, Ausführen von Anwendungsfunktionalität und schließlich Überprüfen, ob sich die App spezifikationskonform verhält. Für kontextsensitive Anwendungen in denen die Anwendermobilität eine Rolle spielt, muss dem Gerät oder dem Emulator während der Durchführung des Tests ebenfalls der Standort des Anwenders gemäß Spezifikation zugeführt werden. Hierzu kommen mehrere Vorgehensweisen in Frage, die jeweils auf Emulatoren oder Geräten angewendet werden können.

4.2.3.1 Auswirkungen auf das Testen

Konkret ergeben sich für das Testen von Anwendungen, die solche Kontextparameter verwenden, die durch Mobilität des Anwenders (*In-motion User*, *In-motion Device*, vgl. Book et al. [54, 53, 55]) bestimmt werden folgende Auswirkungen: (1) Es muss für den konkreten Anwendungsfall bestimmt werden, welche Kontextparameter beim Testen der Anwendung berücksichtigt werden müssen, es müssen (2) Testdaten bestimmt werden, die relevante Zustände des SUT adäquat repräsentieren und (3) es muss dem SUT während der Testausführung ein entsprechender Kontext zugeführt werden (vgl. Morla und Davies [256], Broens und van Halteren [59], Diewald et al. [97], Vieira et al. [352]).

Die Herausforderung der ersten Aufgabe, der Bestimmung relevanter abhängiger Kontextparameter, kann nicht technisch im Sinn einer Automatisierung gelöst werden, wenngleich aufgrund inhärenter Abhängigkeiten (z. B. beeinflusst Standort Verfügbarkeit von Netzwerken und GPS) eine systematische Analyse möglich ist (vgl. Henricksen et al. [178, 176, 177], Ayed et al. [21]). Hierzu gilt es den Anwendungsfall zu analysieren, primäre Kontextparameter zu bestimmen und abhängige sekundäre Parameter zu identifizieren und beim Testen zu berücksichtigen. Insbesondere die Konnektivität zu Netzwerken ist ein Faktor, der wesentlich durch Mobilität mitbestimmt wird (vgl. Muccini et al. [258]).

Die zweite zu lösende Problemstellung ist die Erzeugung von Testdaten, mit denen die Anwendung getestet werden soll. Für Kontextparameter, die sich in wenigen diskreten Werten manifesten, wie etwa Art und Verfügbarkeit einer Netzwerkverbindung, ist diese Aufgabe

vergleichsweise einfach zu lösen. Für Fälle in denen Kontextparameter überabzählbar unendlich viele Werte annehmen können, wie z. B. Standortinformationen, stünden einem kontextsensitiven System unendlich viele Testfälle gegenüber (vgl. Vieira et al. [352]). Anforderung an eine Menge von Testfällen für Anwendungen, die den Standort des Anwenders verwenden, ist beispielsweise zu überprüfen, ob sich die Anwendung für alle möglichen Positionen korrekt verhält. Sie verhält sich genau dann korrekt, wenn keiner dieser Tests fehlschlägt.

Für eine mobile Anwendung, die den Kontextparameter Mobilität des Anwenders verwendet, müssen deshalb nicht-trivialer Testdaten erzeugt werden. Die konkrete Auswahl dieser Daten ist hierbei abhängig von dem durch das SUT implementierten Anwendungsfall. Für ortsbasierte Anwendungen ist zu erwarten, dass die Anzahl der zu berücksichtigenden Testdaten groß ist, weil die Menge der Testdaten nicht nur durch eine Vielzahl verschiedener Standorte gebildet wird, sondern für individuelle Standorte zusätzlich mehrere Instanzen mit variierender Präzision gewählt werden müssen (vgl. Broens und van Halteren [59]).

Nachdem eine Menge von Testfällen mit zugehörigen Testdaten erstellt wurde, die geeignet ist, Fehler in der Anwendung aufzudecken, müssen diese Testfälle ausgeführt werden. Hier kommen die Besonderheiten kontextsensitiver Anwendungen gegenüber nicht-kontextsensitiven Anwendungen besonders zu Geltung, weil für eine reproduzierbare Testdurchführung neben den Interaktionen des Anwenders ebenfalls der Kontext als Vorbedingung eines Tests erzeugt werden muss. Für den Kontextparameter Standort bedeutet das konkret, dem SUT muss ein Standortparameter gemäß der Testspezifikation zugeführt werden und es müssen sekundäre Kontextparameter bei der Durchführung des Tests berücksichtigt werden.

Darüber hinaus muss berücksichtigt werden, dass sich der Anwender während der Verwendung von Gerät und App in Bewegung befinden kann (*In-motion User*, *In-motion Device*, vgl. [54, 53, 55]). Der Tester muss also eine Abfolge von Standortdaten in das SUT eingeben, die einem spezifizierten Bewegungsprofil eines Anwenders entspricht. Standortinformationen werden von jeder Lokalisierungstechnologie der im Jahr 2016 verfügbaren mobilen Plattformen mit einem Zeitstempel im Millisekundenbereich assoziiert. Einem menschlichen Tester ist es aus diesem Grund durch manuelle Aktualisierung des simulierten Standorts nicht möglich, Tests mehrmals mit identischen Standortinformationen zu reproduzieren, da der Zeitintervall zwischen einzelnen Standortdaten nicht konstant gehalten werden kann.

Abhilfe schafft hier die Verwendung präparierter Bewegungsprofile in Form einer GPX-Datei[6]. Die SDK-Werkzeuge der mobilen Plattformen verfügen grundsätzlich über die Fähigkeit, dem Emulator simulierte Standortdaten aus GPX-Dateien zuzuführen. Allerdings ist diese Technologie Teil der Entwicklungswerkzeuge und kann nicht zusammen mit Testautomatisierungsrahmenwerken wie beispielsweise JUnit oder anderen Vertretern der *xUnit*-Familie verwendet werden. Hierzu muss im Einzelfall eine individuelle Lösung erarbeitet werden, wodurch die Erstellung von Testfällen selbst zu einer fehleranfälligen und zeitaufwändigen Programmieraufgabe wird. Der hohe Aufwand zur Implementierung konsumiert dann Ressourcen, die der inhaltlichen Entwicklung von Tests dann nicht mehr zur Verfügung stehen.

Unter der Voraussetzung, dass der Standort des Anwenders zu Zwecken des Testens simuliert werden kann, ergeben sich noch eine Reihe weiterer assoziierter Problemstellung aufgrund sekundärer Kontextparameter. Beispielsweise ist die Versorgung mit mobilem Breitbandinter-

[6]Das GPS-Exchange-Format ist ein Datenaustauschformat zur Speicherung von Geodaten.

net nicht flächendeckend homogen. In Metropolregionen kann i. d. R. eine gute Netzabdeckung über alle ISPs hinweg angenommen werden. In ruralen Gegenden ist die Abdeckung mit mobilem Internet hingegen stark heterogen. Wenn also beim Testen der Standort des Anwenders simuliert wird, ist es unter Umständen zweckmäßig oder nötig, weitere Kontextparameter ebenfalls zu simulieren. Die SDK-Werkzeuge der mobilen Plattformen bieten hierzu jedoch kaum Unterstützung. Eigenschaften von Netzwerken können in Emulatoren nur rudimentär konfiguriert werden, auf realen Geräten i. d. R. gar nicht (vgl. Vieira et al. [352]).

Stellt sich also beim Testen die Anforderung, die Funktion einer App unter Berücksichtigung eines Standortes zu prüfen, von dem eine für den Anwendungsfall unzureichende Netzwerkabdeckung vermutet wird, so muss dieser Standort in Ermangelung adäquater Simulationstechnologien für Netzwerkeigenschaften zur Testdurchführung tatsächlich aufgesucht werden, wodurch das zum Testen verfügbare Budget zusätzlich belastet wird (vgl. Haller [164], Joorabchi et al. [204]). Ähnliche Anforderungen ergeben sich auch für Anwendungen, von denen gefordert wird, dass sie auch bei hohen Geschwindigkeiten in der Nähe der Spezifikationsgrenzen von Funknetzwerken verwendet werden sollen (z. B. Medien-Streaming auf ein mobiles Gerät im Zug oder PKW).

Im Fall der Beispielanwendung Mobiler Taxiruf ist der Standort des Anwenders primärer Kontextparameter. Da die Anwendung weiterhin auf eine Netzwerkverbindung angewiesen ist, sind Verfügbarkeit und Qualität einer Netzwerkverbindung ebenfalls zu berücksichtigen. Beim Testen muss die Funktion der App also sowohl mit unterschiedlichen Standorten des Nutzers getestet werden als auch unter Verwendung verschiedener Zustände des Netzwerkadapters. Zu berücksichtigen sind beispielsweise die Zustände keine Netzwerkverbindung, Verbindung via GPRS, UMTS, LTE, Wi-Fi und existierende Netzwerkverbindung ohne Internetzugang. Letzteres kann beispielsweise der Fall sein, wenn sich ein mobiles Gerät in einem Wi-Fi eines Hotels, eines Unternehmens oder in ein öffentliches Wi-Fi (z. B. an Bahnhöfen) anmeldet, der Anwender zur Erlangung von Internetzugriff jedoch Zugangsdaten in ein Web-Formular eingeben muss. Für die Zustände, in denen die App keinen Zugriff auf das Internet erlangt, kann sie ihre Funktion nicht erfüllen. Diese unterschiedlichen Varianten müssen individuell getestet werden, da die Anwendungsspezifikation konkretes Verhalten für diese Zustände definiert.

Am Beispiel der Anwendung Mobiler Taxiruf ist die Menge der Testdaten aufgrund fachlicher Anforderungen auf das Gebiet der Bundesrepublik Deutschland begrenzt, im Fall der Beispielanwendung AR Tourist Information auf das Stadtgebiet. In beiden Fällen kann das Zielgebiet der Anwendung also durch eine polygonal begrenzte Fläche eingeschränkt werden.

Da beide Anwendungen eine Fachlogik implementieren, die bewertet, ob sich der Anwender innerhalb eines gültigen Areals befindet, werden Testdaten benötigt, die sowohl innerhalb als auch außerhalb des jeweilig zulässigen Bereichs liegen. Aufgrund technologischer Eigenschaften der Lokalisierungstechnologien ist es insbesondere in Randbereichen des fachlich zulässigen Areals möglich, dass aufeinanderfolgende Standortinformationen soweit auseinander liegen, dass die Bewertung der Gültigkeit einer Standortinformation jeweils unterschiedlich ausfällt.

Dieser Fall kann auftreten, wenn die Anwendung alternative Lokalisierungstechnologien parallel verwendet und mehrere dieser Technologien widersprüchliche Standortinformationen liefern (z. B. liefert das GPS-Modul einen Standort mit hoher Präzision, eine mobilfunkbasierte Lokalisierungstechnologie jedoch eine Standortinformation die jünger ist, eine geringere

Präzision ausweist und von ersterer Information abweicht). Auch mit einem solchen Fall muss die Anwendungslogik adäquat verfahren. Es sollten also Testfälle erstellt werden, die in rascher Folge Standortinformationen liefern, die sowohl Bereiche innerhalb als auch außerhalb des zulässigen Bereichs abdecken. Hierzu könnte beispielweise eine Reihe von WGS84-Koordinaten ausgewählt werden, die im Rahmen der zu erwartenden Lokalisierungsgenauigkeit in der Nähe der Konturlinie des zulässigen Areals liegen (Abbildung 4.2).

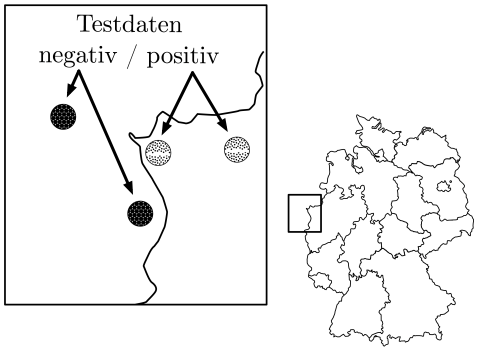


Abbildung 4.2: Auswahl von Testdaten, die sowohl innerhalb als auch außerhalb des zulässigen Areals liegen. Eine Beschränkung auf solche Daten, die entweder nur innerhalb oder nur außerhalb des zulässigen Areals liegen ist nicht ausreichend.

Aus fachlicher Sicht sind hierbei ebenfalls Standorte zu berücksichtigen, die zwar grundsätzlich innerhalb des zulässigen Areals liegen, für die aber dennoch Einschränkungen gelten. Das kann im Fall der Anwendung Mobiler Taxiruf beispielsweise für Inseln gelten, die zwar zum Gebiet der Bundesrepublik Deutschland zählen, aber nicht von Taxiunternehmen bedient werden (z. B. Nordseeinsel Spiekeroog[7]). Polygonal begrenzte Flächen können zudem Segmente enthalten, die kritisch bei der Lokalisierung des Anwenders sind (z. B. enge konkave Areale). Um hier zuverlässige Resultat zu erzeugen, muss der Standort des Anwenders mit hoher Frequenz und hoher Präzision ermittelt werden.

Für die Beispielanwendung Mobiler Taxiruf ist das u. U. relevant, um zu verhindern, dass ein Taxi an einen unzulässigen Ort gerufen wird. Kritische Areale können beispielsweise Flussufer oder Meerbusen sein, wie in Abbildung 4.3 dargestellt. Die technologisch erreichbare Lokalisierungspräzision variiert je nach Technologie. Es könnte daher der Fall eintreten, dass ein fachlich gültiger realer Standort durch Lokalisierung mit geringer Präzision inkorrekt als in einem Ausschlussareal liegend interpretiert wird (und anders herum).

Im Fall der Anwendung AR Tourist Information ist es wiederum denkbar, dass POIs so nahe beieinander liegen, dass individuelle POIs durch unzureichende Präzision der Lokalisierungstechnologie (d. h. der Standort wird vom GPS-Modul mit einer Präzisionsangabe geliefert, die größer ist als der Abstand zwischen zwei POIs) nicht zweifelsfrei unterschieden werden können. Es könnte dann der Effekt auftreten, dass zu einem POI Detailinformationen auf dem UI angezeigt werden, die inhaltlich einem anderen POI zuzuordnen sind.

Da ein Test einen solchen Defekt erkennen soll, gilt es im Rahmen der Testfallerstellung kritische POIs zu identifizieren und entsprechende Testfälle mit präparierten Standortinformationen – also mit einer Konfiguration von Standort und Präzision, die durch Überschneidung

[7] Auf der Nordseeinsel Spiekeroog verkehren mit Ausnahme von Rettungswagen keine PKW.

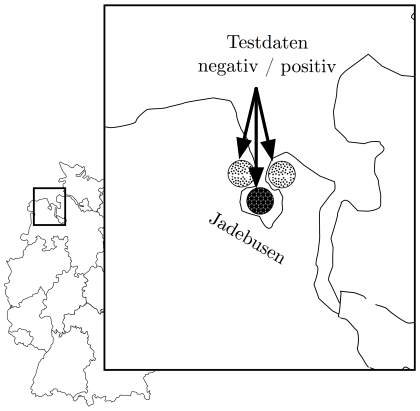


Abbildung 4.3: Kritische Areale müssen beim Testen ortsbasierter Anwendungen besonders berücksichtigt werden. Geologische Besonderheiten könnten hier dazu führen, dass bei unpräziser Standortbestimmung fachlich ungültige Werte von der Anwendung verarbeitet werden. Deshalb gilt es für kritische Areale, insbesondere konkave polygonale Flächenbegrenzungen, Testdaten sowohl in zulässigen als auch in unzulässigen Bereichen zu berücksichtigen.

mit einem anderen POI im Konflikt steht – zu erzeugen, die einen Anwendungsdefekt zuverlässig bestätigen.

Die Auswirkung von Mobilität des Anwenders auf das Testen manifestiert sich also vielgestaltig. Zunächst erfordert die Erstellung von Testfällen und Testdaten genaue Kenntnis über für den Anwendungsfall kritische geographische Regionen oder Regionen, die abhängig von ihren geographischen Eigenschaften problematisch für Lokalisierungstechnologien sind. Diese Kenntnisse variieren unter Umständen mit dem durch das SUT implementierten Anwendungsfall und sind deshalb nur bedingt zwischen Projekten übertragbar. In der Konsequenz erhöht sich also pro Projekt der Aufwand für Planung und Organisation von Testaktivitäten.

Der Aufwand für das Durchführen von Tests wird zusätzlich durch einen hohen Anteil manueller Tätigkeiten zur Simulation von Standortdaten oder Feldtests erhöht. Mobilität des Anwenders ist ein bedeutender Aufwandstreiber, der nicht lediglich durch Bereitstellung weiterer Ressourcen aufgefangen werden kann. Die Möglichkeiten manueller Tests sind zusätzlich stark begrenzt, da einige Parameter des Testens unmöglich vollständig von menschlichen Tester kontrolliert werden können. Bereits die Simulation von Standortdaten stößt hier an Grenzen, wenn ein sich verändernder Standort des Anwenders Gegenstand eines Tests ist. Einige Aspekte der Standortsimulation können hingegen gut durch einen Automatisierungslösung umgesetzt werden. Hierzu ist jedoch zwingend erforderlich, dass Testautomatisierung und eine Standortsimulationsautomatisierung miteinander synchronisiert werden, um temporale Abhängigkeiten von Tests mit der Standortveränderung des Anwenders abzubilden (z. B. Berechnung der Reisegeschwindigkeit aus aufeinanderfolgen Standortdaten).

4.2.3.2 Grenzen existierender Lösungskonzepte und Technologien

Werkzeugunterstützung ist im Jahr 2016 nur in geringem Umfang ausgeprägt, kann nicht hinreichend automatisiert werden und weist überdies den Mangel auf, das Verhalten der in realen Geräten implementierten Lokalisierungstechnologien stark zu idealisieren. Beispielsweise wird

eine Standortinformation, die im Emulator unter Verwendung der SDK-Werkzeuge simuliert wird, ohne Zeitverzögerung wirksam. Dies entspricht jedoch nicht den realen Verhältnissen, da eine variables Zeitintervall bis zum *First Fix* zu erwarten ist (bei GPS u. U. mehrere Minuten).

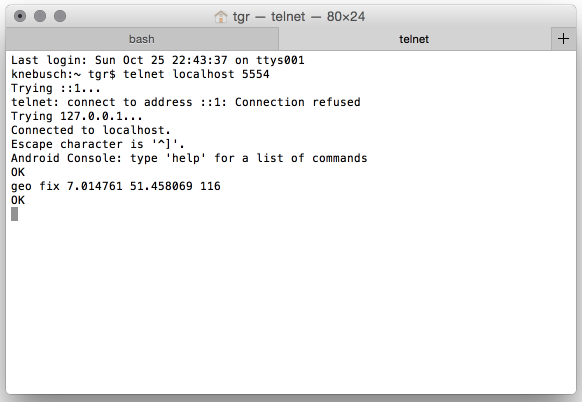
Ein weiterer wesentlicher Mangel ist, dass in Werkzeugen die Signalquelle i. d. R. nicht spezifiziert werden kann. Zwar bieten mobile Plattformen die Möglichkeit von der Signalquelle zu abstrahieren (z. B. die *FusedLocationProvider API* der Plattform Android [148]), es steht dem Entwickler jedoch auch frei, eine Signalquelle seiner Wahl zu verwenden. Unterschiedliche Signalquellen (GPS, mobilfunkbasiert, vgl. Abschnitt 3.1.2.1.3) weisen unterschiedliche Charakteristiken hinsichtlich der Verfügbarkeit, Dauer bis zum First Fix oder der Präzision auf (vgl. Zandbergen [374]). Für Anwendungen, die bei der Lokalisierung des Geräts die Signalquelle differenzieren, ergeben sich hieraus Anforderungen an Tests, die Lokalisierungsfunktion mit verschiedenen Signalquellen zur prüfen. Simulationstechnologie in den SDK-Werkzeugen der mobilen Plattformen unterstützen diese Anforderung im Jahr 2016 nicht. Das hat zur Konsequenz, dass Emulatoren nicht zum Testen von Lokalisierungsfunktionen mit verschiedenen Signalquellen verwendet werden können. Tests können folglich nur solche Situationen abbilden, in denen die Lokalisierung des Geräts bereits erfolgt ist, nicht jedoch Situationen, in denen eine Lokalisierung überhaupt nicht gelingt.

Auf der Plattform Android sind im Jahr 2016 drei Technologien zur Simulation von Standortdaten verfügbar, von denen jedoch zwei auf die Verwendung mit dem Emulator beschränkt sind. Der Android Emulator wird in der virtuellen Maschine QEMU (engl. *Quick Emulator*, schneller Emulator) [38] ausgeführt. Diese kann im Netzwerk des Backend-Systems über eine Verbindung mit dem *Teletype Network* (Telnet) (RFC 15 und 854 [70, 286]) adressiert werden. Der Entwickler einer Android App bzw. ein Tester kann sich auf diesem Weg Zugang zu einer Reihe von Funktionen verschaffen, zu denen ebenfalls die Simulation von Standortdaten gehört. Hierzu muss zunächst eine Telnet-Verbindung mit der Emulatorinstanz hergestellt werden. Anschließend können durch den Befehl

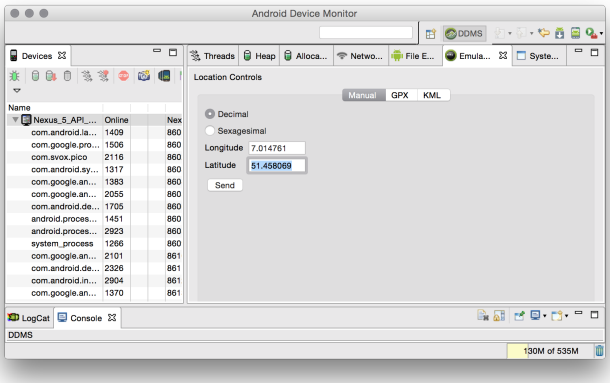
„*geo fix <Längengrad> <Breitengrad> [<Höhe>] [<Anzahl Satelliten>]*“

die Parameter einer Standortdefinition eingegeben werden (Abbildung 4.4a). Die Angabe von Längen- und Breitengrad ist obligatorisch, Höhe über mittlerem Meeresspiegel und Anzahl Satelliten ist optional. Bemerkenswert ist hier, dass mit der Anzahl Satelliten eine für ortsbasierte Anwendung i. d. R. unerhebliche Information gesteuert werden kann. Die Möglichkeit, eine Standortbestimmung ungültig zu machen oder Signalverlust zu simulieren existiert hingegen nicht. Die manuelle Eingabe von Standortdaten während der Testdurchführung fordert dem Tester ein hohes Maß an Aufmerksamkeit ab, insbesondere bei der Durchführung zahlreicher Tests mit unterschiedlichen Standortinformationen. Eine Bewegung des Anwenders lässt sich in dieser Variante nur abbilden, indem innerhalb vorgegebener Zeitintervalle neue Standortinformationen eingeben werden. Darüber hinaus stellt die Telnet-Verbindung allerdings einen möglichen Ansatzpunkt für die Implementierung einer Automatisierungslösung dar. Hier könnte eine Automatisierungslösung Kommandos zur Steuerung des Standorts Geräts absetzen, wäre dann aber auf den Emulator beschränkt.

Die zweite Variante der Simulation von Standortdaten im Android Emulator ist die Verwendung des Werkzeugs *Android Device Monitor* [129] (Abbildung 4.4b). Auch hier ist eine



(a) Simulation des Standorts via Telnet-Verbindung



(b) Simulation des Standorts via Android Device Monitor [129]

Abbildung 4.4: Werkzeuge der Plattform Android zur Simulation von Standortdaten. Die Werkzeuge des Android SDK erlauben die Verwendung einer Telnet-Verbindung zum Emulator (a) und die Verwendung des *Android Device Monitor* Werkzeugs (b).

Dateneingabe nur durch manuelle Interaktion möglich. Diese Variante stellt im Vergleich zur Lösung mit Telnet eine Verkürzung der Funktionalität dar, weil hier weder die Höhe noch die Anzahl Satelliten spezifiziert werden kann. Allerdings kann hier eine Datei im *GPS Exchange Format* (GPX) spezifiziert werden. Es wäre zwar Reproduzierbarkeit von Testdaten gewährleistet, jedoch kann diese Lösung nicht in Automatisierungswerkzeuge integriert werden.

Das Testen mobiler Anwendungen umfasst jedoch neben dem Testen auf Emulatoren ebenfalls das Testen auf realen Geräten. Im Vergleich zu Emulatoren ist auf Geräten die Bereitstellung simulierter Standortdaten nur mit Einschränkungen möglich. Es können hier zwar teilweise die Debug-Werkzeuge der plattformspezifischen SDKs im manuellen Betrieb verwendet werden. Die für Emulatoren geltenden Einschränkungen hinsichtlich der Automatisierbarkeit und der Differenzierung von Signalquellen gelten jedoch ebenfalls für Geräte.

Für die Plattform Android gilt, dass die Standardwerkzeuge zur Simulation von Standortdaten (DDMS, Telnet, vgl. Abschnitt 4.2.3.2) auf realen Geräten nicht verwendet werden können. Die dritte Lösung umfasst die Implementierung einer präparierten Signalquelle, wel-

che dem Android-System Standortinformationen vorgeben kann. Diese Variante ist die einzige Lösung, die auch auf Geräten verwendet werden kann, sofern der Anwender diese Funktion explizit in den Geräteeinstellung freigibt. Das SUT kann dann angewiesen werden, genau diese Signalquelle zur Lokalisierung zu verwenden. Diese Lösung ist auch die Einzige, die eine Spezifikation der Genauigkeit und eines definierten Zeitstempels zulässt. Diese Funktion ist besonders kritisch, da die Handhabung unpräziser Standortinformationen ein Qualitätsmerkmal ortsbasierter Anwendungen ist. Diese implementieren i. d. R. Algorithmen zur Bewertung der Qualität von Standortinformationen, die sich auf die weitere Verarbeitung des Standorts in der App auswirken. Beispielsweise kann eine Fehlermeldung angezeigt werden, wenn der Standort als nicht plausibel eingestuft wird, weil die Information veraltet ist, im Widerspruch zu einer anderen Standortinformation steht oder im Verhältnis zum verstrichenen Zeitintervall und zum zugrundeliegenden Mobilitätsmodell (z. B. Fußgänger, Autofahrer) unrealistisch vom vorherigen Standort entfernt ist. Tests für solche Anwendungen sollten daher Testdaten für Standortinformationen mit unterschiedlichen Werten für Alter der Information und Genauigkeit der Lokalisierung bereitstellen.

Allerdings erfordert diese Lösung eine Modifikation des SUT. Dieses muss so konfiguriert werden, dass es simulierte Standortdaten akzeptiert, wodurch das SUT in zwei Artefakte zerfällt: ein für das Testen optimiertes Kompilat und ein Kompilat zur Auslieferung an den Anwender. Aus Sicherheitsgründen dürfen das testoptimierte Kompilat und das Auslieferungskompilat nicht identisch sein, da hierdurch die dem Anwender übergebene Anwendung einen Angriffsvektor auf die Lokalisierung des Anwenders exponiert. Denkbar wären hier Angriffe unter Ausnutzung der Testschnittstelle, die dem Gerät gezielt manipulierte Standorte vorgeben, um den Anwendungsinhalt zu korrumpieren. Das Testen mit einem eigens hierfür erstellten Kompilat hingegen birgt ebenfalls Risiken. Es stellt sich die Frage, inwiefern Testergebnisse, die durch ein testoptimiertes Kompilat des SUT erzeugt werden, auf das für die Auslieferung intendierte Kompilat übertragbar sind (vgl. Vieira et al. [352]).

4.2.4 Physikalische und logische Kontextparameter

Anwendungen, die Sensoren zur Messung physikalischer Umgebungsparameter´ (vgl. Abschnitt 3.1.2.1) verwenden, stellen den Tester vor besondere Herausforderungen. Zunächst gilt es auch hier eine Menge von Testgeräten zu bestimmen, die für die Zielgruppe des Softwareprodukes relevant sind. Nahezu alle Geräte der Smartphone-Generation verfügen über ein GPS-Modul und die Verwendung von Ortsdaten ist für eine ortsbasierte Anwendung alternativlos. Bei Nichtverfügbarkeit eines GPS-Moduls oder falls es misslingt Standortinformationen automatisch zu bestimmen, können Standortinformationen notwendigenfalls manuell eingegeben werden. Bei den übrigen Sensoren mobiler Geräte besteht diese Möglichkeit nicht, so dass Geräte, die nicht über die für einen Anwendungsfall notwendigen Sensoren verfügen, effektiv aus der Zielgruppe ausgeschlossen sind. In einzelnen Fällen kann die Funktion bestimmter Sensoren durch Aggregation anderer Sensoren substituiert werden.

Es ist einem menschlichen Akteur aufgrund der Funktionsweise des menschlichen Körpers nicht möglich, ein mobiles Gerät um einen exakt spezifizierten Winkel zu neigen, insbesondere dann nicht, wenn die Änderung des Neigungswinkels innerhalb eines vorgegebenen Zeitintervalls erfolgen soll. Zugleich ist es einem menschlichen Akteur kaum möglich, die Korrektheit

der Reaktion der Anwendung mit bloßem Auge einzuschätzen und zu reproduzieren. Einige Sensoren werden bevorzugt verwendet, um solche Parameter zu messen, die sich bei der Bewegung oder Orientierung des Geräts ändern, um Anwendungsinhalte zu steuern. Die Werte, die diese Sensoren erzeugen, werden in diesen Fällen nur selten numerisch auf dem UI angezeigt, sondern manipulieren digitale Objekte der Anwendungslogik. Fälle, in denen eine direkte Beurteilung der Korrektheit einer Implementierung möglich ist, sind auf solche Anwendungen beschränkt, in denen die Auswirkungen der Veränderung von Sensorwerten numerisch auf dem UI abzulesen ist.

Konkret stellen sich beim Testen von Anwendungen, die physikalische Kontextparameter verwenden folgende Herausforderungen:

Präzision Ein menschlicher Tester ist nicht in der Lage, (1) ein mobiles Gerät in eine klar definierte Anfangsposition, beispielsweise eine exakt waagerechte Orientierung, zu bringen, die eine haptische Steuerung weiterhin zulässt. Ebenfalls ist er nicht in der Lage, (2) ein Gerät in einem vorgegeben Zeitintervall um einen definierten Winkel zu neigen oder zu drehen (etwa 5 Grad in 0,6 Sekunden). Hierzu ist die Motorik des menschlichen Körpers nicht ausreichend präzise.

Konzentrationsfähigkeit Ein Tester ist bei der Testdurchführung gezwungen, sich zu verhalten wie ein Anwender, um eine App adäquat zu steuern. Gleichzeitig ist es jedoch seine Aufgabe, zahlreiche Details des durch die App realisierten Inhalts (z. B. Steuerung einer Spielfigur, Ausschnitt einer Kartendarstellung, Punktzählung, Entfernungsangaben, UI, etc.) unter testspezifischen Merkmalen zu beobachten. Das heißt, er muss sich unter den durch eine Testspezifikation vorgegeben Bedingungen (z. B. Zeitrestriktionen) darauf konzentrieren die App zu steuern, muss aber gleichzeitig die Spezifikationskonformität der Software beobachten. Hierdurch wird die Testdurchführung sowohl von der individuellen Geschicklichkeit des Testers als auch von dessen individueller Konzentrationsfähigkeit abhängig, wodurch die Reproduzierbarkeit der Testdurchführung nicht gewährleistet werden kann.

Gerätebauform Die Steuerung einer Software durch Manipulation physikalischer Eigenschaften des Geräts, wie etwa dessen Orientierung, unterliegt subjektiven Eindrücken. Ein verhältnismäßig kleines Smartphone um 10 Grad um die Längsachse zu neigen ist eine andere haptische Erfahrung, als einen Tablet-Computer ebenfalls um 10 Grad um die Langachse zu neigen. Ein Spezifikationsdokument wird hierzu im Idealfall entsprechende Aussagen treffen. In jedem Fall muss die Anwendung jedoch mit beiden Geräteklassen getestet werden. Ein menschlicher Tester könnte nach mehreren Tests mit einem Smartphone eine individuelle Voreingenommenheit hinsichtlich der haptischen Erfahrung der Anwendungssteuerung entwicklen, die ihn für eine objektive Bewertung der Anwendung unter diesen Aspekten auf einer anderen Geräteklasse disqualifizieren. Er könnte zu dem Schluss kommen, die Steuerung der Anwendung durch Neigen des Geräts weiche über das akzeptable Maß von seiner Erwartung ab, da diese Erwartung durch die Spielerfahrung auf einer anderen Geräteklasse vorherbestimmt ist.

Darüber hinaus ergibt sich ebenfalls die Herausforderung adäquate Testdaten bereitzustellen. Es ist eine Frage des konkreten Anwendungsfalls, ob die durch die Gerätesensoren

erzeugten Daten in ihrem Rohformat von der Anwendung verwendbar sind oder ob erst die Aggregation der Daten mehrerer Sensoren verwertbare Daten repräsentieren. Für Anwendungen, die aus Sensorinformationen die Orientierung oder Lage des Geräts berechnen, gilt typischerweise der zweite Fall. Beispielsweise existiert in mobilen Geräten kein Sensor, aus dem sich Roll-, Nick-, und Gierwinkel direkt ablesen lassen. Testfälle, in denen diese Werte für den Inhalt eines Testfalls relevant sind, müssen Testdaten also alternativ bereitstellen.

4.2.4.1 Auswirkungen auf das Testen

Die Verwendung von physikalischen Kontextparametern in mobilen Anwendungen hat die konkrete Auswirkung, dass sich über den erhöhten Aufwand zur Erstellung von Tests und Testdaten und über deren manuelle Durchführung hinaus Grenzen der Machbarkeit auftun.

Es stellen sich konkret zwei Problemstellungen: (1) Es müssen Daten erzeugt werden, die den Testfall entsprechend repräsentieren und (2) diese Daten müssen einem Gerät oder Emulator während des Testens zugeführt werden. Die Lösung des ersten Problems ist nicht trivial, da hier zur Bereitstellung von Testdaten ggf. die Abstraktionsebene der Datenaufbereitung gewechselt werden muss. Für einen Testfall, der fordert eine Anwendung oder eine Komponente einer Anwendung auf korrektes Verhalten zu überprüfen, wenn das Gerät um 30° entlang seiner Längsachse geneigt wird, müssen zunächst Testdaten für die in der Anwendung zur Lagebestimmung verwendeten Sensoren erzeugt werden. Im konkreten Fall müssten Testdaten beispielsweise für den Beschleunigungs- und den Magnetfeldsensor so bestimmt werden, dass sie nach der Berechnung innerhalb der Anwendungslogik den gewünschten Neigungswinkel produzieren. Die Lösung dieses Problem ist mit Mitteln der Mathematik zu erreichen, da der Algorithmus zu Berechnung der jeweiligen Nutzgröße aus den Sensordaten bekannt ist.

Die Lösung der zweiten Problemstellung ist hingegen technischer Natur. Mobile Geräte der Smartphone-Generation um das Jahr 2016 verfügen i. d. R. nicht über Schnittstellen, Sensorinformationen während des Testens zu simulieren. Lediglich Emulatoren bieten hier eingeschränkte Möglichkeiten Sensordaten zu simulieren.

Ein Testfall für eine Anwendung, die beispielsweise auf Neigen des Geräts um die Längsachse reagiert (z. B. Doodle Jump [237], Abbildung 3.9a), könnte zum Beispiel eine präzise zeitliche Abfolge von definierten Neigungswinkeln vorschreiben, um die Korrektheit der Implementierung des auswertenden Algorithmus zu testen. Durch einen White-Box-Test auf der Ebene eines Unit-Test könnte der Algorithmus zwar isoliert getestet werden. Um die komplexen Interaktionen unterschiedlicher Software-Komponenten miteinander zu testen, sind Unit-Tests jedoch häufig nicht ausreichend. Auf der Ebene funktionaler Akzeptanztests, die diese Interaktion aus Anwenderperspektive testen, besteht jedoch gegenwärtig keine technische Automatisierungslösung, so dass auf manuelle Tests zurückgegriffen werden muss. Von einem menschlichen Tester kann jedoch nicht erwartet werden, eine definierte Abfolge von Neigungswinkeln in einem vorgegebenen zeitlichen Zusammenhang zu reproduzieren. In der Konsequenz kann hier nur ein näherungsweises manuelles Testen auf einer Auswahl unterschiedlicher Geräte erfolgen. Über die Übertragbarkeit der Testergebnisse kann lediglich gemutmaßt werden.

Die Anwendung AR Tourist Information implementiert *Augmented Reality* (AR), in welcher ein Live-Bild der Gerätekamera mit zusätzlichen Informationen zum Standort eingeblendet wird. Hierbei ist die Ausrichtung des Geräts von Bedeutung. Ein Test muss überprüfen,

ob zu einem gegebenen Standort in unterschiedlichen Blickrichtungen jeweils die korrekte Einblendung zum Umfeld des Anwenders angezeigt wird. Zur Durchführung dieses Tests müssen zunächst repräsentative Standorte innerhalb des zulässigen Areals ausgewählt werden. Weiterhin muss bestimmt werden, mit welchen Zusatzinformationen das Live-Bild der Gerätekamera je nach Blickrichtung des Anwenders überblendet werden soll. Blickrichtung Osten erfordert beispielsweise eine andere Informationsanzeige als Blickrichtung Norden oder Westen. Dem Anwender ist es freigestellt, seine Position und Ausrichtung frei zu wählen, weshalb nicht diskrete Werte wie Norden, Westen oder Ostern in Testdaten zu verwenden sind, sondern das gesamte Spektrum von 0°-360°. Gegenstand des Tests ist dann nicht nur, ob die korrekte Information angezeigt wird, sonder auch, ob die Information in Abhängigkeit von der Blickrichtung des Anwenders an der richtigen Stelle auf dem Display angezeigt wird.

Eine Option zur Durchführung des Tests ist es, den Test vollständig manuell *in loco*, also vor Ort, durchzuführen. Der Tester muss sich mit einer Geräteauswahl in das Zielgebiet der App begeben, dort relevante POIs aufsuchen, sich mit Hilfe eines Kompasses gemäß der Testspezifikation in einer Gradzahl relativ zum magnetischen Nordpol ausrichten und überprüfen, ob auf dem Display korrekte Inhalte angezeigt werden. Tritt an einem bestimmten POI, bei einer bestimmten Ausrichtung oder bei einem bestimmten Gerät ein Fehler auf, muss die App korrigiert werden. Ist von dieser Korrektur der Algorithmus zur Implementierung des digitalen Kompasses (z. B. Magenetfeld- und Beschleunigungssensor zur Bestimmung von Roll-, Nick- und Gierwinkel (auch Azimut)) oder der Algorithmus zur Berechnung der Darstellungskoordinaten auf dem Display betroffen, so werden alle bisherigen Testergebnisse ungültig. Der Test muss dann unter Inkaufnahme hohen Aufwands wiederholt werden.

Eine weitere Option ist eine teilautomatisierte Ausführung des Tests, wobei wenigstens der Standort, nicht aber die Ausrichtung des Geräts, unter Verwendung von SDK-Werkzeugen (vgl. Abschnitt 4.2.3.2) auf dem Gerät simuliert wird. In diesem Fall kann der Test durchgeführt werden, ohne das Zielgebiet der App tatsächlich aufzusuchen. Die manuelle Ausrichtung des Geräts kann hingegen nicht vermieden werden, so dass hier, wie auch in der ersten Option, Raum für fehlerhafte Testdurchführung gegeben ist.

Beide Optionen erfordern signifikante manuelle Arbeitsleistung. Die zur Testdurchführung benötigte Zeit ist direkt abhängig von der Anzahl der gewählten POIs und Anzahl unterschiedlicher Blickrichtungen pro Testfall. Es handelt sich um eine monotone Tätigkeit, die ein hohes Maß an Aufmerksamkeit erfordert und anfällig für störende Umwelteinflüsse ist. Die Gegenwart magnetischer Gegenstände kann sowohl den in der App implementierten digitalen Kompass als auch den als Vergleichswert verwendeten realen Kompass stören, wodurch eine Reproduktion von Testdaten unmöglich wird.

Dieses Beispiel soll darstellen, dass menschliche Tester für die Durchführung bestimmter Arten von Tests aus objektiven Gründen nicht geeignet sind. Eine Automatisierungslösung kann hier dabei helfen, subjektive Eindrücke menschlicher Tester aus der Bewertung von Testergebnissen zu entfernen. Voraussetzung hierzu ist allerdings eine adäquate Automatisierungstechnologie, die insbesondere in der Lage sein muss, solche Kontextparameter zu simulieren, die für die Auswertung von Orientierung und Bewegung des Geräts verwendet werden.

Zusammenfassend manifestieren sich die Auswirkungen physikalischer Kontextparameter also darin, dass zur Realisierung von Tests, die physikalische Parameter (Umgebungstempe-

ratur, Lichtintensität, Magnetfelder usw., vgl. Abschnitt 3.1.2) berücksichtigen, ein enormer Aufwand entsteht, der Nutzen hingegen zweifelhaft ist. Weil bei einer manuellen Ausführung Umgebungsbedingungen nicht zuverlässig reproduzierbar sind, entstehen durch eine manuelle Ausführung keine belastbaren Ergebnisse.

4.2.4.2 Grenzen existierender Lösungskonzepte und Technologien

Eine Automatisierungslösung, die in Zusammenarbeit mit geeigneten Testschnittstellen Sensordaten künstlich in ein SUT einbringt, kann diese Problematik entschärfen. Auch sensorbasierte Anwendungen können dann automatisiert über ein breites Spektrum von Geräten oder Emulatorkonfigurationen getestet werden. Der Schlüssel ist *Design for Testability* (vgl. Pettichord [281]), eine für das Testen optimierte Implementierung der mobilen Plattform. Zur Entfaltung kommt dieser Ansatz allerdings nur, wenn er auf Betriebssystemebene implementiert wird, d. h. Testschnittstellen nicht im SUT, sondern in der Plattform verankert werden.

Auf der Plattform Android beispielsweise können Sensordaten im Emulator ähnlich der Simulation von Standortdaten über eine Telnet-Verbindung zum Emulator simuliert werden. Allerdings gelten hier eine Reihe von Einschränkungen. Sensordaten unterliegen einem natürlichen technologiebedingten Rauschen. Dieses Rauschen wird durch die Simulation im Emulator nicht nachgebildet. Hierbei handelt es sich um einen Mangel, da gerade der Umgang mit Sensorrauschen eine der wesentlichen Herausforderungen des Umgangs mit Sensoren in mobilen Anwendungen ist. Zudem entstehen praktische Schwierigkeiten. Sensordaten werden i. d. R. im Abstand weniger Millisekunden aktualisiert. Einem menschlichen Akteur ist es keinesfalls möglich, in einer solchen Geschwindigkeit Testdaten in eine Konsole einzugeben.

Lösungen, die Sensordaten ohne Rauschen simulieren, sind daher nur für solche Anwendungen und Sensoren geeignet, bei denen der Einfluss von Sensorrauschen vernachlässigt werden kann und die in hinreichend langen Zeitintervallen aktualisiert werden, so dass diese Aktualisierung unter Verwendung von Simulationswerkzeugen von menschlichen Akteuren durchgeführt werden kann. Weiterhin besteht die Schwierigkeit, dass die meisten mobilen Plattformen solche Testschnittstellen für Sensoren nicht anbieten. Selbst auf der Plattform Android ist diese Technologie auf Emulatoren beschränkt und kann auf Geräten nicht verwendet werden.

In Ermangelung von Testschnittstellen, die außerhalb des SUT Sensordaten auf Systemebene simulieren können, haben sich einige Ersatztechnologien etabliert, die das Verhalten der sensorverarbeitenden Systemkomponenten nachstellen und alternativ in Anwendungen verwendet werden können (z. B. OpenIntents SensorSimulator™ [269] für Android, vimov iSimulate™ [354] für Apple iOS). Diese haben allerdings den Nachteil, dass sie massive Veränderungen am zu testenden System erforderlich machen. Anstelle der Standardsystemkomponenten zur Verarbeitung von Sensordaten müssen die Komponenten dieser Hersteller verwendet werden. Wenngleich beide Technologien über eine Bypass-Funktion verfügen, um außerhalb von Testszenarien die Daten der tatsächlichen Sensoren an die Anwendung weiterzugeben, so öffnen beide Technologien eine Netzwerkschnittstelle innerhalb des SUT, wodurch dieses einen Angriffsvektor exponiert. Diese Technologien sind deshalb aus Sicherheitsgründen ungeeignet, um nach dem Testen in der App zu verbleiben. Eine Modifikation der App nach dem Testen macht allerdings alle Testergebnisse obsolet. Weiterhin sind diese Technologien ebenfalls nicht geeignet, um innerhalb von Automatisierungslösungen verwendet zu werden.

Das in dieser Dissertation untersuchte Konzept zur Testautomatisierung für mobile, kontextsensitiv Anwendungen untersucht ein *Design for Testability* exemplarisch an der Plattform Android durch Implementierung von Testschnittstellen, die Testen sensorbasierter Anwendungen durch Bereitstellung künstlicher Sensordaten ermöglichen. Hierdurch können die Auswirkungen von Kontextsensitivität mobiler Anwendungen auf das Testen hinsichtlich des zusätzlichen entstehenden Aufwands zur Erreichung einer signifikanten Geräte- und Kontextabdeckung wesentlich reduziert werden.

4.3 Testautomatisierung

Je nach Teststufe beinhaltet bereits der Entwicklungsprozess die regelmäßige und häufige Ausführung individueller Tests. Insbesondere bei der Verwendungen von TDD-Vorgehensmodellen, bei denen Tests vor dem eigentlichen Quellcode implementiert werden, ist von einer hohen Anzahl von Wiederholungen von Tests auszugehen. Tests auf einer Quellcode[nahen] Stufen (d. h. Unit-Tests) werden hierbei besonders häufig ausgeführt (z. B. bei jedem *Check-In* in ein SCM-System), während die Anzahl komplexerer Tests bis hin zu Akzeptanztests aufgrund der möglicherweise erforderlichen Involvierung menschlicher Akteure typischerweise abnimmt, in CI-Umgebungen aber immer noch hoch sein kann, z. B. täglich im Rahmen von *Nightly Build*s.

Dem hierdurch entstehenden personellen Aufwand durch manuelles Testen kann durch die Verwendung von Technologie entgegengewirkt werden. Hierzu kommen Testautomatisierungswerkzeuge zum Einsatz, die einzelne Aktivitäten der Qualitätssicherung in unterschiedlichem Maß von menschlicher Arbeitskraft entkoppeln. Hierdurch ergeben sich eine Reihe von Vorteilen, die den Testprozess sowohl inhaltlich als auch wirtschaftlich verbessern. Aspekte, in denen sich die Verwendung von Testautomatisierung als vorteilhaft erweist, sind (1) Geschwindigkeit, (2) Effizienz, (3) Präzision und Akkuratesse und (4) Härte (vgl. Patton [275]).

Der Vorteil in der höheren Geschwindigkeit der maschinellen Testausführung ergibt sich aus der Tatsache, dass Computer in der Lage sind, mehr Tests in einem kürzeren Zeitraum auszuführen als menschliche Akteure, hierbei nicht ermüden und nicht von biologischen Faktoren, wie etwa Konzentrationsfähigkeit, abhängig sind. Für viele Softwaretests gilt, dass das getestete System Interaktionen (z. B. Eingabe von Daten, starten von Operationen) wesentlich schneller verarbeiten kann, als ein menschlicher Akteur in der Lage wäre, Eingaben in das SUT zu tätigen und die Gültigkeit von Nachbedingungen zu überprüfen. [275]

Effizienzvorteile ergeben sich aus der Tatsache, dass die Arbeitskraft eines Entwicklers oder Testers während des manuellen Testens vollständig in Anspruch genommen wird und nicht für andere Tätigkeiten zur Verfügung steht. Es ist deshalb sinnvoll, solche Tätigkeiten, die keine kognitive Kreativleistung erfordern und deshalb automatisierbar sind, durch Maschinen durchführen zu lassen. Menschliche Arbeitskraft kann dann für andere Tätigkeiten freigesetzt werden, z. B. Weiterentwicklung der Software oder Erstellen weiterer Testfälle. [275]

Weiterhin sind Maschinen Menschen hinsichtlich Präzision und Akkuratesse überlegen, da sie auch bei langanhaltenden, monotonen Tätigkeiten nicht ermüden. Fehler durch Konzentrationsdefizite, wie sie bei einem menschlichen Tester beim häufigen Durchführen vieler Tests zu erwarten sind (vgl. Baur und Groth [30]), können bei der Verwendungen maschinenbasierter Technologie sicher ausgeschlossen werden. Computerbasierte Werkzeuge sind in der Lage,

Tests beliebig häufig mit identischer Qualität durchzuführen. [275]

Mit Härte bezeichnet Patton die Eigenschaft eines maschinenbasierten Werkzeugs, Aufgaben beliebig großen Umfangs ungeachtet der zu erwartenden Zeitdauer zuverlässig erfüllen zu können. Das subjektive Gefühl, mit der Durchführung von hunderten oder tausenden Testfällen vor einer scheinbar unlösbaren Aufgabe zu stehen, zeitlich oder kognitiv überfordert zu sein, kann sich demotivierend oder frustrierend auf einen menschlichen Tester auswirken [30]. Dieses Kriterium ist für Maschinen irrelevant.

Darüber hinaus existieren Tests, die aufgrund ihrer inhaltlichen Ausrichtung für eine manuelle Durchführung nicht in Frage kommen. Hierzu zählen beispielsweise *Load Tests* (engl. Lasttests) (oder Stresstests, vgl. Patton [275]), die keine funktionalen Aspekte einer Software zum Gegenstand haben, sondern vielmehr überprüfen sollen, wie lastverträglich ein SUT ist. Das heißt, es wird versucht, ein SUT durch eine große Anzahl möglicherweise paralleler Zugriffe so lange unter Last zu versetzen, bis entweder die in der Spezifikation geforderte Lastverträglichkeit nachgewiesen ist oder das SUT kollabiert. Diese Art von Tests können i. d. R. nur durch Verwendung von Automatisierungstechnologie durchgeführt werden.

Im Umfeld von Desktop- und Server-Software hat Testautomatisierung mit der Integration von Unit-Test-Technologie in gängige IDEs bereits umfangreich Einzug in den Softwareprozess gehalten. Nicht so jedoch im Umfeld mobiler Anwendungen. Insbesondere das Testen kontextsensitiver Anwendungen wird im Jahr 2016 nur rudimentär von Testwerkzeugen unterstützt. Beispielsweise sind Testautomatisierungstechnologien nicht in der Lage, wechselnde Standorte des Anwenders in Testfällen zur berücksichtigen, so dass Tester zum Durchführen von Tests ortsbasierter Funktionen auf manuelle Testverfahren zurückgreifen müssen. Maschinelle Verfahren sind hier geeignet, um nicht nur den manuellen Aufwand zu reduzieren und damit das Budget eines Softwareprojekts zu entlasten, sondern sie sind i. d. R. auch zuverlässiger bei der exakten Reproduktion von Tests und schneller bei deren Ausführung.

Hoffman [183] untersuchte bereits im Jahr 1999 allgemeine Architekturen für Testautomatisierungswerkzeuge. Der Autor identifiziert eine Vielzahl zu lösender Aufgaben der Testautomatisierung. Zunächst einmal erschöpft sich Testautomatisierung nicht in der bloßen automatisierten Ausführung von Tests, d. h. des Einsatzes einer Maschine als Ersatz für einen menschlichen Akteur. Vielmehr beinhalten Testautomatisierungsaktivitäten das gesamte Spektrum von Aufgaben der Erstellung von Testfällen, der Auswahl, Konfiguration bzw. Implementierung von Testwerkzeugen bis hin zur Beobachtung und Interpretation von Testergebnissen. Insbesondere die Auswertung von Ergebnissen ist kritisch bei der Verwendung von Testautomatisierungstechnologie, da hier eine immense Anzahl von Tests in kurzer Zeit ausgeführt werden kann und u. U. eine große Menge von Ergebnissen entsteht, die nicht mehr wirtschaftlich sinnvoll manuell auszuwerten ist.

Mit der Frage, unter welchen Umständen Tests automatisiert werden sollten, beschäftigten sich Ramler und Wolfmaier [291] im Jahr 2006. Die Autoren diskutieren kritisch die Gültigkeit der intuitiven Annahme, dass in einem Softwareprojekt möglichst alle Tests automatisiert werden sollten (vgl. Bertolino [41]). Allerdings geht die Automatisierung von Tests mit u. U. hohen Investitionskosten einher, so dass es einer sorgfältigen Abwägung bedarf, ob die zu erwartenden Gewinne durch schnelle, automatisierte Testausführung durch Automatisierung die Investitionskosten amortisieren. Testautomatisierung ist sinnvoll, wenn bei einer identischen

Anzahl von Testausführungen die Kosten einer automatisierten Ausführung geringer sind als die Kosten der manuellen Ausführung [291]. Unter Vernachlässigung der Investitionskosten kann eine Amortisierung hier schnell erreicht werden, wenn lediglich die Kosten der manuellen Ausführung den Kosten einer automatisierten Ausführung gegenübergestellt werden. Der tatsächliche ROI ist hingegen nur schwer zu bestimmen, da i. d. R. nur die Kosten für die Einführung einer Automatisierungstechnologie bzw. für die Testdurchführung bekannt sind. Die erzielbare Kostenreduzierung durch Automatisierung kann hingegen nur schwer beziffert werden, da automatisierte und manuelle Tests nicht direkt miteinander vergleichbar sind. Es ist nicht nur der Prozess der Durchführung der automatisiert wird, sondern es liegen andere Denkmodelle zugrunde, die sich in unterschiedlicher inhaltlicher und technischer Ausrichtung von manuellen und automatisierten Tests manifestieren, z. B. adressieren sie typischerweise unterschiedliche Defekte (vgl. Bach [22]).

Weiterhin diskutieren Ramler und Wolfmaier das von Biffl et al. [44] unterstellte wertneutrale Testen, das alle Tests und insbesondere alle Defekte als gleichrangig betrachtet. In der Realität trifft diese Annahme nicht zu. Testfälle haben unterschiedliche Prioritäten und Defekte sind für ein Projekt unterschiedlich wichtig (vgl. Abschnitt 4.1.1.3). Insbesondere auf der Ebene von Akzeptanztests können Defekte geringer Priorität u. U. vorläufig akzeptiert werden, ohne den Erfolg eines Produkts insgesamt zu gefährden. Als besonderes Risiko für die Wirtschaftlichkeit von Testautomatisierung identifizieren Ramler und Wolfmaier, dass Kosten für die Automatisierung von Tests sich nicht mehr amortisieren, wenn die getestete Funktionalität spät im Projekt aus dem Softwareprodukt entfernt wird.

Kasurinen et al. [213] untersuchten im Jahr 2010 Projekte in 31 Softwareunternehmen hinsichtlich der Verwendung und der Nützlichkeit von Testautomatisierungswerkzeugen mit dem Ergebnis, dass Testautomatisierung generell geeignet ist, die Nutzung von Ressourcen für Aktivitäten der Qualitätssicherung zu reduzieren. Insbesondere repetitive Aufgaben konnten als geeignete Ziele für Automatisierung identifiziert werden, um rare Ressourcen, wie etwa kognitive Leistung von Entwicklern und Testern, auf wertschöpfendere Aktivitäten zu fokussieren. Kasurinen et al. stellen heraus, dass die Testbarkeit (vgl. *Design for Testability*, Pettichord [281]) einer Software hierbei einen wesentlichen Einfluss auf die Nützlichkeit von Testautomatisierung hat. Das in dieser Dissertation untersuchte Konzept zur Testautomatisierung mobiler, kontextsensitiver Anwendungen hat u. a. die Bereitstellung von Testschnittstellen auf der Ebene des Betriebssystem zum Ziel, so dass ein adäquates *Design for Testabitilty* bereits durch die Zielplattform gegeben und im SUT implizit verfügbar ist. Hierdurch kann der Nutzen von Testautomatisierung in einem Softwareprojekt maximal ausgeschöpft werden, ohne zusätzliche Kosten zur Anpassung des SUT zu verursachen.

Haller [164] argumentiert ebenfalls aufgrund ökonomischer Faktoren für die Verwendung von Testautomatisierung. Mobile Anwendungen sollen i. d. R. einen Markt adressieren, der durch eine ausgeprägte Geräteheterogenität charakterisiert ist. Apps müssen aus diesem Grund auf einer großen, während des Projektverlaufs möglicherweise noch anwachsenden Zahl unterschiedlicher Geräte getestet werden (*Scalable Test Configuration Coverage* [164], vgl. Abschnitt 4.2.1). Hierdurch entsteht selbst bei einer vergleichsweise geringen Anzahl von Tests zur Sicherstellung einer minimalen funktionalen Testabdeckung (*Minimal Functional Coverage* [164]) des SUT großer Aufwand zur Testdurchführung. Insbesondere da zu erwarten ist,

dass viele Tests häufig, z. B. auf täglicher Basis bei Verwendung von CI-Umgebungen beim Erstellen von Nightly Builds ausgeführt werden, entsteht bei menschlichen Testern nicht nur hoher Arbeitsaufwand, sondern auch Frust durch monotone Tätigkeit, die langfristig zu nachlassender Qualität führt (vgl. Baur und Groth [30]). Technische Automatisierungslösungen hingegen können mit geringem Aufwand skaliert werden, z. B. durch Bereitstellung zusätzlicher Hardware, wodurch selbst bei überlinearem Anstieg der Anzahl von Tests oder der Testhäufigkeit ein konstanter, manueller Arbeitsaufwand ausreichend ist.

Die Unterstützung von Testaktivitäten durch maschinengestützte Verfahren bringt jedoch Herausforderungen mit sich. Zunächst einmal müssen automatisierbare Aktivitäten identifiziert werden. Der Grad, zu dem eine Testaktivität automatisiert werden kann, wird durch mehrere Faktoren bestimmt. Grundvoraussetzung ist, dass Vor- und Nachbedingungen im Rahmen der Testdurchführung hergestellt werden können, ohne dass hierzu bei der Testausführung manuelle Interventionen menschlicher Akteure erforderlich ist. Faktoren, die sich grundsätzlich dem maschinengestützten Zugriff entziehen repräsentieren deshalb ein Hindernis, das eine Testautomatisierung für diesen spezifischen Aspekt des Systems ausschließt. Weiterhin stellt sich auch beim automatisierten Testen die Frage nach der Herkunft von Tests in zweierlei Hinsicht. Zunächst einmal muss bestimmt werden, welche Komponenten oder Funktionen einer Software getestet werden sollen und welche Testdaten hierzu verwendet werden sollen. Weiterhin müssen sich Tests in einer für den Menschen und – für die Automatisierung – für Maschinen lesbaren Form manifestieren.

Im Folgenden werden automatisierbare Aktivitäten des Testens (vgl. Abschnitt 4.3.1) sowie einige Lösungsansätze der Testautomatisierung (Abschnitt 4.3.2) diskutiert und in Bezug zu dieser Dissertation gesetzt. Abschnitt 4.3.3 adressiert den Stand der technischen Möglichkeiten der Testautomatisierung im Jahr 2016.

4.3.1 Automatisierbare Aktivitäten in Testprozessen

Das Testen von Software umfasst eine Reihe von Aktivitäten in unterschiedlichen Phasen des Softwareprozesses. Neben administrativen und organisatorischen Aufgaben gehören hierzu insbesondere das Spezifizieren von Testfällen und Testdaten, die Durchführung von Tests und die Analyse von Testergebnissen.

Die Spezifikation von Testfällen umfasst die Festlegung des zu testenden Funktionsumfangs des SUT gemäß der in der Teststrategie festgelegten Rahmenbedingungen und die Manifestierung von Testfällen und Testdaten in einem für die spätere Durchführung geeigneten Format. Jeweils pro Teststufe gehört hierzu die Auswahl von Quellcode[artfakten] (bei Komponententests / Unit-Tests), die Auswahl der Komponenten, die in einem Integrationstest auf Interoperabilität geprüft werden sollen, die Spezifikation von Rahmen- und Einbettungsbedingungen beteiligter Softwaresysteme bei Systemtests, die Auswahl zu testender Pfade durch eine Sofware aus Anwendersicht bei Akzeptanztests sowie die Bestimmung von Testfällen und Wiederholungspunkten bei Regressionstests. Auf allen Teststufen müssen pro Testfall und Testschritt Vor- und Nachbedingungen definiert werden.

Je nach Teststufe sind für die Erstellung von Testfällen unterschiedliche Kenntnisse des SUT erforderlich, die sich jeweils aus unterschiedlichen Abschnitten der Software-Spezifikation ergeben. Für die Erstellung von Komponententests beispielsweise muss der Tester nicht zwin-

gend über Kenntnisse des fachlichen Inhalts des zukünftigen Anwendungsfalls verfügen, wohl aber über Tiefenkenntnisse der technischen Implementierung. Zur Erstellen von Akzeptanztests hingegen verhält es sich genau umgekehrt. Aus dieser Diskrepanz ergeben sich unterschiedliche Anforderungsprofile an den spezifizierenden Tester. Komponententests werden i. d. R. direkt vom Entwickler/Programmierer (auch im Rahmen eines TDD-Vorgehensmodells) spezifiziert, realisiert und durchgeführt. Bedarfsträger für Akzeptanztests sind hingegen Rollen nahe dem Fachbereich oder der Schnittstelle zum Auftraggeber, z. B. *Product Owner*, oder Rollen, die eng mit dem Fachbereich zusammenarbeiten. Typisch für solchen Rolle ist, dass ihr individuelles Anforderungsprofil keine weitreichenden Technologiekenntnisse voraussetzt, so dass eine Testfallspezifikation regelmäßig nicht als technologiegängiges Artefakt vorliegt. Üblich sind natürlichsprachliche textuelle Testfallbeschreibungen, die ihrerseits eine technologische Aufbereitung durch implementierungsnahes Personal (z. B. Entwickler) erfordern, sofern eine nicht-manuelle Durchführung angestrebt wird.

Zur Erstellung einer umfassenden Testsuite, durch die ein Softwareprodukt in seinem wesentlichen (und durch die Teststrategie priorisierten) Umfang getestet werden kann, entsteht aufgrund der unterschiedlichen Profilierung beteiligter Akteure ein hoher Aufwand, insbesondere bei der Kommunikation zwischen fachlich und technisch orientierten Akteuren. Dieser Aufwand kann durch Verwendung von Automatisierungstechnologie reduziert werden.

Insbesondere im Umfeld mobiler Anwendungen gehört zur Spezifikation von Testfällen ebenfalls die Bestimmung relevanter Kontextparameter, die bei der Testdurchführung durch eine geeignete Simulation zu manipulieren sind, um bestimmte Funktionen des SUT zu testen. Die konkrete Auswahl dieser Parameter und die Bestimmung von Testdaten ist im Jahr 2016 eine Aufgabe, deren Automatisierung in der wissenschaftlichen Literatur bislang nicht thematisiert wurde und auch nicht Gegenstand dieser Dissertation ist. Es wird deshalb vorausgesetzt, dass testrelevante Kontextparameter manuell aus Spezifikationsdokumenten extrahiert werden müssen.

Ein weiteres im Umfeld mobiler Anwendungen zu verortendes Problem ist die Auswahl eines testrelevanten Portfolios mobiler Geräte oder Emulatorkonfigurationen. Die Konkretisierung einer solchen Auswahl erfolgt i. d. R. auf Basis einer Markt- und Zielgruppenanalyse und ist deshalb schwer automatisierbar. In der wissenschaftlichen Literatur wurde dieses Thema bis zum Jahr 2016 ebenfalls nicht in nennenswertem Umfang untersucht. Für diese Dissertation wird auch hier eine manuelle Selektion eines Geräteportfolios vorausgesetzt.

Die Generierung von Testfällen, Testdaten, die Testdurchführung und die Analyse von Testergebnissen sind hingegen Probleme, die seit vielen Jahren Gegenstand von Forschungsarbeiten sind. In den folgenden Abschnitten werden automatisierbare Aktivitäten des Testens im Bezug zum hier untersuchten Konzept der Testautomatisierung mobiler, kontextsensitiver Anwendungen diskutiert.

4.3.1.1 Generierung von Testfällen

Die Testfallerstellung ist eine der anspruchsvollsten Aktivitäten des Testens. Im Vergleich zur Testdurchführung wird diese Aktivität zwar weniger häufig durchgeführt, orientiert sich aber sehr stark an der Spezifikation des SUT und erfordert in weit höherem Maße kognitive Leistungen. Das erneute Erstellen von Testfällen ist i. d. R. immer dann erforderlich, wenn sich die

Spezifikation des Softwaresystems verändert hat (z. B. hinzufügen oder entfernen von Anforderungen durch den Auftraggeber, Iteration der inkrementellen Softwareentwicklung oder der Zielplattform) oder wenn im (Test-)Betrieb einer Software Defekte beobachtet werden, deren mögliche Ursache bislang nicht durch Testfälle abgedeckt ist.

Ein wirtschaftliches Risiko der manuellen Erstellung von Testfällen ergibt sich ebenfalls aus der ansteigenden Komplexität des Testcodes mit voranschreitender Entwicklung eines Softwareprodukts. Meszaros et al. [246] argumentieren hier, dass besonders in TDD-Umgebungen der Aufwand zur Pflege von manuell erstellten Tests im Verhältnis zum Aufwand für die inhaltliche Weiterentwicklung der Software überproportional ansteigt. Weiterhin charakterisieren Meszaros et al. Testcode als obsolet aus der Perspektive des Auftraggebers. Das wirtschaftliche Risiko des Testens liegt beim Auftragnehmer, der dem Testen innerhalb eines verfügbaren Budgets angemessen Stellenwert einräumen muss. Steigt der wirtschaftliche Druck auf ein Softwareprojekt, gehört das Testen zu denjenigen Aktivitäten, die als Erstes zum Ziel von Einsparungen werden, weil es nicht konstruktiv zur Produktentwicklung beiträgt [246]. Deshalb kann die Generierung von Tests anstelle einer manuellen Erstellung dazu beitragen, nicht nur die Testkomplexität besser zu beherrschen, sondern auch den Aufwand zu reduzieren.

Um wirtschaftliche Schäden aus defekter Software zu vermeiden ist es erforderlich, Testfälle hoher Qualität mit einer risikoadäquaten Testabdeckung zu erstellen. Aufgrund der hohen Testkomplexität von Softwaresystemen ist diese Aufgabe für menschliche Akteure anspruchsvoll. Beispielsweise können funktionelle Bereiche eines Softwaresystems versehentlich übersehen oder ihre Bedeutung für den qualitativen Gesamteindruck einer Software könnte falsch eingeschätzt werden. In beiden Fällen ist zu befürchten, dass Mängel der Testfallspezifikation erst spät (d. h. nahe dem oder sogar nach Auslieferungszeitpunkt eines Softwareinkrements) erkannt werden und eine Richtigstellung der Testfallspezifikation außerplanmäßige (und vor allem außerbudgetäre) korrektive Maßnahmen erforderlich machen.

Die Automatisierung der Generierung von Testfällen stellt aus verschiedenen Gründen eine signifikante Verbesserung von Testprozessen dar. Zum einen besteht hier das Potenzial, menschliche Fehlerquellen durch technische Unterstützung der Testfallgenerierung zu minimieren, indem ein menschlicher Akteur bei einigen Aktivitäten vollständig durch maschinelle Werkzeugunterstützung abgelöst wird oder indem Softwarewerkzeuge einen menschlichen Akteur so bei seiner Arbeit unterstützen, dass häufige Fehlerquellen (z. B. Übersehen von Anforderungen durch hohe Softwarekomplexität) unwahrscheinlicher werden. Andererseits besteht auch die Möglichkeit, die Revision von Testfällen zu beschleunigen, indem menschliche Akteure durch Werkzeugunterstützung auf solche Bereiche der Software hingewiesen werden, wo Änderungen der Spezifikation Anpassungen von Tests erforderlich machen.

Da Testen von Software auf unterschiedlichen Stufen (vgl. Abschnitt 4.1.1.6) mit stark unterschiedlichem fachlich-inhaltlichem und technischem Fokus stattfindet, steht zur Diskussion, welche Testfälle auf welchen Teststufen automatisch generiert werden können. Die wissenschaftliche Literatur zum Themas befasst sich seit vielen Jahren mit der Automatisierung von Testaktivitäten (vgl. Abschnitt 2.4). Wenngleich technische Umsetzungen in vielen Bereichen der Softwaretechnik noch nicht alle theoretischen Möglichkeiten ausschöpfen, ist doch Konsens, dass die Generierung von Testfällen aus Artfakten des Systementwurfs oder Implementierungsartefakten auf unterschiedlichen Teststufen grundsätzlich möglich ist.

Der in dieser Arbeit vorgestellte Ansatz zur Generierung von Testfällen aus UML-Aktivitätsdiagrammen basiert auf der Annahme, dass Akzeptanztests für kontextsensitive, mobile Anwendungen aus Artefakten des Systementwurfs erzeugt werden können. Anforderungen an die Tragfähigkeit dieses Ansatzes ist, dass aus diesen Entwurfsdokumenten ausreichend viele Informationen über das zu testende System extrahiert werden können. Dies umfasst sowohl statische Information über das SUT (z. B. Anwendungsfälle, Navigationspfade durch die Anwendung) als auch dynamische Informationen, wie etwa Eingabedaten oder Eigenschaften mobiler Betriebsumgebungen (z. B. Standort des Anwenders oder Qualität und Verfügbarkeit einer Netzwerkverbindung). Unter der Voraussetzung der Existenz angemessen detaillierter Spezifikationsdokumente in Form von UML-Aktivitätsdiagrammen aus denen mögliche Pfade durch die Anwendung im Charakter definierter Anwendungsfälle hervorgehen, sind lediglich Informationen zur Einbettung des Anwendungsfalls in ein Anwendungsszenario unter Berücksichtigung relevanter Kontextparameter (vgl. Abschnitt 3.1.2) in diesen Modellen zu ergänzen.

Im Fall mobiler Anwendungen, die häufig in kurzen Zyklen in agilen Vorgehensmodellen (z. B. *Scrum*) auf die Anforderungen eines schnellen Marktes angepasst werden (vgl. Dantas et al. [87], Haller [164]), kann die Existenz angemessen detaillierter Entwurfsdokumente nicht immer vorausgesetzt werden. Doch auch in diesem Fall ist es zweckmäßiger komplexe Anwendungsfälle in UML-Aktivitätsdiagrammen zu modellieren und diese Modelle zur Testfallgenerierung zu verwenden, als Testfälle vollständig manuell zu erstellen. Modelle sind geeignete Werkzeuge, auch komplexe Zusammenhänge so abzubilden, dass Softwarekomplexität auch für menschliche Akteure beherrschbar bleibt. Modelle von Software werden gerade deshalb eingesetzt, weil sie ein Mittel sind, auch komplexe Zusammenhänge für menschliche Akteure greifbar zu machen, wobei der wesentliche Inhalt zwar vom technischen Detail abstrahiert, Anwendungsfälle jedoch in ausreichendem fachlich-inhaltlichen Detail erhalten bleiben, um als solide Basis für die Erstellung von Akzeptanztests verwendet zu werden. Enthielten Softwaremodelle diese Information nicht, so wären sie nicht nur ungeeignet zur Testfallerstellung, sondern ebenfalls nicht geeignet zur Softwarespezifikation.

Die Erstellung von Testfällen ist deshalb eine Aktivität, die signifikant von der Verwendung von Automatisierungstechnologien profitiert. Es steht weniger die Einsparung von Arbeitszeit zur Testfallerstellung im Fokus, sondern vielmehr qualitative Aspekte. Selbst kleine Softwaresysteme (hinsichtlich gängiger Softwaremetriken, z. B. wie etwa *Lines of Code*) erreichen aus der Sicht des Testens schnell hohe Komplexität, die aus der Mächtigkeit des Zustandsraumes von Eingabe- und Navigationsparametern und – bei mobilen, kontextsensitiven Anwendungen – insbesondere durch die Vielfalt relevanter Kontextparameter (vgl. Abschnitt 3.1.2) resultiert. Maschinelle Unterstützung kann (maschinenlesbare Datenformate vorausgesetzt) durch Verwendung geeigneter Technologien wie beispielsweise graphentheoretische Analysemethoden dabei helfen, auch komplexe Workflows durch eine Software unter Berücksichtigung kontextualer Rahmenbedingungen in Testfälle zu transformieren, die einerseits fachliche Inhalte angemessen repräsentieren und andererseits ausreichend vom technischen Detail abstrahieren.

Als Möglichkeit Testfälle für Software automatisch zu erzeugen kommen hierbei unterschiedliche Methoden in Frage. Beispielsweise können Testfälle aus Dokumenten des Systementwurf durch Anreicherung mit Testdaten und durch Anwendung von Modelltransformation erstellt werden. Ebenfalls ist es möglich, den Workflow durch eine Software einmalig bei einer

manuellen Testdurchführung aufzuzeichnen und diese Aufzeichnung durch Automatisierungstechnologie später weiterzuverwenden. Ebenfalls ist es möglich, durch explorative Ansätze bestimmte Artefakte (z. B. graphische Benutzungsoberflächen) zur Laufzeit von Werkzeugen analysieren zu lassen und hieraus Rückschlüsse sowohl auf Workflows als auch auf Testdaten zu ziehen. Insgesamt gehört die Generierung von Testfällen neben der eigentlichen Durchführung von Tests zu denjenigen Aktivitäten, die in besonderem Maße von Automatisierung profitieren. In der Informatik ist die Testfallgenerierung deshalb ein seit langem intensiv beforschtes Gebiet (vgl. Abschnitt 2.4). Methoden zur automatisierten Testfallgenerierung werden in Abschnitt 4.3.2.1 im Detail diskutiert.

4.3.1.2 Generierung von Testdaten

Beim Testen von Softwaresystemen ist es regelmäßig notwendig, das SUT in simulierten Anwenderszenarien mit Eingabedaten zu versorgen. Da es sich beim Testen nicht um einen bestimmungsgemäßen produktiven Einsatz des Systems handelt, werden an verwendete Testdaten besondere Ansprüche gestellt. Einerseits sollen Testdaten den durchschnittlichen (oder typischen) Anwendungsfall angemessen repräsentieren, andererseits ist es ebenfalls wichtig, das SUT auch mit solchen Eingabedaten zu stimulieren, die gerade nicht den typischen Anwendungsfall repräsentieren. Hierzu zählen beispielsweise solche Daten, die sich den Rand- oder Grenzbereichenbereichen der Gültigkeit (fachlich oder technisch) stark annähern oder diese sogar überschreiten (Randbereichstests). Die Erstellung von Testdaten ist deshalb eine anspruchsvolle Tätigkeit, die je nach adressierter Teststufe von Bedarfsträgern mit unterschiedlicher Profilierung wahrgenommen wird. Neben technischen und fachlichen Eigenschaften des SUT sind hierbei unter Umständen auch juristische Restriktionen zu beachten.

Für einige Anwendungsfälle ist es technisch und fachlich möglich, Testdaten automatisch zu generieren. Ein üblicher Ansatz ist beispielsweise, Testdaten anhand der Systemspezifikation unter Verwendung probabilistischer Modelle zufällig auszuwählen, so dass sowohl typische Anwendungsfälle abgedeckt werden als auch solche, die das SUT gezielt an den Grenzbereichen der Datengültigkeit stimulieren. Ebenfalls üblich ist es, existierende Datenbestände unter Beachtung gesetzlicher Datenschutzbestimmungen (z. B. durch Verfremdung von Personendaten) zum Testen zu verwenden.

Ziel der Automatisierung der Generierung von Testdaten ist es vorrangig, die zum manuellen Erstellen der Testdaten notwenige Arbeitszeit zu reduzieren. Auch hier steht zur Diskussion, ob die Komplexität von menschlichen Akteuren beherrschbar ist, so dass hinsichtlich der Testfallqualität eine zufriedenstellende (d. h. eine dem Risiko eines Softwaredefekts und den daraus resultierenden Folgen angemessene) Lösung erreichbar ist. Hier kann ebenfalls Automatisierungstechnologie zum Einsatz kommen, um menschliche Akteure zu unterstützen. Für Informationssysteme deren Funktionalität im Wesentlichen auf die Ein- und Ausgabe von Daten und deren Speicherung und Verwendung zur Erfüllung spezifischer fachlicher Aufgaben beschränkt ist (z. B. Informationssysteme bei Versicherungs- oder Finanzdienstleistern), ist eine automatisierte Testdatengenerierung erfolgversprechend, da Werte- und Gültigkeitsbereiche von Eingabedaten bekannt sind.

Im Umfeld kontextsensitiver mobiler Anwendungen hingegen ist die Erstellung von Testdaten weitaus aufwendiger. Die Anforderungen an die Testdatenerstellung für nicht-mobile

Anwendungen bestehen hier weiterhin. Es kommen allerdings zusätzliche Anforderungen hinzu, wie etwa die Erstellung repräsentativer Daten für relevante Kontextparameter (vgl. Abschnitt 3.1.2). Hier entsteht das Problem, dass zwar für einzelne Kontextparameter die Wert- und Gültigkeitsbereiche durchaus bekannt sind (z. B. Ortsangaben im WGS84 Format), das komplexe Zusammenspiel unterschiedlichster Kontextparameter in mobilen Szenarien aber schnell unüberschaubar werden (z. B. Roamingszenarien bei gleichzeitiger Überquerung der Zeitzonengrenze in Bereiche unterschiedlicher Gültigkeit von Sommer- und Winterzeit).

Aus diesem Grund ist es zweckmäßig, auch für kontextsensitive, mobile Anwendungen Testdaten unter Verwendung von Automatisierungstechnologie zu erstellen. Gerade für diese Art Software ist die Komplexität möglicher Eingabeparameter für einen menschlichen Akteur schnell unüberschaubar, so dass eine vollständige Abbildung des Testraums nicht möglich ist. Aktuell müssen Testdaten basierend auf Annahmen über mögliche zukünftige Betriebsumgebungen und Erfahrungswerten gemutmaßt werden. Notwendige Voraussetzung für eine sinnvolle Generierung von Testdaten für kontextsensitive mobile Anwendungen sind nach aktuellem Stand der Technik nicht erfüllt. Aus diesem Grund gilt im Fokus dieser Arbeit die Generierung von Testdaten unter Berücksichtigung von Kontextparametern als nicht automatisierbar. Methoden zur Automatisierung der Testdatengenerierung werden im Rahmen dieser Arbeit nicht weiter betrachtet.

4.3.1.3 Testdurchführung

Während die Testfallerstellung aus Gründen der Komplexitätsbeherrschung ein Risiko für den wirtschaftlichen Erfolg des Softwareprozesses ist, ist die Testdurchführung aufgrund des verhältnismäßig hohen manuellen Aufwands (vgl. Myers et al. [261], Tassey [341]) gegenüber anderen Aktivitäten des Softwareprozesses ein wirtschaftlich bedeutsamer Faktor. Auf unterschiedlichen Teststufen müssen zur Qualitätssicherung viele Testaktivitäten von unterschiedlichen Bedarfsträgern häufig und wiederkehrend ausgeführt werden. Oftmals handelt es sich hierbei um mechanische Tätigkeiten, deren intellektueller Anspruch über korrekte Befolgung von Testanweisungen und die Dokumentation der Ergebnisse nicht hinaus geht. Dennoch ist die Testdurchführung ein kritischer Teil des Softwareprozesses, der wenig Raum für Unachtsamkeit lässt, da ein fehlerhaftes Softwareprodukt ein Risiko für den Projekterfolg ist.

Da viele Testaktivitäten häufig (und oftmals unverändert) wiederkehrend ausgeführt werden, kann die Ablösung menschlicher Akteure unter mehreren Perspektiven vorteilhaft sein. Einerseits kann so vergleichsweise teure menschliche Arbeitskraft eingespart werden und somit das Projektbudget entlastet werden (vgl. Hoffman [183], Ramler und Wolfmaier [291], Harrold [170], Meszaros et al. [246]). Andererseits können menschliche Fehler – die bei einer wiederkehrenden monotonen Arbeit zu erwarten sind – ausgeschlossen werden. Zudem ergibt sich der Vorteil, dass die Durchführung umfangreicher Testsuites auf Zeiten verlagert werden kann, die überlicherweise außerhalb normaler Arbeitszeiten liegen, so dass das Voranschreiten des Softwareprozesses nur um ein Mindestmaß durch Testaktivitäten beeinträchtigt wird.

Die Durchführung von Tests ist oberhalb der Teststufe Komponententest eine Simulation der tatsächlichen Verwendung eines Softwaresystems, einer Komponente oder eines Quellcode[artefakts] durch den zukünftigen Anwender. Es gilt den Workflow durch ein Softwaresystem durch geeignete Technologie abzubilden. Der rein technische Vorgang der Testdurchführung

beschränkt sich daher auf die Manipulation des SUT durch Werkzeugverwendung in derselben Weise wie es durch einen menschlichen Akteur geschehen würde. Eine solche Technologie wird in der Terminologie des Softwaretestens als *Driver* (vgl. Patton [275]) bezeichnet.

Aktuelle Technologien der Softwareentwicklung erlauben die Manipulation eines Softwaresystems unter Verwendung sogenannter Instrumentierung durch Softwarewerkzeuge anstelle eines menschlichen Anwenders. Der Begriff Instrumentierung bezeichnet hierbei einerseits die Anreicherung von Quellcode um Messtechniken, die eine spätere Analyse des Programmablaufs ermöglichen. Andererseits bezeichnet Instrumentierung ebenfalls die labormäßige Fernsteuerung einer Software durch andere Software anstelle des menschlichen Anwenders (vgl. IEEE [192], Liggesmeyer [236]). Dieses Vorgehen ermöglicht die automatisierte Durchführung von Softwaretests.

Eine Automatisierungstechnologie kann dann angewendet werden bzw. ein Softwaretest ist nach Hoffman [184] dann automatisierbar, wenn die Automatisierungstechnologie in der Lage ist, (i) mindestendes zwei Testfälle auszuführen, (ii) die Testausführung auf eine Untermenge aller Testfälle beschränkt werden kann, (iii) nach dem Starten der Testausführung keine Eingriffe in den Test notwenig sind, (iv) relevante Parameter der Testumgebung durch ein Werkzeug hergestellt oder simuliert werden können, (v) der Testfall automatisch ohne Beteiligung eines menschlichen Akteurs ausgeführt werden kann, (vi) Ist-Ergebnisse automatisch mit Soll-Ereignissen verglichen und Abweichungen aufgezeigt werden und (vii) ein Testreport generiert wird. Meszaros et al. [246] definieren in ihrem Manifest zur Testautomatisierung einen ähnlichen Satz an Anforderungen. Diese Kriterien werden an eine Automatisierungstechnologie gestellt um zu gewährleisten, dass ihre Verwendung einen tatsächlichen Mehrwert gegenüber der manuellen Testdurchführung generiert.

Voraussetzungen zur Verwendung von Automatisierungstechnologie ist, dass Testfälle auf der jeweilig adressierten Teststufe in geeigneten maschinenlesbaren (d. h. strukturiert, formal und in den meisten Fällen digital) Formaten repräsentiert werden können. Auf der Ebene von Komponententests ist diese Voraussetzung regelmäßig erfüllt, da die Testfallbeschreibung selbst in Quellcode erfolgt und deshalb dieselbe algorithmische Mächtigkeit besitzt wie die Implementierung des SUT. Häufig kommt hierbei zur Implementierung von Testfällen und dem SUT dieselbe Programmiersprache zum Einsatz und Testfälle teilen sich denselben Namens- und Adressraum. Innerhalb der im Testfall implementierten Logik kann deshalb auf Variablen und Methoden des konkret untersuchten Softwareartefakts zugegriffen werden. Für Teststufen höherer Ebenen (z. B. Systemtest, Akzeptanztest), die häufig als Black-Box-Tests durchgeführt werden, muss die Erfüllung dieser Voraussetzungen zur Automatisierung durch Verwendung weiterer Technologie hergestellt werden. In der Regel geschieht dies durch Verwendung von Rahmenwerken, die Tests von höheren Teststufen auf Tests auf Ebene von Komponententests zurückführen, wobei auf plattformnative Instrumentierungstechnologie zurückgegriffen wird. Diesen Ansatz verfolgen beispielsweise die Technologien UIAutomator (vgl. Abschnitt 2.5.2.3), Robotium (vgl. Abschnitt 2.5.2.1) und insbesondere Calabash (vgl. Abschnitt 2.5.2.6), das in dieser Dissertation als Basistechnologie für die kontextsensitive Testautomatisierung verwendet wird.

Insgesamt ist die Testdurchführung aufgrund ihres imperativen Charakters von allen Aktivitäten des Testens am besten automatisierbar. Nachdem Testfälle einmal erstellt wurden,

müssen diese lediglich in einem maschinenlesbaren Format dokumentiert werden, so dass die Instrumentierung verwendet werden kann, um das SUT in testrelevanten Aspekten zu steuern und zu beobachten. Insbesondere für kontextsensitive mobile Anwendungen kommt hier noch die Anforderung hinzu, auch kontextuelle Parameter der Betriebsumgebung durch Instrumentierungstechnologie steuern zu können. Es ist technisch nicht möglich, Parameter der zukünftigen Betriebsumgebung einer Software direkt zu manipulieren. Aber es ist technisch durchaus möglich, im Rahmen der Testdurchführung eine künstliche Betriebsumgebung zu erzeugen, die den Spezifikationen des konkreten Testfalls genügt. Gegebenenfalls muss hierzu in die Art, wie ein SUT Eigenschaften seiner Betriebsumgebung misst, eingegriffen werden. Dies kann beispielsweise durch Simulation und Manipulation von Sensormesswerten erfolgen. Dieser Ansatz wird in dieser Dissertation verfolgt. Zur Integration von Kontextparametern in Tests wird unter Ausnutzung technischer Möglichkeiten der Instrumentierungstechnologie der Zielplattform eine künstliche Betriebsumgebung simuliert, die in ihren Eigenschaften der realen Betriebsumgebung so ähnlich ist, dass vom Verhalten des SUT in der Testumgebung auf das Verhalten des SUT im realen Einsatz geschlossen werden kann.

4.3.1.4 Analyse von Testergebnissen

Wesentlicher Aspekt des Testens von Software ist die Analyse von Ergebnissen, also konkret der Vergleich der tatsächlich durch das SUT erzeugten Ergebnisse mit den durch die Spezifikation geforderten Ergebnissen. Anforderung an eine Testautomatisierungslösung ist es deshalb, auch diesen Vergleich von Soll- und Ist-Ergebnis ohne Beteiligung eines menschlichen Akteurs durchführen zu können. Die Analyse von Testergebnissen erstreckt sich jedoch nicht nur auf diese Aspekte. Nicht-funktionale Testarten produzieren Ergebnistypen, die anders quantifiziert und qualifiziert werden, als der bloße Abgleich von Soll- und Ist-Ergebnis.

Smoke-Tests beispielsweise erzeugen i. d. R. keine Ergebnisse, die durch einen Vergleich mehrerer Werte interpretiert werden können. Ihre Aufgabe ist es zu überprüfen, ob eine Anwendung unter bestimmten Umständen unerwartet terminiert. Eine solche Überprüfung kann i. d. R. nicht durch bloße Inspektion des funktionalen Verhaltens erfolgen. Vielmehr müssen Ausgaben und Meldung der Laufzeitumgebung des SUT, z. B. der *Java Virtual Machine*, der *Dalvik Virtual Machine* oder der *Ahead-of-Time-Compiler Android Runtime* (ART), auf Fehlermeldungen analysiert werden. Insbesondere im Umfeld mobiler Plattformen, auf denen eine App zu jeder Zeit vom Betriebssystem suspendiert werden kann, ist es nicht immer trivial zu entscheiden, ob eine App oder eine ihrer Komponenten unerwartet terminiert oder ob die App tatsächlich vom Betriebssystem in einen Ruhezustand versetzt wurde. Beide Fälle sind anhand des UI nicht trivial unterscheidbar. Bei App-Komponenten, die kein eigenes UI anbieten, wird dieses Problem noch verschärft.

Bei Load Tests wird versucht, eine Anwendung so lang unter Last zu setzen, bis nichtfunktionale Anforderungen verletzt oder bestätigt werden. Bei einer Web-Anwendung kann beispielsweise untersucht werden, wie viele Aufrufe zeitgleich erfolgen dürfen, bevor die Antwortzeit einen kritischen Wert überschreitet. Diese Problematik ist auch im Umfeld mobiler Anwendungen relevant, wenn diese in mehreren Threads Informationen von einem Web-Service herunterladen. Die Ergebnisse von Load Tests können i. d. R. ebenfalls nicht direkt am SUT gemessen werden, sondern erfordern zusätzliche Infrastruktur oder Messwerkzeuge.

Im Fokus dieser Dissertation stehen funktionale Tests, bei welchen der Kern der Testaktivität der Vergleich von Soll- und Ist-Ergebnis ist. Hierzu wird das SUT durch Simulation von UI- und Kontextereignissen manipuliert, wie es ein tatsächlicher Nutzer durch Interaktion mit dem UI der Anwendung oder mit dem Gerät tun würde. Hierbei spielt neben der Stimulation des SUT ebenfalls die Beobachtung und Bewertung des Verhaltens des SUT eine herausgehobene Rolle.

Hoffman [183, 184] argumentiert, dass der Zweck automatisierter Tests u. a. die Reduzierung des manuellen Arbeitsaufwands ist. Automatisierungswerkzeuge können in kurzer Zeit hunderte oder tausende von Tests ausführen und dabei eine große Menge von Ergebnisdaten produzieren. Es ist nicht sinnvoll, den manuellen Aufwand zur Testdurchführung durch eine Technologie zu reduzieren, die ihrerseits manuellen Aufwand zur Auswertung der Ergebnisse produziert. Für große Mengen von Testergebnissen gilt beispielsweise, dass neben syntaktischen auch semantische Aspekte bei der Bewertung der Korrektheit einer Software zu beachten sind. Gegenstand eines Softwaretests könnte beispielsweise eine Funktion sein, die als Ergebnis eine Liste gleichartiger Daten erzeugt, die nicht nur individuell auf korrekten Inhalt, sondern auch auf eine korrekte Reihenfolge (z. B. bei der Anzeige in einem UI) geprüft werden müssen[8]. Eine einfache Aufgabe, wenn die Liste nur wenige Einträge enthält, jedoch für einen menschlichen Tester sehr aufwändig zu lösen, wenn die Software hundert oder mehr Listeneinträge erzeugt, die geprüft werden müssen, jedoch immer nur ausschnittsweise auf dem UI angezeigt werden. Analog ist es für einen menschlichen Akteur nur schwer zu überprüfen, wenn eine solche Reihenfolge durch ein abstraktes, dem menschlichen Bewusstsein nicht intuitives Sortierkriterium bestimmt wird. Die Anwendung AR Tourist Information zeigt beispielsweise nahegelegene POIs in einer Listendarstellung an. Die korrekte Reihenfolge dieser Liste ist abhängig vom Standort des Anwenders, so dass bei der Analyse der Testergebnisse ein zusätzliches Kriterium zu beachten ist. Weiter verschärft wird die Problematik, wenn neben der Entfernung des Anwenders vom jeweiligen POI zusätzlich eine Priorisierung von POIs die Sortierreihenfolge bestimmt. Weiterhin können Funktionen von Software Gegenstand von Tests sein, die keine menschenlesbaren Ausgabe erzeugen, z. B. *Octet-Streams* (geordnete Sequenz von Bytes, deren Semantik erst durch Metainformationen bestimmt wird), die menschliche Akteure weder auf syntaktische noch semantische Korrektheit überprüfen können. [183, 184]

Es ist daher eine Kernanforderung an Testautomatisierungstechnologie, dass neben der Testausführung ebenfalls die Analyse von Ergebnissen automatisiert, d. h. ohne manuelle Intervention, erfolgt. Im Normalfall kann für ausgewählte Aspekte des Testens diese Analyse maschinell sogar besser erfolgen als manuell. Beispielsweise können neben fachlichen Inhalten auch sonstige Eigenschaften von UIs maschinell mit einer Genauigkeit geprüft werden, die für einen menschlichen Akteur nur schwer zu erreichen ist, z. B. Farbwerte oder pixelgenaue Dimensionen von UI-Elementen. Nach der Testdurchführung ist deshalb die Analyse von Ergebnissen eine Aktivität, die technisch nicht nur besonders gut automatisierbar ist, sondern bei der durch Automatisierung ein besonderer Mehrwert entsteht.

[8] Die mobile Anwendung ERGO App [109] beispielsweise implementiert eine Funktion, die eine Liste der sich in der geographischen Nähe der Ausführung befindlichen Außendienstbüros des Unternehmens anzeigt. Zu testen wäre diese Funktion an unterschiedlichen Standorten und jeweils der korrekte Inhalt und die korrekte Sortierung der Suchergebnisse.

4.3.2 Lösungsansätze zur Testautomatisierung

Wie Testautomatisierung technisch umgesetzt werden kann, wird bestimmt vom konkreten Gegenstand der Automatisierung, d. h. Generierung von Testfällen oder Testdurchführung und Analyse der Ergebnisse. In der Vergangenheit waren bereits unterschiedliche Herangehensweisen an die Testautomatisierung Gegenstand der Informatik. Insbesondere die Testfallgenerierung ist nicht nur eine technische, sondern auch eine konzeptuelle Herausforderung.

In den folgenden Abschnitten werden Eigenschaften und Unterschiede unterschiedlicher Lösungsansätze zur Generierung von Testfällen diskutiert und in den Zusammenhang dieser Dissertation eingebettet.

4.3.2.1 Generierung von Testfällen

Für die Generierung von Softwaretests haben sich zahlreiche methodische Ansätze herausgebildet, die grob in modellbasierte Technologien und solche, die Tests zur späteren Wiederverwendung aufzeichnen, klassifiziert werden können. Ziel beider Technologien ist es, den Anteil menschlicher Arbeitskraft bei der Erstellung von Tests zu reduzieren, gleichzeitig jedoch die Anzahl, Qualität und Abdeckung von Tests zu erhöhen.

In den folgenden Abschnitten werden die beiden methodischen Ansätze modellbasierte Testfallgenerierung und *Capture and Replay* (C'n'R) (engl. Aufzeichen und Wiederabspielen) diskutiert und gegenübergestellt. Insbesondere wird herausgestellt, warum in dieser Dissertation ein modellbasierter Ansatz verfolgt wird und C'n'R-Ansätze im Umfeld mobiler, kontextsensitiver Anwendungen nur bedingt geeignet sind.

4.3.2.1.1 Modellbasierte Methoden

Modellbasierte Lösungsansätze verschieben den Anteil menschlicher Tätigkeiten bei der Erstellung von Tests von der konkreten Testerstellung, d. h. Programmierung von Unit-Tests oder der schriftlichen Dokumentation von Tests, auf das Erstellen von Modellen. Zwar erfordert auch das Erstellen geeigneter Modelle manuelle Arbeitsleistung, doch i. d. R. können aus einem Modell eine Reihe von Tests abgeleitet werden, so dass der Aufwand für die Modellierung ingesamt geringer ausfällt als für eine manuelle Testerstellung. Zudem verwenden viele Ansätze des modellbasierten Testens die Modelle des Systementwurfs, so dass es nicht notwendig ist, für die Testerstellung völlig neue Modelle zu entwerfen.

Testen von Softwaresystemen ist eine Aktivität, die untersucht, ob die in der Systemspezifikation dokumentierten Anforderungen durch das SUT erfüllt werden. Auch Systemmodelle sind Artefakte, die auf den Ergebnissen der Anforderungsanalyse basieren. Sowohl strukturelle Aspekte des Systems als auch dessen Verhalten sind somit im Modell dokumentiert. Beim modellbasierten Testen, auch als *Model Driven Testing* (MDT) bezeichnet, werden Tests aus Modellen erstellt, die entweder eigens für diesen Zweck erstellt werden oder es werden Modelle des Systementwurfs wiederverwendet (vgl. Utting et al. [347], Baker et al. [23]).

Nach Heckel und Lohmann [173] umfasst das MDT die Automatisierung der drei wesentlichen Aufgaben: (1) Generierung von Testfällen aus Modellen nach gegebenen Kriterien, (2) die Generierung eines Orakels zur Bestimmung erwarteter Resultate von Tests sowie (3) die Ausführung von Tests in entsprechenden Testumgebungen. So soll gewährleistet werden, dass

die Spezifikation von Tests früh in den Entwicklungsprozess integriert wird, um zu verhindern, dass bei einer nachträglichen Testerstellung Teile des Systems ausgelassen werden.

Während sich eine Vielzahl individueller Lösungskonzepte herausgebildet haben, die unterschiedliche Zieltechnologien in variierenden Domänen zum Gegenstand haben, entspricht das prinzipielle Vorgehen vom Grundsatz her dem in Abbildung 4.5 dargestellten Schema. Ein gemeinsames Systemmodell bildet die Grundlage sowohl für die Systemimplementierung als auch für die Testerstellung. Testmodelle werden hierzu aus den Entwurfsmodellen des Systems abgeleitet. Das geschieht durch Auswahl einer Untermenge von Teilmodellen und durch Ergänzung des Systemmodells um testrelevante Informationen. Sowohl das Systemmodell als auch das Testmodell liegen zunächst in einer plattformunabhängigen Darstellung, *Platform Independent Model* (PIM) sowie *Platform Independent Test* (PIT), vor. Durch Modelltransformation werden diese Modelle im Anschluss zu plattformspezifischen Modellen, *Platform Specific Model* (PSM) sowie *Platform Specific Test* (PST), verfeinert. Hierbei findet i. d. R. eine Anreicherung um plattformspezifische Informationen statt. Im Anschluss daran erfolgt eine weitere Modelltransformation zu Erzeugung von Code. Links ist in Abbildung 4.5 grob vereinfacht der Prozess des *Model Driven Software Development* (MDSD) dargestellt, bei welchem angestrebt wird, das gesamte Softwaresystem aus Modellen zu generieren. Methoden des MDSD liegen nicht im Fokus dieser Dissertation und werden im Weiteren nicht mehr betrachtet. Rechts abgebildet ist der Prozess des MDT, bei dem Tests für Software aus Modellen generiert werden. Dieser Prozess wird in dieser Dissertation auf die Domäne der mobilen, kontextsensitiven Anwendungen angewendet.

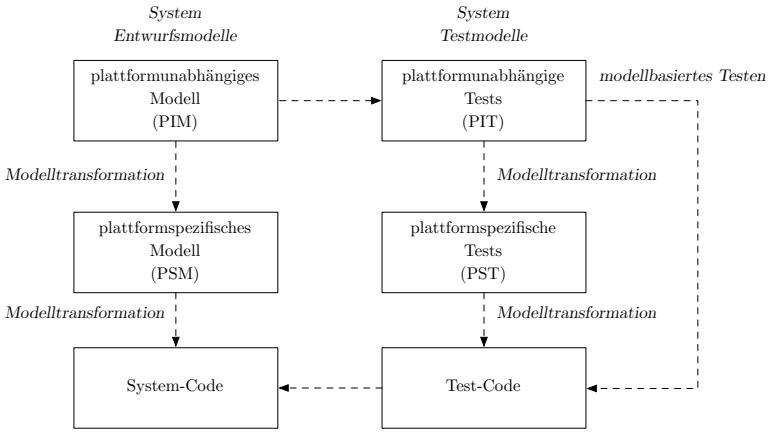


Abbildung 4.5: Schema des modellbasierten Testens in Anlehnung an [375]. Durch Modelltransformation werden aus einem Systemmodell zunächst plattformunabhängige Testartefakte, z. B. Testmodelle, erstellt und diese in einem zweiten Schritt zu plattformspezifischen Tests weiterverarbeiten.

Der Prozess des modellbasierten Testens wurde durch zahlreiche Forschungsarbeiten bereits in unterschiedlichen Technologievarianten realisiert. Im Umfeld kommunikationsbasierter Systeme hat sich beispielsweise die *Testing and Test Control Notation* (TTCN-3) etabliert. Hierbei handelt es sich um eine DSL zur Spezifikation von Tests für Kommunikationsprotokolle, insb. in eingebetteten Systemen (vgl. Utting et al. [347], Zander et al. [375]). Eine weitere Technologie im Umfeld des MDT ist das von der *Object Management Group* (OMG)

vorgestellte *UML 2.0 Testing Profile* (U2TP). Während die UML die Modellierung von Softwaresystemen unter zahlreichen Perspektiven ermöglicht, ist Testen bislang ein Bereich, für den die UML bislang kein dediziertes Metamodell oder Diagrammarten bereitstellt. Um diese Lücke zu schließen wurde U2TP entwickelt, um die Verwendung von Modellen für die Spezifikation und Durchführung von Tests innerhalb der Entwurfsparadigmen der MDA zu ermöglichen. Allerdings handelt es sich beim U2TP um einen stark abstrahierenden Ansatz, der für das hier untersuchte Konzept der Testautomatisierung mobiler Anwendungen als nicht geeignet bewertet wird (vgl. Abschnitt 2.4).

Im Jahr 2002 untersuchten beispielsweise Cavarra et al. [74] eine Konzept zur Generierung von Tests, bei dem ähnlich dem in dieser Dissertation untersuchten Konzept ein UML-Profil verwendet wird, um Modelle des Systementwurfs um testrelevante Informationen zu ergänzen, allerdings mit dem Unterschied, dass Cavarra et al. mit UML-Klassen- und Objektdiagrammen sowie UML-Zustandsdiagrammen andere Eingabemodelle verwenden und Software außerhalb der Domäne der mobilen Anwendungen adressieren. In vom Cavarra et al. untersuchten Konzept zur Testgenerierung werden Modelle durch Anwendung des UML-Profils angereichert und dann durch eine Modelltransformation vor der Generierung technologiespezifischer Tests in ein Zwischenformat überführt. Dieses dient als Austauschformat und ermöglicht es, Testmodelle an weitere Werkzeuge zu übertragen, mit denen die eigentliche Generierung technologiespezifischer Tests erfolgt. Das von Cavarra et al. untersuchte Konzept endet allerdings am Austauschformat. Zur Weiterverarbeitung wird auf andere Werkzeuge verwiesen. Aufgrund der geringen problemspezifischen Orientierung und der Verwendung von UML-ZustandsDiagrammen anstelle von UML-Aktivitätsdiagrammen kann dieser Ansatz jedoch nicht auf die hier untersuchte Domäne angewendet werden.

Einen ähnlichen Ansatz untersuchen Rutherford und Wolf [309] im Jahr 2003 für die Domäne der Web-Anwendungen. Auch hier werden UML-Modelle verwendet, um strukturelle und dynamische Aspekte des Systems zu modellieren. Aus diesen wird unter Hinzuziehung von Datenobjekten ein Testmodell erzeugt. Diese Datenobjekte werden ähnlich dem in dieser Dissertation untersuchten Ansatz ebenfalls als UML-Modell, konkret als UML-Objektdiagramm, modelliert. Allerdings bieten die Autoren im Unterschied zum hier untersuchten Konzept kein domänenspezifisches Metamodell zur Modellierung von Testdaten an. Eine Modelltransformation zur Generierung automatisierbarer Tests stünde hier vor der Herausforderung, die Struktur und Semantik der Testdaten zu analysieren. Bei einer Übertragung dieses Ansatzes auf die hier untersuchte Testautomatisierung für kontextsensitive Anwendung stellt sich hier das Problem, dass beispielsweise Sensordaten unterschiedlicher Sensoren strukturell identisch sind. Ohne eine gesonderte Auszeichnung auf einer Metaebene könnten diese bei der Testausführung dann nicht mehr zugeordnet werden.

Linzhang et al. [239] untersuchten im Jahr 2004 einen Ansatz zur modellbasierten Generierung von Tests unter Verwendung von UML-Aktivitätsdiagrammen. Unter der Voraussetzung Tests aus der Perspektive des Anwenders zu erzeugen, entwickeln die Autoren ein Werkzeug, das aus einem UML-Aktivitätsdiagramm eine Menge von Testfällen erzeugt, die alle Pfade durch den Graphen des Aktivitätsdiagramms abdeckt. Ein ähnliches Vorgehen wird ebenfalls in dieser Dissertation untersucht, allerdings unter Berücksichtigung von Testdaten, die durch ein UML-Profil in das Systemmodell eingebracht werden.

Neben UML-basierten Ansätzen existieren ebenfalls solche, die eine eigene DSL anstelle der UML verwenden, um Tests zu modellieren. Im Jahr 2008 untersuchten Im et al. [197] einen solchen Ansatz, bei dem Tests mit einer eigens entwickelten Sprache modelliert werden. Der Vorteil der Verwendung dieser Modellierungssprache gegenüber einer direkten Implementierung von Tests im Code ergibt sich hierbei daraus, dass diese Modellierungssprache einem Tester ohne Programmierkenntnisse leichter zugänglich ist. Ein direkter Vorteil aus der Verwendung einer spezifischen DSL gegenüber der Verwendung der UML ist hingegen nur insofern erkennbar, als dass die UML eine generische, domänenunabhängige Modellierungssprache ist, deren Komplexität ohne einen spezifischen Wissenerwerb nur schwer beherrschbar ist. Das in dieser Dissertation untersuchte Konzept zur Testautomatisierung verwendet ein Mischmodell, in dem UML-Aktivitätsdiagramme zunächst manuell mit testrelevanten Informationen angereichert werden und anschließend in ein spezielles Testmodell (vgl. Abschnitt 5.2.3.2) überführt werden. Dieses erfüllt die Aufgabe eines Austauschformats zur anschließenden Generierung technologiespezifischer Tests, z. B. JUnit oder Calabash.

Ein weiterer Ansatz zur Testgenerierung wird von Gupta und Surve [163] im Jahr 2011 vorgestellt. Auch hier stehen Tests auf einer hohen Teststufe mit einem besonderen Fokus auf UI-Tests. Der von Gupta und Surve vorgestellte Ansatz ist insofern herauszuheben, als dass die Autoren in Abgrenzung von Systemmodellen der Softwaretechnik Geschäftsprozessesmodelle (BPMN) verwenden. Die Anreicherung des so erzeugten Testmodells erfolgt ebenfalls nicht durch Modellierung, sondern durch Kombination der modellbasierten Testgenerierung mit einer C'n'R-Technologie, bei der Testdaten bei einer initialen manuellen Ausführung des SUT in das Testmodell eingefügt werden.

Insgesamt ist die Generierung von Softwaretests aus Modellen ein intensiv erforschtes Feld der Informatik, das für viele Domänen bereits eine Vielzahl unterschiedlicher Lösungen hervorgebracht hat. Aufgrund der hervorgehobenen Stellung der UML im Softwareentwurf sind viele dieser Lösungen auf UML-Modelle fokussiert. Die Verwendung anderer Modellierungssprachen ist i. d. R. auf besondere Domänen beschränkt oder wird in Fällen bevorzugt, in denen die Mächtigkeit der verwendeten Modelle bewusst eingeschränkt werden soll.

Konzepte, die eine modellbasierte Technologie zur Generierung von Tests für mobile, kontextsensitive Anwendungen zum Gegenstand haben, sind im Jahr 2016 jedoch nur in geringem Umfang erforscht. In dieser Dissertation liegt der Fokus deshalb insbesondere auf der Generierung von Tests aus Modellen mobiler, kontextsensitiver Anwendungen. Hierzu ist es erforderlich, dass diese Modelle Kontexteigenschaften adäquat abbilden. Existierende Modelle tun dies i. d. R. nicht, weshalb für die Umsetzung des hier untersuchten Konzepts der Testautomatisierung für mobile Systeme nur limitiert auf existierende Lösungen zurückgegriffen werden kann.

4.3.2.1.2 Capture and Replay

Die Erstellung von Softwaretests kann in unterschiedlichem Grad automatisiert werden. In der Mitte des Spektrums zwischen vollständig manueller und vollständig automatisierter Testfallerstellung existieren teilautomatisierte Methoden, denen *Capture and Replay* (C'n'R) (engl. Aufzeichen und Wiederabspielen) zuzuordnen ist. Hierbei werden Tests, die in der Folge von einem Testautomatisierungswerkzeug verarbeitet werden sollen, zunächst manuell erstellt und

ausgeführt. Bei der Ausführung des Tests werden einzelne Schritte und verwendete Eingabedaten aufgezeichnet und für eine spätere automatisierte Wiederholung durch ein Werkzeug aufbereitet. Zur Erstellung von Testfällen ist bei diesem Ansatz manuelle Arbeitskraft notwendig, die Einsparungen durch Automatisierung der Testdurchführung bleiben jedoch mit Einschränkungen (z. B. bedingte Übertragbarkeit auf andere mobile Geräte) erhalten.

Der Grundgedanke bei C'n'R-Techniken ist es, eine initiale manuelle Testdurchführung vollständig aufzuzeichnen (d. h. Interaktionen des Anwenders, Eingaben in das SUT, Ausgaben des SUT usw.) und in einer maschinenlesbaren Form zu speichern. Grundlage zur Verwendung von C'n'R-Techniken ist die Möglichkeit, Interaktionen eines Anwenders auf einer Ebene aufzuzeichnen, die so in das SUT, die Plattform oder das Betriebssystem eingebettet ist, dass maschinell lesbare Daten entstehen. C'n'R-Werkzeuge wie z. B. *Android GUITAR* [325], *Abbot* [357] oder *GUI Crawler* [6] verwenden i. d. R. das sogenannte *Keyword-Action* Paradigma (vgl. Gomez et al. [126], Abschnitt 2.3.3), um UI-Interaktionsereignisse nach erfolgter Aufzeichnung wiederzugeben. Das *Keyword-Action* Paradigma abstrahiert von konkreten UI-Interaktionsereignissen wie beispielsweise einem Klick-Event spezifischer Dauer an einem bestimmten Koordinatenpaar auf dem Display eines spezifischen Geräts, um eine Darstellung auf einer höheren Ebene zu erreichen, beispielsweise Anklicken eines UI-Elements des Typs Schaltfläche (fachlich häufig als *Button* bezeichnet) mit einer bestimmten Textmarke (z. B. Button „Absenden"). Diese Abstraktion ist notwendig, weil UI-Ereignisse, die vom Anwender als atomar wahrgenommen werden, je nach Plattform und Implementierung tatsächlich aber durch eine Event-Sequenz realisiert sind. Abbildung 4.6 illustriert beispielsweise die Sequenz technischer UI-Events, wenn ein Anwender auf der Plattform Android die Schaltfläche „*Home*" anklickt, die ihn zum Startbildschirm der Bedienoberfläche des Betriebssystems zurückführt.

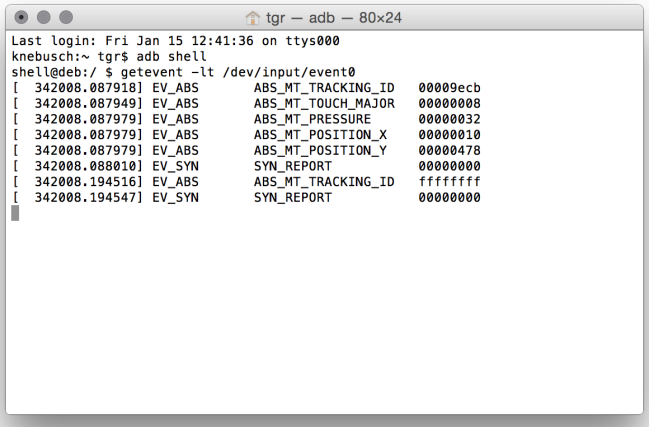


Abbildung 4.6: Sequenz atomarer UI-Events beim Anklicken des *Home*-Buttons der Plattform Android in der Version 6.0.1 Marshmallow, Build-Nummer MMB29K auf einem Nexus 7 Tablet-Computer, aufgezeichnet mit dem Werkzeug *Getevent* [134] des Android-SDK.

Zu erkennen sind in Abbildung 4.6 insgesamt acht technische UI-Events, jeweils mit Zeitstempel, Bezeichnung und einem Wert, dessen Bedeutung je nach Event definiert ist. Für ein Testskript, das auf demselben Gerät in einer anderen Bildschirmorientierung oder auf ande-

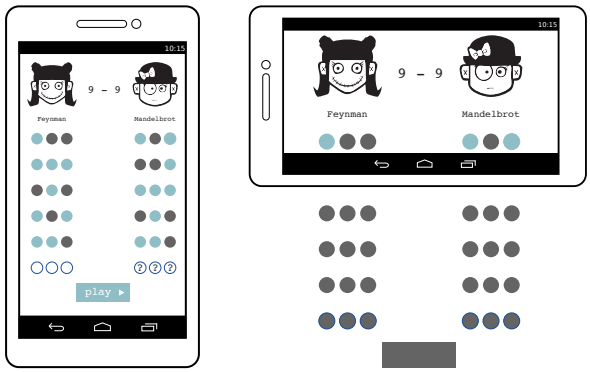
ren Geräten wiederverwendet werden soll, ist diese Darstellung zu detailliert, da beispielweise die Pixelkoordinaten des UI-Events für eine bestimmte Schaltfläche, z. B. dem Home-Button, mit der Displaygröße und -auflösung variieren. Deshalb abstrahieren C'n'R-Werkzeuge hier auf eine höhere Abstraktionsebene, um eine eingeschränkte Portierbarkeit eines solchen aufgezeichneten Tests zu ermöglichen. In der Regel bleiben hierbei allerdings Informationen zu Pixelkoordinaten erhalten, wodurch derselbe Test auf Geräten einer anderen Bauform i. d. R. nicht ohne Anpassung lauffähig ist.

Diese Portierbarkeit findet ihre Grenzen, wenn auf Geräten bestimmter Klassen (Smartphone vs. Tablet-Computer) derselbe Anwendungsfall durch ein völlig anderes UI abgebildet wird, um die UI-Darstellung an die Möglichkeiten und Einschränkungen bestimmter Geräteklassen[9] anzupassen. Weiterhin ist dieses Vorgehen ungeeignet, wenn ein UI-Element zwar grundsätzlich in der Hierarchie der UI-Elemente einer Benutzungsoberfläche existiert, aber nicht innerhalb des darstellbaren Bereichs angezeigt wird. Dieser Fall kann eintreten, wenn durch Unachtsamkeit bei der Entwicklung ein UI mit statischer Platzierung von UI-Elementen erstellt wurde, deren Koordinaten außerhalb des Bereichs eines Geräts mit kleinem Display liegen. In diesem Fall liegt ein UI-Defekt vor, der durch einen C'n'R-Ansatz der Testautomatisierung jedoch nicht entdeckt wird, weil das UI-Element zwar nicht im sichtbaren Bereich liegt, seine Funktion jedoch in der Theorie weiterhin erfüllt. Bei der automatisierten Wiederholung des Tests durch künstliche Injektion von UI-Events fällt dieser Defekt nicht auf. Einem menschlichen Akteur stellt sich die Situation hingegen anders dar, da dieser das fragliche UI-Element nicht erreichen kann und konsequent die damit verbundene Funktionalität nicht ausführen kann. Abbildung 4.7 illustriert diesen Sachverhalt anhand eines Beispiels einer App, die ähnlich dem Quizduell [113] ein Quizspiel realisiert[10]. Teilabbildung 4.7a bildet das UI der App im Hochformat ab, in welchem das gesamte UI in der verfügbaren Displayfläche angezeigt wird. Teilabbildung 4.7b hingegen zeigt die App im Querformat, wobei wesentliche Teile des UI außerhalb der darstellbaren Displayfläche liegen. Dies ist ein Defekt, der beim Testen entdeckt werden sollte. Wird jedoch ein C'n'R-Test der App mit der Variante in Teilabbildung 4.7a aufgezeichnet, entsteht ein Testskript, welches den Button mit der Textmarke „*play*“ als interaktionsfähiges UI-Element beinhaltet. Wird dieser Test nun auf einem Gerät im Querformat wie in Teilabbildung 4.7b ausgeführt, versucht die Testausführung erfolgreich diese Schaltfläche zu erreichen, da diese zwar nicht sichtbar ist, aber dennoch in der UI-Hierarchie enthalten ist. Ein menschlicher Tester hingegen könnte diese Schaltfläche nicht erreichen und würde an dieser Stelle einen Defekt des UI feststellen. Ein automatisierter Test im C'n'R-Verfahren entdeckt diesen Defekt hingegen nicht, weil das Testskript die Existenz und die Sichtbarkeit der Schaltfläche ungeprüft voraussetzt.

Sowohl manuelle als auch generative modellbasierte Ansätze sind hier überlegen, da sie entweder bewusst eine solche Sichtbarkeitsprüfung (manuell) beinhalten oder (generiert) bei geeigneter Parametrisierung des Testgenerators einen Testschritt zur Sichtbarkeitsprüfung der

[9] Die Plattform Android verwendet beispielsweise sogenannten *Fragments* (engl. Fragmente) [133], um komplexe UIs je nach Anforderungen der Geräteklasse (Smartphone oder Tablet-Computer) flexibel in individuelle Komponenten zu verfeinern.

[10] Im Rahmen der Lehrveranstaltung Mobile Software Engineering (MSE) des Lehrstuhls für Software Engineering, insb. mobile Systeme erarbeiten Studenten im Themengebiet Android-Entwicklung ein Praxisbeispiel. Im Sommersemester 2014 wurde hierbei ein Quizduell-Klon erstellt.



(a) Intaktes UI im Hochformat.

(b) Defektes UI im Querformat. Teile des UI liegen außerhalb der Displayfläche.

Abbildung 4.7: UI der Quiz-App Mobile Science Challenge im Hochformat (4.7a) und im Querformat (4.7b). Im Querformat liegt ein Defekt vor, der bei der Generierung von Tests durch C'n'R-Technologien konzeptuell nicht erkennbar ist.

Schaltfläche vorsehen. Der C'n'R-Ansatz kann beides nicht leisten, da bei der Aufzeichnung des Tests von einem intakten UI auf einer spezifischen Gerätekonfiguration ausgegangen wird.

C'n'R-Ansätzen ist gemeinsam, dass sie ausschließlich die Ausführung von Tests automatisieren. Die Erstellung von Tests obliegt weiterhin menschlichen Akteuren. Da Software schnell eine für einen menschlichen Akteur unüberschaubare Komplexität erreicht, entsteht das Risiko, dass wesentliche funktionale oder nicht-funktionale Anforderungen bzw. technische Besonderheiten bei der Testerstellung unberücksichtigt bleiben.

Mit einem Fokus auf UI-Tests untersuchten Kaasila et al. [209] den C'n'R-Ansatz *Testdroid* [45] für die Plattform Android (vgl. Abschnitt 2.3.3). Hier müssen Testfälle manuell erstellt werden, bevor durch eine automatisierte Wiederholung wirtschaftliche Effekte durch Einsparung weiterer manueller Aktivitäten bemerkbar werden. Allerdings adressieren Kaasila et al. [209] das Problem der Geräteheterogenität, indem atomare UI-Events (vgl. Abbildung 4.6) zu abstrakten UI-Events transformiert werden. Hierdurch entstehen bei der Aufzeichnung Testskripte, die technischen Details wie beispielsweise Pixelkoordinaten, abstrahieren. Diese können dann denselben Test auf Geräten mit unterschiedlicher Hardware ausführen.

Einen ähnlichen Ansatz verfolgten Liu et al. [240] (vgl. Abschnitt 2.3.3), die ebenfalls eine manuelle Testausführung zur Aufzeichnung eines Testskripts voraussetzen. Um testrelevante UI-Events zu identifizieren nehmen die Autoren Anpassungen am SUT vor, die die Einführung eines weiteren UI-Elements als Wurzelkonten aller in der App verwendeten UI-Elemente beinhalten. Dieses zeichnet alle UI-Events auf und gibt sie unverändert an die darunterliegenden Schichten der UI-Hierarchie weiter. Auch in diesem Ansatz wird Robotium verwendet, um UI-Events auf eine Ebene zu abstrahieren, die über unterschiedliche Geräte hinweg eine Wiederverwendung zulässt. Analog zum von Kaasila et al. untersuchten Ansatz ist diese Technologie jedoch ebenfalls auf UI-Events beschränkt, Kontextparameter finden keine Berücksichtigung, so dass Tests für kontextsensitive Anwendungen nicht möglich sind.

Eine herausgehobene Stellung im Umfeld der C'n'R-Technologien nimmt die von Gomez et al. [126] (vgl. Abschnitt 2.3.3) vorgestellte Technologie *RERAN* ein. Zwar handelt es sich

hier ebenfalls um einen C'n'R-Ansatz, der ein manuelles Erstellen von Tests erfordert, ist aber bei der wiederholten Ausführung von Tests in der Lage, neben Events des UI ebenfalls Sensor-Events zu berücksichtigen. Hierzu greifen die Autoren den Event-Strom nicht auf der Ebene des UI ab, sondern analysieren den Event-Strom auf der Ebene des Betriebssystem durch Verwendung des Werkzeugs *Getevent* [134] (vgl. Abbildung 4.6) des Android SDK, welches ein Event-Monitoring auf atomarer Ebene zulässt. In Abgrenzung zu den von Kaasila et al. [209] und Liu et al. [240] vorgestellten C'n'R-Technologien erfolgt jedoch keine Abstraktion, so dass eine Wiederverwendung auf anderen Geräten ausgeschlossen ist.

Zusammenfassend lässt sich feststellen, dass C'n'R-Technologien im Jahr 2016 nur in einem beschränkten Umfang zur effizienten Automatisierung von Tests für mobile Anwendungen beitragen. Prinzipiell zeichnen sie sich durch eine weiterhin manuelle Erstellung von Testfällen aus. Technisch ergibt sich hier zwar der Vorteil, dass Tester Tests nicht explizit programmieren müssen, sondern Testskripte implizit bei der Auszeichnung einer manuellen Testdurchführung erstellt werden. Die kognitive Leistung einer problemadäquaten Auswahl und Priorisierung von Testfällen obliegt jedoch weiterhin dem Tester. C'n'R-Technologien können deshalb nicht zu einer Verbesserung der Qualität von Tests, z. B. durch eine bessere Testabdeckung des SUT, beitragen, sondern automatisieren lediglich den Akt der Testdurchführung.

Ein weiterer Nachteil von C'n'R-Techniken ist, dass sie i. d. R. kaum Unterstützung für Kontextsensitivität in mobilen Anwendungen bieten. Hierzu müsste eine Technologie implementiert werden, die in der Lage ist, bei der Aufzeichnung von Tests ebenfalls Kontextparameter aufzuzeichnen und später bei der wiederholten Testausführung in das SUT zu injizieren. Zwar existieren vereinzelt Ansätze die auch Sensor-Events unterstützen, z. B. RERAN von Gomez et al., bei denen sich allerdings das Problem stellt, dass nicht sichergestellt werden kann, dass während der Testaufzeichnung der Kontext ungestört ist. Beispielsweise könnten C'n'R-Tests unbemerkt an einem Ort aufgezeichnet werden, an dem eine zuverlässige Standortbestimmung aufgrund externer Einflüsse (z. B. Urban Canyoning, vgl. Abschnitt 3.1.2.1.3) unmöglich ist. Ähnliches gilt für weitere physikalische Kontextparameter, z. B. das Erdmagnetfeld, das bei der Testaufzeichnung durch externe Einflüsse künstlicher Magnetfelder überlagert sein könnte.

Ein menschlicher Akteur ist unfähig zu erkennen, ob bei der Aufzeichnung von C'n'R-Tests ein verfälschter Kontext vorliegt, weil sich die meisten physikalischen Kontextparameter (vgl. Abschnitt 3.1.2.1) entweder vollständig der menschlichen Sinneswahrnehmung entziehen (z. B. magnetische Flussdichte) oder der Mensch nicht in der Lage ist, diese Kontextfaktoren mit einer quantifizierbaren Präzision wahrzunehmen[11]. Es ist deshalb nicht zu erwarten, dass mit C'n'R-Techniken aufgezeichnete Tests eine adäquate Abdeckung des SUT gewährleisten oder Kontextparameter mit Qualitätseigenschaften aufzeichnen, die den zukünftigen Anwendungsfall adäquat und realistisch repräsentieren. C'n'R-Tests, die unter solchen Umständen zustandekommen, bilden einen verfälschten Kontext ab und sind für das weitere Testen nicht verwendbar. Zu erwarten ist dann, dass in einem SUT vermeintliche Defekte korrigiert werden oder spezifikationsgemäß korrektes Verhalten durch falsche Rückschlüsse aus defekten Tests

[11] Das Gleichgewichtsorgan im Innenohr reagiert zwar selbst auf kleine Beschleunigungskräfte empfindlich, der Wahrnehmungsapparat ist aber nur fähig diese Kräfte abstrakt zu quantifizieren oder zeitlich versetzt miteinander zu vergleichen.

zerstört wird. Besonders schwerwiegend wirkt sich dieser Effekt in TDD-Umgebungen aus, in denen ein solcher defekter Test zwangsläufig zu einer inkorrekten Implementierung führt.

4.3.3 Testautomatisierungstechnologien für mobile Anwendungen

Gegenüber der manuellen Erstellung und Durchführung von Softwaretests bietet die Verwendung von Testautomatisierung im Kern zwei Optimierungspotenziale. Im Vergleich zur Testdurchführung ist die Erstellung von Tests die weniger häufig durchgeführte Aktivität. Optimierungspotenzial ergibt sich für die Testfallerstellung primär in qualitativer Hinsicht.

Tests werden weit häufiger ausgeführt als weitere Tests erstellt werden. Ein besonderes Potenzial menschliche Arbeitskraft einzusparen ergibt sich deshalb aus der Automatisierung der Durchführung von Tests. Für Quellcode[nahe] Tests existieren bereits Automatisierungslösungen, beispielsweise die Familie der *xUnit*-Technologien wie z. B. JUnit. Diese werden i. d. R. in derselben Programmiersprache implementiert wie das zu testende System. Die Testausführung selbst kann daher als die regelmäßige Ausführung des SUT oder seiner Komponenten unter Verwendung eines speziellen Einstiegspunkts verstanden werden.

Während auf der Ebene von Unit-Tests individuell adressierbare Codeartefakte getestet werden, haben Tests auf höherer Ebene, wie die in dieser Dissertation untersuchten funktionalen Akzeptanztests, das Gesamtsystem zum Gegenstand. Der Akt der Testdurchführung ist aus der Perspektive des SUT nicht von einer gewöhnlichen Benutzung des Systems unterscheidbar. Das Ziel der Systemausführung ist im Rahmen des Testens dann abweichend von der gewöhnlichen Benutzung nicht das Erfüllen einer fachlichen Ausführung, sondern die Beobachtung des Systems und die Bewertung seiner Spezifikationskonformität.

Lösungsansätze, den Prozess des funktionalen Testens zu automatisieren beinhalten deshalb i. d. R. die Verwendung einer Treibertechnologie, die sich dem SUT gegenüber so verhält wie ein menschlicher Akteur. Je nach Zielplattform haben sich eine Reihe von Technologien etabliert, die sich weniger in ihrer Funktionalität als in ihrer technischen Realisierung unterscheiden. Stellvertretend für den Anwender führen sie das SUT oder einzelne Komponenten des SUT aus, geben Testdaten in das SUT ein und vergleichen Ausgaben und Verhalten des SUT mit Sollwerten, die durch die Systemspezifikation vorgegeben werden.

Im Fokus dieser Dissertation stehen funktionale Tests, insbesondere Akzeptanztests, für mobile, kontextsensitive Anwendungen. Diese produzieren i. d. R. Ergebnisse, die direkt im SUT ermittelt werden können, beispielsweise durch Inspektion des UI des SUT nach einer Interaktion des Anwenders. Instrumentierungstechnologie ermöglicht es, diesen Schritt zu automatisieren. Die Analyse von Ergebnissen funktionaler Tests ist i. d. R. Kernfunktionalität von Testautomatisierungstechnologie. Sie setzt ein Test-Framework voraus, das es zulässt, das SUT während der Testausführung zu beobachten. Insbesondere auf mobilen Plattformen stellt sich hier das Problem, dass Apps i. d. R. durch Sandbox-Technologien vor fremden Zugriffen auf den eigenen Adressraum geschützt werden. Hierbei handelt es sich um eine Sicherheitstechnologie, die zum Ziel hat, Inhalte einer App vor unberechtigtem Zugriff zu schützen. Im Normalbetrieb ist es i. d. R. nicht wünschenswert, dass technische Voraussetzungen existieren, die es einem Prozess ermöglichen, Elemente des UI oder andere Schnittstellen einer App in einem anderen Prozess zu manipulieren, beispielsweise um den Anwender vor sogenannten *Phishing*-Angriffen bei der Eingabe sensibler Daten zu beschützen.

Mobile Plattformen implementieren besondere Sicherheitsvorkehrungen, um einen solchen Zugriff zu unterbinden. Die Plattform Apple iOS beispielsweise führt Apps in Sandboxes aus, die einen Zugriff auf den Speicherbereich anderer Apps verhindern. Auf der Plattform Android wird ein Sicherheitssystem direkt im Betriebssystemkern verankert, bei dem jede App in einem eigenen Prozess des Betriebssystems mit einer eigenen UNIX-Benutzer-Id ausgeführt wird und zusätzlich in einer privaten Instanz der ART, d. h. einer eigenen virtuellen Java-Laufzeitumgebung, ausgeführt wird.

Beim funktionalen Testen handelt es sich aus technischer Perspektive jedoch genau um den Fall, bei dem auf Inhalte des SUT durch eine dritte Instanz, ersatzweise für den tatsächlichen Anwender, zugegriffen wird. Für funktionale Tests auf der Ebene von Unit-Tests entstehen hier i. d. R. keine Probleme, weil diese als Bestandteil des SUT regelmäßig innerhalb des Prozesses des SUT ausgeführt werden. Bei der Automatisierung von Akzeptanztests werden hingegen Interaktionen des Anwenders maschinell simuliert. Diese Simulation kann nur effizient erfolgen, wenn sie ohne Verwendung besonderer Hardware direkt in Software realisiert wird. Das heißt beispielsweise für solche Tests, die Interaktionen eines Anwenders simulieren, dass UI-Elemente nicht direkt durch physikalische Interaktion mit dem Geräte erfolgen, sondern UI-Ereignisse künstlich in das SUT injiziert werden. Analog werden Ausgaben des SUT nur selten durch nicht-invasive Technologien wie optische Bildanalyse eines Bildschirmabzugs des SUT bewertet, sondern i. d. R. durch direkten Abgriff von Informationen am getesteten UI-Element. Beispielsweise muss ein Test zur Analyse von Testergebnissen in der Lage sein, direkt auf den Inhalt eines UI-Elements, z. B. eines Textfeld, zuzugreifen, um dessen Inhalt durch einen Textvergleich mit einem Soll-Zustand zu vergleichen. Um hier die Verwendung von Testautomatisierungtechnologien zu ermöglichen, bieten mobile Plattformen jeweils ein Test-Framework an, das eine solche Manipulation und Inspektion des SUT ermöglicht, ohne dass die Sicherheitsmechanismen der jeweiligen Plattform dies verhindern.

Die Plattform Android beispielsweise erzwingt hierzu die technische Realisierung der Testautomatisierung durch Implementierung von Tests in besonderen *Test Packages*, die gleichberechtigt neben dem SUT installiert werden und dessen Adressraum teilen (vgl. Abbildung 4.8). Hieraus folgt, dass für mobile Plattformen keine Testautomatisierungswerkzeuge existieren, die nicht auf der Testtechnologie der jeweiligen Plattform basieren, wodurch stets eine Abhängigkeit der Testautomatisierungstechnologie von der Zielplattform impliziert wird. Da Apps i. d. R. für mehrere Plattformen implementiert werden, wird der insgesamt durch Testautomatisierung erzielbare Einspareffekt durch die Plattformmultiplizität vermindert, da Automatisierungstechnologien nicht grundsätzlich plattformübergreifend verwendet werden können, wenngleich einige Technologien plattformübergreifendes Testen ermöglichen (z. B. Calabash [369], Xamarin [370], Appium [14]).

Die Plattformunabhängigkeit dieser Werkzeuge ergibt sich hierbei allerdings nicht inhärent aus der Technologie, sondern wird mit hohem Aufwand von den jeweiligen Betreibern von Web-basierten Dienstleistungen gewährleistet, die ergänzend zur jeweiligen Testautomatisierungstechnologie digitale Dienstleistungen anbieten, wie etwa Xamarin oder *AppThwack* [19]. Zu diesen Dienstleistungen gehört beispielsweise der Betrieb von Gerätefarmen, die zur Unterstützung von Tests auf einer heterogenen Gerätelandschaft eine Vielzahl unterschiedlicher Geräte vorhalten, die Testern im Teilzeitnutzungsverfahren angeboten werden. Der Tester

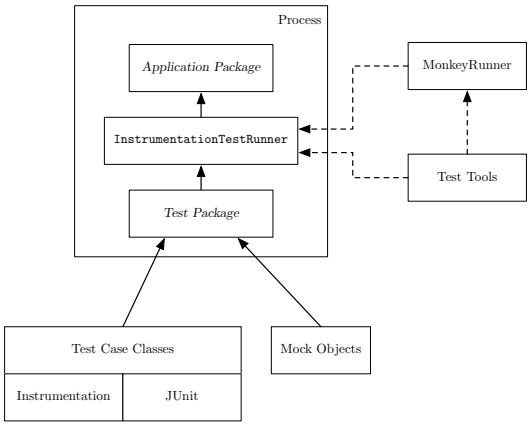


Abbildung 4.8: Architektur der Testautomatisierung der Plattform Android nach [145]

muss das SUT zum Zweck des Testens im Portal des jeweiligen Anbieters der Testfarm hochladen, wo dieses dann bei Verfügbarwerden der festgelegten Geräteressourcen ausgeführt wird. Hierbei handelt es sich um ein Geschäftsmodell, das auf der Bereitstellung einer plattformübergreifenden Testautomatisierungstechnologie basiert, die Kunden von der Notwendigkeit des Betriebs einer eigenen Gerätefarm befreit.

Darüber hinaus ist der Markt der Testautomatisierungstechnologien hinsichtlich der technischen Realisierung einzelner Werkzeuge oberhalb der Testschnittstellen der jeweiligen mobilen Plattform stark fragmentiert. Die Technologie Calabash beispielsweise ist sowohl für die Plattform Android (Calabash-Android [121]) als auch für Apple iOS (Calabash-iOS [122]) verfügbar. Beide verwenden zwar auf der Ebene der Testfalldefinition eine gemeinsame technologische Basis (vgl. Abschnitt 2.5.2.6), verwenden aber plattformspezifische Treibertechnologien. Das führt dazu, dass individuelle Testschritte, d. h. *Steps* in der Calabash-Terminologie, nicht beliebig zwischen den Plattformen austauschbar sind.

Die Relevanz dieser Betrachtung für diese Dissertation ergibt sich daraus, dass die im Jahr 2016 verfügbaren Testautomatisierungstechnologien über Standortinformationen hinaus keine Unterstützung für das Testen kontextsensitiver Anwendungen bieten, weil sie nicht in der Lage sind, Kontextparameter während der Testausführung zu simulieren (vgl. Abschnitt 2.5). Neben der modellbasierten Generierung von Tests für kontextsensitive Anwendungen ist die Bereitstellung einer Automatisierungstechnologie, die Kontextsimulation während der Testausführung unterstützt, ein Kernbeitrag dieser Dissertation.

4.4 Zusammenfassung

In den vorangegangen Abschnitten wurde das in dieser Arbeit untersuchte Konzept der Testautomatisierung für mobile, kontextsensitive Anwendungen in den Gesamtkontext Testen und Testautomatisierung in der Softwaretechnik eingebettet. Zunächst wurden hierzu die in dieser Dissertation verwendeten Begriffe erläutert und für eine Verwendung in Kapitel 5 aufbereitet (Abschnitt 3.1.1). In Abschnitt 4.1.2 werden Klassen von Softwaredefekten diskutiert. Insbesondere werden solche Defektklassen im Detail betrachtet, denen im Umfeld mobiler Anwendungen ein besonderer Stellenwert zukommt. Hierzu zählen beispielsweise Konnektivi-

tät, Energie, UI, außerplanmäßige Programmunterbrechung sowie insbesondere die Klasse der kontextinduzierten Defekte. Hierbei handelt es sich um solche Defekte, die durch die Verarbeitung von Kontextparametern (vgl. Abschnitt 3.1.2) eintreten können, wie etwa UI-Defekte aufgrund von Plattformheterogenität, fehlerhafte Darstellung oder Berechnung von Standort, Geräteorientierung oder anderer physikalischer Parameter.

Aus der Verwendung von Kontextparametern ergeben sich eine Reihe von Auswirkungen auf Testaktivitäten (vgl. Abschnitt 4.2). Gemeinsam ist diesen Auswirkungen, dass sie zusätzliche Komplexität in das Testen einführen. Zunächst müssen in einer zu testenden Anwendung Abhängigkeiten von Kontextparametern identifiziert werden. Anschließend müssen entsprechende Testdaten erstellt werden, die jeweils gültige und ungültige Werte für diesen Kontextparameter umfassen. Zusätzlich muss sowohl ein Konzept und eine Technologie erarbeitet werden, die es zulässt, Kontextparameter während der Testdurchführung so zu simulieren, dass kontextinduzierte Defekte erkannt werden können, z. B. durch Injektion ungültiger oder unplausibler Standortdaten oder Sensorinformationen in das SUT zur Laufzeit des Tests.

Abschnitt 4.3 untersucht Technologien und Konzepte der Testautomatisierung. In der Softwaretechnik existieren bereits für die Bereiche Generierung von Tests, Generierung von Testdaten, für die Durchführung von Tests und für die Analyse von Testergebnissen zahlreiche Konzepte, Methoden und Technologien. Sofern es sich um Standardwerkzeuge handelt sind Werkzeuge zur Testautomatisierung teilweise auf die Domäne der mobilen Anwendungen übertragbar. Gilt es jedoch auch Kontextfaktoren beim Testen zu berücksichtigen, sind existierende Lösungen nur mit Einschränkungen in der Lage, effektive und effiziente Tests zu automatisieren. Insbesondere Technologien zur Generierung kontextsensitiver Test (Abschnitt 4.3.2.1) sind im Jahr 2016 de facto nicht existent. Weder modellbasierte Methoden noch C'n'R-Methoden bedienen diese Domäne bislang in einem praxisrelevanten Umfang.

Für die Generierung von Tests wird in dieser Dissertation die Verwendung von modellbasierter Technologien gegenüber der Verwendung von C'n'R-Technologien präferiert. Zwar lassen sich auch mit C'n'R-Technologien signifikante Effizienzsteigerungen beim Testen erzielen, da Tests i. d. R. um ein Vielfaches häufiger ausgeführt werden als neue Tests erstellt werden. Diese Effizienzsteigerungen ergeben sich bei der Verwendung modellbasierter Methoden ebenfalls. Darüber hinaus ergibt sich im Umfeld mobiler, kontextsensitiver Anwendungen bei einer Aufzeichnung von Tests im C'n'R-Verfahren jedoch der Nachteil, dass einige Kontextfaktoren für den menschlichen Tester mit den Sinnesorganen nicht wahrnehmbar sind und deshalb in unbekannter Größe in die Testaufzeichnung eingehen. Es kann dann nicht abgeschätzt werden, ob bei der Testaufzeichnung eine Störung des Kontext vorliegt, die zu minderer Qualität von Tests führt. Die Verwendung modellbasierter Lösungsansätze vermeidet dieses Problem, indem relevante Kontextfaktoren im Testmodell explizit gemacht werden.

Basierend auf diesen Betrachtungen ist es Ziel dieser Dissertation, diese gegenwärtige Lücke in der methodischen und technologischen Unterstützung von Testautomatisierung für mobile, kontextsensitive Anwendungen zu schließen. Hierzu wird in Kapitel 5 eine Methode und ein Werkzeug zur modellbasierten Testautomatisierung kontextsensitiver, mobiler Anwendungen erarbeitet. Hierzu werden die in Abschnitt 4.3.2 diskutierten existierenden Lösungsansätze auf die Anforderungen mobiler Anforderungen erweitert.

Kapitel 5

Methode und Werkzeug zur modellbasierten Testautomatisierung kontextsensitiver mobiler Anwendungen

In den vorangegangen Kapiteln wurde dargelegt, dass sich Mobilität und Kontextsensitivität signifikant auf die Entwicklung, den Betrieb und das Testen von Software auswirken. Diese Auswirkungen manifestieren sich in einer erhöhten Komplexität aller Aktivitäten der Qualitätssicherung. Bislang existieren jedoch kaum Methoden und Werkzeuge, die den besonderen Anforderungen kontextsensitiver Anwendungen an das Testen gerecht werden.

Nachdem in Kapitel 3 die Bedeutung von Mobilität und Kontextsensitivität für mobile Softwaresysteme und in Kapitel 4 die besonderen Anforderungen an das Testen solcher Softwaresysteme diskutiert wurden, untersucht Kapitel 5 ein konstruktives Konzept einer modellbasierten Technologie zur Generierung von Softwaretests aus Modellen des Systementwurfs sowie ein Werkzeug zur automatisierten Testdurchführung mit Kontextsimulation.

Untersucht wird in diesem Kapitel eine konkrete Anwendung des MDSD-Verfahrens auf die Erstellung von Softwaretests. Voraussetzung ist, dass zu einer zu testenden mobilen Software Entwurfsmodelle existieren, in denen sich die Anforderungen an diese Software gemäß ihrer Spezifikation manifestieren. Es wird nicht vorausgesetzt, dass die Software selbst durch MDSD-Technologien aus diesen Modellen generiert wird.

Das Konzept zur Testgenerierung und Ausführung wird in Abschnitt 5.1 diskutiert und ähnlichen Ansätzen aus der wissenschaftlichen Literatur gegenübergestellt. In Abschnitt 5.2 erfolgt die Untersuchung der ersten Lösungskomponente, der Modellierung von Tests als Bestandteil der Systemmodellierung. Ebenfalls wird hier ein UML-Profil zur Verfeinerung des Systemmodells durch Anreicherung mit Testdaten erläutert. Die dritte Lösungskomponente, die Generierung plattformunabhängiger Testmodelle aus Systemmodellen, wird in Abschnitt 5.3 diskutiert. Abschnitt 5.4 behandelt die Transformation von Testmodellen in technologiespezifische Tests. Kapitel 5 schließt in Abschnitt 5.5 mit der Untersuchung der vierten Lösungskomponente, der automatisierten Ausführung von Tests mit Simulation des relevanten Kontext.

5.1 Methode zur modellbasierten Testautomatisierung

Das hier untersuchte Konzept zur automatisierten Generierung und Ausführung von Tests für mobile, kontextsensitive Anwendungen besteht aus fünf Komponenten und folgt dem Vorgehensmodell des *Model Driven Software Development* (MDSD), in welchem Software – in diesem Fall Softwaretests – basierend auf Modellen erstellt wird.

Zugrunde liegt hierbei die Prämisse, dass eine stringente Verwendung von Modellierungs- und Transformationswerkzeugen, die ein zunächst grobes Systemmodell sukzessiv unter Anreicherung von Modellen mit immer detaillierteren Informationen über Technologie und Plattform bis hin zum Quellcode verfeinern, einen hohen Grad an Kongruenz zwischen Spezifikation, Modell und Code gewährleisten. Konkret umfasst das Konzept diese fünf Phasen:

1. Modellierung von Verhaltensaspekten als Bestandteil der Systemmodellierung
2. Verfeinerung von Systemmodellen mit testspezifischen Details
3. Generierung plattformunabhängiger Testmodelle aus Systemmodellen
4. Transformation von Testmodellen in technologiespezifische Tests
5. Automatisierte Ausführung von Tests mit Simulation des relevanten Kontext

Die Vorgehensweise ist in Abbildung 5.1 abgebildet. Im ersten Schritt werden Verhaltensaspekte des zu testenden Systems im Systemmodell modelliert. Der hier vorgestellte Ansatz setzt voraus, dass dieses Verhaltensmodell als UML-Aktivitätsdiagramm vorliegt. Diese Einschränkung basiert auf der im Rahmen dieser Dissertation erarbeiteten Implementierung, ist vom Grundsatz her jedoch auch auf andere Modellierungssprachen übertragbar.

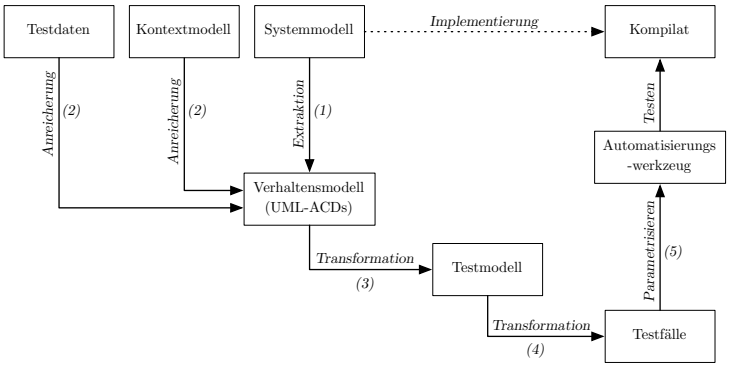


Abbildung 5.1: Schema der modellbasierten Testautomatisierung für mobile, kontextsensitive Anwendungen

Im zweiten Schritt wird das Verhaltensmodell, d. h. das UML-Aktivitätsdiagramm, mit testspezifischen Details angereichert. Zu diesen zählen Testdaten und insbesondere ein Kontextmodell. Diese Anreicherung erfolgt mit dem in Abschnitt 5.2.4 diskutierten UML-Profil zur Integration der Testfallmodellierung in die Systemmodellierung. Das Profil erweitert das Metamodell der UML um das in Abschnitt 5.2.2.2 betrachtete Metamodell zur Kontextmodellierung sowie das in Abschnitt 5.2.3.2 betrachtete Metamodell zur Testfallmodellierung.

Durch Anwendung von Modelltransformation wird dieses angereicherte UML-Aktivitätsdiagramm im dritten Schritt in ein plattformunabhängiges Testmodell überführt, welches testrelevante Aspekte des Verhaltensmodells isoliert. Es bildet die Grundlage zur weiteren Verarbeitung zu technologiespezifischen Tests.

Durch eine weitere Modelltransformation unter Hinzuziehung von Plattforminformationen werden aus dem Testmodell im vierten Schritt technologie- und plattformspezifische Testfälle erstellt. Es ist dabei nicht Voraussetzung, dass das plattformunabhängige Testfallmodell aus einem UML-Aktivitätsdiagramm generiert wurde. Manuell unter Verwendung des in Abschnitt 5.2.3.2 diskutierten Metamodells erstellte Testfallmodelle sind ebenfalls gültige Eingaben in die Modelltransformation, die ausführbare Tests generiert. Die hier untersuchte Methode zur Testautomatisierung macht hierbei keine konkreten Vorgaben hinsichtlich der Zieltechnologie. Es wird lediglich gefordert, dass die Zieltechnologie (z. B. JUnit, Calabash, UIAutomator) über Instrumente verfügt, ein Automatisierungswerkzeug zu instruieren während der Testausführung Kontextparameter zu simulieren.

In dieser Arbeit erfolgt die Generierung technologie- und plattformspezifischer Testfälle am Beispiel der Technologie Calabash (vgl. Abschnitt 5.4). Das heißt, Produkt der Phase vier ist eine Menge von Calabash-Features, die durch ein modifiziertes Calabash-Framework (vgl. Abschnitt 5.5.3) ausgeführt werden. Die Standardimplementierung des Calabash-Frameworks kann nicht verwendet werden, weil sie nicht über die notwendigen Funktionen verfügt, um während der Testausführung Kontextparameter (vgl. Abschnitt 3.1.2) im SUT zu simulieren. Diese Einschränkung gilt ebenfalls für andere verfügbare Testautomatisierungstechnologien (vgl. Abschnitt 2.5), von denen im Jahr 2016 keine in der Lage ist, Kontextsensitivität in einem adäquaten Umfang in Tests abzubilden. Calabash hat gegenüber den anderen Testautomatisierungstechnologien den Vorteil, dass Tests nicht im Quellcode einer Programmiersprache realisiert werden, sondern auf einer quasi natürlichsprachlichen Syntax basieren und deshalb für Stakeholder aller Rollen gleichermaßen zugänglich sind.

Die Ausgabe des vierten Schritts wird im fünften Schritt verwendet, um ein Automatisierungswerkzeug zu parametrisieren, welches die Tests ohne manuelle Intervention auf Geräten, Emulatoren oder einer Geräte- bzw. Emulatorfarm ausführt und hierbei den Kontext gemäß der Testfallspezifikation im SUT simuliert. Die Simulation von Kontext umfasst die Manipulation des SUT sowie der Ausführungsplattform, insbesondere zur Einspeisung künstlicher Kontextereignisse, wie etwa Standortinformationen oder Sensordaten.

Im Jahr 2016 bieten mobile Plattformen nur sehr eingeschränkte Schnittstellen zur Manipulation von Kontextparametern. Sowohl Emulatoren als auch Geräte lassen zwar in beschränktem Umfang eine Einspeisung von Standortinformationen zu. Die Manipulation von Sensoren beim Testen ist jedoch nicht in einem Umfang möglich, der zu einer effizienten Testautomatisierung verwendet werden kann (vgl. Abschnitt 4.2). Auf Geräten ist die Simulation von Sensorereignissen nicht möglich. Geräte implementieren Hardware-Sensoren, deren Werte ohne Möglichkeiten eines *Bypass* (d. h. ohne Schnittstellen, die eine testspezifische Abänderung zur Einspeisung von Testdaten zulassen) direkt in Anwendungen gelangen. Emulatoren verfügen nicht über Sensoren, so dass hier überhaupt nur eine künstliche Provisionierung in Frage kommt, die aufgrund der existierenden technischen Realisierung (vgl. Abschnitt 4.2.4.2) jedoch nicht zur Testautomatisierung einsetzbar ist.

Zu Realisierung einer Testautomatisierung mobiler, kontextsensitiver Anwendungen wird in dieser Dissertation aus diesem Grund eine Modifikation der Android-Plattform erarbeitet, die Schnittstellen zur Manipulation von Kontextparametern bereitstellt. Diese in Abschnitt 5.5.2 diskutierte Modifikation des Android-Betriebssystems überträgt das Paradigma *Designed for Testability* (vgl. Pettichord [281]) vom SUT in die Plattform.

In dieser Dissertation wird das Konzept am Beispiel der Plattform Android demonstriert, weil es sich bei Android um ein OSS-Projekt handelt und der Quellcode des Android-Betriebssystems frei zugänglich ist und angepasst werden kann. Das fertige Kompilat kann dann sowohl in Emulatoren als auch auf Geräten installiert werden, so dass ein SUT sowohl in emulierten Umgebungen als auch auf Geräten ausgeführt und getestet werden kann. Das modifizierte Android kann weiterhin an die Android-Plattforminfrastruktur angebunden werden, so dass die integrierten Plattformfunktionen wie beispielsweise der Zugang zum Google Play Store und anderen Plattformdiensten (z. B. Google Health [149]) erhalten bleiben. Vom Grundsatz her sind die Modifikationen am Android-Betriebssystem in funktional identischer Weise auch auf andere mobile Plattformen übertragbar. Diese sind jedoch keine OSS-Projekte und werden deshalb in dieser Dissertation nicht herangezogen.

Schritt eins (Modellierung von Verhaltensaspekten des Systems) des hier untersuchten Ansatzes zur modellbasierten Testautomatisierung ist ein regelmäßiger Bestandteil des Softwareprozesses und repräsentiert somit keine Besonderheit des hier vorgestellten Konzepts. Die Anreicherung des Modells mit Testdaten in Schritt zwei ist Bestandteil von MDT-Prozessen. Alleinstellungsmerkmal des hier untersuchten Testautomatisierungskonzepts gegenüber anderen MDT-Ansätzen ist die zusätzliche Anreicherung des Modells um Kontextparameter, die gleichrangig neben anderen Testdaten als Vorbedingung in die Testausführung einfließen. Die Schritte drei bis fünf sind ebenfalls Standardprozeduren von MDT-Technologien, hier allerdings mit der Besonderheit der Integration von Kontextparametern bei der Erstellung technologie- und plattformspezifischer Testfälle und der Simulation dieser Kontextparameter bei der Testausführung im Zielsystem.

Ähnliche Methoden zur Generierung von Tests wurden bereits in der wissenschaftlichen Literatur untersucht (vgl. Abschnitt 2.4). Während individuelle Ansätze in Details variieren, besteht Konsens über die grundsätzliche Machbarkeit des Vorgehens. Trotz Differenzen in einzelnen Schritten der Testfallgenerierung folgen individuelle Ansätze weitgehend einem einheitlichen Muster, an dem sich auch der hier vorgestellte Ansatz orientiert.

Das größte Unterscheidungsmerkmal von Ansätzen zur Testgenerierung ist die Verwendung unterschiedlicher Arten von UML-Modellen als Ausgangspunkt für die Testfallgenerierung. In der Regel basieren Ansätze zur Generierung von Tests, Testdaten oder Testorakeln auf der Verwendung von Verhaltensdiagrammen, wobei einige Ansätze zusätzlich Strukturmodelle (z. B. Klassendiagramme) verwenden. Neben Aktivitätsdiagrammen definiert die UML weitere Verhaltsmodelle, die potenziell zur Generierung von Testfällen verwendet werden können. Hierzu gehören Anwendungsfalldiagramme, Zustandsautomaten, Sequenzdiagramme, Interaktionsübersichtdiagramme, Kommunikationsdiagramme und Zeitverlaufsdiagramme.

Im Jahr 1999 untersuchten Offutt und Abdurazik [264] einen Ansatz zur Generierung von Softwaretests aus UML-Zustandsdiagrammen. Die Autoren argumentieren, dass das Zustandsdiagramm aus dem Repertoire der Verhaltensmodelle der UML die am besten formalisierte

Modellart ist. Zustandsdiagramme definieren, unter welchen Stimuli ein Softwaresystem von einem Zustand in einen anderen überführt wird. Diese Stimuli können beispielsweise Interaktionen von Anwendern sein. Ähnlich dem in dieser Dissertation vorgestellten Ansatz setzen auch Offutt und Abdurazik voraus, dass vor der Testgenerierung ein expliziter Zusammenhang zwischen Zustandsübergangsstimuli und Testdaten hergestellt wird. Der von den Autoren vorgestellte Ansatz transformiert Zustandsdiagramme in Sequenzen von Stimuli, die je nach gewähltem Abdeckungskriterium sicherstellen, dass bestimmte Zustände des SUT durch einen Test adressiert werden. Darüber hinaus liegt der von Offutt und Abdurazik untersuchte Ansatz außerhalb der Domäne mobiler Anwendungen und ist zudem auf einem implementierungsnahen Abstraktionsniveau angesiedelt.

Diese Dissertation hat Akzeptanztests zum Gegenstand. Diese überprüfen eine Software aus der Perspektive des Anwenders. Interne Zustände der Software sind für Akzeptanztests nicht interessant. Bei der Modellierung von Tests nach dem Ansatz von Offutt und Abdurazik ist hingegen Wissen über interne Zustände der Software notwendig. Der in dieser Dissertation verfolgte Ansatz zielt in Abgrenzung zu Offutt und Abdurazik darauf ab, nur solches Wissen über die Software zu berücksichtigen, dass sich auch aus der Perspektive von Black-Box-Tests erschließt. Ebenfalls adressiert die Arbeit der Autoren lediglich die Generierung von Tests, nicht jedoch deren automatisierte Durchführung.

Briand und Labiche [58] untersuchten 2002 einen ebenfalls auf der UML basierenden Ansatz zur Generierung von Tests aus Systemmodellen. Ziel des Ansatzes ist es, funktionale Tests, Orakel und sogenannte *Test Driver* aus verfeinerten Systemmodellen zu erzeugen, wobei das Artefakt *Test Driver* hierbei eine Testautomatisierungstechnologie bezeichnet. Konkret verwenden Briand und Labiche Anwendungsfalldiagramme mit zugehörigen Beschreibungen, Sequenz- und Kollaborationsdiagramme sowie Klassendiagramme zur Testfallgenerierung. Die Verwendung von Anwendungsfalldiagrammen dient im Ansatz der Autoren jedoch lediglich der Ableitung übergeordneter Systemfunktionen, die durch Tests abgedeckt werden sollen. Anwendungsfalldiagramme veranschaulichen Systemfunktionen auf einer hohen Abstraktionsebene, geben unmittelbar jedoch keine Auskunft hinsichtlich zulässiger Ausführungssequenzen individueller Aktionen. Aus diesem Grund analysieren die Autoren zusätzlich die Geschäftslogik der Anwendung zur Bestimmung zulässiger Funktionssequenzen auf Basis von UML-Aktivitätsdiagrammen, die einzelnen Anwenderrollen des Systems zugeordnet werden. Hierzu muss zusätzlich zum Anwendungsfalldiagramm Fachwissen eines Domänenexperten hinzugezogen werden. In Abgrenzung zu Briand und Labiche wird in dem in dieser Dissertation vorgestellten Ansatz durch die Annotation individueller Aktivitäten und Aktionen in UML-Aktivitätsdiagrammen die Notwenigkeit zusätzlichen Expertenwissens obsolet.

In Übereinstimmung mit dem in dieser Dissertation vorgestellten Ansatz ist der folgende Schritt bei Briand und Labiche die Analyse von Aktivitätsdiagrammen zur Identifikation möglicher Anfang-bis-Ende-Sequenzen individueller Aktionen, die einem Anwender zugänglich sind und deshalb durch einen Test abgebildet werden sollten. Durch die Möglichkeit von Schleifen in der Graphstruktur des Aktivitätsdiagramms existiert die Möglichkeit potenziell unendlich langer Sequenzen, die in Tests nicht sinnvoll abbildbar sind. Hier gilt es eine geeignete Strategie zum Umgang mit solchen Strukturen zu bestimmen (vgl. Abschnitt 5.3.3.1.3), die sich letztlich in einem konkreten Testabdeckungskritierum manifestiert.

Die Ausführung von Tests erfordert die Bereitstellung von Testdaten, mit denen das SUT bei der Durchführung von Tests parametrisiert wird. Beim von Briand und Labiche untersuchten Ansatz erfolgt die Definition von Testdaten durch Annotation individueller Aktionen in den Aktivitätsdiagrammen mit informalen Datenobjekten. In der in dieser Dissertation untersuchten Methode werden zu diesem Zweck die Modellierungselemente der Metamodelle zur Testfallmodellierung (vgl. Abschnitt 5.2.3.2) und zur Kontextmodellierung (vgl. Abschnitt 5.2.2.2) gemeinsam mit dem UML-Profil zur Integration der Testfallmodellierung in die Systemmodellierung verwendet (vgl. Abschnitt 5.2.4). In Abgrenzung zum Ansatz von Briand und Labiche wird hierdurch das Modell durch die Verfeinerung mit Testdaten um Kontextparameter ergänzt. Diese bleiben im von Briand und Labiche diskutierten Ansatz vollständig unberücksichtigt. Zur eigentlichen Generierung von Tests wird im Ansatz von Briand und Labiche der Graph des Aktivitätsdiagramms unter Berücksichtigung eines Abdeckungskriteriums in eine Baumstruktur expandiert. Dieses Vorgehen wird auch in dem in dieser Dissertation untersuchten Ansatz verfolgt. Der von Briand und Labiche untersuchte Ansatz ist weiterhin insofern von dem in dieser Dissertation untersuchten abzugrenzen, als dass die Autoren die Erstellung eines Testmodells zum Ziel haben, nicht aber die automatisierte Ausführung dieser Tests. In der Überführung des gerichteten Graphen des UML-Aktivitätsdiagramms in eine Baumstruktur, deren individuelle Pfade jeweils Testfälle repräsentieren, existieren allerdings Überscheidungen.

Cavarra et al. [74] untersuchen im Jahr 2002 ebenfalls ein Konzept zur Testgenerierung, das zahlreiche Parallelen zu dem in dieser Dissertation vorgestellten aufweist. Zwar haben Cavarra et al. weder mobile, kontextsensitive Anwendungen im Fokus noch erstreckt sich das von den Autoren untersuchte Konzept auf die automatisierte Ausführung von Tests. Jedoch basiert der Ansatz von Cavarra et al. (vgl.Abschnitt 2.4) ebenfalls auf der Verwendung von Objektdiagrammen, mit denen Testdaten in das Modell eingepflegt werden, bevor die Transformation zu Tests erfolgt. Ähnlich dem von Offutt und Abdurazik [264] untersuchten Konzept kommen bei Cavarra et al. ebenfalls Zustandsdiagramme zum Einsatz.

Mit der Verwendung des *UML 2.0 Testing Profile* (U2TP) zur Aufwertung von Systemmodellen zur Testgenerierung befassten sich Dai et al. [86, 85] im Jahr 2004. Das U2TP führt vier logische Konzepte zur Abdeckung von Testaspekten in die Modellierung ein: Testarchitektur, Testverhalten, Testdaten und Testzeit. Zusammen definieren diese eine Modellierungssprache zur Spezifikation, Visualisierung, Analyse, Konstruktion und Dokumentation eines Testsystems. Ähnlich wie dem in dieser Dissertation untersuchen Konzept der Generierung eines Testmodells aus einem Systemmodell, schlagen Dai et al. die Verwendung des U2TP zur Transformation eines Systemmodells in ein Testmodell vor. Allerdings setzt die Verwendung des U2TP die Erstellung eines völlig neuen Teilmodells voraus, bei dem lediglich Teile des bereits existierenden Systemmodells wiederverwendet werden. Der in dieser Dissertation untersuchte Ansatz vermeidet die Notwendigkeit einer manuellen Erstellung eines testspezifischen Modells durch Verfeinerung existierender Modelle. Der von Dai et al. untersuchte Ansatz unterscheidet sich jedoch grundlegend in seiner Zielsetzung von dem hier untersuchten Konzept. Dai et al. zielen mit der Verwendung des U2TP auf den Entwurf eines vollständigen Testsystems ab. In dieser Dissertation wird jedoch die Generierung von Tests verfolgt, die durch eine separate Automatisierungstechnologie ausgeführt werden können. Weiterhin stellt das U2TP keine

Domänenmodelle zur Spezifikation von Testdaten bereit. In diesem Sinn ist das von Dai et al. vorgestellte Konzept zur Verwendung des U2TP nicht auf die hier untersuchte Methode der Testautomatisierung für mobile, kontextsensitive Systeme übertragbar.

Einen ebenfalls auf UML-Aktivitätsdiagrammen basierenden Ansatz zur Generierung von Tests untersuchten Linzhang et al. [239] sowie Mingsong et al. [251] in den Jahren 2004 bzw. 2006. Zur automatisierten Analyse definieren die Autoren zunächst ein formales Modell zur Beschreibung von UML-Aktivitätsdiagrammen. Dieses Modell wird von den Autoren als formale Basis zum Entwurf eines Algorithmus zur Traversierung von Aktivitätsdiagrammen herangezogen. Diese Traversierung dient der Generierung einer Reihe von Sequenzen individueller Aktionen, die in ihrer geordneten Hintereinanderausführung einen Testfall repräsentieren. In dieser Dissertation wird in Abschnitt 5.3.3.1 ebenfalls ein Algorithmus zur Generierung solcher Sequenzen entworfen. In Abgrenzung zu Linzhang et al. werden hier Tests nicht direkt aus Aktivitätsdiagrammen generiert, sondern es werden zunächst plattformunabhängige Testmodelle aus Systemmodellen generiert, um Flexibilität hinsichtlich der Zielplattform und der konkreten Automatisierungstechnologie zu erhalten. Die technische Realisierung der Transformation eines Systemmodells zu einem Testmodell verwendet hierzu in einem Zwischenschritt eine Transformation zu Petri-Netzen (vgl. Abschnitt 5.3.3.1.2) vor der eigentlichen Generierung plattformunabhängiger Testmodelle. Zu diesem Zweck ist das durch das Metamodell der UML vorgegebene Modell von Aktivitätsdiagrammen ausreichend. Die Analyse möglicher Ausführungssequenzen erfolgt dann auf dem Erreichbarkeitsgraphen des Petri-Netzes. Hierdurch wird eine wesentliche Einschränkung des Konzepts von Linzhang et al. aufgefangen, welches nicht dafür ausgelegt ist, parallele Kontrollflüsse zu verarbeiten. Petri-Netze sind durch ein formales Modell besser zur Analyse möglicher Sequenzen von Aktionen geeignet als das semiformale Metamodell der UML. Darüber hinaus ist die Generierung möglicher Sequenzen von Aktionen aus Aktivitätsdiagrammen einziger Gegenstand der Arbeit von Linzhang et al., der Fokus liegt nicht auf den besonderen Herausforderungen des Testens mobiler, kontextsensitiver oder adäquaten Testautomatisierungstechnologie.

Die hier vorgestellte Methode zur Testfallgenerierung aus Systemmodellen basiert auf der Verfeinerung dieser Modell mit Testdaten, welche sowohl aus Testdaten im engeren Sinne, also z. B. von Eingaben eines Anwenders, als auch aus Kontextparametern zusammengesetzt sind. Diese Verfeinerung erfolgt in der technischen Realisierung durch Verwendung des in Abschnitt 5.2.4 diskutierten UML-Profils zur Integration der Testfallmodellierung in die Systemmodellierung. Der Ansatz Datenobjekte in die Systemmodellierung zu integrieren, wurde bereits außerhalb der Domäne Testautomatisierung angewendet. Stefanov et al. [334] untersuchten im Jahr 2005 einen ähnlichen Ansatz, UML-Aktivitätsdiagramme im Kontext von *Data Warehouses* (DWH) um *Business Intelligence Objects* (d. h. fachliche Geschäftsobjekte in digitaler Darstellung) zu erweitern. Wenngleich das von Stefanov et al. untersuchte Konzept die Relation zwischen Anwendung und DWH in Systemmodellen explizit herauszustellen in keiner Weise mit der hier untersuchten Testautomatisierungstechnologie in Zusammenhang steht, gibt es jedoch Parallelen in der technischen Herangehensweise. Auch Stefanov et al. definieren ein Metamodell zur strukturellen Modellierung von DWHs mit den für den Anwendungsfall maßgeblichen Entitäten. Durch ein UML-Profil erzeugen die Autoren eine explizite Verbindung zwischen diesem Metamodell und UML-Aktivitätsdiagrammen. Schritt

zwei (vgl. Abbildung 5.1) des in dieser Dissertation vorgestellten Konzepts zur Testautomatisierung realisiert die Verfeinerung von Systemmodellen in einer analogen Technologie.

Testfallgenerierung auf der Basis von UML-Aktivitätsdiagrammen wurde ebenfalls von Heinecke et al. [174] mit einem anwenderzentrierten Fokus auf Systemmodelle untersucht. Auch dort wurde ein Vorgehen verfolgt, ein Aktivitätsdiagramm mit Testdaten zu verfeinern und im Anschluss durch Modelltransformation in eine Darstellung mit spezifischem Fokus, dem *Interaction Flow Diagram* (IFD), zu transformieren. Dadurch wird die Anwenderinteraktionen durch Abstraktion vom restlichen Modell in den Vordergrund gerückt. Aus dem IFD wurden nach der Transformation durch einen Tiefensuchealgorithmus, analog zu Linzhang et al., Sequenzen von Interaktionen eines Anwenders mit dem SUT extrahiert. In Abgrenzung zu dieser Dissertation standen bei Heinecke et al. nicht mobile Anwendungen im Fokus, so dass das Vorgehen keine Unterstützung kontextsensitiver Aspekte des SUT bietet. Die Verfeinerung des Systemmodells mit Testdaten ist hingegen in ähnlicher Weise realisiert. Im Ansatz von Heinecke et al. erfolgt die Verfeinerung mit Testdaten durch Annotation von Aktionen des Aktivitätsdiagramms mit den durch das Metamodell der UML definierten Elementen zur Abbildung informeller Kommentare, die im Stil von UML-Stereotypen mit Informationen zur jeweiligen Rolle des ausführenden Anwenders ausgezeichnet werden. In dem in dieser Dissertation untersuchten Konzept erfolgt die Integration von Testdaten hingegen über die Verwendung von UML-Stereotypen, die durch das in Abschnitt 5.2.4 diskutierte UML-Profil definiert werden. Während bei Heinecke et al. eine explizite Unterscheidung von Akteuren erfolgt, ist diese Unterscheidung in dem hier untersuchten Ansatz nicht relevant. Weiterhin produziert das von Heinecke et al. diskutierte Verfahren ausschließlich eine textuelle Testfallbeschreibung, während das hier untersuchte Vorgehen direkt ausführbare Testfälle generiert.

Zusammenfassend wird festgestellt, dass das hier vorgestellte Vorgehen bei der Erstellung von Tests aus Systemmodellen in seiner wesentlichen Struktur mit anderen bereits wissenschaftlich untersuchten Verfahren vergleichbar ist. Wenngleich Technologien existieren, die auf anderen Modellierungssprachen basieren (z. B. BPMN, Guerra [162], Gupta und Surve [163] oder *BPEL*, Yan et al. [372] sowie Petri-Netze Xu [371]), hat sich die Verwendung von UML-Modellen auch im Bereich der Testgenerierung als de facto Standard etabliert. Das in dieser Dissertation vorgestellte Verfahren ist jedoch insofern von anderen Ansätzen abzugrenzen, als dass es explizit mobile, kontextsensitive Anwendungen zum Gegenstand hat. Durch die besonderen Herausforderungen des Testens dieser Klasse von Anwendungen sind existierende Konzepte und Technologien nur rudimentär auf das hier verfolgte Ziel übertragbar. Insbesondere bieten existierende Konzepte keine Möglichkeiten, Kontextparameter als Testdaten zu verwenden und den vollständigen Prozess von der Systemmodellierung bis hin zur automatisierten Testausführung mit Kontextsimulation abzubilden.

5.2 Modellierung von Testfällen für kontextsensitive mobile Anwendungen

Während im Bereich der Testdurchführung Automatisierungstechnologien bereits zur Effizienzsteigerung von Softwareprozessen beigetragen haben, ist das Erstellen von Testfällen weiterhin ein Prozess, der nicht nur ressourcenintensiv, sondern darüber hinaus auch anfällig

für Fehler ist. Von Hand geschriebener Code zur Implementierung von Testfällen, beispielsweise JUnit-Tests, unterliegt hinsichtlich der Fehleranfälligkeit denselben Kriterien wie die Software, die durch ihn getestet werden soll. Das heißt, fehlerhaft programmierte Tests führen dazu, dass Defekte im zu testenden System nicht gefunden werden oder vermeintliche Defekte erkannt werden, obwohl sich ein System spezifikationskonform verhält.

Ebenfalls ein kritischer Schritt in Testprozessen ist die Bereitstellung von Testdaten, insbesondere bei manuellen Tests. Es besteht das Risiko, dass Fehler bei der Durchführung (etwa Falscheingabe von Testdaten) zur fehlerhaften Bewertung der Spezifikationskonformität des SUT führen. Menschliche Fehler bei der Erstellung und Durchführung von Tests können durch eine sorgfältige Spezifikation und Dokumentation von Testfällen reduziert werden. Hierzu ist es jedoch notwendig, Testfälle so zu spezifizieren, dass sie unabhängig von der durchführenden Person reproduzierbar sind. Textuelle Beschreibungen bieten in dieser Hinsicht Interpretationsspielraum (vgl. Apfelbaum und Doyle [11]).

Im Systementwurf hat sich aus vergleichbaren Gründen die Verwendung von Modellen zur Systemspezifikationen etabliert. Auch im Bereich Testen sind modellbasierte Technologien in den Fokus von Forschungsarbeiten gerückt (vgl. Abschnitt 2.4). Für die automatisierte Generierung und Durchführung von Tests, insbesondere beim MDT, sind Modelle zentrale Artefakte. Geeignete Modellelemente bilden Kernartefakte des Testens strukturiert ab, so dass kein Spielraum für Fehlinterpretation gegeben ist. Modellelemente, die Artefakte des Testens (z. B. Eingabedaten, Kontextparameter usw.) abbilden, erfüllen zugleich die Funktion einer technologieunabhängigen Repräsentierung von Testfällen, die durch geeignete Werkzeuge zu technologiespezifischen Testimplementierungen weiterverarbeitet werden können.

Einleitend zur Generierung von Testmodellen wird im Folgenden zunächst in Abschnitt 5.2.1 auf die Rolle der Systemmodellierung in der Softwaretechnik im Allgemeinen eingegangen, Abschnitt 5.2.2 diskutiert die Integration von Kontextmodellierung in die Systemmodellierung und Abschnitt 5.2.3 hat die Modellierung plattformunabhängiger Testmodelle zum Thema. Ein UML-Profil zur Testfallmodellierung wird in Abschnitt 5.2.4 diskutiert.

5.2.1 Modellierung von Softwaresystemen

Softwareentwicklung ist ein gemeinschaftlich durchgeführter Prozess, in dem die Beteiligten unterschiedliche Sichten auf das zu entwickelnde System haben. Je nach Stakeholderperspektive stehen beispielsweise wirtschaftliche, prozessorientierte oder technische Aspekte eines Softwareprojektes im Fokus. Um diese Differenzen auszugleichen und eine einheitliche Wahrnehmung eines Softwarevorhabens bei allen Stakeholdern zu gewährleisten, ist Modellbildung ein Werkzeug, das es erlaubt, je nach konkreter Perspektive relevante Systemaspekte in den Fokus zu stellen und von irrelevanten Aspekten zu abstrahieren (vgl. Baumeister et al. [29]).

Als Modell (it. *modello* für Muster, von lat. *modulus* für Maß oder Maßstab [227]) wird in der Informatik eine verkürzende Abstraktion eines Ausschnittes der Realität bezeichnet, die für die jeweilige Domäne typische und für den in Software zu realisierenden Anwendungsfall bedeutsame Eigenschaften isoliert und weniger relevante oder nicht allgemeingültige Aspekte ignoriert (vgl. Stachowiak [328]). Ein Softwaremodell wird verwendet, um musterhaft die Struktur und Dynamik eines Softwaresystems abzubilden. Modelle selbst sind zunächst nichtgegenständliche Artefakte des Denkens. Um ihre Funktion der vereinfachenden konsensualen

Abbildung der Realität zur Entwicklung eines gemeinsamen Verständnis mehrerer Akteure über eine Sache zu erfüllen, bedarf es einer Modellierungssprache, mit deren Hilfe ein Modell lesbar für den Menschen (und ggf. für Maschinen) verschriftlicht werden kann. Von einer Modellierungssprache wird gefordert, einen eindeutigen Zusammenhang zwischen einem Modellierungselement (d. h. einem darstellbaren Artefakt) und seiner Semantik herzustellen. Das wird durch das sogenannte Metamodell zu einer Modellierungssprache gewährleistet, durch welches Modellierungselemente sowie deren Relationen untereinander definiert werden.

Softwaresysteme werden sowohl hinsichtlich ihrer strukturellen als auch ihrer dynamischen Eigenschaften modelliert. Strukturmodelle bilden den strukturellen Zusammenhang zwischen einzelnen Komponenten ab, d. h. es wird modelliert, aus welchen Komponenten ein System besteht, wie diese zueinander in Beziehung stehen (z. B. Assoziation, Aggregation, Vererbung) und wie Komponenten im Betrieb auf (virtuelle) Maschinen verteilt sind. Sie beschreiben den Aufbau eines Softwaresystems. Dem gegenüber stehen dynamische Modelle, die das Verhalten eines Softwaresystems beschreiben. Sie bilden die Abfolge individueller Aktivitäten zur Realisierung eines fachlichen Ziels ab. Insgesamt wird ein Softwaresystem durch eine Kombination von Struktur- und Verhaltensmodellen beschrieben. Je nach Detaillierungsgrad individueller Modelle gehen die Verwendungsmöglichkeiten über die Dokumentations- und Kommunikationsaspekte hinaus. Klassenmodelle können beispielsweise als Eingabe für Quellcodegeneratoren verwendet werden, um solche Codeartefakte automatisiert zu erzeugen, die nach etablierten Mustern der Softwareentwicklung zu erstellen sind, beispielsweise die Klassengerüste für Datentypen. Es wird erwartet, dass Generatoren Quellcode erzeugen, der manuell implementiertem Code semantisch gleichwertig, syntaktisch in Qualitätsaspekten jedoch überlegen ist (vgl. Herrington [181], Beydeda et al. [43], Stahl et al. [329], Kelly und Tolvanen [216]). Hinzu kommt, dass Generatoren Quellcode in einem Bruchteil der Zeit erzeugen können, die bei einer manuellen Implementierung aufzuwenden wäre.

In der praktischen Anwendung ist die Verbindung von Modell und Quellcode unterschiedlich ausgeprägt. Kelly und Tolvanen [216] unterscheiden hier fünf Zustände des Zusammenhangs zwischen Modell und Quellcode, wobei Modellen jeweils eine unterschiedliche Funktion zuteil wird. Nach dieser Klassifikation, dargestellt in Abbildung 5.2, wird das eine Extrem durch die Variante *Code Only* gebildet, in welcher Anforderungen an ein Softwaresystem direkt und ohne jede Modellierung in Code realisiert wird. Diese Variante verwendet keine Modelle und wird deshalb im Folgenden nicht weiter betrachtet. Das andere Extrem manifestiert sich in der Variante *Domain-Specific Modeling*, in welcher ein Softwareprodukt vollständig auf Modellen basiert, wenngleich nicht gefordert wird, dass das Softwareprodukt unter Verwendung von MDSD-Technologien aus Modellen generiert wird. Zwischen diesen extremen Ausprägungen existieren die Mischmodelle *Separate Model and Code*, *Code Visualization* und *Round Trip*. In der Variante *Separate Model and Code* sind Softwaremodelle selbst nicht unmittelbarer Gegenstand der Softwareentwicklung, sondern werden losgelöst vom Code angefertigt. Ihre Funktion ist dann auf Dokumentation beschränkt, die Stakeholder beim Verstehen des Codes bzw. des Softwaresystems unterstützt. Eine ähnliche Funktion hat die Variante *Code Visualization*, wobei hier gewährleistet ist, dass das Modell nicht unzulässig von technischen Eigenschaften des Codes abstrahiert. Die einzige Variante in diesem Kontext, die gewährleistet, dass Code und Modell ständig kongruent sind, ist *Round Trip*. Hier führt eine Änderung

des Modells zwangsweise zu einer Änderung des Codes und eine Änderung des Codes zieht eine Anpassung des Modells nach sich. Diese Variante erfordert hohe Sorgfalt und erzeugt einen kaum manuell beherrschbaren Aufwand (vgl. Balz [27]).

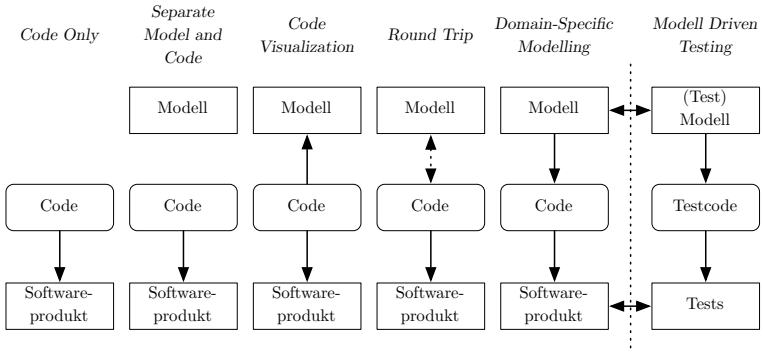


Abbildung 5.2: Beziehung zwischen Modell und Code nach Kelly und Tolvanen [216]. Von links nach rechts fällt der Modellierung eine zunehmend dominante Rolle im Softwareprozess zu. Die Grafik nach Kelly und Tolvanen wurde durch den in dieser Dissertation thematisierten Aspekt der modellbasierten Testfallgenerierung ergänzt. Dieser ist auf alle Ausprägungen mit Ausnahme der Variante *Code-Only* anwendbar und verlangt nicht, dass das zu testende System vollständig oder in Teilen aus Modellen generiert wurde.

Es wurden unterschiedliche Modellierungssprachen und -werkzeuge entwickelt, die je nach Anwendungsfall in unterschiedlichem Maß für die Verwendung in einem konkreten Softwareprojekt geeignet sind. Hierzu gehören beispielsweise *Ereignisgesteuerte Prozessketten* (EPK), *Business Process Model and Notation* (BPMN) und die *Unified Modeling Language* (UML). Einige Modellierungssprachen haben eine solche Beschreibungsmächtigkeit ausgeprägt, dass sie als gleichrangige Artefakte neben Quellcode in der Softwareentwicklung verwendet werden (vgl. Hartman et al. [171]). Technologien wie *Model Driven Architecture* (MDA), *Model Driven Software Development* (MDSD) oder *Model Driven Testing* (MDT) verwenden Modelle zur Generierung technischer Artefakte, wie z. B. Quellcode, Maschienencode oder Tests.

MDSD verwendet Softwaremodelle intensiv und zielt darauf ab, den Entwicklungsprozess weitgehend durch die Verwendung von Modelltransformation von einer manuellen Implementierung von Code zu entkoppeln. Mit einem starken Fokus auf fachliche Anforderungen wird Software im MDSD zunächst auf einem hohen Abstraktionsniveau beschrieben und später in techniknahen Modellen näher spezifiziert (vgl. Gruhn et al. [161], Sendall und Kozaczynski [318]). Diese bilden dann wiederum die Basis für die eigentliche Implementierung. Diese Idee wird in der MDA weiter vorangetrieben. Über die Konzepte des MDSD wird hier versucht, Softwareartefakte durch Verwendung von Werkzeugen aus Modellen zu generieren. Bei diesen Artefakten kann es sich um Quellcode oder auch um Testfälle handeln. Insbesondere letzterer Fall wird als *Model Driven Testing* (MDT) bezeichnet.

Die Modellierungssprache UML hat sich zum Quasistandard in der Softwareindustrie entwickelt (vgl. Baumeister et al. [29], Rumbaugh et al. [306], Lange et al. [233]) und wird daher in Wissenschaft und Praxis besonders häufig im Kontext des MDSD, MDA und MDT eingesetzt. Gegenüber anderen Modellierungssprachen zeichnet sie sich dadurch aus, dass sie ein

umfangreiches erweiterbares Metamodell zur grafischen Modellierung unterschiedlicher Aspekte eines Systems anbietet. Die UML ermöglicht die semiformale Modellierung von Systemen in beliebigem Detaillierungsgrad, wobei je nach konkretem Fokus eines individuellen Modells der Abstraktionsgrad frei wählbar ist. Strukturelle und dynamische Systemeigenschaften können unter Verwendung von insgesamt 14 Diagrammtypen abgebildet werden. Diese entfallen auf die Kategorien Strukturdiagramm und Verhaltensdiagramm. Insbesondere stellt die UML mit den UML-Profilen einen Mechanismus zur Verfügung, mit dem die Modellierungssprache domänenspezifisch angepasst werden kann, d. h. das Metamodell der UML kann domänenspezifisch um weitere Modellierungselemente erweitert oder existierende Modellierungselemente um neue Eigenschaften ergänzt werden.

Die in dieser Dissertation untersuchten Forschungshypothesen setzen die Verwendung von Modellen zur Testfallgenerierung voraus. Es ist hingegen keine Voraussetzung, dass eine zu testende mobile, kontextsensitive Anwendung selbst aus Modellen generiert wurde. Alle in der Abbildung 5.2 dargestellten Varianten mit Ausnahme von *Code Only* sind deshalb geeignet für eine modellbasierte Generierung von Testfällen. Unter der funktionsorientierten Perspektive von Akzeptanztests kann eine dedizierte Modellierung einer Anwendung zum Zweck der Testfallgenerierung sogar von Vorteil sein, da auf diese Weise gezielt eine adäquate Abstraktionsebene gewählt werden kann.

Es existiert im Jahr 2016 keine Modellierungssprache, die universell für alle Anwendungsbereiche gleichermaßen geeignet ist. Vielmehr sind Modellierungssprachen im Software Engineering Werkzeuge, die bedarfsgerecht weiterentwickelt werden, um den jeweiligen Anforderungen der jeweiligen Technologie-Ära oder der spezifischen Anwendungsdomäne gerecht zu werden. Mobile Anwendungen Anwendungen sind im Kontext der Geschichte des Software Engineering verhältnismäßig jung. Der Beginn der Smartphone-Ära kann auf den Zeitpunkt des Erscheinens der ersten Generation des Apple iPhone (2007) bzw. des ersten Android Smartphone HTC Dream (2008) datiert werden. Erst ab diesem Zeitpunkt begannen Smartphones die zur Implementierung kontextsensitiver Anwendungen notwendigen technischen Voraussetzungen alltagstauglich auszubilden. Seitdem haben sich die Modellierungssprachen und -werkzeuge des Software Engineering nicht nennenswert im Hinblick auf alleinstellende Merkmale mobiler Anwendungen weiterentwickelt (vgl. Henricksen und Indulska [177]). Eine vollständige Spezifikation mobiler Anwendungen unter Berücksichtigung von Kontextsensitivität ist deshalb praktisch ausgeschlossen. Während die technische Entwicklung von Smartphones und den zugehörigen SDKs zügig voranschreitet und die Verfügbarkeit von GPS-Hardware oder anderer Sensorik zeitnah zur Hardwareentwicklung in die APIs mobiler Plattformen überführt wird, finden diese Technologien nur langsam Einzug in Modellierungssprachen und -werkzeuge, mit der Konsequenz, dass etablierte Modellierungssprachen keine adäquaten Sprachmittel und Modellierungselemente zur Berücksichtigung dieser Faktoren in Systemmodellen anbieten.

5.2.2 Modellierung von Kontextsensitivität in mobilen Softwaresystemen

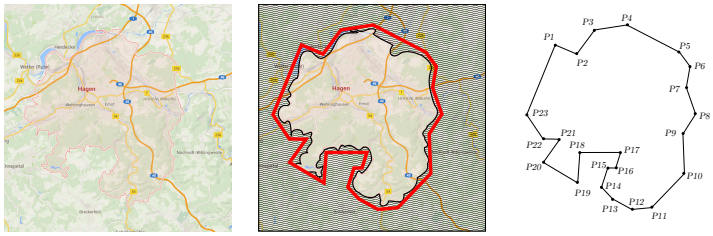
Das in dieser Dissertation verfolgte Ziel, Testfälle für mobile Anwendungen aus Systemmodellen zu generieren und anschließend automatisiert und unter Berücksichtigung eines definierten Betriebskontextes auszuführen setzt voraus, dass das zur Testgenerierung verwendete

Systemmodell neben den fachlichen Testdaten ebenfalls benötigte Kontextparameter enthält. Existierende Modellierungssprachen der Informatik sind für diese Aufgabe nur bedingt geeignet. Insbesondere die UML stellt keine entsprechenden Modellierungselemente bereit. Deshalb muss eine Modellierungssprache durch ein entsprechendes Metamodell definiert oder erweitert werden, welches Modellierungselemente bereitstellt, um Kontextparameter zu beschreiben.

In der Softwaretechnik werden Modelle verwendet, um strukturelle Eigenschaften und das dynamische Verhalten einer Software so zu fixieren, dass alle Stakeholder ein gemeinsames Verständnis des Systems auf einer Abstraktionsebene entwickeln, die eine fachliche Beschreibung des Systems zulässt, ohne dass durch die Detailtiefe technischer Aspekte fachliche Auseinandersetzung mit den Anforderungen aus dem Fokus geraten. Diese Anforderungen können Inhalte zum Gegenstand haben, die mit den Mitteln der UML oder anderen Modellierungssprachen nicht abgebildet werden können, sondern auf separaten Dokumentationswegen, z. B. natürlichsprachliche Beschreibung, in die Systemspezifikation einfließen müssen.

Es existieren mehrere Optionen, diese fachliche Anforderung in einem Modell abzubilden. Weder die UML noch andere Modellierungssprachen machen Vorgaben hinsichtlich der Abstraktionsebene eines Modells, wenngleich einige Modellierungssprachen hier im Bezug auf die mögliche Detaillierungstiefe limitiert sein können. Aus der fachlichen Perspektive ist es ausreichend, hier einfache Fallunterscheidung zu modellieren, die in Abhängigkeit von der Gültigkeit der Standortinformation das weitere Verhalten der App steuert. Ebenfalls wäre es möglich, die Bewertung der Gültigkeit des Standortes in einem verfeinernden Modell näher zu spezifizieren, beispielsweise durch Modellierung eines entsprechenden Algorithmus, der jedoch ebenfalls eine Ausgabe erzeugt, die eine boolsche Bewertung der Zulässigkeit des aktuellen Standorts erlaubt. Bei Verwendung der UML geht in beiden Fällen das Ergebnis der Auswertung in eine UML-*Guard-Condition-Condition* (ein Wächterausdruck, der auswertet ob eine Transition zwischen Knoten eines UML-Aktivitätsdiagramms passierbar ist) an den ausgehenden Kanten eines Entscheidungsknotens ein.

Unabhängig von Abstraktionsebene und Modellierungssprache muss jedoch in jedem Fall das zulässige polygonale Areal definiert werden. Die konkrete Definition des zulässigen Areals ist einerseits Teil der Anforderungsspezifikation und andererseits eine Ressource bei der Erstellungen von Testfällen, da sie Vor- und Nachbedingungen für Tests impliziert, nämlich zulässige und unzulässige Standortinformationen (Vorbedingung) und Fortsetzung der App mit Aktualisierung des UI mit gültigen POI-Daten oder der Anzeige einer Fehlermeldung (Nachbedingung). Stand der Technik im Jahr 2016 ist es, die zulässigen Standortinformationen auf einer geeigneten Abstraktionsebene separat in der Anwendungsspezifikation zu dokumentieren, beispielsweise durch Angabe eines Ortsnamens (Abbildung 5.3a) oder durch Definition eines Polygons (Abbildung 5.3b) oder Polygonzugs, etwa durch Auflistung von (WGS84-Koordinaten) (Abbildung 5.3c). Hierdurch entsteht ein Medienbruch, der diese Information für alle Techniken des MDSD, MDA und insbesondere des MDT von einer Nutzbarkeit durch Softwarewerkzeuge ausschließt. Um diese Information, oder analog andere Kontextparameter wie z. B. Geräteorientierung, für MDSD- oder MDT-Technologien verfügbar zu machen, müssen diese Kontextparameter ohne Medienbruch direkt in Systemmodelle integriert werden. Um das zu gewährleisten muss das Metamodell einer Modellierungssprache um entsprechende Modellierungselemente zur Kontextmodellierung erweitert werden.



(a) Umgangssprachliche Spezifikation „Hagen“

(b) Implizite polygonale Spezifikation

(c) Maschinenlesbare polygonale Spezifikation durch Auflistung von Koordinaten

Abbildung 5.3: Abstraktionsebenen der Spezifikation regionaler Standortinformationen

Das Thema Kontextmodellierung wurde in der Vergangenheit bereits vielfach wissenschaftlich betrachtet und ist auch im Jahr 2016 weiterhin ein intensiv untersuchtes Forschungsfeld, in dem eine Konvergenz möglicher Lösungsansätze noch nicht erkennbar ist. Kleinrock [226] hat beispielsweise bereits im Jahr 1996 erkannt, dass die Modellierung von Kontext bereits in frühen Phasen des Softwareprozesses eine Notwendigkeit ist, um adäquate Softwarearchitekturen zu entwerfen, die den besonderen Anforderungen von Mobilität und Kontextsensitivität gerecht werden (vgl. Abschnitt 2.2.2). Allerdings waren die Fähigkeiten mobiler Geräte im Jahr 1996 wesentlich geringer ausgeprägt als in der Smartphone-Ära, so dass die Erkenntnisse von Kleinrock nur bedingt auf die Anforderungen des Jahrs 2016 übertragbar sind. In den folgenden Jahren dominierten hauptsächlich Überlegungen zu Mobilität von Geräten, Personen und Codeartefakten die wissenschaftliche Literatur im Umfeld kontextsensitiver Software. Erst im Jahr 1999 entwickelten Schmidt et al. [315] ein Kontextmodell (vgl. Abschnitt 3.1.2) unter Berücksichtigung einer Vielzahl generischer Kontextparameter auch abseits der Netzwerkkonnektivität, ohne jedoch technische Aspekte der Anwendungsmodellierung zu adressieren.

Frühe Arbeiten zur Kontextmodellierung untersuchen die Eignung von *Key-Value-Pair*-Modellen (engl. Schlüssel- und Wertpaare) bzw. *Markup Models* (engl. beschreibende Auszeichnungsmodell) zur Modellierung von Kontext. Als Modellierungssprache kann beispielsweise *Extensible Markup Language* (XML) verwendet werden. Ein solcher Ansatz Kontextinformationen durch eine Modellierungssprache zu spezifizieren ist das *Composite Capabilities/Preference Profile* (CCPP) [356] (vgl. Reynolds et al. [295]). Es handelt sich hierbei um einen *Resource Description Framework* (RDF)-Ansatz, der das Key-Value-Modell mit einer Markup-Sprache vereinigt. Durch den formalisierten RDF-Ansatz könnten CCPP-Modelle gut maschinell verarbeitet werden. Da sie jedoch keine graphische Notation vorsehen, sind komplexere Modelle für den Menschen schwer zu überblicken. Die Anforderung an eine Modellierungstechnologie, Sachverhalte der Wirklichkeit durch Abstraktion auf relevante Eigenschaften zu reduzieren, wird hierdurch verletzt. Das CCPP-Metamodell gibt Typen von Kontextparametern vor, die den Anforderungen des Jahres 2016 nicht mehr gerecht werden. Beispielsweise fehlen Komponenten zur Modellierung von Standortinformationen.

Eine weitere Modellierungstechnologie ist das von Castelli et al. [73] vorgestellte W4-Kontextmodell. Es basiert ebenfalls auf Key-Value-Paaren, wobei erforderliche Schlüsselwerte durch das 4W-Metamodell verbindlich vorgegeben sind, nämlich wer, was, wo und wann (Kern-

faktoren situativer Kontextbeschreibungen wie bereits von Schilit et al. [311] untersucht). Das Metamodell definiert allerdings lediglich die Struktur von Wissensatomen (*Knowledge Atoms* [73]), nicht jedoch die Datentypen der den Schlüsseln zugeordneten Werte. Es besteht beispielsweise nicht die Möglichkeit, explizit zwischen geodätischen und symbolischen Standortinformationen (vgl. Abschnitt 3.1.2.1.3) zu unterscheiden. Weitere Kontextparameter, die beispielsweise durch Gerätesensoren messbar sind, (vgl. Abschnitt 3.1.2.1) sind im W4-Metamodell ebenfalls nicht vorgesehen, so dass eine Übertragbarkeit dieses Konzept auf die in diese Dissertation untersuchte Anwendung nicht gegeben ist.

Die geringe Ausdrucksmächtigkeit von Key-Value-Paar-basierten Modellen wird durch objektrelationale Modelle erweitert. Hier werden Kontextinformationen nicht einfach in Schlüssel/Wert Paaren gespeichert, sondern Kontextobjekte werden in Anlehnung an objektorientierte Entwurfstechnologien klassifiziert und in voneinander abgegrenzten Entitäten gekapselt. Ein Vertreter dieser Technologie ist *Context Modeling Language* (CML) nach Henricksen und Indulska [178, 176, 177]. CML stellt eine graphische Notation bereit und fügt sich deshalb in den Kanon der graphischen Modellierungssprachen ein. So wie Softwaremodelle mit Hilfe der unterschiedlichen Diagrammtypen der UML graphisch dargestellt werden können, bietet CML die Möglichkeit über alle Entwurfsstadien des Softwareprozesses hinweg verwendet zu werden. Allerdings ist CML darüber hinaus nicht in die UML integriert. Eine direkte Verwendung von CML innerhalb von mit UML erstellten Systemmodellen ist daher nicht möglich. Aufgrund der hohen Durchdringung von UML verwenden Werkzeuge des MDSD, MDA und MDT jedoch i. d. R. UML-Modelle. Um die Anzahl verwendeter Modellierungssprachen und -werkzeuge in einem Softwareprojekt gering zu halten, ist es vorteilhaft, auch Kontextmodelle mit den Werkzeugen und Diagrammen aus dem Umfeld der UML erstellen zu können. Hervorzuheben ist die Eigenschaft von CML, einzelne Kontextobjekte logisch miteinander zu verknüpfen. Ähnlich einer Ontologie ermöglicht das, weitere Informationen aus einem Kontextmodell zu inferieren. Nachteilig ist hingegen, dass CML nur eine Modellierung auf der Ebene individueller Kontextatome erlaubt. Hierarchische Beziehungen zwischen Kontextatomen oder komplexe, zusammengesetzte Kontexte werden nicht unterstützt.

Ein alternativer Ansatz ist die Verwendung von Kontextontologien, die über reine Assoziationsbeziehungen hinaus weitere Semantik für Assoziationen definieren. Eine solche Kontextontologie wird von Korpipää und Mäntyjärvi [229] vorgestellt. Die Autoren definieren eine Reihe von Eigenschaften von Kontextparametern, nämlich Kontexttyp, Bezeichnung, Wert, Konfidenz, Kontextquelle, Zeitstempel. Weiterhin können Kontextklassen definiert werden, die für Kontextparameter prototypische Werte spezifizieren. Instanzen dieser Klassen erfüllen die Funktion von Kontextatomen. Funktionsfähig wird dieser Ansatz in einer kontextsensitiven Anwendung durch Verwendung einer in der Anwendung implementierten Inferenzmaschine. Diese kann Werte aus Gerätesensoren und sonstigen Kontextquellen auf die für diese Anwendung erstellte Ontologieinstanz anwenden und über den Kontext schlussfolgern. Korpipää und Mäntyjärvi entwerfen mit ihrem Metamodell eine Technologie, die zur direkten Verwendung von Kontextmodellen zur Laufzeit einer Anwendung intendiert ist. Das Kontextmodell ist somit kein Artefakt des Systementwurfs, sondern ein laufzeitrelevantes Datenobjekt, welches ohne eine adäquate Inferenzmaschine abseits dieser spezifischen Nutzung nicht verwendet werden kann.

Ein weiteres ontologiebasiertes Kontextmodell wird von Chen et al. [75] betrachtet. Übereinstimmend mit Korpipää und Mäntyjärvi argumentieren die Autoren ebenfalls, dass für die adäquate Implementierung kontextsensitiver Anwendungen maschinelles Inferieren höherklassiger Kontexte aus niederklassigen Kontextparametern unerlässlich ist, wenn diese Anwendungen komplexe situative Kontexte verarbeiten. Gemeint ist hier die Erstellung einer Ontologie, die konkrete Werte oder Wertebereiche bestimmter Kontextparameter auf Konzepte einer höheren Abstraktionsebene überführt. Im Beispiel der Anwendung AR Tourist Information würden nach der von Chen et al. vorgestellten Kontextontologie die definierenden Eckpunkte einer polygonal begrenzten geographischen Ausdehnung (vgl. Abbildung 5.3c) als Instanzen einer Klasse *AtomicPlace* der *Web Ontologie Language* (OWL) modelliert werden. Das Polygon selbst würde als Instanz einer Klasse *CompoundPlace* mehrere Instanzen von *AtomicPlace* aggregieren. Die in *CompoundPlace* definierte Eigenschaft *isSpatiallySubsumedBy* erlaubt dann die Ausführung einer Abfrage der Enthaltenseinsbeziehung unter Angabe spezifischer Koordinaten. Das Resultat dieser Abfrage ist entweder leer oder liefert eine Instanz der Klasse *Place*, ihrerseits Superklasse von *AtomicPlace* und *CompoundPlace*. Die Transition von Wissen von einer niederen Abstraktionsebene (Koordinaten) auf eine hohe Abstraktionsebene (Stadtgebiet, Campus, etc.) kann auf diese Weise maschinell vollzogen werden. Auch hier liegt die Verwendung der Kontextontologie zur Laufzeit der Anwendung im Fokus, nicht die Verwendung als Artefakt des Systementwurfs.

Bettini et al. [42] untersuchen ebenfalls einen ontologiebasierten Ansatz zur Repräsentierung von Kontext. Die Autoren argumentieren, dass Ontologien ein wesentliches Werkzeug der Wissensrepräsentation und vor allem für den maschinellen Wissensaustausch sind. Gerade diese Eigenschaft hebt Ontologien als Wissensträger im Umfeld der Entwicklung mobiler, kontextsensitiver Anwendungen heraus, da hier viele heterogene Entitäten Kontextinformationen, ggf. im gegenseitigen Austausch, interpretieren und zur Steuerung von Kontroll- und Datenflüssen verwenden. In dieser Hinsicht sind ontologiebasierte Ansätze Key-Value-basierten oder objektrelationalen Ansätzen überlegen. Bettini et al. stellen diese unterschiedlichen Ansätze einander gegenüber und kommen zu dem Ergebnis, dass keiner der Ansätze den anderen insgesamt so deutlich überlegen ist, dass er diese de facto obsolet macht. Es ist vielmehr abhängig vom konkreten Anwendungsfall, ob der Aufwand zur Erstellung und Pflege einer Kontextontologie gerechtfertigt ist. Für das in dieser Dissertation vorgestellte Konzept der Generierung kontextsensitiver Testfälle wird die Mächtigkeit ontologiebasierter Kontextmodelle nicht benötigt. Schlussfolgern über den Kontext ist Teil der zu testenden Anwendungslogik. Aufgabe einer Testautomatisierungsumgebung ist jedoch die Simulation von Kontext, so dass einfachere Kontextmodelle ausreichend sind. Ziel der in dieser Dissertation vorgestellten Methode und Technologie ist es nicht, Kontextparameter zu interpretieren und daraus Schlussfolgerungen über die Situation des Anwenders zu ziehen. Vielmehr soll das Testen solcher Funktionen erleichtert werden, indem Kontextparameter reproduzierbar auf einer niederen Abstraktionsebene für das Testen bereitgestellt werden.

Baumeister et al. [29] untersuchen im Jahr 2003 ein Konzept, Standortinformationen in UML-Modelle zu integrieren. Zum Zeitpunkt der Untersuchung waren mobile, kontextsensitive Anwendungen noch nicht in der Form realisierbar wie es mit den Geräten der Smartphone-Ära möglich ist. Deshalb unterstützt die von Baumeister et al. entworfene UML-Erweiterung außer

Ortsinformationen keine weiteren Kontextparameter. Zur Unterstützung von Standortinformationen entwerfen die Autoren ein Metamodell, in dem UML-Klassen mit Stereotypen annotiert werden können, die abbilden, dass modellierte Entitäten in Softwaremodellen ortsbasiert sind. Die Gemeinsamkeit zwischen dem in diese Dissertation untersuchten Ansatz zur Testautomatisierung für mobile Anwendungen und dem von Baumeister et al. [29] untersuchten Ansatz zur Modellierung von Anwendungen ist die Verwendung von UML-Profilen zur Modellierung von Kontext. In dieser Dissertation werden jedoch vorrangig dynamische Aspekte mobiler Anwendungen betrachtet, weshalb das von Baumeister et al. vorgestellte UML-Profil nicht wiederverwendet werden kann. Es fokussiert auf strukturelle UML-Modelle und ist daher für die Generierung von Akzeptanztests, also funktionaler Tests, nicht verwendbar.

Ein ebenfalls UML-basiertes Metamodell zur Kontextmodellierung untersuchten Grassi et al. [156] mit dem Ziel, Mobilität von Code oder Anwender bei der Modellierung von Softwaresystemen zu berücksichtigen. Grassi et al. entwerfen hierzu ein Metamodell, mit dem es möglich ist, Entitäten der Systemmodellierung mit der UML durch Anwendung von UML-Stereotypen auf Modellelemente als mobil zu kennzeichnen. Durch eine Verankerung dieser Stereotypen auf einer hohen Abstraktionsebene des UML-Metamodells lassen es die Autoren zu, alle Knotentypen, die durch das UML-Metamodell bereitgestellt werden, in einen Mobilitätskontext einzubetten. Verwendungszweck dieser Stereotypen ist es, im Anwendungsentwurf darstellbar zu machen, welchen Einfluss Mobilität voraussichtlich auf eine Software haben wird. Was der von Grassi et al. vorgestellte Ansatz hingegen nicht zu leisten vermag, ist die Abbildung konkreter Standortinformationen. Vielmehr betrachten die Autoren Mobilität und Standorte als abstrakte Entitäten, deren konkrete Zustände nicht durch das Modell abbildbar sind. Zudem unterstützt das Metamodell auch keine anderen Kontextparameter. Deshalb ist auch der Ansatz abseits der Erweiterung der UML nicht geeignet, um zum Zweck der Augmentierung von Systemmodellen um kontexttuale Testdaten verwendet zu werden.

Im Jahr 2007 untersuchten Ayed et al. [21] ein Konzept zur Integration von Kontext in die Modellierung von Software mit der UML. Das von den Autoren vorgestellte UML-Profil ermöglicht es, Abhängigkeiten von Kontextparametern durch UML-Stereotypen zu Modellen hinzuzufügen. Obwohl das von Ayed et al. vorgestellte Metamodell aus der frühen Phase der Smartphone-Ära stammt, in der nur wenige Gerätemodelle mit vergleichsweise geringer Ausstattung verfügbar waren, ermöglicht es dennoch die Berücksichtigung eines umfangreichen Sortiments an Sensoren und alternativen Kontextressourcen. Wenngleich Ayed et al. ihr Metamodell auch einsetzen, um Verhaltensaspekte einer Software zu modellieren, erweitert das UML-Profil keine derjenigen UML-Metaklassen, die i. d. R. zur Modellierung von Verhaltensaspekten von Software verwendet werden. Vielmehr zielt das Metamodell nach Ayed et al. darauf ab, solche Modelle zu erstellen, die in MDSD-Prozessen zur Generierung von Software verwendet werden, nämlich UML-Klassendiagramme, nicht jedoch solche, die das Verhalten der Software aus der Perspektive des Anwenders abbilden. Deshalb kann das Metamodell nach Ayed et al. nicht auf die Generierung kontextsensitiver Testfälle übertragen werden. Doch auch zu diesem Zweck stellen sich Anforderungen an ein Metamodell, die den von Ayed et al. diskutierten ähnlich sind. Relevant für diese Dissertation ist die Arbeit von Ayed et al. vor allem, weil die Autoren diejenigen Phasen des Softwareprozesses explizit herausstellen, die Alleinstellungsmerkmale für kontextsensitive Anwendungen sind.

Hierzu gehört insbesondere die Identifikation relevanter Kontextparameter und die Bestimmung von deren Einfluss auf das Laufzeitverhalten einer Anwendung. In Abgrenzung zu den in den vorherigen Abschnitten diskutierten Konzepten allerdings mit einem Fokus auf MDSD-Technologien und nicht auf MDT-Technologien. In einem sechs Phasen umfassenden Prozess (vgl. Abschnitt 2.2.2) sollen nach dem von Ayed et al. vorgestellten Ansatz kontextsensitive Anwendungen mit MDSD-Werkzeugen aus Modellen generiert werden. Teil dieses Prozesses ist es, Komponenten einer Anwendung zu identifizieren, in denen Kontextsensitivität implementiert wird. Hierzu erstellen die Autoren ein UML-Profil, welches Stereotypen definiert, die Kontextparameter in Softwaremodellen auszeichnen. Allerdings legen die Autoren einen engen Fokus auf kontextsensitive Variabilitätspunkte in Anwendungen. Ziel ist es, im Systemmodell zu veranschaulichen, an welcher konkreten Stelle in einer Anwendung konkrete Kontextparameter zu berücksichtigen sind. Ein Objektmodell, welches die Definition konkreter Parameter, wie beispielsweise eine geographisch regionale Gültigkeitsbeschränkung in ein Softwaremodell integrieren kann, ist mit dem von Ayed et al. vorgestellten Metamodell allerdings nicht möglich.

Ein weiteres UML-basiertes Metamodell zur Modellierung von Kontext wird von Vieira et al. [353] vorgeschlagen. Die Autoren entwerfen eine Erweiterung für UML-Aktivitätsdiagramme, die es erlaubt, Kontext in Verhaltensdiagrammen zu modellieren. Das von den Autoren vorgeschlagene Metamodell ist allerdings auf ein aktivitätszentriertes Kontextkonzept (vgl. Prekop und Burnett [287]) ausgelegt und berücksichtigt die Vielfalt physikalischer Kontextparameter nicht. Auf das in dieser Dissertation vorgestellte Konzept der Testautomatisierung mobiler, kontextsensitiver Anwendungen ist das UML-Profil nach Vieira et al. deshalb nicht übertragbar. Weiterhin zielt das von Vieira et al. vorgeschlagene UML-Profil nicht auf diejenigen Modellierungselemente von UML-Aktivitätsdiagrammen ab, die in dieser Dissertation als maßgeblich für die Abbildung von Testfällen eingestuft werden.

In einem dem in dieser Dissertation vorgestellten Konzept zur Testautomatiserung naheliegenden Ansatz untersucht auch Decker [89] eine auf UML-Aktivitätsdiagrammen basierende Methode, um Kontextmodellierung direkt in ein Softwaremodell zu integrieren. Mit einem Fokus auf Standortinformationen entwerfen die Autoren ein rudimentäres Kontextmetamodell, mit dessen Hilfe Standortinformationen auf unterschiedlichen Abstraktionsebenen modelliert werden können. Weiterhin ergänzen die Autoren das UML-Metamodell für Aktivitätsdiagramme in Form eines UML-Profils, um Modellierungselemente die es zulassen, standortabhängige *Contraints* (engl. Zwangsbedingungen) in ein Modell einzuführen. Grundsätzlich wäre dieser Ansatz auf den in dieser Dissertation untersuchten Anwendungsfall übertragbar. Allerdings berücksichtigt das von Decker erstellte Kontextmetamodell neben dem Standort keine anderen Kontextparameter. Zudem wird durch den verwendeten Syntax ein zu den UML-Guard-Conditions paralleles Modellierungselement geschaffen, so dass durch diese Ambiguität zusätzliche Komplexität in die Modellierung eingeführt wird.

Zusammengefasst kann festgestellt werden, dass in der wissenschaftlichen Literatur Konsens darüber besteht, dass Kontextsensitivität eine Eigenschaft von Software ist, die durch ihre Wurzeln in den konkreten Anforderungen eines Geschäftsprozesses nicht zu einem späteren Zeitpunkt zu einem Softwaresystem hinzugefügt werden kann, sondern stattdessen bereits während des initialen Systementwurfs berücksichtigt werden muss und sich in der Konsequenz

auch in den Modellen des Systementwurfs manifestieren sollte. Zahlreiche Forschungsarbeiten haben eine große Anzahl unterschiedlicher Ansätze und Metamodelle hervorgebracht, Kontextsensitivität in der Systemmodellierung zu berücksichtigen. Hierbei werden unterschiedliche Abstraktionsebenen adressiert, die das gesamte Spektrum von einer vagen Indikation über die mögliche Relevanz von Kontextparametern bis hin zu konkreten Kontextentitäten und prototypischen Werten abdecken.

Unter den existierenden Metamodellen zur Kontextmodellierung konnte keines identifiziert werden, das die Anforderungen an den in dieser Dissertation vorgestellten Ansatz zur Generierung und Automatisierung von Tests für kontextsensitive Anforderungen erfüllt. Um bei der automatisierten Durchführung von Tests Kontextparameter durch eine Simulationstechnologie zu berücksichtigen, muss aus dem für die Testgenerierung verwendeten Modell hervorgehen, an welchen Komponenten, Aktivitäten oder Fallunterscheidungen Kontextparameter einfließen, um den weiteren Kontroll- oder Datenfluss zu steuern. Für die Generierung von Tests bedarf es weiterhin einer Spezifikation von Testdaten, die während der Durchführung als Eingabeparameter bzw. als Parameter zur Kontextsimulation verwendet werden können.

Zur Bereitstellung eines hierzu geeigneten Metamodells werden in Abschnitt 5.2.2.1 zunächst Anforderungen an ein Metamodell zur Kontextmodellierung für die Integration in Systemmodelle zur automatisierten Generierung von Tests diskutiert. Abschnitt 5.2.2.2 stellt das in dieser Dissertation verwendete Metamodell zur Kontextmodellierung vor.

5.2.2.1 Anforderungen an ein Metamodell zur Kontextmodellierung

Die Untersuchungen im vorangegangen Abschnitt haben ergeben, dass die Modellierung von Kontext in Softwaresystemen eine Voraussetzung ist, diese Informationen in MDSD- und MDT-Technologien nutzbar zu machen. Das hier untersuchte Konzept der modellbasierten Generierung von Testfällen und deren automatisierte Ausführung vereinigt Konzepte des MDSD und des MDT. Es handelt sich um einen MDSD-Ansatz, weil aus Modellen von Softwaresystemen durch sukzessive Verfeinerung des Modells und Fokussierung auf eine adäquate Abstraktionsebene ausführbare Code-Artefakte erzeugt werden. Es ist ebenfalls ein MDT-Ansatz, weil es sich bei diesen Code-Artefakten nicht um Code der eigentlichen Software handelt, sondern um Code, in dem sich Tests des modellierten Systems manifestieren.

Im Fokus dieser Arbeit steht die Testautomatisierung mobiler, kontextsensitiver Anwendungen. Deshalb liegt insbesondere auf dem Aspekt der Lesbarkeit sowohl durch Menschen als auch Maschinen besonderes Gewicht. Systemmodelle werden zur Entwurfszeit von menschlichen Akteuren erstellt. Es ist daher einerseits notwendig, dass die Komplexität der Berücksichtigung von Kontextparametern in Entwurfsaktivität durch Menschen beherrschbar bleibt. Andererseits sollen diese Informationen in späteren Phasen des Softwareprozesses maschinell verarbeitet werden, insbesondere zur automatisierten Generierung und Durchführung von Tests. Es ist demnach notwendig, Kontextparameter in Formaten in die Systemmodellierung einfließen zu lassen, die einer maschinellen Verarbeitung nicht entgegenstehen.

Um Kontextparameter im Sinne der in dieser Dissertation verwendeten Definition (vgl. Abschnitt 3.1.1) in die Modellierung eines Softwaresystems zu integrieren, bedarf es eines adäquaten Metamodells, an das sich eine Reihe von Anforderungen stellen, um das in dieser Dissertation verfolgt Ziel zu unterstützen. In den folgenden Abschnitten werden Anforderun-

gen an ein solches Metamodell definiert und im Bezug zur Verwendung im hier untersuchten MDSD-/MDT-Konzept zur automatisierten Testfallerstellung diskutiert.

Grundsätzlich kann zwischen fachlich-funktionalen und technischen Anforderungen unterschieden werden. Fachlich-funktionale Anforderungen beschreiben solche Eigenschaften von Modellen, die zur Abbildung fachlich-funktionaler Aspekte eines Softwaresystems im Modell erforderlich sind. Als Teil der Systemspezifikation beschreiben diese, welche Funktionen eines Softwaresystems kontextsensitiv sind und wie Kontextparameter das Systemverhalten beeinflussen. Die technischen Anforderungen an Kontextmodelle sind solche, die einerseits sicherstellen, dass Kontextinformationen unter Verwendung standardisierter Methoden und Werkzeuge in die Systemmodellierung integriert werden können. Andererseits sollen sie ebenfalls sicherstellen, dass die zur Entwurfszeit im Modell spezifizierten Kontextparameter während aller Phasen des Softwareprozesses sowohl durch Menschen als auch durch Maschinen aus dem Modell gelesen werden können.

Um dem Einsatzzweck von Kontextmodellen für Entwurfs- und Testaktivitäten gerecht zu werden, werden im Folgenden eine Reihe von Anforderungen an Kontextmodelle definiert.

REQ-MM-Kontext 1: Generizität

> Modellierungssprachen des Software Engineering sind für den Entwurf von Softwaresystemen geeignet, wenn sie eine Menge von Modellierungselementen bereitstellen, die es ermöglichen, vielfältige fachliche Anforderungen abzubilden, ohne hierbei auf eine bestimmte Anwendungsdomäne festgelegt zu sein. Die Entwicklung mobiler, kontextsensitiver Systeme ist zwar insofern auf eine Domäne festgelegt, als dass sie eine bestimmte Klasse von Anwendungen zum Gegenstand hat, die durch ihre besonderen Eigenschaften gegenüber Desktop- und Serveranwendungen abgegrenzt werden können. Inhaltliche Aspekte dieser Anwendungen werden hierdurch jedoch nicht eingeschränkt. Von einem Metamodell einer Kontextmodellierungssprache wird gefordert, Kontextaspekte losgelöst von der spezifischen inhaltlichen Anwendungsdomäne darstellen zu können (vgl. Castelli et al. [73], Baldauf et al. [25]).
>
> Auch Broens und van Halteren [59] sowie Korpipää und Mäntyjärvi [229] identifizieren Generizität als Kernanforderung an Kontextmodelle, wobei Broens und van Halteren insbesondere eine Technologie zur Simulation von Kontext während des Testens untersuchen, Korpipää und Mäntyjärvi hingegen eine Kontextontologie zur Kontextanalyse während der Laufzeit einer Software. Generizität ist demnach eine universelle Anforderung an ein Metamodell, die auch außerhalb der spezifischen Anwendung zur Kontextmodellierung von Modellierungssprachen gefordert und gewährleistet wird, beispielsweise der UML.

REQ-MM-Kontext 2: Unvollkommenheit

> Henricksen et al. [178] und Bettini et al. [42] diskutieren in ihren Forschungsarbeiten zu Kontextmodellierung die Anforderung an ein Metamodell, Unvollkommenheit zuzulassen. Diese Unvollkommenheit kann sich zweigestalt manifestieren.

Einerseits muss es möglich sein, Kontext unvollkommen im Sinne der Fehlerhaftigkeit modellieren zu können. Dies ist beispielsweise bei Standortinformationen zweckmäßig, die durch GPS-Technologie oder andere Lokalisierungstechnologien nie mit absoluter Präzision vorliegen, sondern stets fehlerbehaftet sind.

Andererseits muss es möglich sein, Kontext unvollkommen im Sinn der Unvollständigkeit zu modellieren. Der Kontext einer mobilen Software kann sich aus unzähligen Parametern zusammensetzen, von denen i. d. R. nur eine geringe Anzahl tatsächlich für einen Anwendungsfall relevant sind.

Von einem Metamodell zur Kontextmodellierung wird daher gefordert, bei der Modellierung Raum für Unvollkommenheit zu gewähren.

REQ-MM-Kontext 3: Lesbarkeit durch Menschen

Modelle haben vielfältige Aufgaben, insbesondere dienen sie als Kommunikationsmittel zwischen Fachexperten und Anwendungsentwicklern, indem sie erlauben, Semantik auf einer technikfernen Abstraktionsebene zu repräsentieren (vgl. Stachowiak [328]). Ebenfalls ermöglichen Modelle die Darstellung von Sachverhalten unter Abstraktion von der Zieltechnologie. Diese Abstraktion von konkreter Technologie wird durch graphische Darstellungsformen, i. d. R. Diagramme, erreicht.

Modelle werden i. d. R. von menschlichen Akteuren erstellt. Hieraus ergibt sich unmittelbar die Anforderung an ein Metamodell zur Kontextmodellierung, dass Modelle in einer Form repräsentiert werden müssen, die direkt durch Menschen gelesen und bearbeitet werden kann. In Einklang mit der Anforderung REQ-MM-Kontext 6 (Integrationsfähigkeit in die UML) wird daher gefordert, dass Kontextmodelle durch eine graphische Notation repräsentiert werden können.

REQ-MM-Kontext 4: Lesbarkeit durch Maschinen

Die Generierung von Testfällen aus Systemmodellen kann nur dann automatisiert erfolgen, wenn diese Modelle maschinell verarbeitet werden können. Es wird deshalb an ein Metamodell zur Kontextmodellierung die Anforderung gestellt, Modellelemente so zu repräsentieren, dass sie algorithmisch durch Maschinen (d. h. Computer) gelesen, geschrieben, interpretiert und verarbeitet werden können.

REQ-MM-Kontext 5: Objektrelationale Modellierung

Die Analyse wissenschaftlicher Literatur hat ergeben, dass jüngere Ansätze zur Modellierung von Kontext i. d. R. objektrelationale Modelle sind [42, 336]. Deren Vorteile gegenüber Schlüssel-Wert Paaren wurden im vorangegangen Abschnitt diskutiert. Im Gegensatz zu Schlüssel-Wert Paaren sind objektrelationale Modelle besser geeignet, Objekte der realen Welt auf einer intuitiven Abstraktionsebene abzubilden.

Von einem Metamodell zur Kontextmodellierung wird deshalb gefordert, Kontextparameter objektrelational abzubilden. Hierdurch wird ebenfalls die Grundlage für REQ-MM-Kontext 6 bereitet. Im konkreten Anwendungsfall ist es dann möglich, eine Testumgebung als Objektmodell konkreter Kontextobjekte zu modellieren, die einem SUT während der Testdurchführung durch eine Simulationstechnologie zugeführt werden.

REQ-MM-Kontext 6: Integrationsfähigkeit in die UML

In dieser Dissertation wird ein Konzept zur Generierung von Testfällen aus Systemmodellen vorgestellt, mit dem Ziel den Aufwand für die Erstellung von Tests für mobile, kontextsensitive Anwendungen zu reduzieren. Zur Herabsetzung von Akzeptanzhürden ist es zweckmäßig, hierbei solche Modellierungssprachen und Technologien zu adressieren, die zum Kanon der etablierten Werkzeuge des Software Engineering gehören. Eine ähnliche Anforderung wurde bereits von Bettini et al. [42] und Henricksen et al. [178] mit einer nahezu identischen Begründung formuliert.

Eines dieser Werkzeuge ist die Modellierungssprache UML, die ebenfalls in anderen Technologien des MDSD und MDT zur Anwendung kommt. Sie ermöglicht es, durch UML-Profile, das UML-Metamodell zu erweitern und um Fachkonzepte zu ergänzen. Von einem Metamodell zur Modellierung von Kontext in Systemmodellen wird gefordert, dass die unter Verwendung dieses Metamodells erstellten Kontextmodelle in UML-Modelle integriert werden können.

REQ-MM-Kontext 7: Kontextparameter Zeit

In Abschnitt 3.1.2.1.2 wurde der Kontextparameter Zeit im Detail diskutiert. Dieser Kontextparameter tritt in der praktischen Verwendung zweigestalt auf. Einerseits wird Zeit explizit durch Angabe von Datum und Uhrzeit in Softwareanwendungen verwendet, andererseits kann aber auch ein anderer Kontextparameter als Substitut für Zeit oder einen konkreten Zeitpunkt verwendet werden, z. B. ein Ort. Der absolute Zeitpunkt einer Substitution einer Zeitangabe durch eine Ortsangabe wird dann durch das Eintreffen oder Verlassen dieses Orts bestimmt. An ein Metamodell zur Kontextmodellierung wird deshalb die Anforderung gestellt, sowohl explizite Zeitangaben als auch äquivalente Substitute abbilden zu können. Wie im speziellen Fall mit einer solchen Substituierung verfahren wird, ist Gegenstand der Implementierung einer Testautomatisierungslösung. Diese könnte beispielsweise ein Substitut erkennen und die Kontextsimulationsumgebung anweisen, anstelle einer Modifikation der Uhrzeit auf dem ausführenden Emulator oder Gerät entsprechende Lokalisierungsinformationen bereitzustellen.

Da Zeitangaben in Software häufig als Zeitintervalle verwendet werden, wird zusätzlich die Anforderung gestellt, neben konkreten Zeitpunkten ebenfalls Zeitintervalle abbilden zu können. Beim Testen einer Anwendung, die den Kontextparameter

Zeit verwendet, kann dann eine Menge individueller Testfälle generiert werden, die nicht nur die Randbereiche des Zeitintervalls abdecken, sondern auch individuelle Zeitpunkte innerhalb des Intervalls in einer definierten Auflösung.

REQ-MM-Kontext 8: Kontextparameter Standort (statisch, geodätisch)

In Abschnitt 3.1.2.1.3 wurden grundlegende Eigenschaften von Standortinformationen diskutiert. Hierbei wurde das WGS84 als Standardreferenzsystem zur Abbildung geodätischer Koordinaten identifiziert. In den APIs der gegenwärtig in Smartphone- und Tablet-Computer-Plattformen implementierten Technologie bildet das WGS84 die Grundlage zur Verarbeitung von Standortkoordinaten. Hierzu stellen APIs der SDKs mobiler Plattformen Klassen bereit, die wesentliche Merkmale von WGS84-Koordinaten abbilden. Von Metamodellen zur Integration von Kontextinformationen in Systemmodellen wird gefordert, dass Standorte so modelliert werden können, dass die einzelnen Bestandteile eines WGS84-Datums darstellbar sind. Eine Anforderung an Kontextmodelle, den Standort von Komponenten oder Akteuren abbilden zu können, wurde bereits von Roman et al. [303], Henricksen et al. [178] sowie Bettini et al. [42] formuliert, dort allerdings ohne expliziten Bezug zu WGS84-Koordinaten.

In der Praxis ist es häufig nicht ausreichend, auf ein einzelnes Koordinatenpaar als geografische Referenz zurückzugreifen. Lokalisierungstechnologie ist nicht ausreichend präzise, um zu gewährleisten, dass die durch das GPS-Modul eines Smartphones gelieferten Daten exakt dem tatsächlichen Standort des Geräts entsprechen. Ein Modellierungselement muss deshalb Ungenauigkeit abbilden können. Diese Anforderung geht ebenfalls auf eine von Henricksen et al. [178] und Bettini et al. [42] formulierte Anforderung zur Abbildbarkeit von Unvollkommenheit in Kontextmodellen zurück.

Hierzu bietet es sich an, diese Ungenauigkeit in Übereinstimmung mit der technischen Implementierung der Lokalisierungs-API in mobilen Plattformen als Kreisradius zu definieren, innerhalb dessen der tatsächliche Standort des mobilen Geräts mit einer bestimmten Wahrscheinlichkeit vermutet wird. Die Größe des Kreisradius wird in der Praxis durch die Anzahl und Qualität der Satellitensignale bestimmt, während die Wahrscheinlichkeit der Korrektheit der Standortinformation durch die Bauart des GPS-Moduls bestimmt wird.

Zur Abbildung einzelner Koordinatenpaare wird vom Metamodell gefordert, Lokalisierungsinformationen mit den konkreten Werten für geographische Breite, geographischen Länge, Höhe über dem mittleren Meeresspiegel und einer Präzision abzubilden (d. h. die Bestandteile des WGS84-Formats) abzubilden.

REQ-MM-Kontext 9: Kontextparameter Standort (statisch, symbolisch)

Für Anwendungen, die eine Positionsbestimmung innerhalb von Gebäuden ermöglichen, ist eine Ortsbestimmung unter Verwendung von GPS-Technologie i. d. R.

nicht möglich. Darüber hinaus existieren eine Reihe von Anwendungen, in denen eine geodätische Standortbestimmung nicht erwünscht oder nicht zweckmäßig ist (Vergleich Abschnitt 3.1.2.1.3). Aus diesem Grund wird von einem Metamodell zur Kontextmodellierung gefordert, dass es Modellelemente zur Modellierung symbolischer Standortinformationen bereitstellt.

Mobile Plattformen verfügen über mehrere Technologien zur Bestimmung des Standorts des Geräts (vgl. Abschnitt 3.1.2.1.3), die unabhängig von der tatsächlich verwendeten Signalquelle WGS84-Koordinaten erzeugen, sofern eine absolute Positionsbestimmung Inhalt des Anwendungsfalls ist. Für diesen Fall wird von einem Metamodell zur Kontextmodellierung gefordert, symbolische Standortinformationen auf geodätische abbilden zu können.

Darüber hinaus können mobile Anwendungen den Standort des Geräts auch aus alternativen Informationen schlussfolgern, z. B. der Gegenwart von Wi-Fi-Netzwerken mit bestimmten SSIDs (engl. *Service Set Identifier* (SSID), Konfigurationsmerkmal eines Wi-Fi-Netzwerks). Vom Metamodell zur Kontextmodellierung wird deshalb nicht gefordert, solche symbolischen Standortinformationen abzubilden, die nicht auf geodätische Koordinaten zurückgeführt werden können.

Ein intuitives Substitut für WGS84-Koordinaten zur Modellierung symbolischer Standortinformationen ist eine Standortspezifikation als Postanschrift, welche ihrerseits einem strukturierten, standardisierten Aufbau folgt (z. B. ISO/IEC [200]). Unter Abstraktion von Informationen wie Name des Adressaten, Namenszusätzen usw. sind Postanschrift auf WGS84-Koordinaten abbildbar, d. h. jeder Postanschrift kann ein Koordinatenpaar oder eine polygonale Region zugeordnet werden.

Es ist deshalb Anforderung an ein Metamodell zur Kontextmodellierung symbolischer Standortinformationen, Postanschriften abbilden zu können. Hierdurch entsteht zugleich die Anforderung an eine Technologie zur Simulation von Kontext beim Testen, diese Postanschrift zu WGS84-Koordinaten zu decodieren.

REQ-MM-Kontext 10: Kontextparameter Standort, Regionen

Mobile, kontextsensitive Anwendungen können Anwendungsfälle implementieren, in denen nicht nur der absolute Standort des Anwenders relevant ist, sondern Menge gültiger Standorte Einschränkungen unterliegt oder die Annäherung an einen bestimmten Ort oder eine Region in sonstiger Weise für den Anwendungsfall relevant ist (z. B. Golf GPS & Scorecard [338], die Beispielanwendungen Mobiler Taxiruf, AR Tourist Information, Abbildung 5.3). Diese als Geofencing bezeichnete Funktion operiert mit einer regionalen Begrenzung zulässiger oder relevanter Standorte.

Die Definition einer Region kann auf unterschiedliche Arten erfolgen. Gebräuchlich ist die Spezifikation einer kreisförmigen Region durch ein WGS84-Koordinatenpaar und eines Radius. Durch Kombination mehrerer Kreisflächen können komplexere Regionen definiert werden. Alternativ können Regionen ebenfalls polygonal

begrenzt werden. Hierzu werden die Ecken des Polygons jeweils durch WGS84-Koordinatenpaare bestimmt.

An ein Metamodell zur Kontextmodellierung wird zur Unterstützung von Geofencing die Anforderung gestellt, Modellierungselemente zur Abbildung begrenzter geographischer Regionen anzubieten. Diese Information kann sowohl während der Entwicklung einer mobilen (ggf. ortsbasierten) Anwendung als Teil der Anwendungsspezifikation verwendet werden als auch während des Testens. Eine Automatisierungstechnologie kann die Spezifikation der geographisch begrenzten Fläche aus dem Modell extrahieren und Testdaten in der gewünschten Anzahl und Auflösung generieren, um eine Anwendung mit geringem Aufwand unter einer Vielzahl unterschiedlicher Standortinformationen zu testen, auch in Kombination mit REQ-MM-Kontext 11 und REQ-MM-Kontext 12.

REQ-MM-Kontext 11: Kontextparameter Standort (dynamisch, geodätisch)

In Abschnitt 3.1.2.1.3 wurde diskutiert, dass Standortangaben statisch oder dynamisch sein können. Als statische Standortangaben wurden solche definiert, die sich auf in Ruhe befindliche Entitäten beziehen oder für die Weiterverarbeitung nur der konkrete Standort zum Zeitpunkt der Messung interessant ist.

Mobile, kontextsensitive Anwendungen können auch in Mobilitätsszenarien verwendet werden, in denen sich Gerät und Anwender in Bewegung befinden (*In-motion Device*, *In-motion User*, vgl. Book et al. [54, 53, 55]) und diese Bewegung für den Anwendungsfall relevant ist. Für diesen Fall wird von einem Metamodell zur Kontextmodellierung gefordert, die Änderung des Standorts des Geräts bzw. des Anwenders während eines definierten Zeitintervalls abbilden zu können.

Mobile Plattformen erfahren Mobilität des Anwenders als Veränderung des Standorts zwischen zwei diskreten Erfassungszeitpunkten, d. h. zwei aufeinanderfolgende API-Aufrufe erzeugen unterschiedliche Standortinformationen. Für das Metamodell ist es daher ausreichend, eine Abfolge aufeinanderfolgender Standortinformationen gemäß REQ-MM-Kontext 8 in einem definierten Zeitabstand darstellen zu können. Eine Technologie zur Simulation von Kontext während der Testausführung kann konkrete Standortinformationen interpolieren.

REQ-MM-Kontext 12: Kontextparameter Standort (dynamisch, symbolisch)

REQ-MM-Kontext 11 gilt analog für symbolische Standordinformationen unter Berücksichtigung der Einschränkung aus REQ-MM-Kontext 9.

REQ-MM-Kontext 13: Kontextparameter Orientierung (statisch)

Mobile Geräte sind bauartbedingt dazu geeignet, während der Verwendung frei bewegt und orientiert zu werden (vgl. Abschnitt 3.1.2.2.1).

Die Bestimmung der Orientierung des Geräts basiert auf der Fusion der Informationen unterschiedlicher Sensoren. Als Abstraktion von der Modellierung individueller Sensordaten (vgl. REQ-MM-Kontext 16) wird von einem Metamodell zur Kontextmodellierung gefordert, eine reellwertige beliebige Orientierung (d. h. beliebige Werte für Roll-, Nick- und Gierwinkel) durch ein geeignetes Modellierungselement zu ermöglichen.

REQ-MM-Kontext 14: Kontextparameter Orientierung (statisch, diskret)

Um eine Anpassung des UI einer Anwendung an die jeweilige Orientierung zu gewährleisten, wird die Orientierung eines mobilen Geräts häufig in vier diskrete Klassen eingeteilt: (1) Hochformat, (2) Hochformat auf dem Kopf stehend, (3) Querformat links und (4) Querformat rechts (vgl. Abschnitt 3.1.2.2.1, Abbildung 3.8). Eine Einteilung in vier Klassen ist deshalb sinnvoll, weil viele mobile Geräte bauartbedingte Eigenschaften aufweisen, die im Hochformat eine Seitenkante als *oben* und eine als *unten* auszeichnen (z. B. Position der Kamera oder des Mikrofons). Analog gilt im Querformat, dass eine Seitenkante als *links* (entspricht *oben* im Hochformat) und eine als *rechts* ausgezeichnet wird. Für die Realisierung vieler Anforderungen ist eine solche Unterscheidung nicht sinnvoll. Allerdings gibt es Anwendungen, die eine bestimmte Orientierung des Geräts fordern, beispielsweise Videoaufzeichnung.

Als Abstraktion von reellwertig beliebiger Orientierung (d. h. beliebige Werte für Roll-, Nick- und Gierwinkel) wird von einem Metamodell zur Kontextmodellierung gefordert, eine Modellierung der Geräteorientierung in diesen vier Klassen zu ermöglichen.

Der technische Hintergrund dieser Anforderung ist, dass die Bestimmung der Orientierung des Geräts auf der Fusion der Informationen unterschiedlicher Sensoren basiert. Eine Festlegung einer bestimmten Orientierung erfordert daher die Auswahl und Simulation von Sensordaten, beispielsweise für Beschleunigungssensor und Magnetfeldsensor. Eine Modellierung individueller Sensorzustände würde gegenüber der Modellierung einer diskreten Geräteorientierung einen erhöhten Modellierungsaufwand erzeugen.

REQ-MM-Kontext 15: Kontextparameter Orientierung (dynamisch)

Analog zu REQ-MM-Kontext 11 und REQ-MM-Kontext 12 wird von einem Metamodell zur Kontextmodellierung gefordert, Modellierungselemente bereitzustellen, mit denen eine Veränderung der Geräteorientierung während eines Zeitintervalls modelliert werden kann. Hierdurch soll abgebildet werden, dass Roll-, Nick- und Gierwinkel eines mobilen Geräts innerhalb eines vorgegebenen Zeitintervalls von einem initialen Wert auf einen finalen Wert verändert werden, beispielsweise um darzustellen, dass ein Anwender das Gerät vom Hochformat ins Querformat überführt.

Das Metamodell soll jeweils ein Modellierungselement bereitstellen, mit dem sowohl der Wechsel von einer diskreten Orientierung (vgl. REQ-MM-Kontext 14) zu einer anderen als auch ein kontinuierliche Veränderung der Orientierung beschrieben werden kann.

In einem Systemmodell kann der Entwickler diese Informationen verwenden, um einen entsprechenden Algorithmus für einen Anwendungsfall zu implementieren, in dem eine Anwendung auf die Veränderung der Geräteorientierung reagiert. Für die hier untersuchte Methode der Testautomatisierung kann diese Information von einer Kontextsimulationstechnologie verwendet werden, um während der Testausführung die Werte einzelner Sensoren entsprechend der modellierten Geräteorientierung einzustellen.

REQ-MM-Kontext 16: Modellierung von Sensordaten (statisch)

Mobile Geräte der Smartphone-Generation verfügen über unterschiedlicher Sensoren, die neben der Berechnung der Orientierung des Geräts für unterschiedliche Zwecke verwendet werden können. Hierzu zählen u. a. Beschleunigungssensor, Magnetfeldsensor, Temperatursensor, Helligkeitssensor, Annäherungssensor, Luftfeuchtigkeitssensor.

Sensoren erzeugen im vorgegebenen Zeitintervallen Daten-Tupel, die je nach spezifischem Sensor unterschiedliche Länge haben. Magnetfeldsensor und Beschleunigungssensor erzeugen beispielsweise ein Feld aus drei Fließkommawerten, die den jeweiligen Messwert entlang den Achsen eines Referenzsystems (d. h. das Koordinatensystem des Geräts) repräsentieren. Ein Sensor zur Messung der Umgebungstemperatur hingegen erzeugt i. d. R. nur einen einzelnen Wert.

Von einem Metamodell zur Kontextmodellierung wird die Bereitstellung eines Modellierungselements gefordert, mit dem Sensoren abgebildet werden können. Ziel ist es, in einem Systemmodell die Verwendung oder Abhängigkeit einer Komponente oder eines Moduls von den Messwerten eines Sensors oder mehreren Sensoren darzustellen. Bei der Generierung von Testfällen aus Systemmodellen kann es verwendet werden, um Testdaten zu modellieren, die während der Testdurchführung von einer Kontextsimulationstechnologie in ein SUT eingespeist werden sollen.

REQ-MM-Kontext 17: Modellierung von Sensordaten (dynamisch)

Sensoren erzeugen einen Strom von Messwerten, die in festgelegten Zeitintervallen allen Abonnenten im System übergeben werden. Insbesondere wenn sich der Kontext des mobilen Geräts, d. h. eine Eigenschaft der Betriebsumgebung, verändert, erzeugt ein Sensor aufeinanderfolgend unterschiedliche Messwerte.

Dreht der Anwender ein Gerät beispielweise vom Querformat ins Hochformat, verändern sich die Messwerte entlang der drei Achsen des Referenzsystems des Geräts entsprechend der Orientierung des Geräts, bis sie am Ende der Drehung mit neuen Werten stabil werden (vorbehaltlich sensorspezifischen Rauschens).

Von einem Metamodell zur Kontextmodellierung wird die Bereitstellung eines Modellierungselements gefordert, mit dem Veränderungen von Sensorenmesswerten nach der Zeit abgebildet werden können.

Im Rahmen der Verwendung des Kontextmodells zur Testgenerierung können hieraus Testdaten interpoliert werden, die einen Testfall für eine sensorbasierte Anwendung repräsentieren.

REQ-MM-Kontext 18: Modellierung von Sensordaten (Rauschen)

Sensoren erzeugen grundsätzlich Rauschen, welches einerseits die maschinelle Interpretation von Sensordaten erschwert, weil durch Anwendung geeigneter Filter Rauschen aus dem Signal entfernt werden muss und andererseits eine Interpretation der Daten für menschliche Akteure nahezu unmöglich ist, weil diese i. d. R. nicht in der Lage sind den Anteil des Rauschens am Signal einzuschätzen.

Die Entfernung von Rauschen ist ein wesentlicher Teil der Verarbeitung von Sensordaten und sollte deshalb intensiv getestet werden. Hierzu wiederum ist es erforderlich, reproduzierbare Testdaten zu erzeugen.

Von einem Metamodell zur Kontextmodellierung wird deshalb gefordert, in einem Modellierungselement zur Modellierung von Sensorinformationen den Einfluss von Signalrauschen modellieren zu können. Es können dann Testfälle erzeugt werden, die die Qualität rauschunterdrückender Filter in einer Anwendung zum Gegenstand haben.

Auf Basis dieser Anforderungen wird im folgenden Abschnitt 5.2.2.2 ein Metamodell entworfen, mit dem Kontextparameter in Systemmodelle integriert werden können. Ziel ist es nicht, ein Kontextmodell zu entwerfen, welches durch die Logik einer Anwendung zur Interpretation ihres Kontext verwendet werden soll. Es ist dazu bestimmt, Kontextsensivität bereits bei der Systemmodellierung auf Modellebene zu verankern. Auf eine begleitende textuelle Beschreibung des Kontext, relevanter Kontextparameter und deren Einfluss auf das System kann dann verzichtet werden. Hieraus ergibt sich der Vorteil, dass alle Faktoren, die ein Softwaresystem beeinflussen, auf einer einheitlichen Ebene dargestellt werden können. Weiterhin können hierdurch Medienbrüche reduziert werden, wodurch das Kontextmodell einer automatisierten Verarbeitung durch Softwarewerkzeuge zugänglich wird.

Die im Vorfeld der Erstellung dieser Anforderungen untersuchte Literatur (z. B. Bettini et al. [42], Broens und van Halteren [59], Vieira et al. [352], Morla und Davies [256], Decker [89], Ridene und Barbier [296]) formuliert für solche Kontextmodelle, die eine algorithmische Interpretation der Nutzungssituation einer mobilen Anwendung unterstützen sollen, die Anforderung der maschinellen Schlussfolgerung abstrakten Kontexts aus niederen Detailinformationen. Das hier untersuchte Konzept zur Testautomatisierung mit Kontextsimulation stellt diese Anforderung nicht. Ziel ist es hier, den Kontext einer Anwendungsnutzung im Rahmen des Testens zu simulieren. Hierzu ist ein Objektmodell ausreichend, mit welchem Testdaten auf einer für einen menschlichen Akteur fassbaren Ebene modelliert werden können.

In Abschnitt 3.1.2.1.1 und Abschnitt 4.2.1.1 wurden die Kontextparameter Plattform und Gerät und deren Auswirkungen auf den Betrieb und das Testen von Software diskutiert. Diese Eigenschaften sind überwiegend statisch, d. h. sie verändern sich nicht zur Laufzeit einer Anwendung. Ausnahmen sind lediglich Ressourcen des Geräts, die in gegenseitiger Beeinflussung mit der Ausführung von Software stehen, wie etwa freie Kapazität des Arbeitsspeichers oder die freie Kapazität eines Massenspeichermediums. Die übrigen Parameter wie etwa Displaygröße oder -auflösung, Bauform, Version des Betriebssystems usw. ändern sich zur Laufzeit i. d. R. nicht oder sind Ausnahmesituationen von solcher Signifikanz für das Gesamtsystem, dass sie in einem Softwaretest nicht adressiert werden müssen. Eine Aktualisierung des Betriebssystems beispielsweise zieht immer einen Neustart des Geräts nach sich, welcher die Ausführung einer App zwingend terminiert.

In der Konsequenz ist es nicht sinnvoll, das Verhalten einer App bei spontaner Veränderung solcher Kontextparameter zu überprüfen. Vielmehr gilt es im Vorfeld der Testausführung zu spezifizieren, welche Tests auf welchen Geräte- bzw. Emulatorkonfigurationen auszuführen sind. Es ist daher keine Anforderung an ein Metamodell zur Kontextmodellierung, Modellierungselemente zur Abbildung von Eigenschaften von Plattform und Gerät anzubieten. Von einem Testautomatisierungswerkzeug wird hingegen gefordert, hinsichtlich der Menge der Zielgeräte oder Emulatoren für einen Test konfigurierbar zu sein.

5.2.2.2 Metamodell zur Kontextmodellierung

In Abschnitt 5.2.2.1 wurden Anforderungen an ein Metamodell zur Beschreibung von Kontextmodellen definiert. Das im Folgenden diskutierte Metamodell erfüllt diese Anforderungen durch Bereitstellung von Modellierungselementen. Es unterstützt die Modellierung von Kontextparametern als Testdaten im Rahmen der Testmodellierung. Die Modellierung von Kontextparametern als Testdaten ist hierbei Teil der Anreicherung von Systemmodellen unter Verwendung des UML-Profils zur Integration der Testfallmodellierung in die Systemmodellierung, die detailliert in Abschnitt 5.2.4 diskutiert wird.

In Abschnitt 5.2.1 wurde die UML als Standardwerkzeug der Modellierung von Softwaresystemen diskutiert. Für den Entwurf des Metamodells zur Modellierung von Standortinformationen wurde die Modellierungssprache UML gewählt, weil somit sichergestellt ist, dass die auf diesem Metamodell basierenden Modelle unter Beibehaltung der UML in die Modellierung anderer Aspekte eines Softwaresystems integriert werden können. Hierdurch werden die Anforderungen REQ-MM-Kontext 3 (Lesbarkeit durch Menschen), Anforderung REQ-MM-Kontext 4 (Lesbarkeit durch Maschinen) sowie Anforderungen REQ-MM-Kontext 5 (Objektrelationale Modellierung) und Anforderungen REQ-MM-Kontext 6 (Integrationsfähigkeit in die UML) erfüllt. Die Lesbarkeit für Menschen ergibt sich aus der graphischen Repräsentation der Modellierungssprache UML, die zu jedem Modellelement eine graphische Darstellungen bereitstellt. Die Lesbarkeit durch Maschinen wird durch standardisierte Datei- und Austauschformate gewährleistet (z. B. *XML Metadata Interchange* (XMI), ein Standardformat zum Austausch von UML-Modellen mit XML-Dokumenten).

In Abbildung Abbildung 5.4 ist das Metamodell zur Kontextmodellierung als UML-Klassendiagramm auf einer hohen Abstraktionsebene verkürzt abgebildet. Das Metamodell definiert ein Modellierungselement *Context::Context* als Wurzelelement eines Kontextmodells.

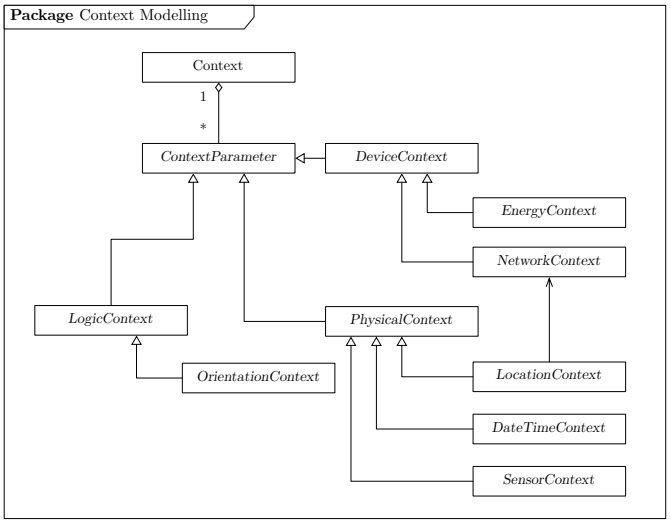


Abbildung 5.4: Metamodell Kontextmodellierung. Aufgrund der begrenzten Darstellungsfläche ist die Abbildung verkürzt, es werden nur die Modellierungselemente der obersten Ebene angezeigt.

Dieses Element dient als Ankerpunkt bei der Integration in ein Systemmodell bzw. zur Modellierung von Kontextparametern als Testdaten. Es aggregiert beliebig viele Elemente des Typs *Context::ContextParameter*[1], die in ihrer Gesamtheit den Kontext repräsentieren. Das Metamodell lässt hier null als Kardinalität zu. Somit wird nicht gefordert, dass in einem Kontextmodell alle Subklassen von *Context::ContextParameter* enthalten sind, wodurch Anforderung REQ-MM-Kontext 2 (Unvollkommenheit) erfüllt wird.

Als direkte Subklassen von *Context::ContextParameter* stellt das Metamodell die Klassen *Context::PhysicalContext*, *Context::DeviceContext* sowie *Context::LogicContext* bereit. Diese Klassen repräsentieren Kontextparameter gemäß der in Abschnitt 3.1.2 definierten Semantik und kapseln logische und physikalische Kontextparameter. Die Klasse der physikalischen Kontextparameter beinhaltet solche Parameter, die von einer Anwendung singulär, z. B. ohne Verknüpfung mit anderen Parametern, verwendet werden können, z. B. Werte einzelner Sensoren. Die Klasse der logischen Kontextparameter bildet solche Parameter ab, die durch Aggregation einzelner Parameter gebildet werden, wie etwa die Geräteorientierung als Aggregat aus Magnetfeldsensor und Beschleunigungssensor. Parameter, die dem System auf andere Weise als mit Sensoren zugeführt werden, z. B. der Status der Netzwerkverbindung, werden durch Subklassen von *Context::DeviceContext* modelliert.

Unterhalb dieser Klassen sind Modellierungselemente definiert, die individuelle Kontextparameter abbilden. Diese werden in den folgenden Abschnitten im Detail erörtert. Die konkrete Auswahl der Modellierungselemente orientiert sich an den Parametern, auf die mobile Plattformen i. d. R. durch Verwendung plattformspezifischer APIs zugreifen können. Besonderes Augenmerk liegt hier auf den Elementen vom Typ *Context::PhysicalContext*.

[1] Die Schreibweise *‹Metamodell›::‹Modellierungselement›* wird verwendet, um Verwechslungen von gleichnamigen Modellierungselementen aus unterschiedlichen Metamodellen zu verhindern.

5.2.2.2.1 Modellierungselement *Context::LocationContext*

In Abschnitt 3.1.2.1.3 sowie in Abschnitt 4.2.3 wurde die Bedeutung des Kontextparameters Standort für den Betrieb und seine Auswirkungen auf das Testen von Softwaresystemen diskutiert. Das Metamodell zur Kontextmodellierung stellt Modellierungselemente bereit, mit denen Standortinformationen in Modellen abgebildet werden können.

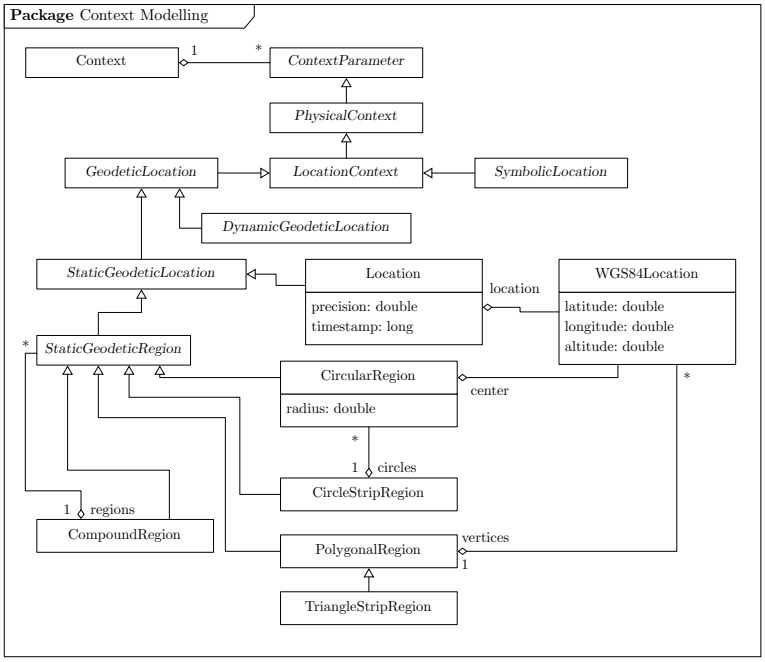


Abbildung 5.5: Metamodell Kontextmodellierung, Ausschnitt *Context::LocationContext*, statische, geodätische Standortdaten

Dargestellt ist in Abbildung 5.5 der Ausschnitt des Metamodells, in dem Elemente zur Modellierung geodätischer, statischer Ortsinformationen (Anforderung REQ-MM-Kontext 8) definiert werden. Modellierungselemente höherer Abstraktionsebene basieren auf dem Element *Context::WGS84Location*, welches die Kerninformationen eines WGS84-Koordinatenpaares abbildet, nämlich Längen- und Breitengrad sowie die Höhe über dem mittleren Meeresspiegel. In einem Softwaremodell kann dieses Modellierungselement verwendet werden, indem ein UML-Klassen- oder UML-Aktivitätsdiagramm um ein UML-Objektdiagramm ergänzt wird, welches Standortinformationen mit Instanzen des Elements *Context::WGS84Location* modelliert. Im Fokus dieser Dissertation steht die Generierung von Tests aus Systemmodellen. Deshalb ist die intendierte Verwendung des Elements *Context::WGS84Location* die Modellierung von Testdaten im Zusammenhang mit weiteren Vorbedingungen (vgl. Abschnitt 5.2.3.2). Ziel ist es, Instanzen des Elements zu erzeugen, die mit weiteren Testdaten bei der Ausführung von Tests als Eingabe in das SUT verwendet werden.

Zur Verwendung durch ein Testautomatisierungswerkzeug zur Simulation unterschiedlicher Standorte im Rahmen des Testens stellt das Metamodell zur Kontextmodellierung das Element *Context::Location* bereit. Es ergänzt das Element *Context::WGS84Location* um ein Attribut zur Abbildung der Präzision einer Standortinformation. Hintergrund dieses Entwurfs ist, dass die Bestimmung des Standorts durch API-Funktionen mobiler Plattformen grundsätzlich fehlerbehaftet ist. Dieser Fehler manifestiert sich durch die üblicherweise in

Metern bestimmte Präzision der Standortinformation, wodurch effektiv ein Umkreis um ein WGS84-Koordinatenpaar gebildet wird. Innerhalb dessen wird der tatsächliche Standort des Anwenders mit einer durch die Spezifikation des GPS-Moduls gegebenen Wahrscheinlichkeit vermutet. Zur Modellierung von Testdaten können Instanzen des Elements *Context::Location* mit variierenden Präzisionsinformationen in einem UML-Objektmodell modelliert werden. Ein Testautomatisierungswerkzeug kann dann eine Sequenz von Tests ausführen, in welchen der Standort des Geräts gemäß dem Testdatenmodell simuliert und die Spezifikationskonformität des SUT hinsichtlich der Verarbeitung unpräziser Standortinformationen überprüft wird.

Zur Unterstützung von Geofencing-Anwendungen, in denen eine Funktionalität einer Anwendung kontextsensitiv bezüglich des Standorts des Anwenders innerhalb einer definierten geographischen Region ist, stellt das Metamodell die Elemente *Context::CircularRegion*, *Context::CircleStripRegion*, *Context::PolygonalRegion* sowie *CContext::TriangleStripRegion* bereit (Anforderung REQ-MM-Kontext 10). Wie alle Modellierungselmente des Metamodells erfüllen sie die Funktion als Vorbedingung in Testdatenmodellen verwendet zu werden. Sie ersetzen die Notwendigkeit einer begleitenden textuellen Dokumentation des Systemmodells und ermöglichen die Verwertung dieser Informationen durch MDSD- und MDT-Technologien. Insbesondere Letztere können diese Informationen verwenden, um zur Generierung von Tests automatisiert Sequenzen positiver und negativer Testdaten zu generieren.

In Abschnitt 3.1.2.1.3 sowie in Abschnitt 4.2.3 wurde dargelegt, dass Standorte dynamisch sein können. Dieser Fall tritt ein, wenn für einen Anwendungsfall nicht der absolute Standort des Anwenders relevant ist, sondern die Bewegung des Anwenders zwischen Standorten. In der technischen Realisierung manifestiert sich diese Situation darin, dass zwei aufeinanderfolgende Standortbestimmungen, z. B. via GPS, unterschiedliche Standortinformationen erzeugen. Das heißt, das Gerät wurde im Zeitintervall zwischen den Standortbestimmungen von einem Standort zu einem anderen bewegt. Typische Nutzung ist etwa eine Navigationsanwendung, die kontinuierlich den Standort des Anwenders bestimmt und daraufhin das UI, etwa eine Kartenansicht, aktualisiert, den zurückgelegten Weg oder die Geschwindigkeit berechnet.

Das Testen solcher Anwendungsfälle erfordert die Bereitstellung von Testdaten, durch welche ein Bewegungsprofil eines Anwenders adäquat abgebildet wird. Das Metamodell zur Kontextmodellierung stellt für diesen Zweck Modellierungselemete bereit, die es Entwicklern oder Testern erlauben, Anwendermobilität auf einem abstrakten Niveau zu modellieren (Anforderung REQ-MM-Kontext 11). Der entsprechende Ausschnitt des Metamodells ist in Abbildung 5.6 dargestellt.

Das Modellierungselement *Context::CardinalDirectedMotion* erlaubt beispielsweise die Modellierung von Anwendermobilität in eine Himmelsrichtung mit einer spezifizierten Geschwindigkeit ausgehend von einem definierten WGS84-Koordinatenpaar. Bei der Modellierung von Testdaten besteht so die Möglichkeit der Abstraktion von einzelnen Koordinatenpaaren, die den Bewegungspfad des Anwenders beschreiben.

Für Fälle, in denen diese abstrakte Modellierung nicht ausreichend ist, bietet das Metamodell ebenfalls das Modellierungselement *Context::Motion* an. In Assoziation mit einem WGS84-Koordinatenpaar kann aus einer Menge von Instanzen von *Context::Location* ein Bewegungspfad interpoliert werden. Hierzu werden die individuellen Standortinformationen mit dem durch die Differenz ihrer Zeitstempel vorgegebenen Zeitabstand in das SUT eingespeist.

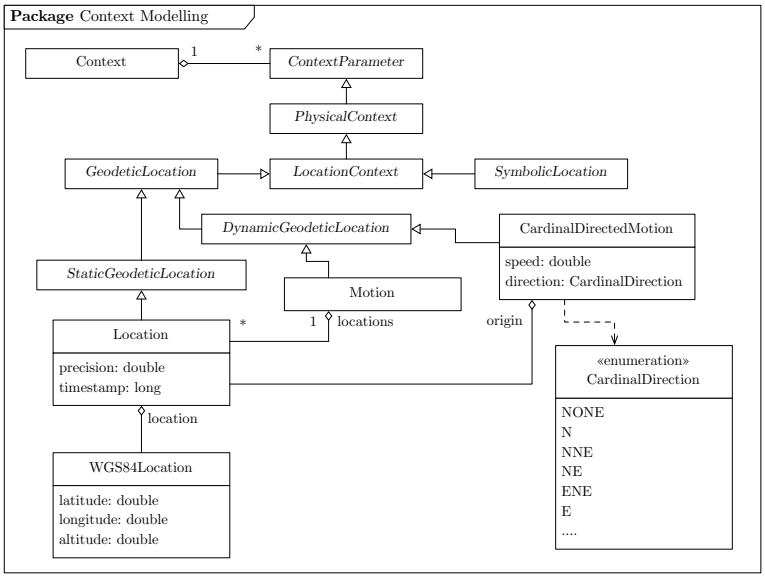


Abbildung 5.6: Metamodell Kontextmodellierung, Ausschnitt *Context::LocationContext*, dynamische, geodätische Standortdaten

Dies wird durch das SUT als sich verändernder Standort des Anwenders wahrgenommen. Durch diesen Entwurf wird es möglich, Daten des GPS-Exchange-Formats (GPX-Dateien) bei der Modellierung von Testdaten zu verwenden.

Neben der Option, Standortinformationen basierend auf absoluten Koordinaten abzubilden, besteht weiterhin die Möglichkeit der Verwendung von symbolischen Standortangaben (vgl. Abschnitt 3.1.2.1.3 sowie in Abschnitt 4.2.3). Diese ermöglichen eine intuitive Handhabung von Standortinformationen, beispielsweise in Form von Postanschriften oder Lokalisierung von Geräten und Personen innerhalb von Gebäuden. Es besteht hier die Schwierigkeit, dass für die Lokalisierung von Personen und Geräten innerhalb von Gebäuden im Jahr 2016 keine standardisierte Technologie existiert und GPS aufgrund starker Signaldämpfung üblicher Baumaterialien i. d. R. nicht verfügbar ist (vgl. Peterson et al. [277], Eissfeller et al. [105])[2]. Es ist somit Aufgabe einer mobilen Anwendung, die einen solchen Anwendungsfall realisiert, geeignete Technologien (z. B. BLE-Beacons) und Algorithmen zu implementieren. Die Abbildung solcher Technologien im Metamodell zur Kontextmodellierung ist deshalb im Rahmen der Standortmodellierung nicht möglich. Es besteht hier lediglich die Möglichkeit, Elemente zur Modellierung des Sensorkontext (vgl. Abschnitt 5.2.2.2.2) zu verwenden.

Geocoding und Reverse Geocoding sind hingegen Technologien, die über alle mobile Plattformen hinweg in den jeweiligen SDKs verfügbar sind. Sie ermöglichen die wechselseitige Zuordnung von WGS84-Koordinaten und Postanschriften. Das Metamodell bietet hierfür ebenfalls ein Modellierungselement an (Anforderungen REQ-MM-Kontext 9 und REQ-MM-Kontext 12). Mobile Geräte sind indes nicht in der Lage, ein solches Datenformat unmittelbar zu verwenden, da alle Lokalisierungstechnologien Daten im WGS84-Format generieren. Es obliegt daher einer Testautomatierungslösung symbolische Standortinformationen aus Modellen zu extrahieren und durch Geocoding oder vergleichbare Technologien in WGS84-Koordinaten

[2] GPS-Signale, die eine Wand durchdringen, werden in Abhängigkeit von den Baumaterialien um den Faktor 100 und mehr gedämpft. Für raumgenaue Positionsbestimmung in Innenräumen ist der Einsatz differentieller GPS-Verfahren oder alternativer Technologien unvermeidbar. [277, 105]

zu transformieren, um sie als simulierten Kontextparameter in das SUT einzuspeisen. Ein entsprechendes Modellierungselement ist in Abbildung 5.7 dargestellt.

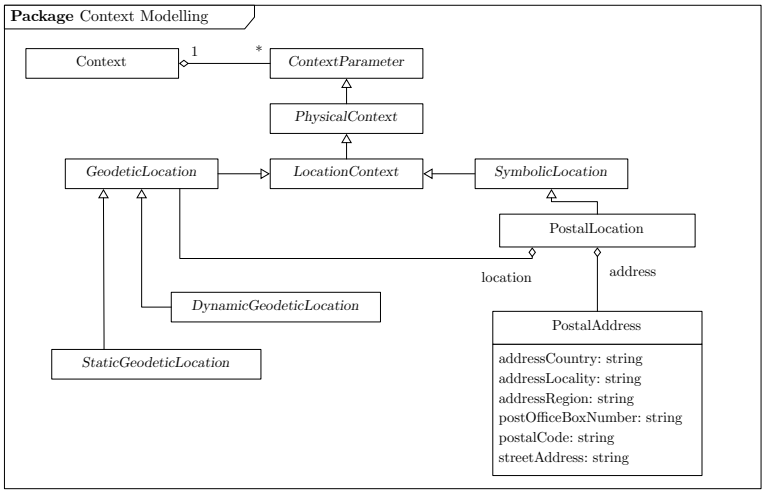


Abbildung 5.7: Metamodell Kontextmodellierung, Ausschnitt *Context::LocationContext*, symbolische Standortdaten

Zur Modellierung symbolischer Standortinformationen ist das Metamodell auf das Element *Context::PostalLocation* beschränkt. Es ermöglicht die Definition von Postadressen und bietet optional eine Referenz auf ein Element des Typs *Context::GeodeticLocation*. Wird dieses modelliert, ist die Standortinformation dem Testwerkzeug direkt verfügbar. Anderenfalls muss sie durch Geocoding ermittelt werden.

Die beiden in dieser Dissertation verwendeten Beispielanwendungen Mobiler Taxiruf und AR Tourist Information sind jeweils durch einen Geofence auf eine geographische Region beschränkt. Die Anwendung Mobiler Taxiruf ist beispielsweise auf das Gebiet der Bundesrepublik Deutschland beschränkt, wobei einige Teile des Bundesgebiets (z. B. bestimmte Inseln, vgl. Abschnitt 4.2.3) nicht zum zulässigen Gebiet gehören. Eine Approximation dieses Geofence durch eine Instanz von *Context::CircularRegion* oder *Context::CircleStripRegion* wäre hier im Vergleich zur Nutzung einer Instanz von *Context::CompoundRegion* ungünstig, da die Anwendung per Definition auf Regionen innerhalb des Geofence beschränkt ist und keine Toleranz gewährt wird. Unter Verwendung des Modellierungselements *Context::CompoundRegion* kann der Geofence jedoch mit ausreichender Präzision modelliert werden, beispielsweise aus einem Polygon, welches das Bundesgebiet ohne Inseln abdeckt und mit weiteren Polygonen zur Modellierung zulässiger Inselregionen kombiniert wird.

Eine Modellierung des Geofence unter Verwendung des Metamodells zur Kontextmodellierung hat hier gegenüber einer textuellen Anforderungsdokumentation Vorteile. Einerseits sind durch konkrete WGS84-Koordinaten im Modell Missverständnisse ausgeschlossen. Andererseits liegen die Daten so bereits in einem maschinell lesbaren Format vor. Eine direkte Wiederverwendung in der Implementierung der Anwendung als auch bei der Testfallgenerierung ist so gewährleistet. Letztere kann gezielt Koordinaten innerhalb und außerhalb des Geofence generieren, um eine Vielzahl von Tests zu erzeugen.

Oberflächlich betrachtet kann hier der Eindruck entstehen, dass ein hoher Aufwand zur Erstellung des Kontextmodells entsteht. Der Aufwand zur Bestimmung aller die zulässige poly-

gonale Region definierenden WGS84-Koordinaten kann zur Realisierung des Anwendungsfalls jedoch in keinem Fall vermieden werden, so dass hier unmittelbar ein Vorteil aus der Wiederverwendbarkeit des Modells (Anwendungsimplementierung, Testfallgenerierung) entsteht. Aufgrund der maschinellen Verarbeitung des Modells gegenüber einer manuellen, textuellen Dokumentation besteht hier zusätzlich die Option der Verwendung eines graphischen Editors, der die Definition des Geofence auf einer digitalen Kartenansicht ermöglicht, wodurch der Aufwand zur Erstellung des Modells erheblich gesenkt würde.

5.2.2.2.2 Modellierungselement *Context::SensorContext*

Mobile Anwendungen nehmen ihre Umgebung durch die Verwendung von Sensoren wahr (vgl. Abschnitt 3.1.2.1.4, Abschnitt 3.1.2.1.5 sowie Abschnitt 4.2.4). Ein Kontextmodell muss deshalb Modellierungselemente anbieten, mit denen der Zustand von Sensoren abgebildet werden kann. Weiterhin muss ein Modellierungselement in einer geeigneten Weise in ein Systemmodell bzw. ein Testmodell integriert werden können, um einerseits einem Entwickler den Einfluss von Sensormesswerten auf das System zu verdeutlichen und andererseits einem Testgenerierungswerkzeug Testdaten in einem maschinenlesbaren Format verfügbar zu machen.

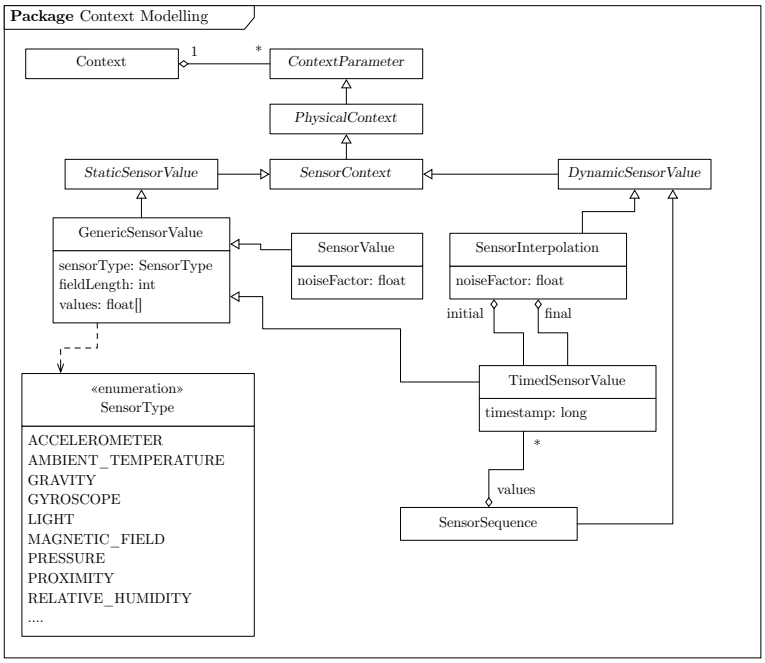


Abbildung 5.8: Metamodell Kontextmodellierung, Ausschnitt *Context::SensorContext*

In Abbildung 5.8 ist der Ausschnitt des Metamodells zur Kontextmodellierung dargestellt, der die Modellierungselemente zur Modellierung sensorbasierter Kontextparameter umfasst. Sensorereignisse können statisch oder dynamisch in Software einfließen (vgl. Abschnitt 3.1.2.1.4, Abschnitt 3.1.2.1.5 und Abschnitt 4.2.4). Je nach Anwendungsfall ist eine Momentaufnahme eines Messwertes relevant oder der Anwendungsinhalt wird durch die Veränderung von Sensorwerten über die Zeit bestimmt. Deshalb stellt das Metamodell auch hier Modellierungselemente bereit, um beide Fälle abzudecken (Anforderungen REQ-MM-Kontext 18, REQ-MM-Kontext 16 und REQ-MM-Kontext 17).

Das Element *Context::GenericSensorValue* bildet die Basis zur Modellierung von Sensorereignissen. Es stellt Attribute bereit, die den Typ des modellierten Sensors abbilden, die Anzahl der Messwerte die in einem atomaren Messdatum enthalten sind sowie die eigentlichen Werte. Sensoren in mobilen Geräten messen unterschiedliche physikalische Größen, einige davon eindimensional (z. B. Umgebungstemperatur), andere mehrdimensional, wie z. B. der Magnetfeldsensor, der die magnetische Flussdichte in den drei Achsen des Referenzsystems des Geräts misst. Hierin ist die variable Länge des Attributs *values* begründet.

Sensoren erzeugen i. d. R. Rauschen, d. h. eine das zu messende Signal überlagernde Störgröße. Diese kann natürlichen Ursprungs sein oder durch die Bauart des Sensors bedingt sein. Das Element *Context::GenericSensorValue* berücksichtigt diese Größe nicht. Bei einer Verwendung dieses Elements in einem Testdatenmodell kann hierdurch ein rauschfreies Signal abgebildet werden, welches zwar in der Realität nicht zu erwarten ist, aber dabei helfen kann, die Qualität eines Algorithmus zu prüfen, der Sensordaten verarbeitet. Das Metamodell bieten zur Abbildung von Rauschen das Element *Context::SensorValue* an, dieses stellt ein Attribut zur Auszeichnung einer Störung bereit. Ein Testautomatisierungswerkzeug mit Kontextsimulation kann diesen Wert verwenden, um sensorspezifisches Rauschen zu modellieren.

Das Metamodell stellt die Elemente *Context::SensorInterpolation* und *Context::SensorSequence* zur Modellierung dynamischer Sensorereignisse bereit. Ersteres ermöglicht die Modellierung der Veränderung des Wertes eines Sensors von einem initialen auf einen finalen Wert innerhalb eines Zeitintervalls. Die Kontextsimulation wird während der Testausführung eine Sequenz individueller Werte interpolieren und in das SUT einspeisen. Das Element *Context::SensorSequence* kann zur Modellierung komplexer Werteverläufe verwendet werden und kann insbesondere für C'n'R-Tests verwendet werden, d. h. zur wiederholten Einspeisung zuvor aufgezeichneter Werte in das SUT.

5.2.2.2.3 Modellierungselement *Context::OrientationContext*

Die Orientierung eines mobilen Geräts während der Benutzung kann Auswirkungen auf die Funktion und das UI einer mobilen Anwendungen haben (vgl. Abschnitt 3.1.2.2.1 und Abschnitt 4.2.4). Zur Abbildung dieser Parameter in einem Kontextmodell bietet das Metamodell Modellierungslemente an, die bei der Modellierung von Testdaten verwendet werden können, um die Orientierung des Geräts anstelle einer expliziten Modellierung einzelner Sensorwerte (vgl. Abschnitt 5.2.2.2.2) auf einer abstrakten Ebene abzubilden. Abbildung 5.9 stellt den Ausschnitt des Metamodells zur Kontextmodellierung dar, der die Modellierungselemente zur Abbildung der Geräteorientierung enthält.

Zur Modellierung einer diskreten Geräteorientierung stellt das Metamodell das Element *Context::DiscreteDeviceOrientation* bereit. Es dient der Abbildung von Zuständen, in denen konkrete Werte für Roll-, Nick- und Gierwinkel nicht relevant sind, sondern lediglich eine Orientierungsklasse festlegt werden soll (z. B. Querformat, Anforderung REQ-MM-Kontext 14). Wird dieses Element bei der Modellierung von Testdaten verwendet, generiert die Kontextsimulationstechnologie Daten für die an der Klassifikation der Geräteorientierung beteiligten Sensoren (z. B. Beschleunigungssensor und den Magnetfeldsensor, Orientierungssensor).

Das Modellierungselement *Context::DeviceOrientation* ist für Anwendungsfälle bestimmt, bei denen konkrete Werte für Roll-, Nick- und Gierwinkel bedeutsam sind (Anforderung REQ-

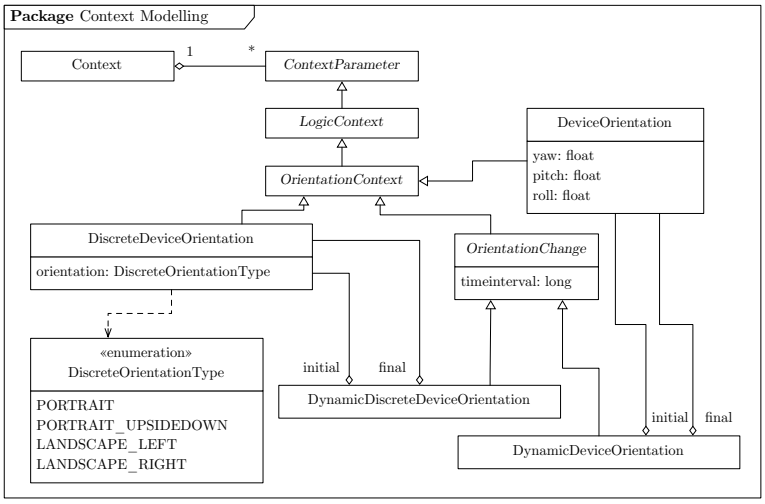


Abbildung 5.9: Metamodell Kontextmodellierung, Ausschnitt *Context::OrientationContext*

MM-Kontext 13). Beispielsweise bei AR-Anwendungen, bei denen die Pixelkoordinaten zur Darstellung von Inhalt auf dem Display aus der Orientierung des Geräts berechnet werden (z. B. AR Tourist Information). Auch hier gilt, dass eine Kontextsimulationstechnologie diese Werte aus dem Modell extrahiert und im SUT entsprechende Sensorwerte simuliert.

Das Metamodell bietet ebenfalls Modellierungslemente (*Context::DynamicDiscreteDeviceOrientation* sowie *Context::DynamicDeviceOrientation*) an, mit denen eine Veränderung der Geräteorientiertung über die Zeit abgebildet werden kann (Anforderung REQ-MM-Kontext 15). Sie finden Verwendung im Rahmen der Testmodellierung, um den Fall abzubilden, dass ein Anwender das Gerät willkürlich von einer Orientierung in eine andere überführt (z. B. von Hochformat auf Querformat).

In der Testmodellierung können Instanzen von *Context::OrientationContext* dazu verwendet werden, ein Modell mit dem Ziel zu annotieren, die Testautomatisierungstechnologie mit einer Kontextsimulation zu instruieren, den Wechsel der Geräteorientierung durch Manipulation von Sensorwerten in das SUT einzuspeisen.

5.2.2.2.4 Modellierungselement *Context::DateTimeContext*

In Abschnitt 3.1.2.1.2 sowie Abschnitt 4.2.2 wurde die Bedeutung von Datum und Uhrzeit als Kontextparameter diskutiert. Die Implementierung einer Anwendung, deren Funktionalität vom Ausführungszeitpunkt abhängt, erfordert eine präzise Definition der Zeitpunkte oder Zeitintervalle, zu denen diese Funktionalität verfügbar sein soll. Um diese Information auch für modellbasierte Technologien wie etwa MDT verfügbar zu machen, wird in der verwendeten Modellierungssprache ein entsprechendes Modellierungselement benötigt.

Zwar bieten Programmiersprachen i. d. R. dedizierte Datentypen zur Repräsentierung von Datum und Uhrzeit an, die in der etablierten Praxis in der UML als generischer Datentyp *Date*, *Time* oder auch *Timestamp* verwendet werden. Zur Modellierung komplexerer Zusammenhänge wie etwa Zeitintervallen oder maskierten Zeitintervallen (z. B. Mai bis Oktober in jedem Jahr, also Maskierung der Jahresangabe) sind diese generischen Typen jedoch nicht ausreichend. Insbesondere bei einer Verwendung in einem Kontextmodell zur Generierung

von Softwaretests, die zeitsensitive Anwendungen zum Gegenstand haben, müssen neben der eigentlichen Anwendung auch entsprechende Testdaten im Modell enthalten sein.

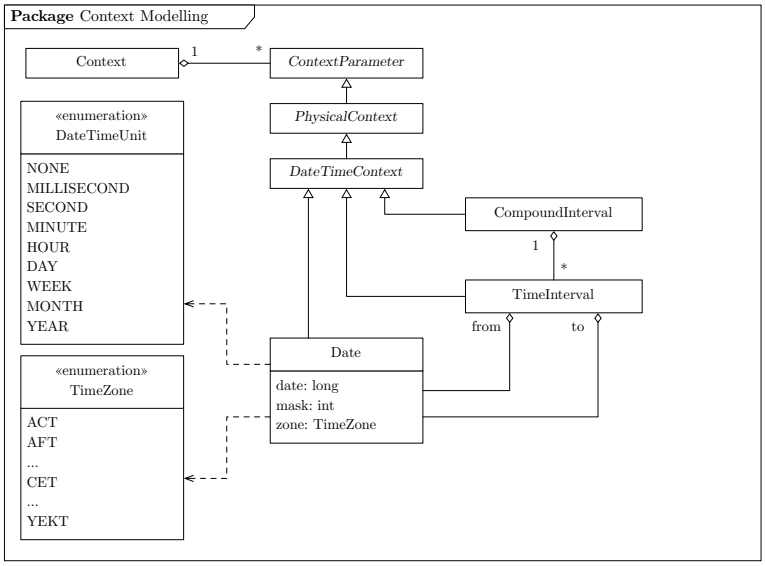


Abbildung 5.10: Metamodell Kontextmodellierung, Ausschnitt *Context::DateTimeContext*

Aus diesem Grund stellt das Metamodell zur Kontextmodellierung zur Erfüllung der Anforderung REQ-MM-Kontext 7 (Kontextparameter Zeit) entsprechende Modellierungselemente bereit, mit denen auch komplexere Zeitintervalle abgebildet werden können.

Das Element *Context::Date* repräsentiert eine einfache Datumsangabe, welche als Anzahl vergangener Sekunden seit einem definierten Zeitpunkt im Modell abgebildet ist[3]. Darüber hinaus enthält das Modellierungselement Informationen zur Zeitzone und zur Maskierung der einzelnen Komponenten der Datumsangabe (Sekunde, Minute, Stunde, usw.). Zeitintervalle können mit dem Modellierungselement *Context::TimeInterval* modelliert werden. Es referenziert einen Anfangs- und einen Endzeitpunkt vom Typ *Context::Date*. Zusammengesetzte Zeitintervalle können mit *Context::CompoundInterval* modelliert werden.

5.2.2.2.5 Modellierungselement *Context::NetworkContext*

Neben dem Standort des Anwenders und Sensorenmesswerten wurde in Abschnitt 3.1.2 und insbesondere in Abschnitt 3.1.2.1.1 ebenfalls die Verfügbarkeit von Netzwerkkonnektivität als Kontextparameter eingeführt. Mobile Apps sind häufig auf Netzwerkverbindungen angewiesen, so dass das Verhalten einer App bei Veränderung der Eigenschaften der Netzwerkverbindung ein relevanter Gegenstand des Testens ist.

Um diesen Kontextparameter bei der Modellierung zu berücksichtigen, bietet das Metamodell zur Kontextmodellierung Elemente zur Abbildung unterschiedlicher Netzwerkeigenschaften an. In Abbildung 5.11 ist der entsprechende Ausschnitt dargestellt. Das Metamodell definiert die Modellierungselemente *Context::NetworkContext* und *Context::DynamicNetworkContext*, mit denen Eigenschaften der Netzwerkverbindung modelliert werden können.

[3]Unixzeit, ein POSIX-Standard der die vergangenen Sekunden seit dem 1. Januar 1970 00:00 Uhr UTC als Basis verwendet.

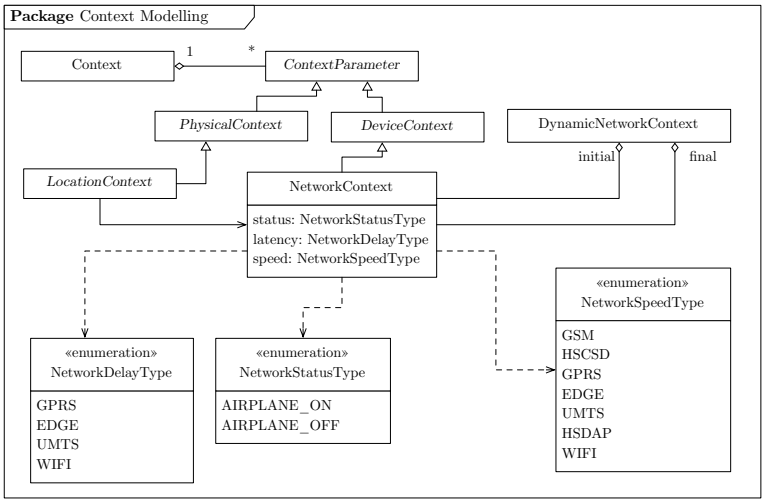


Abbildung 5.11: Metamodell Kontextmodellierung, Ausschnitt *Context::NetworkContext*

Die Verwendung dieser Komponenten (vgl. Abschnitt 5.2.3.3) ist beim Anreichern von Systemmodellen mit Testparametern zweckmäßig. So kann an einem Modell annotiert werden, welche Aktivitäten abhängig vom Status der Netzwerkverbindung sind. Aus diesem Modell können dann Tests generiert werden, die die betreffende Aktivität unter unterschiedlichen Status der Netzwerkverbindung ausführen.

5.2.2.2.6 Modellierungselement *Context::EnergyContext*

Der Status der Energieversorgung gehört ebenfalls zu den Kontextparametern (vgl. Abschnitt 3.1.2.1.1) und repräsentiert einen Parameter des Geräts, der sich zur Laufzeit einer App verändern kann (im Gegensatz zu statischen Geräteparametern, wie etwa Displayauflösung).

Um auch diesen Parameter bei der Systemmodellierung und der Testfallmodellierung zu berücksichtigen, stellt das Metamodell zur Kontextmodellierung Elemente zur Abbildung unterschiedlicher Status der Energieversorgung bereit. In Abbildung 5.12 ist der entsprechende Ausschnitt dargestellt. Das Modellierungselement *Context::EnergyContext* kann verwendet werden, um im Rahmen der Testmodellierung das Verhalten einer App bei einer Veränderung des Status der Energieversorgung zu modellieren.

5.2.2.3 Verwendung des Metamodells zur Kontextmodellierung

Das im Abschnitt 5.2.2.2 entworfene Metamodell zur Kontextmodellierung ist geeignet, Kontextparameter als Testdaten in einem Objektmodell darzustellen. Die Analyse wissenschaftlicher Literatur zum Thema Kontextmodellierung (vgl. Abschnitt 2.2.2) hat eine Reihe unterschiedlicher Ansätze identifiziert, wie Kontext modelliert und in Software verwendet werden kann. Diese Ansätze haben i. d. R. algorithmisches Schlussfolgern abstrakten Kontexts aus atomaren Kontextparametern zum Gegenstand. In Gegensatz hierzu steht das in dieser Dissertation verfolgte Ziel, Kontext während der Testausführung zu simulieren. Zu diesem Zweck muss abstrakter Kontext in atomare Kontextparameter zerlegt werden und diese müssen durch eine geeignete Technologie während des Testens in das SUT eingespeist werden.

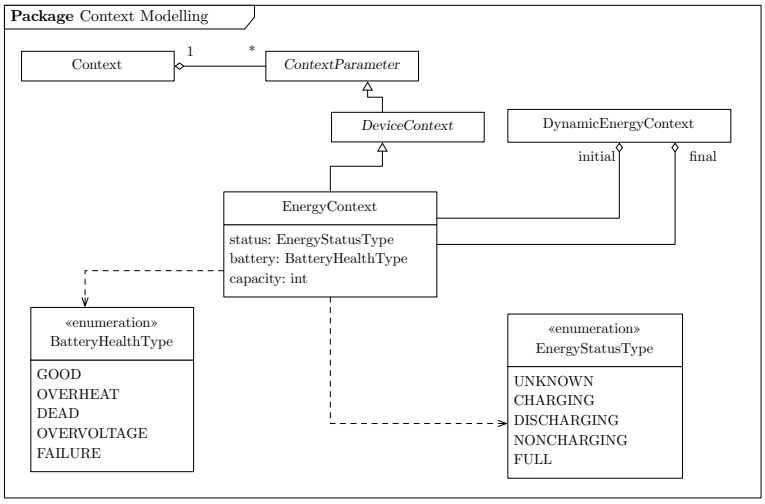


Abbildung 5.12: Metamodell Kontextmodellierung, Ausschnitt *Context::EnergyContext*

Aufgrund der generischen Natur der Entität Kontext besteht kaum Aussicht ein allgemeingültiges Kontextmetamodell zu entwerfen. Vielmehr orientiert sich der hier dargestellte Entwurf an den technischen Möglichkeiten, die aufgrund der im Jahr 2016 typischerweise in mobilen Geräten zu erwarten sind.

Entwurf und Implementierung einer kontextsensitiven Anwendung setzten stets voraus, relevante Kontextparameter in der Planungsphase zu identifizieren und deren Auswirkungen auf das zu entwickelnde Softwaresystem zu definieren. Die Informationen zur Erstellung eines für diese Anwendung gültigen Kontextmodelles liegen nach Abschluss dieser Aktivitäten also vor. Das in Abschnitt 5.2.2.2 vorgestellte Metamodell soll ein Werkzeug sein, diese Informationen in Form von Testdaten im Systemmodell zu verankern, so dass die in Abschnitt 5.3 diskutierte Methode zur Generierung von Softwaretests diese Information direkt als Eingabedaten in Form von Vorbedingungen nutzbar macht.

Seine bestimmungsgemäße Verwendung findet das Metamodell zur Kontextmodellierung deshalb in einer Verwendung als Datenmodell bei der Integration von Testdaten in Systemmodellen durch Verwendung des in Abschnitt 5.2.4 vorgestellten UML-Profils zur Integration der Testfallmodellierung in die Systemmodellierung.

5.2.3 Testfallmodellierung

Eine Option zur Spezifikation von Softwaretests ist die natürliche Sprache. Diese Variante ist insofern vorteilhaft, als dass sie hinsichtlich der Notwendigkeit spezieller Technologiekenntnisse keine besonderen Anforderungen an den Akteur stellt. Nachteilig ist hingegen, dass natürliche Sprache Raum zur Interpretation lässt, unvollständig ist oder auf jene Aspekte des zu testenden Systems fokussiert, die besonders gut verstanden sind, andere Aspekte jedoch vernachlässigt. Natürlichsprachliche Tests entziehen sich zudem i. d. R. einer maschinellen Verarbeitung, wodurch manueller Aufwand beim Testen entsteht (vgl. Robinson [299]).

Das andere Extrem bildet die Spezifikation von Tests direkt in Code, der die Testausführung automatisiert (z. B. JUnit). Entgegen einer natürlichsprachlichen Spezifikation ist hier das Abstraktionsniveau gering und Interpretationsspielraum existiert nicht. Allerdings kann

diese Art der Testspezifikation nur von Akteuren mit weitreichenden Technologiekenntnissen erstellt und nachvollzogen werden. Einem der Softwareentwicklung fernen Stakeholder ist es hier u. U. noch nicht einmal möglich nachzuvollziehen, ob ein in Code realisierter Test tatsächlich die anforderungsgemäße Funktionalität einer Anwendung prüft.

Zwischen diesen beiden Extremen existieren zahlreiche weitere Optionen, Tests so zu spezifizieren, dass einerseits alle Stakeholder unabhängig von technischen Kenntnissen ein gemeinsames Verständnis der Spezifikation erlangen und andererseits eine hohe Präzision erreicht wird. Eine dieser Optionen ist die Abbildung von Tests in DSL-basierten Spezifikationssprachen auf einem hohen Abstraktionsniveau, wie etwa Sprachen der BDD-Familie, zu welcher beispielsweise Calabash (vgl. Abschnitt 2.5.2.6 und Abschnitt 5.5.3) zuzuordnen ist. Hierdurch wird die Testspezifikation allerdings früh an eine konkrete Automatisierungstechnologie gebunden. Die Entscheidung, zu einem späteren Zeitpunkt zu einer anderen Automatisierungslösung zu wechseln, macht dann die bis zu diesem Zeitpunkt erstellten Tests obsolet.

Deshalb bietet sich die Verwendung von Modellen zur Testspezifikation an. Hartman et al. [171] diskutieren in ihrer Forschungsarbeit Technologien zur Modellierung von Tests unter Verwendung generischer Modellierungssprachen (z. B. UML) und domänenspezifischer Modellierungssprachen (DSML). Generische Sprachen haben den Vorteil, dass sie auch unterschiedliche Aspekte eines Systems auf einer einheitlichen Abstraktionsebene abbilden können. Der Vorteil einer DSML zur Beschreibung von Testmodellen ist andererseits, dass ist besser auf die konkreten Anforderungen der Testmodellierung zugeschnitten sind und beispielsweise geeignete Modellierungselemente zur Beschreibung von Vor- und Nachbedingungen anbieten.

Die Verwendung von Modellen zur Generierung von Tests wurde bereits von Robinson [299] untersucht. Auch hier basiert die Argumentation auf der Absenkung von Zugangshürden für Stakeholder zur Testspezifikation bei gleichzeitigem Erhalt hoher Präzision. Modellbasiertes Vorgehen ermöglicht es auch weniger technisch versierten Stakeholdern, ein abstraktes Verständnis des SUT zu entwicklen und die Testspezifikation auf wesentliche Merkmale zu fokussieren. Hieraus leitet Robinson die Idee der ausführbaren Spezifikation ab, die primär auf der Verwendung von Modellen zur Repräsentierung von Tests basiert. Das vom Autor vorgeschlagene Vorgehen, aus einem Systemmodell zunächst ein plattformunabhängiges Testmodell zu erzeugen und dieses erst in einem späteren Schritt zu technologiespezifischen Tests zu transformieren, ermöglicht eine Separierung von Modell und Technologie, die auch zu einem späten Zeitpunkt Entscheidungen zur verwendeten Automatisierungstechnologie zulässt.

Das in dieser Dissertation vorgestellte Konzept beschreitet ebenfalls den Weg, aus Modellen einer generischen Modellierungssprache, nämlich der UML, noch vor der Generierung plattformspezifischer Tests ein plattformunspezifisches Testmodell zu erzeugen. Die Systemmodellierung kann ohne Berücksichtigung besonderer Anforderungen des Testens erfolgen. Erst die Anreicherung mit Testdaten unter Verwendung des Metamodells zur Kontextmodellierung setzt den Fokus auf das Testen kontextsensitiver Anwendungen.

Vom Grundsatz her bestünde die Möglichkeit, zur Modellierung von Tests ebenfalls die UML zu verwenden. Die UML bietet zwar eine Reihe unterschiedlicher Diagrammarten und Modellierungselemente, die geeignet sind Verhaltensaspekte von Anwendungen abzubilden. Zur Modellierung von Tests für mobile, kontextsensitive Anwendungen bietet die UML jedoch keine Diagrammart oder spezielle Modellierungselement an. Um ein Testmodell jedoch

auf solche Eigenschaften des zu testenden Systems zu fokussieren, die für das Testen von besonderer Bedeutung sind und von solchen zu abstrahieren, die beim Testen nicht relevant sind, ist es zweckmäßig ein dediziertes Metamodell zur Testfallmodellierung zu definieren, welches adäquate Modellierungselemente bereitstellt und gleichzeitig keinen Raum für unerwünschte Komplexität eröffnet. Es ist sinnvoll, die durch das Metamodell definierten Modellierungselemente auf solche zu beschränken, die zur Testfallmodellierung unverzichtbar sind.

In Quellcodefragment 5.1 ist beispielhaft ein Calabash-Android-Test abgebildet, der innerhalb eines *Scenario* einen individuellen Testschritt bestehend aus der Spezifikation der Vorbedingungen, der Durchführung einer Aktion und der Überprüfung der Nachbedingungen beschreibt. Die Vorbedingung beschreibt hierbei den Zustand des zu SUT vor der im Rahmen des Tests durchzuführenden Aktion, also die Vorbereitung des SUT auf den eigentlichen Test. Eine Aktion beschreibt i. d. R. die Interaktion des Anwenders mit dem System, z. B. die Eingabe von Daten in ein spezifiziertes Eingabefeld der Benutzungsoberfläche. Einem Eingabedatum oder einer Klasse von Eingabedaten wird hierbei i. d. R. eine Menge definierter Nachbedingungen gegenübergestellt, von denen erwartet wird, dass sie nach der Testausführung gültig sind. Dateneingabe im klassischen Sinn sind hierbei nicht die einzig möglichen Interaktionen, ebenfalls sind Schüttel- oder Wischgesten denkbar, durch welche das Verhalten der Anwendung gesteuert wird. Erfüllt der Zustand des SUT die Nachbedingungen nicht, so wird der Testfall als nicht bestanden interpretiert. Als Ursache hierfür wird angenommen, dass sich das SUT nicht erwartungskonform verhält (d. h. im SUT liegt ein Defekt vor). Konkret ist in Quellcodefragment 5.1 ein Ausschnitt der Testsuite der Anwendung Mobiler Taxiruf dargestellt.

Quellcodefragment 5.1: Calabash-Test der Anwendung Mobiler Taxiruf, manuelle Adresseingabe

```
1  Feature: Mobiler Taxiruf
2
3    Scenario: Anwender verweigert Verwendung von GPS, ersatzweise ↙
       ↳ manuelle Adresseingabe
4
5      # general preconditions
6      Given my app is running
7
8      # preconditions for show location mode selection dialogue
9      Then I wait for the view with id "dialog_locationmode" to appear
10     Then I wait for the view with id "button_manual" to appear
11
12     # actions location mode
13     Then I press view with id "button_manual"
14
15     # preconditions for show address dialogue
16     Then I wait for the view with id "editTextAdresse" to appear
17     Then I wait for the view with id "editTextStadt" to appear
18     Then I wait for the view with id "button_commit" to appear
19
20     # actions for show address dialogue
21     Then I enter "Gerlingstrasse 16" into field with id "↙
         ↳ editTextAdresse"
22     Then I enter "Essen" into field with id "editTextStadt"
23     Then I press view with id "button_commitadress"
24
25     # postconditions
26     Then I wait for the view with id "map" to appear
```

Zeilen 6, 9, 10 und 16-18 des Testfalls repräsentieren Vorbedingungen, in denen das SUT in einen definierten Zustand überführt wird. Die Vorbedingungen im konkreten Codefragment umfassen das Starten der App (Zeile 6) und das Sicherstellen, dass bestimmte UI-Elemente

sichtbar sind (Zeilen 9, 10, 16-18). Im illustrierten Beispiel handelt es sich um Texteingabefelder. Nur wenn diese Bedingungen erfüllt sind kann der eigentliche Test ausgeführt werden. Dieser beinhaltet die Manipulation von UI-Elementen (Zeilen 13, 21-23) und die Auswertung von Nachbedingungen. Der Test umfasst genau eine Nachbedingung, nämlich die Überprüfung, ob in Reaktion auf eine Adresseingabe in die hierfür vorgesehenen Textfelder mit anschließender Bestätigung eine Kartenansicht auf dem Display angezeigt wird.

Der hier untersuchte Ansatz zur Testautomatisierung basiert auf der Transformation von UML-Modellen zu Testfallmodellen, aus denen in einem weiteren Arbeitsschritt die technologiespezifischen Tests generiert werden. Zur Erzeugung des plattformunspezifischen Testfallmodells bedarf es eines Metamodells, das geeignete Modellierungselemente zur Modellierung von Tests für mobile, kontextsensitive Anwendungen anbietet. Im Folgenden werden zunächst in Abschnitt 5.2.3.1 Anforderungen an ein solches Metamodell diskutiert und anschließend wird in Abschnitt 5.2.3.2 ein Metamodell entworfen.

5.2.3.1 Anforderungen an ein Metamodell zur Testfallmodellierung

Der hier vorgestellte Lösungsansatz zur Generierung von Testfällen aus spezifischen Systemmodellen basiert auf Modelltransformationstechnologien, welche das Quellmodell (das angereicherte UML-Aktivitätsdiagramm) auf ein Zielmodell (Testfallmodell) abbildet (vgl. Abschnitt 5.3). Bei dieser Transformation werden Modellelemente des Quellmodells so auf Elemente des Zielmodells abgebildet, dass einerseits die Semantik des Quellmodells erhalten bleibt und andererseits die für Testaktivitäten besonders relevanten Aspekte in den Vordergrund rücken. Die durch die Modelltransformation erzeugten Modelle repräsentieren die Menge von Testfällen, durch welche die Funktionen des zu testenden System abgedeckt werden.

Grundlage der Modelltransformation ist neben einem Metamodell für das Quellmodell (gegeben durch das Metamodell der UML) sowie einem Metamodell zur Kontextmodellierung (vgl. Abschnitt 5.2.2.2) ein Metamodell zur Modellierung von Testfällen. Im Folgenden werden die Anforderungen an ein solches Metamodell diskutiert.

REQ-MM-Testmodell 1: Modellierung von Testsuiten

> Im Rahmen eines Anwendungsfalls durchzuführende Aktionen sind u. U. voneinander abhängig. Das heißt, die Durchführung einer Aktion hängt ggf. von der Ausgabe der vorher ausgeführten Aktion ab, so dass je nach Testfall und Testdaten einige Aktionen eines Testfalls nicht oder nur bedingt ausgeführt werden.
>
> Um dennoch eine Abdeckung aller Aktionen zu erreichen, werden die Vorbedingungen für mehrere Testfälle so gewählt, dass jede Aktion von mindestens einem Testfall abgedeckt wird. Da diese unterschiedlichen Testfälle jedoch denselben Anwendungsfall abdecken, besteht ein semantischer Zusammenhang zwischen den Testfällen.
>
> Vom Metamodell wird daher gefordert, dass Testfälle, die in einem semantischen Zusammenhang stehen, in einem geeigneten Modellelement aggregiert werden können. Metainformationen wie etwa Bezeichnung der Testsuite, eine textuelle Beschreibung, Datum der Erstellung oder verantwortlicher Ansprechpartner sollen dazu dienen, den Umfang der Testsuite zu beschreiben.

REQ-MM-Testmodell 2: Modellierung von Testfällen

Gemäß der in Abschnitt 4.1.1 diskutierten Einführung des Begriffs repräsentiert ein Testfall eine Menge individuell durchzuführender Aktionen im Rahmen eines Anwendungsfalls. Individuelle Aktionen stehen hierbei in Relation zueinander, welche die Aktionen in eine definierte Ausführungsreihenfolge strukturieren.

Es wird an ein Metamodell zur Modellierung von Testfällen die Anforderung gestellt, dass individuelle Aktionen zu Testfällen zusammengefasst werden können. Dies ist durch entsprechende Modellelemente zu gewährleisten. Zusätzliche Metainformationen wie etwa Bezeichnung eines Testfalls, eine textuelle Beschreibung oder Datum der Testfallerstellung sollen dazu dienen, komplexe Testfälle für den Anwender unterscheidbar zu machen.

REQ-MM-Testmodell 3: Spezifikation von Relationen zwischen Testschritten

Anwendungsfälle in Softwaresystemen sind aus individuellen, aufeinander folgenden Aktionen zusammengesetzt. Der Kontrollfluss durch eine Anwendung definiert hierbei die Abfolge, in welcher einzelne Aktionen nacheinander aufrufbar sind. Gegebenenfalls enthält der Kontrollfluss Entscheidungspunkte, an welchen alternative Aufrufreihenfolgen unterschieden werden. Ebenfalls kann der Kontrollfluss Zyklen enthalten, durch welche ein wiederholtes Aufrufen einzelner Aktionen oder Abfolgen von Aktionen abgebildet werden.

Beim Testen von Softwaresystemen werden Anwendungsfälle unter definierten Vorbedingungen durchgeführt und die Erfüllung entsprechender Nachbedingungen überprüft. Dieser Prozess umfasst die Durchführung einzelner Aktionen oder Abfolgen von Aktionen. Die Reihenfolge der Durchführung individueller Aktionen ist nicht beliebig, sondern ist wesentliches Merkmal funktionaler Anwendungstests. Gegebenenfalls bildet die Durchführungsreihenfolge individueller Aktionen das Unterscheidungsmerkmal verschiedener Tests.

An ein Metamodell zur Durchführung von Testfällen wird daher die Anforderung gestellt, dass Modellelemente, durch welche individuelle Aktionen repräsentiert werden, miteinander in Relation gesetzt werden können. Diese Relation strukturiert einzelne Aktionen in einen gerichteten Graphen.

Beim Testen von Softwaresystemen müssen die zu testenden Aspekte des Systems so durch Testfälle repräsentiert werden, dass alle zu testenden Funktionen tatsächlich durch eine Abfolge durchzuführender Aktionen abgedeckt werden. Hierbei ist die Durchführungsreihenfolge einzelner Aktionen so zu wählen, dass sie einem zu spezifizierenden Abdeckungskriterium genügt. Sofern diese Durchführungsreihenfolge alternative Kontrollflüsse beinhalten, sind Testfälle so zu gestalten, dass alle Alternativen durch Abfolgen von Aktionen, beginnend mit der ersten und endend mit der letzten, entsprechend dem Abdeckungskriterium durchgeführt werden.

Um einzelne Testfälle voneinander abzugrenzen und in einer sowohl für den Menschen als auch für Maschinen lesbarer Form zu repräsentiert, ist es zweckmäßig,

von einem Testfallmodell Zyklenfreiheit zu fordern. Durch diese Forderung an das Metamodell zur Modellierung von Testfällen wird der gerichtete Graph, durch welchen einzelne Aktionen strukturiert werden, in eine Baumstruktur überführt. Unterschiedliche Pfade vom Wurzelknoten zu verschiedenen Blattknoten bilden dann individuelle Testfälle ab.

REQ-MM-Testmodell 4: Modellierung von Testschritten

Testfälle sind Mengen aufeinanderfolgender Testschritte. Jeder Testschritt repräsentiert die Durchführung einer individuellen Aktion des zu testenden Anwendungsfalls (vgl. Abschnitt 4.1.1.8). Um mit der Durchführung eines Testschritts aussagefähige Ergebnisse zu erzielen, muss diese Durchführung mit definierten Vorbedingungen erfolgen. Nur so ist sichergestellt, dass bei wiederholten Durchführungen einzelner Testschritte bei Abweichungen der jeweiligen Ergebnisse auf einen Fehler im zu testenden System geschlossen werden darf.

Ob sich das zu testende System erwartungskonform verhält wird durch eine Überprüfung der Nachbedingungen eines Testschritts bestimmt. Sind nach der Durchführung die gemäß der Spezifikation des Systems zu erwartenden Nachbedingungen nicht erfüllt, erfüllt das System seinen Zweck nicht bestimmungsgemäß.

Neben der Menge der Vorbedingungen und der Menge der Nachbedingungen spezifiziert ein Testschritt insbesondere die konkret durchzuführende Aktion. Diese kann beispielsweise die Interaktion mit Elementen der Benutzungsoberfläche sein.

Von einem Metamodell zur Modellierung von Testfällen wird deshalb gefordert, dass Modellelemente bereitgestellt werden, die sowohl Vorbedingungen und Nachbedingungen abbilden als auch die durchzuführende atomare Aktion. Die durchzuführende Aktion kann vielgestaltig sein, beispielsweise die Manipulation eines Elements des UI oder, im Kontext mobiler Anwendungen, das Verfügbarwerden einer Netzwerkverbindung oder eine Änderung des Standorts des Anwenders oder sonstiger Kontextparameter.

REQ-MM-Testmodell 5: Spezifikation konkreter Testaktionen in Testschritten

Von einem Metamodell zur Modellierung von Testfällen wird ein Modellelement gefordert, mit welchem elementare Interaktionen mit dem zu testenden System eindeutig definiert werden können. Hierunter werden sowohl Interaktionen mit der Benutzungsoberfläche als solche Interaktionen zusammengefasst, die zwar nicht die Benutzungsoberfläche zum Gegenstand haben (z. B. Anklicken eines Elements der Benutzungsoberfläche, Texteingabe in ein Textfeld), aber dennoch eine Interaktion mit der Anwendung repräsentieren.

Ridene et al. [297] entwerfen einen Prüfstand für mobile Anwendungen. Auch hier hat die Interaktion des Anwenders mit der Anwendung eine zentrale Bedeutung. Um diese zu realisieren entwerfen die Autoren eine DSL, mit welcher Interaktionen

mit der zu testenden Anwendung ohne Programmierkenntnisse modelliert werden können. Neben Interaktion, die direkt auf dem Smartphone ausgeführt werden, führen Ridene et al. ebenfalls Interaktionen ein, die nicht durch den Anwender erfolgen. Hierunter fallen beispielsweise eingehende Anrufe oder Textnachrichten, die Smartphones-Apps üblicherweise unterbrechen bzw. pausieren.

In Fällen, in denen eine Interaktion mit einem spezifischen Element der Benutzungsoberfläche erfolgt, muss durch das entsprechende Modellelement das betreffende Element eindeutig in Form eines Interaktionszielelements identifiziert werden können. Sind zudem (z. B. Texteingabe) zusätzliche Informationen notwendig, so müssen diese durch ein Modellelement entsprechend abgebildet werden können.

REQ-MM-Testmodell 6: Modellierung von Vorbedingungen

Zur reproduzierbaren Durchführung von Testfällen ist es notwendig, das System vor der Durchführung individueller Aktionen in einen definierten Zustand zu überführen. Dieser Zustand wird durch die Menge der für die zu testende Aktion relevanten Parameter bestimmt. Vorbedingungen wiederum können sowohl für einen vollständigen Testfall als auch für einzelne Testschritte gelten. Insbesondere für mobile, kontextsensitive Anwendungen werden durch diese Vorbedingungen ebenfalls Kontextparameter wie z. B. Standort oder Sensormesswerte erfasst.

Insbesondere für die in dieser Dissertation untersuchte Methode zur Testautomatisierung mobiler, kontextsensitiver Anwendungen umfassen diese Vorbedingungen die Spezfikation von Kontextparametern. Demnach wird von einem Metamodell zur Testfallmodellierung gefordert, dass die Modellierung von Kontextparametern als Vorbedingungen für durchzuführende Testschritte ermöglicht wird.

REQ-MM-Testmodell 7: Modellierung von Nachbedingungen

Ziel des Testens ist die Verifikation der Anforderungskonformität eines Softwaresystems. Hierzu werden Funktionen eines Softwaresystems exemplarisch unter definierten Vorbedingungen aufgerufen und das erzeugte Resultat mit dem erwarteten Resultat verglichen. Das erwartete Resultat wird durch Nachbedingungen definiert. Genügt der Zustand des Systems nicht den Nachbedingungen, verhält sich das System nicht anforderungskonform. In der Praxis heißt das, einer Menge von Vorbedingungen wird eine Menge Nachbedingungen zugeordnet, von denen erwartet wird, dass sie das System nach der Testdurchführung erfüllt.

Nachbedingungen für individuelle Testaktionen können vielgestaltig sein. Beispielsweise kann gefordert werden, dass nach der Durchführung ein bestimmter Text in einem bestimmten Element der Benutzungsoberfläche auf dem Display dargestellt wird. Ebenfalls denkbar sind Nachbedingungen die fordern, dass der gesamte Bildschirminhalt einem zuvor hergestellten Prototypen entspricht. Dieser wiederum könnte entweder durch einen Grafiker oder durch Anfertigung einer Momentaufnahme des Bildschirminhalts einer bereits lauffähigen App bereitgestellt werden.

Weiterhin denkbar sind Nachbedingungen, die sich nicht auf die Benutzungsoberfläche der Anwendung, sondern auf die Existenz oder den Inhalt von Dateien beziehen oder Eigenschaften vorhandener Peripheriegeräte.

Es wird von einem Metamodell zur Testfallmodellierung gefordert, dass Eigenschaften der Benutzungsoberfläche durch entsprechende Modellelemente abgebildet werden können. Hierbei müssen ggf. Elemente der Benutzungsoberfläche eindeutig identifiziert werden und ein Testorakel vorgegeben werden können.

Auf Basis dieser Anforderungen wird im folgenden Abschnitt 5.2.3.2 ein Metamodell entworfen, mit welchem Testmodelle von Softwaresystemen abgebildet werden können. Grundsätzlich existiert hier die Option einer manuellen Erstellung solcher Testmodelle. Im Kontext dieser Arbeit ist die intendierte Verwendung des Metamodells jedoch die Abbildung solcher Modelle, die durch die Modelltransformation des Testfallgenerators (vgl. Abschnitt 5.3) aus UML-Modellen erzeugt werden.

5.2.3.2 Metamodell zur Testfallmodellierung

In diesem Abschnitt wird der Entwurf eines Metamodells zur Testmodellierung vorgestellt und im Detail diskutiert, welches die in Abschnitt 5.2.3.1 definierten Anforderungen erfüllt. In dem hier untersuchten Ansatz zur Testautomatisierung erfüllt es zweierlei Funktion. Erstens definiert es die im UML-Profil zur Integration der Testfallmodellierung in die Systemmodellierung (vgl. Abschnitt 5.2.4) zulässigen Modellierungselemente bei der modellbasierten Spezifikation von Tests. Zweitens dient es der Modellierung UML-unabhängiger Testfallmodelle, wie sie von der Modelltransformation im ersten Schritt zur Testfallgenerierung aus UML-Aktivitätsdiagrammen erzeugt werden.

In Abbildung 5.13 ist das Metamodell dargestellt. Aus Platzgründung wurden einige Modellelemente verkürzt dargestellt, diese werden an späterer Stelle im Detail erläutert. Ein auf diesem Metamodell basierendes Testfallmodell enthält die zur Testdurchführung notwendigen Informationen, wie z. B. den Einstiegspunkt des Testfalls sowie eine geordnete Menge durchzuführender Aktionen mit zugehörigen Vorbedingungen und Nachbedingungen.

Softwareanwendungen bieten Funktionen an, die voneinander abhängig sein können, d. h. eine Funktion kann u. U. nur dann ausgeführt werden, wenn zuvor eine andere Funktion aufgerufen wurde. Gegebenenfalls bietet eine Anwendung die Möglichkeit, bestimmte Aktionen mehrfach hintereinander auszuführen (z. B. der Aufruf einer Aktualisierungsfunktion), wobei der erneute Aufruf möglicherweise mit vom vorhergehenden Aufruf verschiedenen Parametern erfolgt. Die sich hieraus ergebenden Strukturen im Kontrollfluss einer Anwendung sind beim Testen zu berücksichtigen. Um Anwendungsdefekte auszuschließen, die aus der mehrfach hintereinander folgenden Ausführung einzelner Funktionen oder Gruppen von Funktionen resultieren, kann im Rahmen des Testens gefordert werden, die betroffenen Aktionen eine definierte Anzahl von Wiederholungen mit jeweils individuellen Parametern auszuführen.

Gemäß der in Abschnitt 4.1.1 diskutierten Begriffsdefinition stellen Fälle mit unterschiedlichen Zyklusiterationen jeweils individuelle Testfälle dar. Gleiches gilt für Alternativen im Kontrollfluss. Aus diesem Grund stellt das Metamodell keine Elemente zur Modellierung von

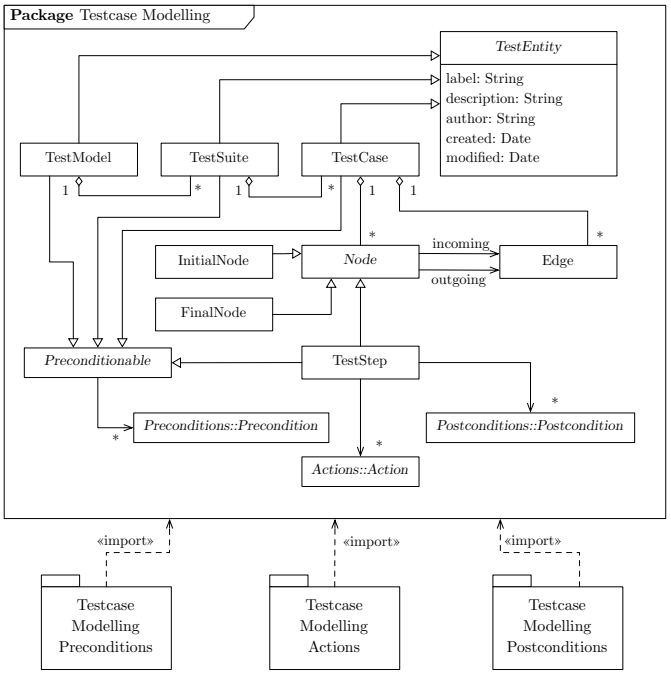


Abbildung 5.13: Metamodell zur Testfallmodellierung

Entscheidungspunkten oder Kontrollflussverzweigung bereit. Im Folgenden werden die Elemente des Metamodells zur Testfallmodellierung erläutert und in ihrer Funktion diskutiert.

5.2.3.2.1 Modellierungselement *Test::TestEntity*

Das Element *Test::TestEntity* generalisiert die Modellelemente *Test::TestModel*, *Test::TestSuite* sowie das Element *Test::TestCase* mit dem Ziel, Metainformationen in diesen Modellelementen homogen verfügbar zu machen. Es handelt es sich um eine abstrakte Klasse, die nicht direkt instanziiert werden kann. Durch dieses Element werden die Anforderungen erfüllt, bestimmte Modellelemente mit Metainformationen anreichern zu können.

5.2.3.2.2 Modellierungselement *Test::TestModel*

Das Element *Test::TestModel* erbt von der Klasse *Test::TestEntity* und repräsentiert einen Container für eine Menge von *TestSuite*-Elementen. In der praktischen Verwendung wird durch eine Instanz dieser Klasse das gesamte zu testende System abgebildet. Es aggregiert eine Menge von *Test::TestSuite*-Elementen. Einzelne *Test::TestSuite*-Elemente bilden Funktionsbereiche des SUT auf Anwendungsfallebene ab. Diese enthalten ihrerseits Testfälle, die mögliche Pfade durch den Graphen des Kontrollflusses der Anwendung repräsentieren.

5.2.3.2.3 Modellierungselement *Test::TestSuite*

Das Element *Test::TestSuite* (vgl. Abschnitt 4.1.1.9) erbt von der Klasse *Test::TestEntity* und aggregiert *Test::TestCase*-Elemente. Funktionale Aufgabe dieses Elements ist die Strukturierung von Testfällen in semantisch zusammengehörige Mengen. Die Bestimmung der Zugehörigkeit ist nicht Teil des Metamodells, sondern liegt im Ermessen des Anwenders. Das Element *Test::TestSuite* erfüllt Anforderung REQ-MM-Testfall 1 (Modellierung von Testsuiten).

5.2.3.2.4 Modellierungselement *Test::TestCase*

Das Element *Test::TestCase* repräsentiert einen Testfall, welcher individuelle Testschritte aggregiert und strukturiert. Dies wird durch die Aggregationsassoziationen realisiert, welche einzelne Testschritte bzw. Relationen zwischen diesen repräsentieren. Die Reihenfolge der *Test::Step*-Elemente innerhalb einer *Test::TestCase*-Instanz wird durch *Test::Edge*-Elemente bestimmt. Das Element *Test::TestCase* genügt damit der Anforderung REQ-MM-Testfall 2.

5.2.3.2.5 Modellierungselement *Test::TestStep*

Das Element *Test::Step* repräsentiert einen Testschritt gemäß der Begriffsdefinition in Abschnitt 4.1.1.8, durch welchen elementare Aktionen, die während des Testens durchzuführen sind, abgebildet werden (Anforderung REQ-MM-Testfall 4). Eine *Test::Step*-Instanz verfügt über eine Assoziation zu einer *Test::Action*.

Weiterhin verfügt *Test::Step* über Assoziationen zu den Elementen *Test::Precondition* und *Test::Postcondition*, durch welche in Testfallmodellen Vor- und Nachbedingungen modelliert werden können. Zu den Vorbedingungen gehören insbesondere Kontextparameter, die durch Verwendung des in Abschnitt 5.2.3.2 diskutierten Metamodells modelliert werden.

Die Modellierungselemente des Metamodells sind in Abbildung 5.14 in einer rudimentären graphischen Notation dargestellt.

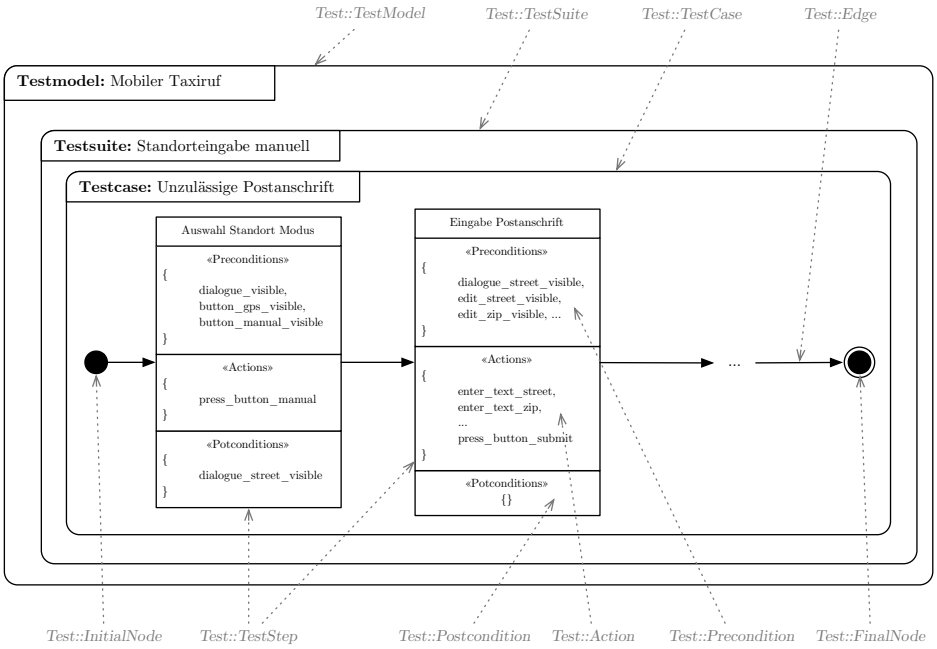


Abbildung 5.14: Ausschnitt des Testmodelles der App Mobiler Taxiruf in einer rudimentären graphischen Notation zur Veranschaulichung der Verwendung der Modellierungselemente des Metamodells zur Testfallmodellierung.

5.2.3.2.6 Modellierungselement *Test::Action*

Anforderung REQ-MM-Testfall 5 (Testaktionen) an ein Metamodell zur Testmodellierung wird durch das Element *Test::Action* erfüllt. Dieses repräsentiert konkrete während eines

Testschritts durchzuführende Aktionen. Hierbei kann es sich u. a. um Interaktion mit der Benutzungsoberfläche handeln oder es kommen auch solche Interaktionen mit der zu testenden Anwendung in Frage, die sich nicht direkt auf Elemente der Benutzungsoberfläche beziehen, z. B. Abwarten eines Zeitintervalls oder Verändern der Orientierung des Smartphones.

Das Metamodell definiert eine Reihe unterschiedlicher Subklassen von *Test::Action*, die hier in ihrer Gesamtheit nicht aufgezählt werden können. Beispielhaft werden deshalb ausgewählte Subklassen erläutert, um die Vielgestaltigkeit dieser Subklassen anzudeuten.

Physikalische Kontextparameter sind i. d. R. naturgegeben und können durch einen menschlichen Akteur nicht verändert werden. Eine Ausnahme hiervon bilden einige logische Kontextparameter, deren Semantik sich durch das Verhältnis des SUT, des Geräts oder des Anwenders zum Kontext bestimmt. Beispielsweise kann ein menschlicher Akteur das Magnetfeld der Erde nicht manipulieren, aber die Wahrnehmung dieses Parameters durch die Sensoren des Geräts, indem das Gerät gezielt orientiert wird (Roll-, Nick- und Gierwinkel). Um solche Aktionen im Testmodell abbilden zu können, definiert das Metamodell eine Klasse von Modellierungselmenten *Test::ContextAction*, mit denen die Manipulation bestimmter Kontextparameter modelliert werden können.

In Abbildung 5.15 ist beispielhaft der Ausschnitt des Metamodells dargestellt, der das Modellierungselement *Test::SetOrientationAction* definiert. Es referenziert das Element *Context::OrientationContext* des in Abschnitt 5.2.2.2 entworfenen Metamodells zur Kontextmodellierung. Ein menschlicher Tester, der das Element *Test::SetOrientationAction* im Testmodell vorfindet, wird das Gerät als Teil des Tests in die durch das Modell spezifizierte Orientierung bringen. Bei einer Testdurchführung mit einer Automatisierungstechnologie wird das Element *Test::SetOrientationAction* zusammen mit dem assoziierten *Context::OrientationContext*-Element zur Simulation von Kontextparametern verwendet, in diesem Beispiel zur Simulation spezifischer Werte für Magnetfeldsensor und Beschleunigungssensor.

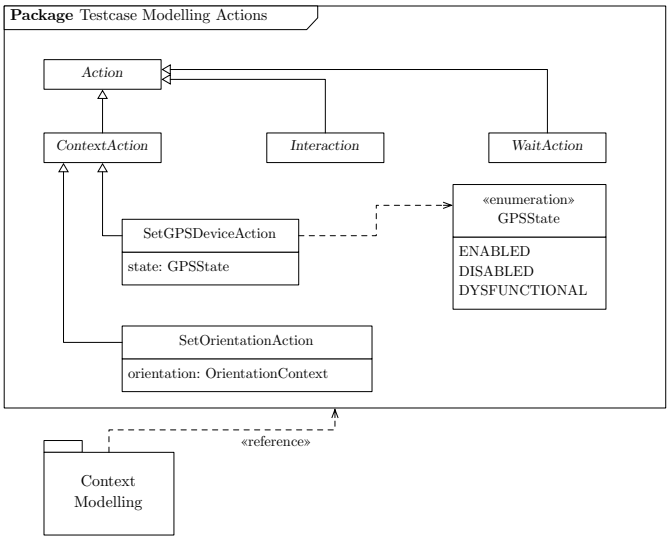


Abbildung 5.15: Metamodell zur Testfallmodellierung, Ausschnitt *Test::ContextAction*

Test::Action-Elemente können ebenfalls Interaktionen mit der Benutzungsoberfläche des SUT beschreiben. Hier muss zwischen solchen Interaktionen unterschieden werden, die ein spe-

zifisches Darstellungselement der Benutzungsoberfläche zum Gegenstand haben (z. B. Texteingabe in ein spezifisches Textfeld) und solchen, die sich auf die gesamte Benutzungsoberfläche beziehen (z. B. Wischgeste über die gesamte Displayfläche).

Das Element *Test::UserInterfaceInteraction* repräsentiert atomare Aktionen, die sich zwar auf die Benutzungsoberfläche, nicht jedoch auf spezifische Elemente darin beziehen. Dabei kann es sich z. B. um Touch-Gesten handeln. Hierbei ist es nicht notwendig, dass die Geste auf einem spezifischen Element der Benutzungsoberfläche ausgeführt wird, sondern die Interaktion wird auf der gesamten Displayfläche akzeptiert. Eine solche Interaktion kann mit Hilfe des Metamodellelements *Test::SwipeGestureAction* modelliert werden. Hierbei können Startpunkt, Endpunkt und Geschwindigkeit, in welcher die Touch-Geste ausgeführt wird, spezifiziert werden. Analog kann das Element *Test::ShakeGestureAction* dazu verwendet werden, eine Schüttelgeste des gesamten Smartphone zu modellieren. Der entsprechende Ausschnitt des Metamodells ist in Abbildung 5.16 dargestellt.

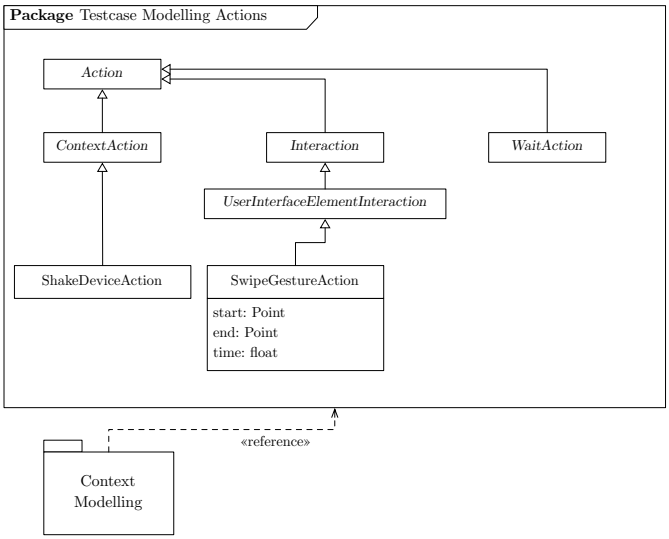


Abbildung 5.16: Metamodell zur Testfallmodellierung, Ausschnitt *Test::UserInterfaceInteraction*

Das Element *Test::UserInterfaceElementInteraction* ermöglicht die Modellierung atomarer Interaktionen mit spezifischen Elementen der Benutzungsoberfläche, etwa eine Texteingabe in ein bestimmtes Textfeld oder Anklicken eines Elements der Benutzungsoberfläche. In diesen Fällen ist neben Eingabedaten ebenfalls eine eindeutige Identifikation des betreffenden Elements zu spezifizieren. Gültige Interaktionsziele sind Elemente der Benutzungsoberfläche, die durch ein eindeutiges Identifikationsmerkmal[4] oder durch ihre Position in der Hierarchie der Benutzungsoberfläche modelliert werden. Texteingaben können durch das Element *Test::TextInputAction* modelliert werden, Anklicken von Elementen der Benutzungsoberfläche mit dem Metamodellelement *Test::ClickUIComponentAction* bzw. *Test::ClickListRowAction*. Der entsprechende Ausschnitt des Metamodells ist in Abbildung 5.17 dargestellt.

Das Metamodell definiert darüber hinaus eine Reihe weiterer Aktionen, die nicht unmittelbar eine Interaktion des Anwenders repräsentieren. Element *Test::WaitTimeIntervallAction*

[4] Auf der Plattform Android verfügt jedes UI-Element über eine eindeutige Identifikationsnummer. Alternativ könnte ebenfalls der durch die Accessibility-Richtlinien zu definierende und zum UI-Element gehörige Beschreibungstext verwendet werden.

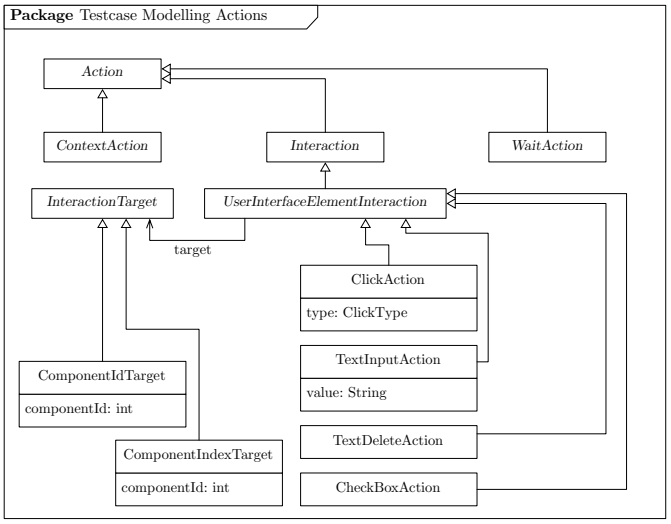


Abbildung 5.17: Metamodell zur Testfallmodellierung, Ausschnitt *Test::UserInterfaceElementInteraction*

beispielsweise ermöglicht die Spezifikation von Aktionen, die lediglich das Abwarten eines definierten Zeitintervalls zum Inhalt haben. Anwendung findet diese Aktion z. B. in Fällen, in denen eine Anforderungsspezifikation definiert, dass eine Aktion nicht länger als einen bestimmten Zeitraum in Anspruch nehmen darf. Denkbar wäre hier beispielweise die Anforderung, dass das Laden von Inhalten aus einer lokalen Datenbank nicht länger als zwei Sekunden dauern darf bevor eine Aktualisierung der Benutzeroberfläche erfolgen muss. Ein Testfall könnte beispielsweise das Auslösen einer anderen Aktion in einem vorherigen Testschritt beinhalten, dann das Abwarten des Zeitintervalls und anschließend die Überprüfung der Nachbedingung (Benutzungsoberfläche).

5.2.3.2.7 Modellierungselement *Test::Precondition*

In Übereinstimmung mit den Anforderungen REQ-MM-Testfall 6 (Vorbedingungen) an ein Metamodell zur Testfallmodellierung ermöglichen Elemente des Typs *Test::Precondition* die Modellierung von Kontextparametern. Das Metamodell sieht hierbei Klassen zur Modellierung physikalischer und logischer Kontextparameter vor.

Instanzen des Elements *Test::Precondition* können mit Instanzen der Elemente *Test::TestModel*, *Test::TestSuite*, *Test::TestCase* und *Test::Step* assoziiert werden. Hierdurch ist es möglich, Vorbedingungen in Abstufungen auf ein ganzes Testmodell oder untergeordnete Hierarchien anzuwenden. Es entbindet den Entwickler des Testmodells davon, gleichbleibende Vorbedingungen für jede Instanz von *Test::Step* einzeln zu definieren, wenn diese Vorbedingung für einen gesamten Testfall oder sogar für das gesamte Modell gelten sollen (z. B. eine Standortinformation, die sich über mehrere Testfälle innerhalb einer Testsuite nicht verändert). Der entsprechende Ausschnitt des Metamodells ist in Abbildung 5.18 dargestellt.

Das Metamodell zur Testmodellierung definiert Subklassen zur Beschreibung unterschiedlicher Vorbedingungen. Kontextparameter werden durch die Subklasse *Test::ContextPrecondition* modelliert. Es referenziert das Element Metamodell zur Kontextmodellierung (vgl. Abschnitt 5.2.2.2) und kann alle dort definierten Kontextparameter modellieren, wie etwa Sen-

sorwerte (*Context::SensorValue*, *Context::SensorInterpolation*), die Geräteorientierung (*Context::DeviceOrientation*, *Context::DiscreteDeviceOrientation*), Standort oder Bewegung des Anwenders (*Context::Location*, *Context::Motion*).

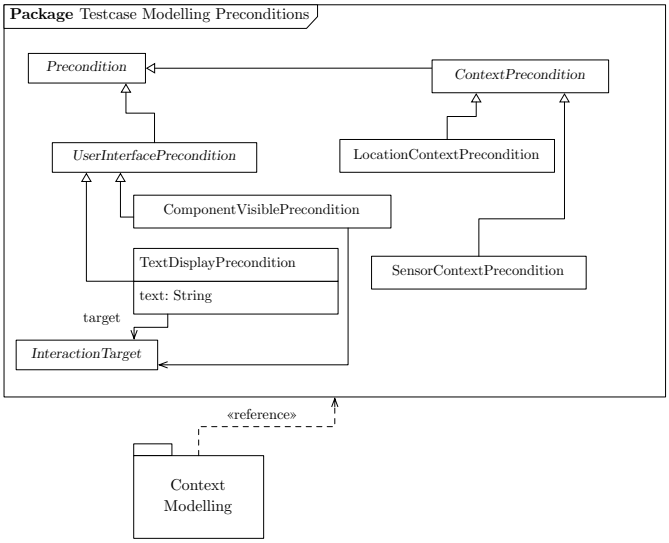


Abbildung 5.18: Metamodell zur Testfallmodellierung, Ausschnitt *Test::Precondition*

Das Element *Test::Precondition* repräsentiert die für die Durchführung einer konkreten Aktion gültigen Vorbedingungen. Zu diesen gehören neben dem Kontext ebenfalls der Zustand des SUT. Das Metamodell definiert deshalb Modellierungselemente, um den Zustand abzubilden, etwa die Sichtbarkeit bestimmter UI-Elemente oder die Anzeige bestimmten Textes in UI-Elementen.

5.2.3.2.8 Modellierungselement *Test::Postcondition*

Zur Realisierung der Anforderung REQ-MM-Testfall 7 (Nachbedingungen) an ein Metamodell zur Testfallmodellierung stellt das Metamodell Elemente des Typs *Test::Postcondition* bereit. Diese dienen der Überprüfung des tatsächlichen Systemzustands nach der Durchführung einer Aktion auf Gültigkeit der Nachbedingungen.

Die Gültigkeit der Nachbedingungen charakterisieren einen Testschritt genau dann als erfolgreich bestanden, wenn der tatsächliche Systemzustand nach der Durchführung einer Aktion die für diese Aktion geltenden Nachbedingungen erfüllt. Durch das Element *Test::Postcondition* wird die Anforderung REQ-MM-Testfall 7 erfüllt, indem durch spezialisierende Elemente charakteristische Nachbedingungen modelliert werden. Hierzu zählt beispielsweise die Überprüfung konkreter Elemente der Benutzungsoberfläche auf Spezifikationskonformität (z. B. Vergleich des Inhaltes eines Textfeldes mit einem Orakel). Eine mögliche Anwendung ist etwa der Vergleich des gegenwärtigen Bildschirminhalts mit einer im Vorfeld erstellen Vorlage (*Test::ScreenshotAssertion*). Ein anderer Anwendungsfall ist die Überprüfung spezifischer Elemente der Benutzungsoberfläche auf bestimmte Eigenschaften, wie etwa Sichtbarkeit oder Textinhalt (*Test::AssertContainsTextPostcondition*). Abbildung 5.19 illustriert den entsprechenden Ausschnitt des Metamodells.

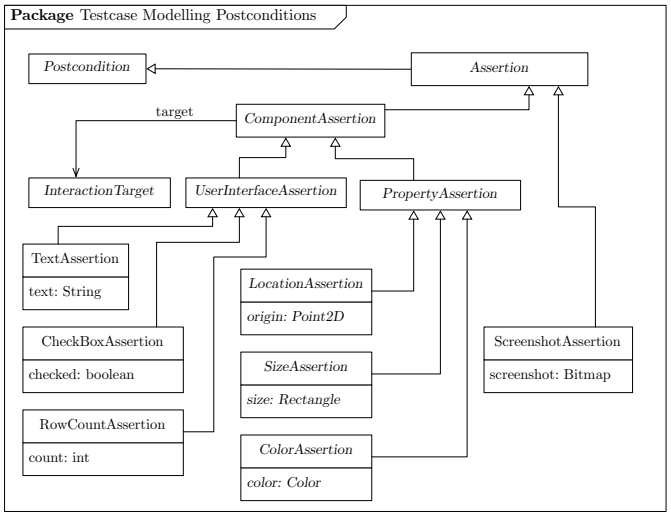


Abbildung 5.19: Metamodell zur Testfallmodellierung, Ausschnitt *Test::Postcondition*

5.2.3.2.9 Modellierungselement *Test::Edge*

Das Element *Test::Edge* dient der Strukturierung von *Test::Step*-Elementen durch gerichtete Kanten. Hierzu verfügt es über Assoziationen zu Quell- und Zielknoten. In der praktischen Anwendung repräsentieren *Test::Edge*-Elemente Verbindungen zwischen einzelnen *Test::Step*-Elementen, wodurch explizit deren Ausführungsreihenfolge definiert wird. Hierdurch wird Anforderung 3 an Testfallmodelle erfüllt.

5.2.3.2.10 Modellierungselement *Test::InitialNode* und *Test::FinalNode*

Das Metamodell zur Testfallmodellierung strukturiert Testfälle in sequenziell miteinander verkettete Instanzen von *Test::Step*. Um sowohl die visuelle Identifikation des Beginns und des Endes eines Testfalls zu erleichtern als auch die maschinelle Verarbeitung von Testmodellen zu vereinfachen, definiert das Metamodell zwei Steuerknoten zur Auszeichnung des Anfangs und des Endes individueller Testfälle innerhalb eines Testmodells.

Das Metamodell lässt es ausdrücklich zu, dass identische Sequenzen von *Test::Step*-Elementen in mehreren *Test::TestCase*-Elementen vorkommen dürfen. Ebenfalls ist es zulässig, dass unterschiedliche *Test::TestCase*-Elemente strukturell identisch aufgebaut sind, d. h. dieselben *Test::Step*-Elemente in identischer Reihenfolge enthalten, für welche jedoch unterschiedliche Vorbedingungen gelten.

Das Element *Test::InitialNode* repräsentiert den Startpunkt eines Testfalls. Dieser Knotentyp bildet selbst keine funktionalen Aspekte ab. Da es sich beim Element *Test::TestCase* analog zu UML-Aktivitätsdiagrammen ebenfalls um einen gerichteten Graphen handelt, wird der Knotentyp *Test::InitialNode* dediziert zur Markierung des Beginns eines Testfalls verwendet. Er zeichnet sich dadurch aus, dass er über keine eingehenden Kanten verfügt.

Das Element *Test::FinalNode* modelliert den Endpunkt eines Testfalls. Der Knoten selbst erfüllt keine weitere semantische Funktion. *Test::FinalNode*-Elemente verfügen über keine ausgehenden Kanten.

5.2.3.3 Verwendung des Metamodells zur Testfallmodellierung

Das in Abschnitt 5.2.3.2 vorgestellte Metamodell zur Testfallmodellierung ist Kernbestandteil der hier untersuchten Methode zur Testautomatisierung. Das Werkzeug zur Generierung von Testfällen (vgl. Abschnitt 5.5.1) identifiziert Elemente des Metamodells zur Testfallmodellierung in angereicherten UML-Aktivitätsdiagrammen. Bei der Transformation des Aktivitätsdiagramms zu Testfallmodellen werden zunächst plattformunabhängige Testfallmodelle erstellt. Hierbei handelt es sich um Instanzen von *Test::TestModel*. Bei dieser Transformation werden Elemente des Metamodells zur Testfallmodellierung aus dem Aktivitätsdiagramm unverändert in die *Test::TestModel*-Instanz übernommen. Bei der weiteren Transformation zu technologiespezifischen Tests werden diese Informationen verwendet, um individuelle Testanweisungen (z. B. Assertions) zu parametrisieren.

Gegenüber einer direkten, technologiespezifischen Realisierung von Tests in Code (z. B. JUnit oder auch Calabash) bleiben so Vorteile modellbasierter Ansätze erhalten. Einerseits hilft eine Modellierung von Tests unter Abstraktion von technischen Details auf inhaltliche Aspekte des Testens zu fokussieren. Andererseits kann bei einem modellbasierten Vorgehen die Entscheidung für eine bestimmte Testautomatisierungstechnologie auf einen möglichst späten Zeitpunkt verlagert werden und muss auch dann nicht endgültig festlegt werden (vgl. Abschnitt 5.4). Anpassungen an einem Modell sind i. d. R. für einen menschlichen Akteur einfacherer und effizienter realisierbar, als die manuelle Anpassung einer großen Anzahl von Testfällen, die bereits in Code vorliegen.

Das in dieser Dissertation untersuchte Konzept der modellbasierten Testautomatisierung für mobile, kontextsensitive Anwendung ist von der Intention her dafür ausgelegt, einen Mehrwert durch automatische Generierung von technologiespezifischen Testfällen auch in Softwareprozessen zu erzeugen, die eine formale Modellierung des Systems nicht erzwingen (z. B. agile Vorgehensmodelle wie Scrum). Deshalb lässt die hier untersuchte Testautomatisierungstechnologie das manuelle Erstellen von Testfallmodellen explizit zu.

Einige Elemente des Metamodells finden ihre Anwendung nicht ausschließlich in Testfallmodellen. Die hier untersuchte Methode zur Testgenerierung (vgl. Abschnitt 5.1) sieht in der zweiten Phase vor, noch vor der Transformation zu Testfallmodellen UML-Modelle mit testspezifischen Daten zu verfeinern. Diese Verfeinerung manifestiert sich durch die Spezifikation von Vorbedingungen, Aktionen und Nachbedingungen mit Hilfe der im Metamodell definierten Elemente *Test::Precondition*, *Test::Action* und *Test::Postcondition*. Instanzen dieser Elemente werden dem UML-Aktivitätsdiagramm dann unter Verwendung des UML-Profils zur Integration der Testfallmodellierung in die Systemmodellierung (vgl. Abschnitt 5.2.4) durch Anwendung von Stereotypen auf die Elemente des Aktivitätsdiagramms hinzugefügt.

5.2.4 UML-Profil zur Integration der Testfallmodellierung in die Systemmodellierung mit der UML

Um die in den vorangegangenen Kapiteln diskutierten Metamodelle zur Kontext- bzw. Testfallmodellierung in die Systemmodellierung zu integrieren, wird im Folgenden ein UML-Profil vorgestellt. Mit diesem ist es möglich, UML-Aktivitätsdiagramme um testrelevante Parameter anzureichern, d. h. Kontextparameter und Testdaten an Elementen von Aktivitätsdiagrammen

zu annotieren. Hierzu definiert das Profil Stereotypen für Modellierungselemente der UML. In Abbildung 5.20 ist das UML-Profil in einem Profildiagramm dargestellt.

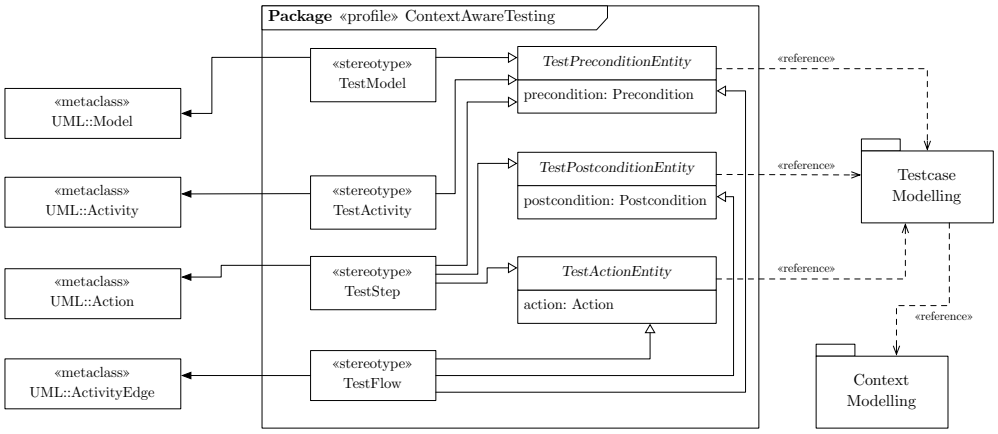


Abbildung 5.20: UML-Profildiagramm zur Integration der Testfallmodellierung in die Systemmodellierung mit der UML

5.2.4.1 Stereotypen zur Testfallparametrisierung

Bei der in Abschnitt 5.3.3 beschriebenen Transformation von UML-Aktivitätsdiagrammen zu plattformunabhängigen Testfallmodellen werden Aktivitätsdiagramme durch einen Algorithmus (vgl. Abschnitt 5.3.3) traviersiert, analysiert und entsprechend der Annotation mit Stereotypen des hier diskutierten Profils zu Testfallmodellen des in Abschnitt 5.2.3.2 vorgestellten Metamodells überführt.

Um UML-Aktivitätsdiagramme für die Generierung von Testfällen nutzbar zu machen, müssen die für die Testdurchführung relevanten Informationen im Systemmodell hinterlegt und mit Modellelementen assoziiert werden. Hierzu stellt das UML-Profil zur Testfallmodellierung die Stereotypen *TestModel*, *TestActivity*, *TestStep* und *TestFlow* bereit, welche zur Annotation von Modellierungselementen des UML-Metamodells verwendet werden.

Von den durch die UML bereitgestellten Modellierungselementen sind einige für die hier untersuchte Testfallgenerierung von besonderer Relevanz. Hierbei handelt es sich um die Elemente, die zur Modellierung von Aktivitätsdiagrammen definiert sind, konkret um die Elemente *UML::Model*, *UML::Activity*, *UML::Action* bzw. dessen Subtypen und Elemente des Typs *UML::ActivityEdge* (vgl. OMG [268, S. 414]). Elemente des Typs *UML::Action* beschreiben Aspekte des Systemverhaltens in individuellen Aktionen, *UML::ActivityEdge*-Elemente setzen Aktionen in strukturierte Relation zueinander.

Die im Folgenden diskutierten Stereotypen *TestModel*, *TestActivity*, *TestStep* und *TestFlow* können bei der Modellierung des Systemverhaltens auf diese Modellierungselemente appliziert werden (Fachterminologie der Modellierung mit UML zur Anwendung von Stereotypen).

5.2.4.1.1 Stereotyp *TestModel*

Die UML wird verwendet, um Systemmodelle ganzheitlich und einschließlich statischer und dynamischer Aspekte zu modellieren. Der semantische Zusammenhang zwischen Modellen,

die dasselbe System beschreiben, manifestiert sich in der technischen Implementierung durch eine hierarchische Beziehung unterschiedlicher Teilmodelle unterhalb eines übergeordneten Systemmodells des Typs *UML::Model*.

Zur Vereinfachung der Annotation von Modellen mit Kontext- und Testdaten zur Testmodellierung definiert das hier diskutierte UML-Profil den Stereotyp *TestModel*. Er erweitert das Modellierungselement *UML::Model* des Metamodells der UML und erlaubt die Definition solcher Vorbedingungen, die für jeden einzelnen aus dem Modell generierten Testfall gelten. Der Stereotyp referenziert die in den Abschnitten 5.2.2.2 sowie 5.2.3.2 definierten Metamodelle zur Kontext- bzw. Testfallmodellierung. Dem Anwender ist es hierdurch möglich, ein Objektmodell für global gültige Elemente des Typs *Test::Precondition* zu definieren und durch Anwendung des Stereotyps *TestModel* mit einem UML-Modell zu assoziieren. Bei der Generierung von Testfällen erzeugt der Algorithmus hieraus Testfallmodelle, die diese Objekte als Vorbedingungen für das gesamte Testfallmodell spezifiziert.

Denkbar ist beispielsweise die Verwendung des Stereotyp *TestModel* zur Spezifikation einer Instanz von *Context::DiscreteDeviceOrientation* (vgl. Abschnitt 5.2.2.2.3) für Anwendungen, die ausschließlich für die Verwendung eines mobilen Geräts im Hoch- oder Querformat konzipiert sind. Der Testfallgenerator erzeugt hieraus ein solches Testfallmodell, das eine global über alle individuellen Testfälle hinweg gültige Vorbedingung zur Geräteorientierung spezifiziert. Eine Testautomatisierungstechnologie mit Kontextsimulation (vgl. Abschnitt 5.5.1) identifiziert diese Vorbedingung und generiert bei der Testausführung Werte für Kontextparameter, die der spezifizierten Geräteorientierung entsprechen. Im konkreten Beispiel also Werte für Beschleunigungssensor, Gravitationssensor und Magnetfeldsensor.

5.2.4.1.2 Stereotyp *TestActivity*

Modelle von Softwaresystemen können mehrere untergeordnete Teilmodelle mit zugehörigen Diagrammen enthalten. Insbesondere kann ein UML-Modell mehrere UML-Aktivitätsdiagramme enthalten, die sich u. U. in einer hierarchischen Relation zueinander befinden (z. B. das Element *UML::CallBehaviorAction* zum Aufruf einer untergeordneten *UML::Activity* aus einem *UML::Action*-Element heraus, vgl. OMG [268, S. 536]).

Aktionen, die in UML-Aktivitätsdiagrammen modelliert werden, spezifizieren unterschiedliche Arten von Verhaltensaspekten. Sie können beispielsweise den Aufruf einer konkreten Operation (z. B. den Aufruf einer Methode) oder den Aufruf einer eingebetteten Verhaltensbeschreibung modellieren. Letztere erlaubt die Spezifikation des Aufrufs einer weiteren *UML::Activity*, durch welche der beschriebene Systemverhaltensaspekt verfeinert und konkretisiert wird. Gegebenenfalls liegt diese Konkretisierung auf einer Abstraktionsebene, die für die Generierung von Testfällen ungeeignet oder nicht relevant ist. Ein Beispiel ist in Abbildung 5.21 dargestellt. Im Beispiel sind zwei Aktionen modelliert, eine davon repräsentiert eine komplexe Aktion (angedeutet durch das Symbol unten rechts), die durch ein separates Aktivitätsdiagramm modelliert wird.

Für die Generierung von Testfällen ist eine Möglichkeit zur Differenzierung relevanter und irrelevanter Teilmodelle erforderlich. Einerseits könnte nur eine Teilmenge der Aktivitätsdiagramme eines Systemmodells einer Anwendung durch automatisierte Tests adressierbar sein. Dies kann z. B. der Fall sein, wenn das Systemmodell sowohl mobile Komponenten, d. h. Apps,

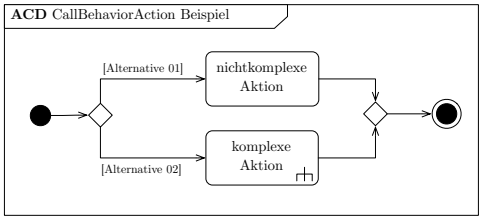


Abbildung 5.21: *UML::CallBehaviorAction* repräsentiert separat modelliertes Systemverhalten

als auch Komponenten des Backend-Systems spezifiziert, die nicht gemeinsam unter Verwendung derselben Testautomatisierungstechnologie getestet werden können. Andererseits muss es möglich sein, eine willkürliche Selektion von Teilmodellen einer automatisierten Testgenerierung zuzuführen und andere Teilmodelle unberücksichtigt zu lassen, z. B. weil in einem iterativen Softwareprozess Teile der Anwendung erst in späteren Iterationen realisiert werden sollen und deshalb früh im Softwareprozess nicht zum Testen bereitstehen.

Um *UML::Activity*-Elemente, die bei der Testfallgenerierung berücksichtigt werden sollen, von solchen zu unterscheiden, die ignoriert werden sollen, stellt das hier diskutierte UML-Profil den Stereotyp *TestActivity* bereit. Dieser erweitert die Metaklasse *UML::Activity* und erbt analog zum Stereotyp *TestModel* von der abstrakten Klasse *TestPreconditionEntity* (vgl. Abbildung 5.20). Diese definiert ein Referenzattribut zur Repräsentierung einer Menge von Instanzen des Typs *Test::Precondition* aus dem Metamodell zur Testfallmodellierung.

Der Stereotyp *TestActivity* erfüllt zwei Funktionen. Zunächst dient er als Markierung derjenigen *UML::Activity*-Elemente, die bei der Transformation von UML-Aktivitätsdiagrammen zu Testfallmodellen berücksichtigt werden sollen. Nur solche Aktivitätsdiagramme, auf welche der Stereotyp *TestActivity* angewendet wurde, werden bei der Transformation explizit berücksichtigt (vgl. Definition 2, Abschnitt 5.3.2). Aktivitätsdiagramme, auf welche der Stereotyp nicht angewendet wurde, können jedoch implizit bei der Testfallgenerierung berücksichtigt werden, wenn auf ein Element innerhalb des nicht annotierten Aktivitätsdiagramms einer der anderen durch das UML-Profil definierten Stereotypen angewendet wurde. Ein Tester kann hierdurch gezielt Aktivitätsdiagramme modellieren, die selbst nicht direkt zu Testfällen transformiert werden sollen, aber Kontext- und Testdaten spezifizieren, die in Testfällen verwendet werden, die aus anderen Modellen desselben Softwaresystems generiert werden.

Die zweite Funktion des Stereotyps *TestActivity* ist die Spezifikation von Vorbedingungen, die lokal beschränkt auf das annotierte Aktivitätsdiagramm gelten sollen (analog Stereotyp *TestModel*, Abschnitt 5.2.4.1.1). Es entbindet den Tester von der Modellierung dieser Vorbedingungen für individuelle *UML::Action*- oder *UML::ActivityEdge*-Elemente.

In Abbildung 5.22 ist die Verwendung der Stereotypen am Beispiel der App Mobiler Taxiruf dargestellt. Die Abbildung zeigt einen Ausschnitt des Modells der Anwendung. Auf das Modell selbst wurde der Stereotyp *TestModel* angewendet. Auf die Aktivität „Ortsbasierter Taxiruf“ wurde der Stereotyp *TestActivity* angewendet. Bei der Transformation des Modells wird dieses Aktivitätsdiagramm deshalb explizit adressiert, d. h. nach der Transformation existiert ein *Test::TestSuite*-Element mit identischem Titel, in welchem alle durch die Transformation generierten Testfälle als *Test::TestCase*-Instanzen enthalten sind.

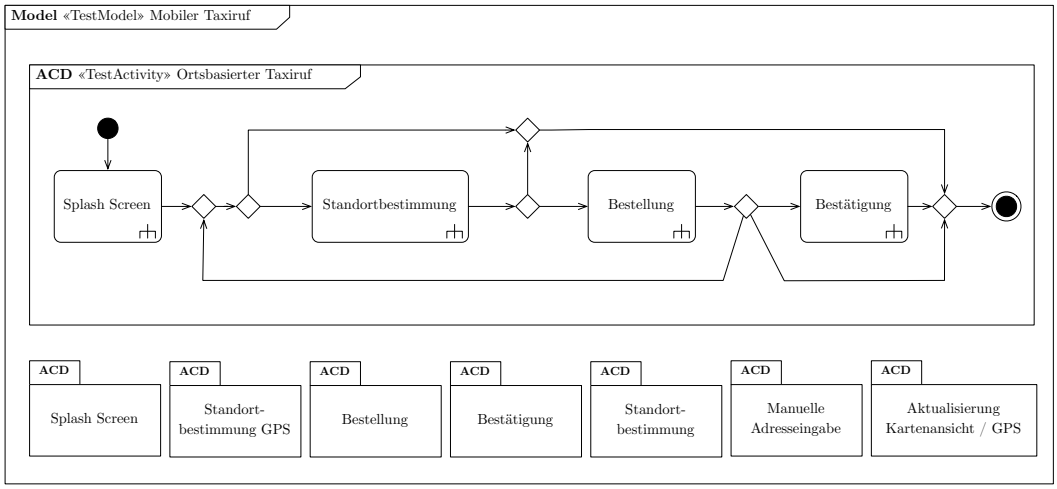


Abbildung 5.22: Die Stereotypen *TestModel* und *TestActivity* am Beispiel der App Mobiler Taxiruf. Dargestellt ist ein Ausschnitt des Systemmodells, das acht Aktivitätsdiagramme zeigt, von denen das Aktivitätsdiagramm Ortsbasierter Taxiruf mit *TestActivity* annotiert ist. Es wird deshalb explizit bei der Transformation zu Testfallmodellen adressiert.

Auf die übrigen Aktivitäten wurde der Stereotyp *TestActivity* nicht angewendet. Diese werden nach der Transformation deshalb nicht durch *Test::TestSuite*-Elemente repräsentiert. Diese Aktivitäten spezifizieren jedoch auf einer feineren Detailstufe das Verhalten der Anwendung, so dass individuelle Aktionen innerhalb dieser Aktivitäten als *Test::Step*-Elemente im resultierenden Testfallmodell enthalten sein können, wenn der Stereotyp *TestStep* auf diese Aktionen angewendet wird.

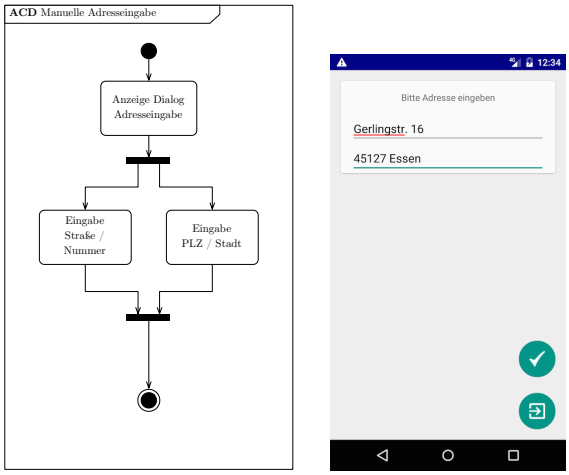
5.2.4.1.3 Stereotyp *TestStep*

Der Stereotyp *TestStep* erweitert die UML-Metaklasse *UML::Action* um Attribute, die das Metamodell zur Testfallmodellierung (vgl. Abschnitt 5.2.3.2) referenzieren. In Abgrenzung zu den Stereotypen *TestModel* und *TestActivity* definiert der Stereotyp *TestStep* neben dem Referenzattribut auf Instanzen von *Test::Precondition* ebenfalls Referenzattribute auf *Test::Action* und *Test::Postcondition* (vgl. Abbildung 5.20).

Die Anwendung von *TestStep* auf *UML::Action*-Elemente ermöglicht die Spezifikation von Vorbedingungen, Nachbedingung und konkretisiert die in der *UML::Action* durchzuführende Aktion insofern, als dass zusätzlich zur Bezeichnung der Aktion eine spezifische Interaktion mit dem zu testenden System modelliert werden kann. Die Eigenschaftsdefinitionen des Stereotyps *TestStep* repräsentieren Vorbedingungen, Nachbedingungen sowie atomare Aktionen.

Aufgabe der Eigenschaftsdefinition der Vorbedingungen ist die Aggregation einer beliebigen Anzahl von Instanzen des Typs *Test::Precondition*, welche bei der Durchführung des Testschritts dazu verwendet werden, die zur Durchführung notwendigen Vorbedingungen einschließlich der Simulation von Kontextparametern herzustellen. Entspricht der Zustand des zu testenden Systems nicht den Vorbedingungen, muss es durch technische Maßnahmen in einen Zustand versetzt werden, der alle kontextspezifischen Vorbedingungen erfüllt. Dies kann beispielsweise durch Einspeisen künstlicher Werte für Sensoren erreicht werden. Ebenfalls kann

sie zur Modellierung weiterer Kontextparameter (z. B. Standortinformationen) verwendet werden, die zur Ausführung des Testschritts gültig sein müssen.



(a) Aktivitätsdiagramm (b) Bildschirmabdruck

Abbildung 5.23: Spezifikation von Interaktion in Aktivitäten. In der Regel werden Aktionen in Aktivitätsdiagrammen mit sprechenden Bezeichnungen versehen. Aus diesen Bezeichnungen sinnvolle Parameter für eine Testautomatisierung abzuleiten ist nicht trivial und wird in dieser Dissertation nicht untersucht.

Abbildung 5.23 zeigt ein Aktivitätsdiagramm ohne Anwendung des Stereotyps *TestStep*. Ein menschlicher Akteur ist in der Lage, der Bezeichnung individueller Aktionen Semantik zuzuordnen. Für die maschinelle Verarbeitung ist das hingegen ein nicht triviales Problem, welches in dieser Dissertation nicht betrachtet wird. Das Beispiel zeigt ein Teilmodell der App Mobiler Taxiruf, in dem der Anwender aufgefordert ist, Standortinformationen durch manuelle Eingabe bereitzustellen. Das UI der App verfügt hierzu über zwei Textfelder, in die in beliebiger Reihenfolge Standortinformationen im Format einer partiellen Postanschrift einzutragen ist. Dem menschlichen Akteur erschließt sich unmittelbar, welcher Teil der Postanschrift in welches der beiden Textfelder einzutragen ist, einer maschinellen Testautomatisierungstechnologie hingegen nicht. Dies muss explizit instruiert werden, in welches der Textfelder Testdaten in welchem Format einzutragen sind.

Die Erweiterung der Metaklasse *UML::Action* um eine Referenz zu *Test::Action* beschreibt die durchzuführende Aktion. Was hier am Beispiel der Texteingabe ausgeführt ist, gilt in dieser Form analog für andere Arten der Interaktion mit der Anwendung oder dem Gerät, etwa Einstellen von Schiebereglern, Anklicken von Schaltflächen oder Rotation des Geräts in eine vorgegebene Orientierung. Hierbei muss es sich um solche Aktionen handeln, die im Rahmen eines Anwendungstest für den testenden Akteur bzw. die Testautomatisierungstechnologie erreichbar ist. Aktionen, die in Hintergrundprozessen autonom vom System ohne Möglichkeit zur Interaktion durch einen Anwender durchgeführt werden, fallen nicht in diese Kategorie und entziehen sich dem Zugriff in Akzeptanztests.

Spezifiziert werden können als gültige Aktionen solche, die durch das Metamodell zur Testfallmodellierung als *Test::Action* definiert werden. Beispielsweise können hier die Typen *Test::UserInterfaceInteraction* oder *Test::UserInterfaceElementInteraction* verwendet werden.

Auf diese Weise kann im Aktivitätsdiagramm eine konkrete *UML::Action* durch Anwendung des Stereotyp *TestStep* modellieren, dass an dieser Stelle eine Interaktion mit dem UI erfolgt, etwa Eingabe von Text oder Anklicken einer Schaltfläche (*Test::TextInputAction* bzw. *Test::ClickAction*). Zusammen mit dem Metamodell zur Kontextmodellierung ist es hier durch den Typ *Test::ContextAction* ebenfalls möglich, Interaktionen mit dem Gerät selbst zu modellieren, etwa die Orientierung des Geräts im Hoch- oder Querformat (*Text::SetOrientationAction*) oder auch eine Schüttelgeste (*Test::ShakeDeviceAction*).

Weiterhin definiert der Stereotyp *TestStep* ein Referenzattribut auf *Test::Postcondition*-Elemente. Es dient als Container für eine beliebige Anzahl von Nachbedingungen. Die Definition der Nachbedingungen entspricht (analog zu den Vorbedingungen) der Definition des Typs *Test::Postcondition* im Metamodell zur Testfallmodellierung.

Bei der Durchführung von Testschritten werden zunächst die notwendigen Vorbedingungen hergestellt und anschließend die durch den Testfall spezifizierten Aktionen durchgeführt. Die Qualifizierung eines Testschritts als bestanden oder nicht bestanden erfolgt anhand eines Vergleichs des tatsächlichen Systemzustands nach der Durchführung mit den durch die Nachbedingungen des Testschritts referenzierten Instanzen des Typs *Test::Postcondition*, konkretisiert durch die Subklassen *Test::UserInterfaceAssertion* oder *Test::PropertyAssertion*, wie z. B. *Test::TextAssertion* zur Überprüfung, ob ein UI-Element den korrekten Text anzeigt.

Abbildung 5.24 zeigt das Aktivitätsdiagramm aus Abbildung 5.23a. Hier wurde auf einige *UML::Action*-Elemente der Stereotyp *TestStep* angewendet. Modelliert ist konkret die Eingabe von Werten in die Texteingabefelder des in Abbildung 5.23b dargestellen UI. Die von den *UML::Action*-Elementen durch Anwendung des Stereotyps *TestStep* referenzierten Vorbedingungen und Aktionen, Typen von *Test::Action*, werden durch ein rudimentäres UML-Objektmodell abgebildet. Ebenfalls modelliert ist die Sicherstellung der Sichtbarkeit der beteiligten UI-Elemente durch *Test::Precondition*-Elemente.

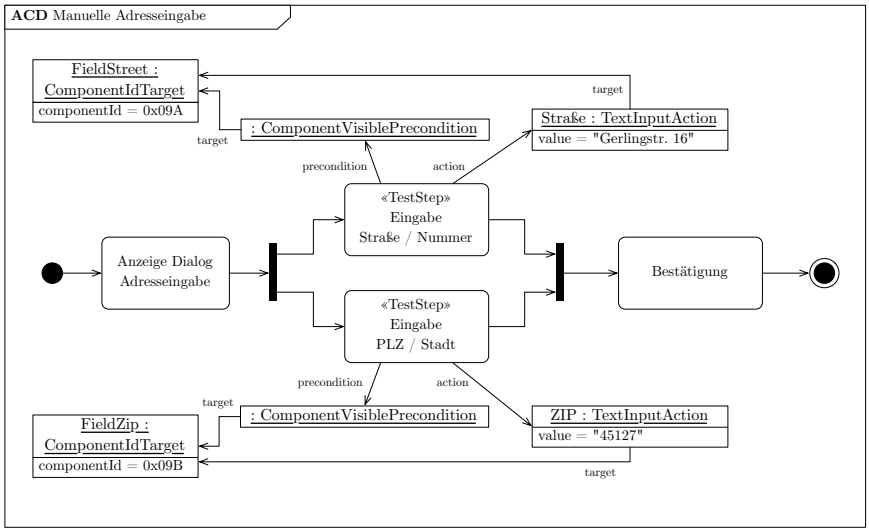


Abbildung 5.24: Der Stereotyp *TestStep* angewendet auf *UML::Action*-Elemente zur Modellierung von Interaktion mit dem UI der App.

Auf das Aktivitätsdiagramm selbst wurde der Stereotyp *TestActivity* nicht angewendet. Hintergrund dieser Modellierung ist, dass das in Abbildung 5.24 dargestellte aktivitätsdia-

gramm nicht der Einstiegspunkt in die App ist. Aus diesem Grund ist der Stereotyp *TestActivity* auf das übergeordnete Aktivitätsdiagramm (ACD) in Abbildung 5.22 angewendet. Dieses spezifiziert das in Abbildung 5.24 dargestellte ACD „Standortbestimmung“ als *UML::CallBehaviorAction*.

5.2.4.1.4 Stereotyp *TestFlow*

Zur Modellierung alternativer Kontroll- bzw. Datenflüsse definiert das Metamodell der UML das Element *UML::DecisionNode*. Es handelt sich hierbei um ein Element mit mindestens zwei abgehenden Kanten vom Typ *UML::ActivityEdge* (konkretisiert durch die zwei Typen *UML::ControlFlow*, *UML::DataFlow*), die den Kontrollfluss/Objektfluss in voneinander verschiedenen Elementen fortführen. Bedingungen, die Auswahlkriterien zwischen diesen Alternativen beschreiben, werden in der UML textuell oder durch Ausdrücke der *Object Constraint Language* (OCL) direkt am Kontrollfluss, durch sogenannte Guard-Conditions, annotiert.

Wird die Bedingung einer Guard-Condition positiv ausgewertet, so folgt der Ablauf diesem Fluss und die Zielaktion des Flusses wird als nächstes ausgeführt. Für mehrere abgehende Kontroll-/Objektflüsse aus einer *UML::DecisionNode* gilt, dass diese mit gegenseitig exklusiven Guard-Conditions annotiert werden sollten, da für das Element *UML::DecisionNode* definiert ist, dass der Kontroll-/Objektfluss nur an einer der ausgehenden Kanten fortgesetzt wird (vgl. OMG [268, S. 430]). Sind mehrere abgehende Kanten qualifiziert, ist nicht definiert, an welcher abgehenden Kante der Kontrollfluss fortgesetzt wird. Diese Situation führt nicht-deterministisches Verhalten in ein Modell ein und sollte deshalb vermieden werden.

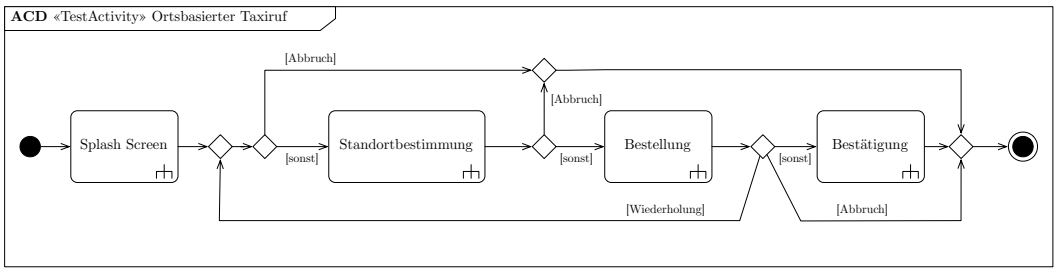


Abbildung 5.25: Verwendung *UML::DecisionNode*

In Abbildung 5.25 ist in einem Aktivitätsdiagramm die Verwendung des Elements *UML::DecisionNode* exemplarisch dargestellt. Das Element selbst wird durch das Symbol ◇ dargestellt, welches ebenfalls zur Repräsentierung des Elements *UML:MergeNode* verwendet wird. Durch gemeinsame Verwendung desselben Symbols wird die Funktion dieser Modellierungselemente ebenfalls vereinigt, d. h. es dürfen mehrere eingehende und mehrere ausgehende Kanten modelliert werden. Diese Zusammenführung der Elemente *UML::DecisionNode* und *UML:MergeNode* wird durch das Metamodell der UML explizit zugelassen (vgl. OMG [268, S. 432]).

Zur Modellierung von Testdaten in angereicherten Aktivitätsdiagrammen ist diese Notation jedoch problematisch, da aufgrund der graphischen Notation der Eindruck entsteht, es wäre Aufgabe des Elements *UML::DecisionNode* über den weiteren Kontrollflussverlauf zu entscheiden. UML-Aktivitätsdiagramme sind jedoch als sogenanntes *Token Game* (engl. Markenspiel) definiert (angelehnt an Petri-Netze), in denen das Systemverhalten durch den Fluss von Marken durch ein Netz modelliert wird. Deshalb ist es so, dass nicht das *UML::Decision-*

Node-Element den weiteren Kontrollflussverlauf steuert, sondern Guard-Conditions an den ausgehenden Kanten von *UML::DecisionNode*-Elementen.

Aus diesem Grund definiert das UML-Profil zur Testfallmodellierung den Stereotyp *TestFlow*, welcher die Metaklasse *UML::ActivityEdge* erweitert. Seine Funktion ist es, Vor- und Nachbedingung sowie Aktionen als Referenzattribute an *UML::ActivityEdge*-Elementen verfügbar zu machen. Insbesondere ist der Stereotyp *TestFlow* dazu intendiert, ausgehende Kanten von *UML::DecisionNode*-Elementen mit testrelevanten Informationen zu ergänzen. Zu diesen gehören auch Kontextinformationen.

Entscheidungsknoten im Kontrollfluss unterscheiden nicht, ob der Auswahl einer Alternative eine willkürliche Entscheidung des Anwenders zugrunde liegt oder ob der Zustand der Anwendung (einschließlich aller Kontextparameter) die Auswahl eines alternativen Kontrollflusses herbeigeführt hat. So kann etwa die Entscheidung darüber, ob für eine kontextsensitive Aktion innerhalb einer Anwendung GPS-Standortinformationen verwendet werden dürfen direkt durch den Anwender getroffen werden (z. B. durch Auswahl in einem entsprechenden Dialog der Benutzungsoberfläche). Ebenfalls ist es möglich, dass die Verwendung von GPS-Standortinformation ausscheidet, weil diese zum Zeitpunkt der Anwendungsausführung nicht ermittelt werden kann bzw. die entsprechende API-Funktion ungültige Werte zurückliefert.

Für die Transformation von UML-Aktivitätsdiagrammen zu Testfällen ist jedoch die Unterscheidung, ob die Auswahl alternativer Kontrollflüsse durch den Anwender oder durch systeminterne Funktionen herbeigeführt wird, maßgeblich. Systeminterne Funktionen können sich dem Einfluss des Anwenders vollständig entziehen, weil sie Parameter als Eingabe verwenden, die durch den Anwender nicht bestimmt werden können (z. B. Verfügbarkeit eines GPS-Signals, Qualität eines Mobilfunksignals usw.). Von einer automatisierten Testfallgenerierung für solche Systeme wird erwartet, dass für solche Entscheidungsknoten im Kontrollflussgraphen jeweils für alle Alternativen gültige Testfälle erzeugt werden. Bei der Testdurchführung können dann für jede Alternative die entsprechenden Vorbedingungen hergestellt werden (etwa durch Simulation von GPS-Koordinaten oder Eigenschaften einer Netzwerkverbindung). Hierbei müssen alle Kontextparameter berücksichtigt werden, die zur Ausführung der Anwendung maßgeblich sind. Insbesondere kommen hier die in Abschnitt 3.1.2 diskutierten Kontextparameter in Frage.

Ebenfalls muss ein UML-Profil zur Integration der Testfallmodellierung in Systemmodellierung in der Lage sein, die Abhängigkeit bestimmter Kontrollflussalternativen von willkürlichen Entscheidungen des Anwenders zu modellieren. Bei der Generierung von Testfällen sind diese insofern zu berücksichtigen, als dass für alle Alternativen entsprechende Testfälle erzeugt werden müssen. Im Unterschied zu den systeminternen Entscheidungsknoten im Kontrollflussgraphen werden hier die Entscheidungskriterien jedoch nicht durch Vorbedingungen aus Kontextparametern, sondern durch alternative Möglichkeiten der Interaktion mit der Benutzungsoberfläche bestimmt (z. B. Texteingabe in unterschiedliche Eingabefelder oder Anklicken unterschiedlicher Schaltflächen).

In Abbildung 5.26 ist erneut ein Ausschnitt des Systemmodells der App Mobiler Taxiruf mit einem Fokus auf die Anwendung des Stereotyps *TestFlow* dargestellt. Abgebildet ist eine Verfeinerung des Modells, bei welcher auf einige Kontroll-/Datenflüsse der Stereotyp *TestFlow* angewendet wurde. Im abgebildeten Modell sind hier *Test::Action*-Subtypen

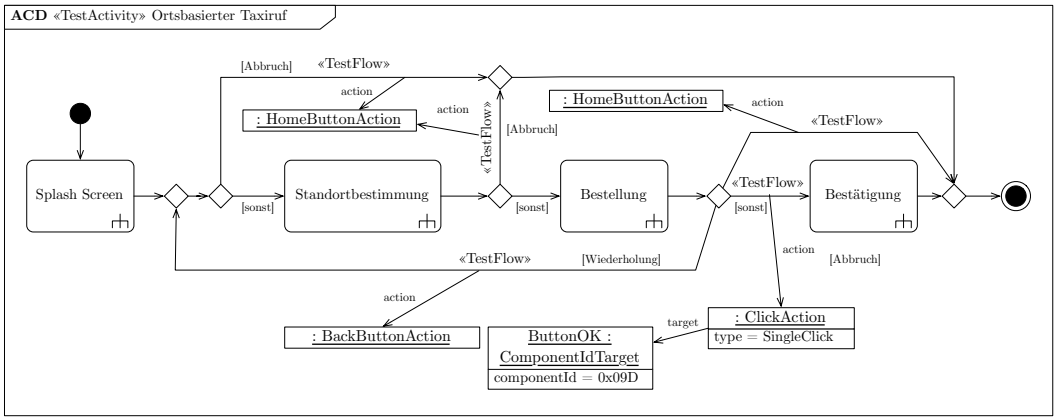


Abbildung 5.26: Der Stereotyp *TestFlow* am Beispiel der App Mobiler Taxiruf. Dargestellt ist ein Ausschnitt des Systemmodells, der eine Interaktion des Anwenders mit Hilfe des Metamodells zur Testfallmodellierung abbildet.

(*Test::BackButtonAction*, *Test::HomeButtonAction* und *Test::ClickAction*) spezifiziert, durch welche die Interaktion des Anwenders mit der App explizit modelliert wird.

5.2.4.2 Verwendung des UML-Profils zur Modellierung von Testfällen

In den vorangegangenen Abschnitten wurde das UML-Profil zur Testfallmodellierung vorgestellt und die einzelnen Stereotypen diskutiert. In diesem Abschnitt wird die Verwendung des UML-Profils im Detail am Beispiel erläutert.

Das Profil stellt Stereotypen bereit, die es ermöglichen, Kontext- und Testdaten direkt im Systemmodell zu modellieren. Somit entfallen Medienbrüche durch eine separate Dokumentation dieser Informationen und stehen dem maschinellen Zugriff durch eine Testfallgenerierungstechnologie zur Verfügung. Während des zweistufigen Generierungsprozess plattformspezifischer Testfälle werden diese Daten zunächst in das plattformunabhängige Testmodell übernommen und manifestieren sich im abschließenden Schritt in einer konkreten Testautomatisierungstechnologie.

In Abbildung 5.27 ist beispielhaft ein rudimentäres Aktivitätsdiagramm dargestellt. Auf das ACD selbst wurde der Stereotyp *TestActivity* angewendet (vgl. Abschnitt 5.2.4.1.2, Abbildung 5.20). Es verfügt deshalb über ein Attribut, welches eine beliebige Anzahl Instanzen von *Test::Precondition* referenziert (vgl. Abschnitt 5.2.3.2.7, Abbildung 5.18). Dieses erlaubt die Spezifikation von Elementen des Typs *Test::UserInterfacePrecondition* und *Test::Precondition*. Hiermit ist es möglich, Vorbedingungen zu spezifizieren, die sich auf die gesamte Aktivität erstrecken. Bei einer späteren Testausführung erzwingt die Automatisierungstechnologie bzw. die Kontextsimulation die Gültigkeit dieser Vorbedingungen für alle *Test::TestCase*-Instanzen, die aus diesem Aktivitätsdiagramm generiert werden.

Instanzen von *Test::UserInterfacePrecondition* sind atomare Testschritte. Sie stellen sicher, dass sich das UI des SUT in einem definierten Zustand befindet. Hierzu gehört beispielsweise die Sicherstellung, dass UI-Elemente sichtbar und aktiv sind oder dass ein bestimmter Text im UI angezeigt wird. Entspricht das SUT nicht den spezifizierten Vorbedingungen, schlägt der Testfall fehl, weil das SUT nicht der Spezifikation entspricht.

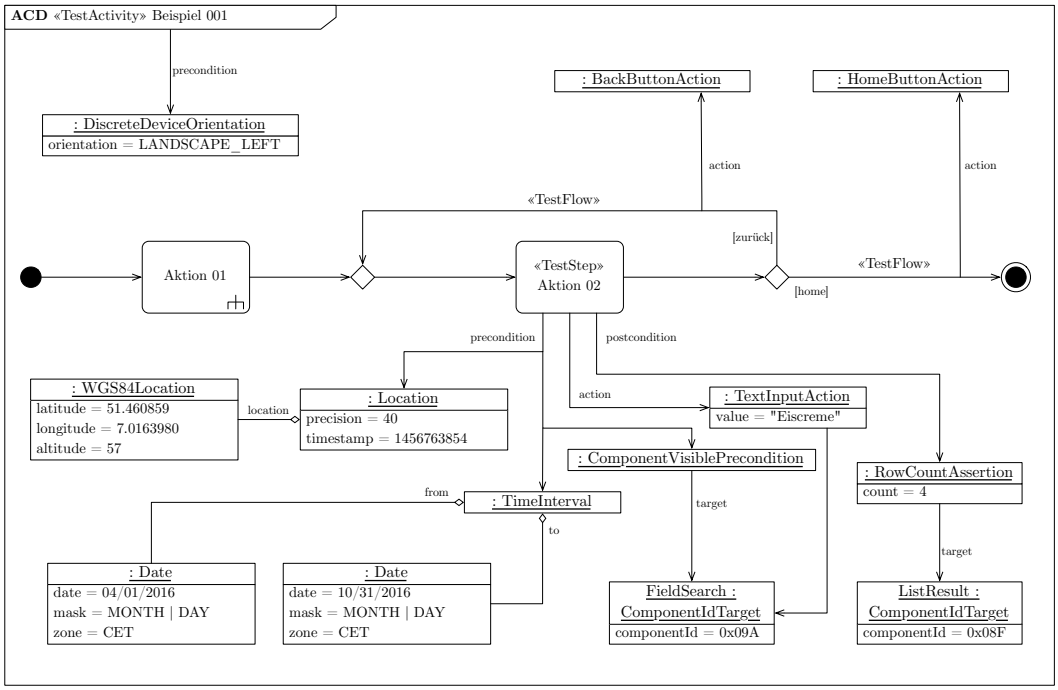


Abbildung 5.27: Beispiel der Anwendung des UML-Profils zur Testmodellierung auf ein Aktivitätsdiagramm

Instanzen von *Test::Precondition* werden von der Kontextsimulation verarbeitet. Sie ist dafür verantwortlich, die in Abschnitt 5.2.2.2 diskutierten Kontextbedingungen herzustellen, beispielsweise durch Konfiguration des Netzwerkadapters oder durch Einspeisung von Testdaten in die Sensoren des SUT.

In Abbildung 5.27 ist die Anwendung des Stereotyp *TestActivity* an einem Beispiel demonstriert, das in einem UML-Objektmodell eine Instanz des Typs *Context::DiscreteDeviceOrientation* spezifiziert. Über das Attribut „precondition“ des Stereotyps *TestActivity* ist diese Instanz mit dem *UML::Activity*-Objekt assoziiert. Konkret modelliert dieses Beispiel, dass die gesamte Aktivität unter der Voraussetzung durchgeführt wird, dass sich das Gerät im Querformat mit Home-Button links befindet. Die Testautomatisierungstechnologie mit Kontextsimulation stellt die Werte der Gerätesensoren entsprechend ein (vgl. Abschnitt 5.5.1).

Das ACD in Abbildung 5.27 spezifiziert zwei Aktionen. Auf eine dieser Aktionen ist der Stereotyp *TestStep* angewendet. Dieser stellt Referenzattribute auf Instanzen von *Test::Precondition*, *Test::Action* und *Test::Postcondition* im *UML::Action*-Element bereit. Im konkreten Beispiel sind als Vorbedingung für einen Test dieser Aktivität die Kontextparameter Zeit (vgl. Abschnitt 3.1.2.1.2) und Standort (vgl. Abschnitt 3.1.2.1.3) spezifiziert.

Bei der Testausführung stellt die Kontextsimulation die Systemzeit des Emulators oder des Geräts auf den im Modell spezifizierten Wert ein und legt den spezifizierten Standort im Gerät fest, so dass API-Aufrufe zur Standortbestimmung innerhalb des SUT genau diesen Wert zurückliefern. Der Standort ist im Beispiel statisch spezifiziert, als Zeit ist hingegen ein Intervall angegeben. Der Testfallgenerator erkennt hier die *Context::TimeInterval*-Instanz und kann, je nach Parametriesierung, Randbereichstests (d. h. Minimum und Maximum der Intervallgrenzen) erzeugen oder eine Sequenz von Tests, die unter Berücksichtigung der Maskierung jedes Datum innerhalb des Zeitintervalls testen. Weiterhin ist eine

Instanz von *Test::TextInputAction* spezifiziert, durch welche im konkreten Beispiel die Eingabe eines Suchbegriffs in ein Textfeld modelliert ist. Als Nachbedingung ist eine Instanz von *Test::RowCountAssertion* modelliert, welche im Beispiel die Anzahl der Einträge in einem Listen-Element des UI überprüft.

Das in Abbildung 5.27 dargestellte Aktivitätsdiagramm ist grob an die Funktionsbeschreibung der App AR Tourist Information angelehnt und demonstriert die Verwendung des UML-Profils am Beispiel der Stereotypen *TestActivity* und *TestStep*.

In Abbildung 5.28 ist ein Ausschnitt des Modells der App Mobiler Taxiruf mit einem Fokus auf die Verwendung des Stereotyp *TestFlow* dargestellt (vgl. Funktionsbeschreibung in Kapitel 3), der eine Reihe von *UML::ActivityEdge*-Elemente enthält, auf welche der Stereotyp *TestFlow* angewendet wurde. Modelliert ist das Verhalten der App beim Bestimmen des Standorts des Anwenders.

Bei Eintritt in die Aktivität wird im ersten Schritt versucht, den letzten bekannten Standort des Anwenders festzustellen. Gelingt dies nicht, wird ein Standardwert angenommen (geographischer Mittelpunkt der Bundesrepublik Deutschland). Der Standardwert wird dem *UML::DataStoreNode*-Element mit dem Titel „Standardwerte“ entnommen. Mit dieser Information wird die Kartenansicht der Anwendung initialisiert. Anschließend wird dem Anwender

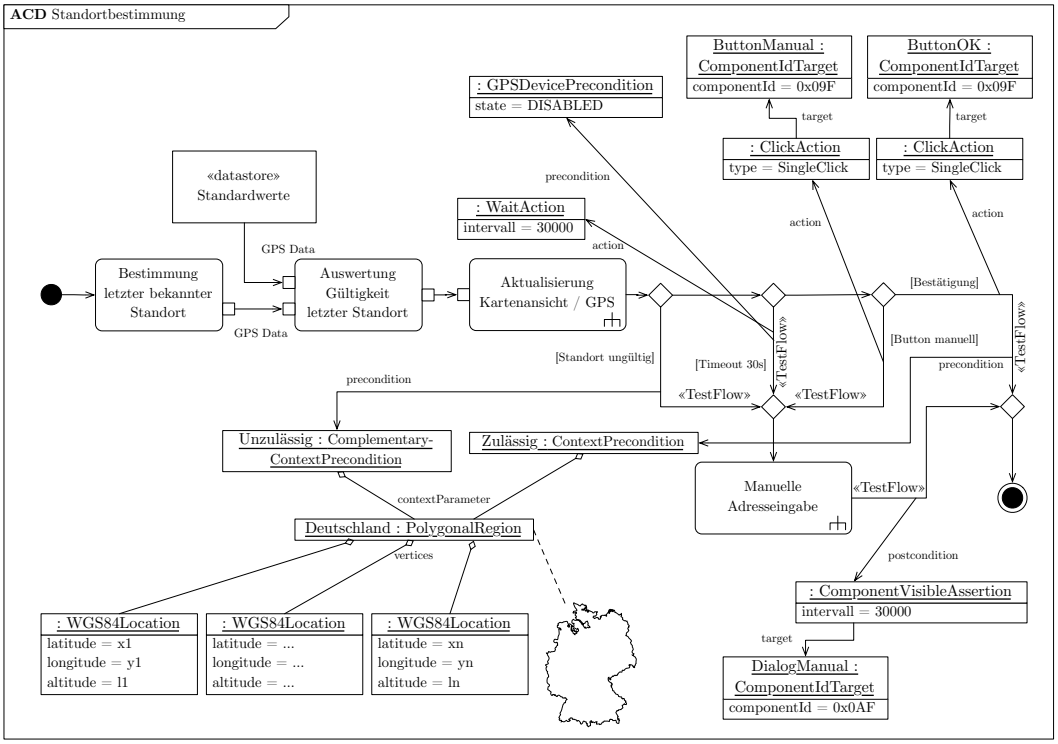


Abbildung 5.28: Beispiel der Anwendung des UML-Profils zur Testmodellierung auf das Verhaltensmodell der App Mobiler Taxiruf. Die Modellierung ist so ausgeführt, dass die Testautomatisierungslösung Testdaten für Standordinformationen generiert.

eine Kartenansicht präsentiert. Hier hat der Anwender die Option, die gegenwärtig gültige Standortinformation zu bestätigen und den Taxibestellprozess fortzusetzen, weitere Versuche den Standort via GPS zu ermitteln zu unterbinden und mit der manuellen Eingabe fortzufahren. Weiterhin überwacht die App die Ermittlung des Standorts via GPS. Gelingt dies

nicht innerhalb eines Zeitraums von 30 Sekunden, wird eine manuelle Standorteingabe durch Aufruf des entsprechenden Dialogs erzwungen.

Die erste Variante, Bestätigung eines gültigen Standorts, wird durch ein *Test::Precondition*-Element im Zusammenwirken mit einem *Test::ClickAction*-Element modelliert. Das *Test::Precondition*-Element referenziert ein Element vom Typ *Context::PolygonalRegion*. In dieser Variante der Modellierung erzeugt die Testausführungsumgebung eine Anzahl gültiger WGS84-Koordinaten, die während der Ausführung des Tests in das SUT eingespeist werden.

Stellt die App fest, dass sich der Anwender außerhalb der zulässigen geographischen Region befindet, wird sofort zur manuellen Eingabe gewechselt. Diese Situation ist ebenfalls mit *TestFlow* modelliert. Das entsprechende *UML::ActivityEdge*-Element referenziert hierzu eine Instanz von *Test::ComplementaryContextPrecondition*, die ihrerseits wiederum die *Context::PolygonalRegion*-Instanz referenziert. Die Automatisierungsumgebung verwendet diese Information zur Generierung ungültiger Standortinformationen.

Ein dritter Fall bildet die Situation ab, dass innerhalb des vorgegebenen Zeitintervalls kein gültiger Standort festgestellt werden kann. Dieser Fall ist modelliert, indem an den entsprechenden Kontrollfluss eine *Test::GPSDevicePrecondition*-Instanz modelliert wurde und ein *Test::WaitAction*-Element spezifiziert wurde. Die Automatisierungstechnologie setzt dieses Modell um, indem sie das GPS-Modul im SUT so blockiert, dass keine Standortaktualisierungen erzeugt werden. Dann wartet die Automatisierungstechnologie das spezifizierte Zeitintervall ab und überprüft die Nachbedingung. Diese ist ebenfalls mit Hilfe von *TestFlow* an einem Kontrollfluss modelliert. Die manuelle Fortsetzung kommt ohne *Test::Precondition*-Element aus und modelliert lediglich ein *Test::ClickAction*-Element und eine Nachbedingung.

5.2.5 Zusammenfassung

Kontextparameter treten bei der in dieser Dissertation untersuchten Technologie zur Testautomatisierung in der Funktion von Testdaten auf. Es wird deshalb ein Metamodell zur Abbildung von Kontext in menschen- und maschinenlesbaren Modellen benötigt. In Abschnitt 5.2.2.2 wird ein Metamodell zur Testfallmodellierung diskutiert. Metamodelle zur Modellierung von Kontext in Softwaresystemen wurden bereits vielfältig wissenschaftlich untersucht. Existierende Ansätze sind jedoch auf die Verwendung von Kontextparametern als Eingabevektoren fokussiert und sind primär dazu intendiert, situativen Kontext höherer Abstraktionsebenen aus individuellen Kontextparametern zu schlussfolgern. Diese Dissertation hat hingegen eine Technologie zum Gegenstand, die kontextsensitive Anwendungen automatisiert zu testen vermag. An das in Abschnitt 5.2.2.2 vorgestellte Metamodell stellen sich aus diesem Grund andere Anforderungen, da maschinelles Inferieren komplexen Kontexts für die hier verfolgten Ziele nicht erforderlich ist. Das Metamodell zu Kontextmodellierung ermöglicht es, individuelle Kontextparameter zu modellieren und zu Kontext höherer Abstraktionsebenen zu aggregieren.

In Abschnitt 5.2.3.2 wird ein Metamodell entworfen, mit welchem Artefakte des Testens in Modellen abgebildet werden können. Diese Testfallmodelle sind zunächst plattform- und technologieunspezifisch hinsichtlich der Testautomatisierungstechnologie. Gegenüber einer direkten Manifestierung von Tests in Code hat die Verwendung von Modellen einige Vorteile. Einerseits sind Modelle durch Abstraktion technischer Details i. d. R. einfacher verständlich als konkrete Realisierungen im Quellcode einer spezifischen Testautomatisierungstechnologie.

Andererseits ergibt sich der Vorteil einer späten Bindung an eine spezifische Testautomatisierungstechnologie, deren Eingabeartefakte aus dem Modell generiert werden können, so dass Flexibilität gegenüber der tatsächlich verwendeten Testtechnologie erhalten bleibt.

Um zu gewährleisten, dass die Modellierung von Kontext und Testfällen keine zusätzlichen Anforderung an das Erlernen einer weiteren Modellierungssprache stellt, wurde in Abschnitt 5.2.4 ein UML-Profil zur Testfallmodellierung diskutiert.

5.3 Generierung plattformunabhängiger Testfälle für kontextsensitive mobile Anwendungen aus Systemmodellen

In den vorangegangenen Abschnitten wurde die Methode zur modellbasierten Testautomatisierung erläutert (vgl. Abschnitt 5.1) und es wurden Metamodelle zur Modellierung kontextsensitiver Testfälle definiert (vgl. Abschnitt 5.2).

In diesem Abschnitt wird die Transformation von UML-Aktivitätsdiagrammen zu Testfallmodellen erläutert. In Abschnitt 5.3.1 wird zunächst das für diese Arbeit zentrale Werkzeug Modelltransformation erläutert. Um eine adäquate Abdeckung einer mobilen, kontextsensitiven Anwendung mit Tests zu gewährleisten, wird in Abschnitt 5.3.2 ein Überdeckungskriterium für Aktivitätsdiagramme definiert. Der Prozess der Transformation wird in Abschnitt 5.3.3 im Detail diskutiert. Insbesondere wird dargelegt, wie Modelltransformation verwendet wird, um Aktivitätsdiagramme in einem ersten Schritt zu Bedingungs-/Ereignisnetzen zu transformieren, aus welchen in einem zweiten Schritt Ausführungspfade durch Aktivitätsdiagramme berechnet werden. Basierend auf diesen Pfaden werden im Anschluss Modellinstanzen des in Abschnitt 5.2.3.2 vorgestellten Metamodells zur Testfallmodellierung erzeugt.

5.3.1 Modelltransformation

Das in dieser Dissertation vorgestellte Konzept der modellbasierten Testautomatisierung mobiler, kontextsensitiver Anwendungen basiert auf der Transformation von Systemmodellen zu Modellinstanzen des in Abschnitt 5.2.3 entworfenen Metamodells zur Testfallmodellierung. Die Anreicherung von Verhaltensmodellen mit Kontext- und Testdaten erfolgt mit dem in Abschnitt 5.2.4 vorgestellten UML-Profil.

Zentrales Werkzeug zur Überführung von Modellen zwischen unterschiedlichen Abstraktionsebenen ist die Modelltransformation, die unter Anwendung bestimmter Transformationsregeln ein Quellmodell auf ein Zielmodell abbildet. Modelltransformation ist ein Prozess, der Elemente des Quellmodells entfernt, anpasst oder im Zielmodell Elemente erzeugt, die aus Elementen mehrerer Quellmodelle aggregiert werden. Im Kontext der MDA und des MDSD wird Modelltransformation insbesondere dazu verwendet, grobe plattformunabhängige Systemmodelle schrittweise in verfeinerte plattformspezifische Modelle umzuwandeln, wobei der Zuwachs des Detailgrads aus Modellen der Zielplattform gewonnen wird (vgl. Sendall und Kozaczynski [318], Ayed et al. [21], Bukhari und Waheed [63]). Kerneigenschaft der Modelltransformation ist es, dass bei der Transformation diejenigen Aspekte des Systems erhalten bleiben, die im Zielmodell besonders relevant sind, während irrelevante Aspekte entfernt werden. Strukturelle Aspekte des Quellmodells bleiben erhalten.

Es werden zwei Arten der Modelltransformation unterschieden: Modell-zu-Modell Transformation (M2M) und Modell-zu-Text Transformation (M2T). Erstere generiert aus einem Quellmodell ein Zielmodell, zweitere erzeugt eine textuelle Repräsentierung eines Eingabemodells. Hierbei kann es sich z. B. um den Quellcode einer Programmiersprache handeln, der im Kontext von MDA-Aktivitäten unter Verwendung automatisierter Modelltransformation aus einem Systemmodell generiert wird. M2T-Transformationen werden als Modelltransformation bezeichnet, weil auch Quellcode ein Modell repräsentiert, welches durch einen Compiler zu Maschinencode verarbeitet wird. In der menschlichen Wahrnehmung repräsentiert Quellcode zwar eine techniknahe Abbildung eines Softwaresystems. Aus der Perspektive der Zielplattform ist Quellcode jedoch als Modell zu betrachten, das durch spezifische Eigenschaften der Zielplattform, etwa der Prozessorarchitektur, weiter konkretisiert wird und sich auf unterschiedlichen Plattformen in unterschiedlichem Byte-Code manifestiert.

Nach Beydeda et al. [43] wird zwischen horizontaler und vertikaler Modelltransformation unterschieden. Modelltransformation wird als horizontal bezeichnet, wenn die inhaltliche Weiterentwicklung des Modells im Fokus steht und Quell- und Zielmodell ein System auf derselben Abstraktionsebene beschreiben. Modelltransformation wird als vertikal bezeichnet, wenn die Transformation zwischen unterschiedlichen Abstraktionsebenen im Fokus steht. Gemäß dieser Klassifikation handelt es sich bei der Transformation von UML-Aktivitätsdiagrammen zu Testfallmodellen um eine horizontale Transformation, da gegenüber dem manuellen Anreichern des Modells mit Kontext- und Testdaten in Phase zwei (vgl. Abschnitt 5.1) in der dritten Phase keine weitere Verfeinerung des Modellinhalts erfolgt. Vielmehr liegt der Fokus auf Aspekten des Modells, die für das Testen des Systems relevant sind, während andere Aspekte aus dem Modell entfernt werden.

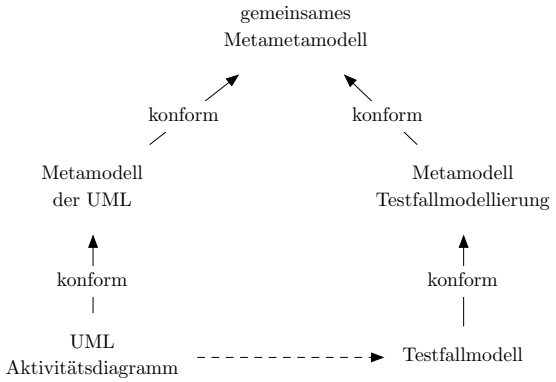


Abbildung 5.29: Modelltransformation überführt Modellinstanzen konform zu einem Metamodell in Modellinstanzen konform zu einem anderen Metamodell. Beide Metamodelle sind konform zu einem gemeinsamen Metametamodell.

Eine M2M-Transformation erzeugt aus einem oder mehreren Eingabemodellen ein oder mehrere Ausgabemodelle. Hierzu kommen Transformationsregeln zur Anwendung, welche beschreiben, wie Elemente des Eingabemodells auf Elemente des Ausgabemodells abzubilden sind. Das Konzept der Modelltransformation von UML-Aktivitätsdiagrammen zu Testfallmodellen ist in Abbildung 5.29 dargestellt.

Voraussetzung zur Durchführung einer Modelltransformation ist, dass Quell- und Zielmodell jeweils konform zu einem gemeinsamen Metametamodell sind. Hierbei handelt es sich um eine technische Anforderung, die durch eine geeignete Technologieauswahl zur Implementierung der beteiligten Metamodelle gewährleistet werden kann.

Im hier untersuchten Ansatz zur Testfallgenerierung kommen beide Varianten Modelltransformation zum Einsatz. Zur Erzeugung des plattformunabhängigen Testfallmodells (vgl. Abschnitt 5.1) kommt in der dritten Phase eine horizontale M2M-Transformation zum Einsatz, die wesentliche strukturelle Aspekte des modellierten Systems erhält, aber von Details abstrahiert, die für die Testdurchführung als nicht relevant ausgezeichnet wurden. In der vierten Phase wird das plattformunabhängige Testfallmodell durch eine vertikale M2T-Transformation in Quellcode der Testautomatisierungstechnologie Calabash überführt. Das Modell wird hierbei im Calabash-Syntax manifestiert, wodurch implizit eine Verfeinerung des Modells um die Semantik der Quellcodeartefakte der Testautomatisierungstechnologie erfolgt.

5.3.2 Testabdeckungskriterium

Das Testen von Softwaresystemen erfordert die Ausführung des SUT unter definierten Bedingungen und die Interpretation des Verhaltens des SUT, um Rückschlüsse auf die Spezifikationskonformität des Systems zu ziehen. Softwaresysteme implementieren i. d. R. mehr als nur eine Funktionalität. Um ein Softwaresystem vollständig zu testen, gilt es, alle Funktionalität beim Testen zu adressieren.

Abdeckungskriterien (auch Überdeckungskriterien) für Softwaretests bestimmen, welche Teile (Funktionen) einer Software durch Tests adressiert werden. In der wissenschaftlichen Literatur wurden Abdeckungskriterien bereits definiert und untersucht. Abdeckungskriterien beziehen sich i. d. R. auf den Kontrollflussgraphen einer Software, dessen Knoten einzelne Anweisungen des Quellcode[s] repräsentieren.

Anweisungsüberdeckungstests (sogenannte C_0-Tests) beispielsweise erfüllen ein Abdeckungskriterium, welches fordert, dass jede im Code des Softwaresystems enthaltene Anweisung durch mindestens einen Testfall adressiert wird, d. h. im Rahmen der Testdurchführung ausgeführt wird (vgl. Patton [275], Myers et al. [261], Liggesmeyer [236]).

Das Kriterium Verzweigungsüberdeckung (auch Zweigüberdeckung) hingegen fordert individuelle Testfälle für jede mögliche alternative Fortsetzung des Kontrollflusses an Verzweigungspunkten. Verzweigungsüberdeckungstests subsumieren Anweisungsüberdeckungstests, weil durch dieses Kriterium zwingend alle Anweisungen des Codes erreicht werden. Zweigüberdeckungstests werden in der Literatur als C_1-Tests bezeichnet (vgl. [275, 261, 236]). Sie gelten als minimales Testkriterium beim kontrollflussorientierten Testen.

Pfadüberdeckungstests (C_2-Tests) fordern, dass alle möglichen Pfade durch den Kontrollflussgraphen durch Tests abgedeckt werden. Diese Kategorie wird weiter in C_{2_a}-, C_{2_b}- und C_{2_c}-Tests unterschieden. Vollständige Pfadüberdeckung, bei der alle möglichen Pfade getestet werden, wird durch C_{2_a}-Tests gefordert. In der Praxis ist C_{2_a}-Abdeckung nicht realisierbar, weil Kontrollflussgraphen zyklische Strukturen enthalten können, die zu unendlich vielen Pfaden führen. Um das zu vermeiden, definiert die C_{2_b}-Abdeckung, dass Zyklen im Kontrollflussgraphen höchstens zweimal durchlaufen werden. Die C_{2_c}-Abdeckung erweitert die C_{2_b}-Abdeckung auf eine vorgegebene Anzahl von Zyklendurchläufen.

Abdeckungskriterien können auf unterschiedlichen Abstraktionsebenen definiert werden. Auf einer techniknahen Abstraktionsebene haben sie i. d. R. Quellcode zum Gegenstand. Knoten im Kontrollflussgraphen referenzieren individuelle Anweisungen im Quellcode. Auf der Ebene von Akzeptanztests, i. d. R. Black-Box-Tests, haben Tests jedoch die Prüfung einer Software gegenüber ihrer Spezifikation zum Gegenstand, ohne dass hierzu zwingend Kenntnisse des Codes der Software vorhanden sein müssen. Das heißt, aus der Perspektive des Anwenders wird geprüft, ob sich das SUT bei definierten Stimuli spezifikationskonform verhält. Durch diese Abstraktion von der Quellcode[ebene] können Abdeckungskriterien für Akzeptanztests ebenfalls nur auf dieser Abstraktionsebene gelten.

Eine direkte Abbildung existierender Testabdeckungskriterien auf Akzeptanztests ist jedoch nicht möglich. Akzeptanztests stellen zwar sicher, dass jede für den Anwender erreichbare Funktion des SUT durch einen Test adressiert wird. Hierdurch ist jedoch nicht gewährleistet, dass jede einzelne Anweisung im Quellcode ebenfalls durch diese Menge von Tests geprüft wird. Es ist aus der spezifikationsorientierten Perspektive von Akzeptanztests unbekannt, ob durch die Gesamtheit aller spezifizierten Anforderungen alle Anweisungen im Quellcode des SUT abgedeckt werden. Eine Abweichung könnte beispielsweise existieren, wenn das SUT Code enthält, der zur Erbringung der Funktionalität nicht erforderlich ist. Moderne Programmierparadigmen machen Gebrauch von Programmierbibliotheken, deren Funktionsumfang die tatsächlich benötigte Funktionalität i. d. R. übersteigt. Das gilt bereits für solche Bibliotheken, die vom SDK mobiler Plattformen bereitgestellt werden, so dass Kompilate fast immer Code enthalten, der zur Leistungserbringung nicht benötigt wird.

Folglich kann Anweisungsüberdeckung (C_0-Überdeckung, analog C_1- sowie C_2-Überdeckung) nicht sinnvoll in der ursprünglichen Bedeutung, fokussiert auf Anweisungen im Quellcode, für Akzeptanztests gefordert werden. Im Kontext dieser Dissertation wird deshalb das Aktivitätsdiagramm der zu testenden Anwendung als Kontrollflussgraph aufgefasst (anstelle des Quellcode[s] des SUT). Die in der Literatur definierten C_0-, C_1- sowie C_2-Überdeckungskriterien können auf Akzeptanztests übertragen werden, wobei sie jedoch nicht Anweisungen im Quellcode als Definitionsbasis der Testabdeckung verwenden, sondern Aktionen im Aktivitätsdiagramm. Um Überdeckungskriterien auf dem Abstraktionsniveau von UML-Aktivitätsdiagrammen zu definieren, werden UML-Aktivitätsdiagramme im Folgenden formal definiert:

Definition 1 Ein UML-Aktivitätsdiagramm ist ein Tupel $ACD = (N, A, C, N_I, N_F, R)$, mit:

$N = A \cup C$ ist die Menge aller Knotenelemente des Typs *UML::ActivityNode*;

A ist die Menge aller Knotenelemente des Typs *UML::Action*;

C ist die Menge aller Knotenelemente des Typs *UML::ControlNode*;

$N_I \subset C$ ist die Menge aller Anfangsknoten, explizit modelliert durch *UML::InitialNode* oder implizit modelliert als Knoten, die keine eingehenden Flüsse haben;

$N_F \subset C$ ist die Menge aller Endknoten, explizit modelliert als *UML::ActivityFinalNode* oder implizit modelliert als Knoten, die keine abgehenden Flüsse haben;

$R \subseteq (N_I \times N \setminus N_I) \cup (N \setminus (N_I \cup N_F) \times N \setminus (N_I \cup N_F) \times (N \setminus N_F \times N_F)$ ist die Menge aller gerichteten Flüsse vom Typ *UML::ActivityEdge*, die Knoten miteinander verbinden.

Aus einem Aktivitätsdiagramm basierend auf dem Metamodell der UML geht nicht unmittelbar hervor, welche Elemente eine Interaktion mit dem Anwender repräsentieren, eine sonstige Manipulation zulassen oder kontextsensitiv sind. In Abschnitt 5.2.4 wurde deshalb ein UML-Profil definiert, welches Stereotypen bereitstellt, die es erlauben, Elemente eines Aktivitätsdiagramms als testrelevant zu kennzeichnen. Im Kontext dieser Dissertation werden Überdeckungskriterien auf eine Abdeckung derjenigen Aktionen ausgelegt, die im Rahmen der bestimmungsgemäßen Verwendung des SUT tatsächlich für den Anwender erreichbar sind und durch Verwendung des UML-Profils zur Testfallmodellierung (Abschnitt 5.2.4) entsprechend annotiert wurden.

Definition 2 Sei ACD ein Aktivitätsdiagramm. Ein Element $e \in N \cup R$ ist *testrelevant*, wenn ein durch das UML-Profil zur Testfallmodellierung definierter Stereotyp auf das Element e angewendet wurde.

Definition 3 Ein Aktionsabdeckungstest ($C_{0_{ACD}}$-Test) für ein Aktivitätsdiagramm ACD ist definiert als Test, bei dessen Ausführung alle testrelevanten Elemente mindestens einmal ausgeführt werden. $C_{0_{ACD}}$-Tests basieren demnach auf der klassischen Definition von C_0-Tests, wobei das Aktivitätsdiagramm des SUT als Kontrollflussgraph zugrundegelegt wird.

Definition 4 Ein Verzweigungsabdeckungkriterium $C_{1_{ACD}}$ für ein Aktivitätsdiagramm ACD ist definiert als Abdeckungskriterium, das fordert, dass alle testrelevanten Verzweigungsalternativen (d. h. ausgehende Kanten an *UML::DecisionNode*-Elementen) im Aktivitätsdiagramm jeweils durch mindestens einen Testfall adressiert werden.

Definition 5 Pfadüberdeckung $C_{2_{ACD}}$-Kriterium für ein Aktivitätsdiagramm ist analog zum klassischen C_2-Test definiert als Testabdeckungskriterium, welches fordert, dass jeder Pfad durch das Aktivitätsdiagramm durch einen Testfall abgedeckt wird.

Analog zum klassischen C_2-Überdeckungskriterium ergibt sich in der Praxis jedoch das Problem, dass selbst für Modelle mit verhältnismäßig geringer Komplexität die Anzahl möglicher Pfade schnell auf ein Maß ansteigt, das selbst bei Verwendung von Computertechnologie nicht beherrschbar ist. Dieses Problem wurde in der wissenschaftlichen Literatur als *State-Explosion-Problem* identifiziert (vgl. Valmari [348]). Im hier untersuchten Ansatz wird dieses Problem vermieden, indem nicht das $C_{2_{ACD}}$-Kriterium vorausgesetzt wird, sondern das im Folgenden diskutierte $C_{2c_{ACD}}$-Kriterium.

Definition 6 Ein Basispfadüberdeckungstest $C_{2c_{ACD}}$ für ein Aktivitätsdiagramm ist definiert als ein $C_{1_{ACD}}$-Test, der analog zum klassischen C_{2_c}-Test jeden Zyklus im Aktivitätsdiagramm bei der Testausführung mindestens eine vorgegebene Anzahl von Durchläufen ausführt, wenn er testrelevante Modellierungselemente enthält, wobei von Permutationen im Pfad enthaltener nebenläufiger Kontrollflussstrukturen abstrahiert wird.

Das Metamodell der UML stellt für Aktivitätsdiagramme Modellierungselemente bereit, durch welche *UML::ActivityNode*-Elemente in einer Graphstruktur modelliert werden können. Hierbei sind sowohl zyklische als auch nebenläufige Strukturen zulässig. Die klassischen Definitionen der Testabdeckungskriterien basieren auf einem Kontrollflussgraphen, der keine nebenläufigen Strukturen enthält.

Hier wird als Kontrollflussgraph jedoch das Aktivitätsdiagramm des SUT zugrundegelegt, in welchem nebenläufige Strukturen zulässig sind. Das Metamodell zur Testfallmodellierung (vgl. Abschnitt 5.2.3.2) sieht hingegen bewusst keine Verzweigungselemente vor, durch welche zyklische oder nebenläufige Strukturen in Testfallmodellen modelliert werden können. Hierdurch soll sichergestellt werden, dass Testfallmodelle unmittelbar in ausführbare Artefakte überführt werden können, so dass die Gewährleistung einer definierten Testabdeckung durch eine geeignete Traversierung des Kontrollflussgraphen nicht zur Aufgabe der Testautomatisierungstechnologie wird. Es wird deshalb eine Erweiterung der klassischen Abdeckungskriterien benötigt, die Nebenläufigkeit berücksichtig.

In Abbildung 5.30 ist ein rudimentäres, aber nicht triviales Aktivitätsdiagramm abgebildet. Es enthält einen Zyklus und zwei nebenläufige Kontrollflussstrukturen. Auf einige Elemente wurden Stereotypen des Profils zur Testfallmodellierung angewendet (das Objektmodell der Kontext- und Testdaten ist aus Gründen der Übersichtlichkeit nicht dargestellt). Sofern Elemente auf nebenläufigen Kontrollflussstrukturen testrelevant sind, sollten unter der Prämisse, dass ein Anwender zu jedem Zeitpunk genau eine Interaktion mit dem SUT durchführen kann, bei der Testdurchführung Sequenzen zulässiger Permutationen dieser Elemente durch Tests abgebildet werden. In Abbildung 5.31 ist eine solche Situation am Beispiel des Aktivitätsdiagramms aus Abbildung 5.30 dargestellt.

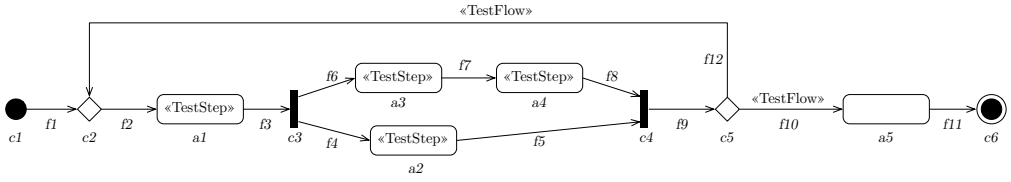


Abbildung 5.30: Beispiel eines UML-Aktivitätsdiagramms

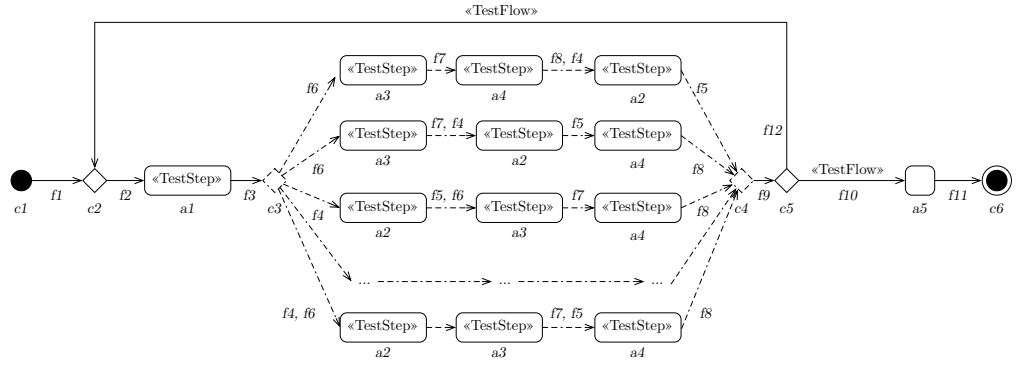


Abbildung 5.31: Beispiel eines UML-Aktivitätsdiagramms mit Permutation nebenläufiger Elemente aus dem Aktivitätsdiagramm in Abbildung 5.30. Aus den nebenläufigen Kontrollflussstrukturen zwischen den Knoten $c3$ und $c5$ sind alternative Kontrollflussstrukturen geworden, die Permutationen nebenläufiger Elemente abbilden.

Definition 7 Eine Sequenzialisierung von Elementen nebenläufiger Kontrollflussstrukturen durch Permutation ist genau dann *zulässig*, wenn Elemente, die auf derselben Kontrollflussstruktur modelliert sind (also nicht nebenläufig), nach der Permutation mit Elementen nebenläufiger Kontrollflussstrukturen in ihrer Ausführungsreihenfolge nicht vertauscht sind.

Definition 8 Ein Permutationssequenzüberdeckungstest $C_{PS_{ACD}}$ für ein Aktivitätsdiagramm ist definiert als $C_{2c_{ACD}}$-Test (Basispfadüberdeckungstest), der jeden Zyklus mindestens einmal durchläuft und zusätzlich zulässige Sequenzialisierungen von Permutationen von testrelevanten Elementen auf nebenläufigen Kontrollflussstrukturen testet. Zur Vermeidung des State-Explosion-Problems ist das $C_{PS_{ACD}}$-Kriterium erfüllt, wenn mindestens eine zulässige Sequenzialisierung durch einen Test abgedeckt wird.

5.3.3 Transformation von UML-Modellen zu Testfallmodellen

Testfälle sind sequenzielle Abfolgen von Testschritten, die entweder manuell durch einen menschlichen Akteur oder automatisch durch ein Werkzeug ausgeführt werden (vgl. Abschnitt 4.1.1.7 und 4.1.1.8). Hierbei wird für jede Sequenz von Testschritten ein definierter Anfang, ein definiertes Ende und eine definierte Reihenfolge dazwischenliegender Testschritte vorausgesetzt. Bei der Generierung von Tests aus UML-Modellen, insb. Aktivitätsdiagrammen, wird die Reihenfolge der Aktionsausführung durch die Struktur des Aktivitätsdiagramms definiert, welches die zu testende Anwendung modelliert.

Die Generierung von Testfällen aus UML-Modellen wurde bereits vielfach wissenschaftlich untersucht. Offutt und Abdurazik [264] befassten sich bereits im Jahr 1999 mit der Generierung von Tests aus Zustandsdiagrammen. Cavarra et al. [74] untersuchten 2002 ebenfalls eine Methode zur Erzeugung von Test aus UML-Modellen mit einem besonderen Fokus auf Objektmodelle zur Abbildung von Testdaten. Einen konkret auf UML-Aktivitätsdiagramme fokussierten Ansatz zur Generierung von Tests untersuchten Linzhang et al. [239], Mingsong et al. [251] sowie Zhao et al. [378]. Im Jahr 2010 untersuchten Heinecke et al. [174] einen Ansatz zur Generierung von Tests aus Aktivitätsdiagrammen, bei dem bereits die Kennzeichnung testrelevanter Modellelemente die Basis für die Berechnung von Ausführungssequenzen war. Die Verwendung von Petri-Netzen als Zwischenschritt der Erzeugung von Tests wurde im Jahr 2010 von Jung und Joo [207] diskutiert, wobei allerdings nicht B/E-Netze verwendet wurden, sondern gefärbte Petri-Netze. Ein anderer Ansatz basierend auf Petri-Netzen wurde von Seiger und Schlegel [317] untersucht.

Alle diese Ansätze basieren auf der Verwendung von Modelltransformation (vgl. Abschnitt 5.3.1) zur Überführung von UML-Modellen in Testfälle, wobei i. d. R. die Abbildung nicht direkt erfolgt, sondern die Transformation in mehreren Schritten, beispielsweise mit Petri-Netzen als Zwischenergebnis, erfolgt. Auch der hier untersuchte Ansatz zur Generierung von Tests aus Aktivitätsdiagrammen basiert auf der Transformation von UML-Modellen in mehreren Schritten. Konkret werden zunächst plattformunabhängige Testfallmodelle aus Aktivitätsdiagrammen erzeugt, die erst in einem späteren Schritt (vgl. Abschnitt 5.4) zu technologiespezifischen Testfällen transformiert werden. Als Zwischenschritt der Generierung plattformunabhängiger Testfallmodelle werden B/E-Netze verwendet, um Ausführungspfade durch Aktivitätsdiagramme zu berechnen, die Zyklen und insb. nebenläufige Kontrollflüs-

se berücksichtigen. Der Prozess der Generierung plattformunabhängiger Testfallmodelle ist schematisch in Abbildung 5.32 dargestellt.

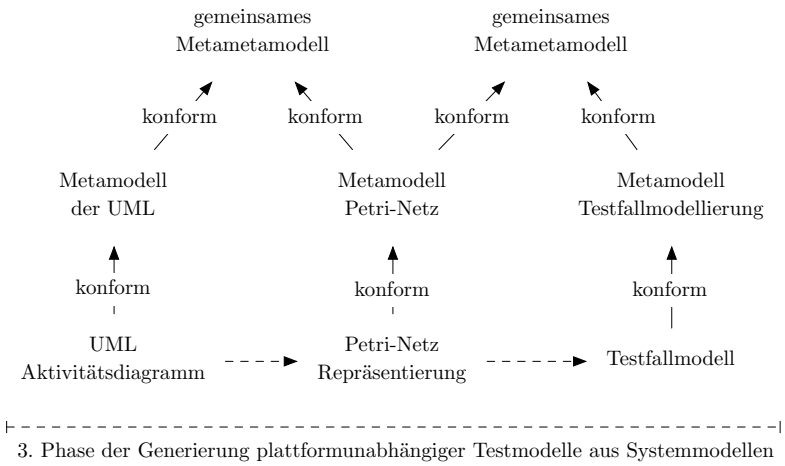


Abbildung 5.32: Modelltransformation von UML-Aktivitätsdiagramm zu einem Petri-Netz mit anschließender Transformation zu Testfallmodell

Um sicherzustellen, dass die aus UML-Aktivitätsdiagrammen generierten Testfälle selbst konform zur Anwendungsspezifikation sind, stellen sich einige Anforderungen an die Transformation:

REQ-Transformation UML-zu-Testfallmodell 1: Erhalt struktureller Eigenschaften

> Bei der Transformation müssen strukturelle Eigenschaften des UML-Aktivitätsdiagramms erhalten bleiben. Das heißt, die Reihenfolge individueller Aktionen (Elemente vom Typ *UML::Action*), die in einer definierten Ausführungsreihenfolge auf einem gemeinsamen Kontrollfluss liegen, dürfen im Testfallmodell in keiner abweichenden Reihenfolge auftreten.

REQ-Transformation UML-zu-Testfallmodell 2: Sequenzalisierung paralleler Flüsse

> Die UML lässt für Aktivitätsdiagramme die Modellierung paralleler Kontrollflüsse zu (z. B. durch Verwendung von *UML::ForkNode*-Elementen). Eine Parallelisierung von Kontrollflüssen hat zur Folge, dass individuelle Aktionen innerhalb einer Aktivität unabhängig voneinander und möglicherweise mit einer zeitlichen Überlappung ausgeführt werden.
>
> Die hier untersuchte Methode zur Generierung von Akzeptanztests aus Aktivitätsdiagrammen impliziert, dass sich nebenläufige Kontrollflüsse in Anwendungen dem Anwender als Abbildung paralleler Sequenzen zu nichtnebenläufigen Sequenzen unter Permutation individueller Aktionen darstellen. Hintergrund ist, dass ein menschlicher Akteur nicht in der Lage ist, nebenläufig mit mehreren Komponenten einer Anwendung gleichzeitig zu interagieren. Es findet stets eine implizite Sequenzialisierung statt, die auf das menschliche Interaktionsvermögen und die

Anwendungsarchitektur mobiler Plattformen zurückzuführen ist, die sequenzielle Workflows durch Paradigmen des UI-Designs erzwingt.

Es ist daher Anforderung an die Transformation von Aktivitätsdiagrammen zu Testfallmodellen, *UML::Action*-Elemente, die auf nebenläufigen Kontrollflussabschnitten liegen, gemäß dem $C_{PS_{ACD}}$-Kriterium zu sequenzialisieren, so dass einerseits mögliche Permutationen der Interaktionsreihenfolge abgebildet werden, andererseits aber die Ausführungsreihenfolge gemäß REQ-Transformation 1 erhalten bleibt.

REQ-Transformation UML-zu-Testfallmodell 3: Isolierung relevanter Elemente

Aus der Perspektive von Akzeptanztests sind nur solche Funktionen einer Software direkt durch Tests adressierbar, die eine Benutzerschnittstelle exponieren. Andere Funktionen werden ggf. indirekt getestet, wenn sie von direkt erreichbaren Funktionen aufgerufen werden.

Es wird an die Transformation von Aktivitätsdiagrammen zu Testfallmodellen die Anforderung gestellt, dass nur für solche Funktionen (Aktionen), auf welche Stereotypen des in Abschnitt 5.2.4 vorgestellten UML-Profils angewendet wurden, unter Beibehalt der Testdaten im Testfallmodell Elemente erzeugt werden.

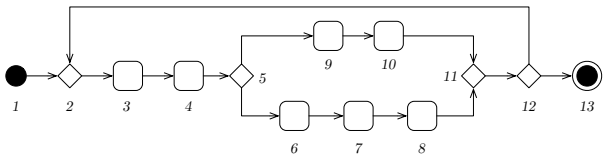
REQ-Transformation UML-zu-Testfallmodell 4: Testabdeckung

Es wird von der Transformation von Aktivitätsdiagrammen zu Testfallmodellen gefordert, dass Testfälle erzeugt werden, die das SUT mindestens entsprechend des Permuationssequenzialiserungsüberdeckungskriteriums $C_{PS_{ACD}}$ (im Detail in Abschnitt 5.3.2 diskutiert) für annotierte UML-Aktivitätsdiagramme abdecken.

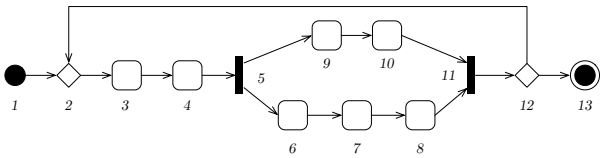
Zur Realisierung einer diesen Anforderungen genügenden Transformation erfolgt die Erzeugung von Testfallmodellen durch eine mehrphasige Modelltransformation. Diese überführt das UML-Aktivitätsdiagramm in einem ersten Schritt in ein Petri-Netz. Aus dessen Erreichbarkeitsgraph wird im Anschluss eine Menge gültiger Sequenzen von Aktionen erzeugt, die zulässige und relevante Workflows durch das Aktivitätsdiagramm repräsentieren. Im letzten Schritt werden aus diesen Sequenzen Modellinstanzen des Metamodells zur Testfallmodellierung erzeugt. Durch dieses Vorgehen wird die dritte Phase der Testfallgenerierung (vgl. Abschnitt 5.1) abgeschlossen.

5.3.3.1 Abbildung von Aktivitätsdiagrammen auf Testfälle

UML-Aktivitätsdiagramme sind Graphen, die sowohl Zyklen als auch nebenläufige Strukturen enthalten können. Das in Abschnitt 5.3.2 diskutierte $C_{PS_{ACD}}$-Abdeckungskriterium erfordert die Bildung zulässiger Sequenzen von *UML::Action*- und *UML::ActivityEdge*-Elementen, d. h. Pfade durch das Aktivitätsdiagramm, die mögliche Workflows abbilden. Für Graphen ohne nebenläufige Strukturen kann diese Aufgabe durch Tiefensuchealgorithmen gelöst werden.



(a) Aktivitätsdiagramm ohne nebenläufige Strukturen



(b) Aktivitätsdiagramm mit nebenläufiger Struktur

Abbildung 5.33: Beispiele von Aktivitätsdiagrammen ohne nebenläufige Struktur (5.33a) und mit nebenläufiger Struktur (5.33b)

Die Anwendung von Tiefen- oder Breitensuchealgorithmen auf Aktivitätsdiagramme kann allerdings im Sinne der Testfallgenerierung unzulässige Ergebnisse erzeugen.

In Abbildung 5.33 sind zwei Beispiele für Aktivitätsdiagramme dargestellt, die frei von nebenläufigen Strukturen sind (Abbildung 5.33a) bzw. die nebenläufige Strukturen enthalten (Abbildung 5.33b). Eine Anwendung eines Tiefensuchealgorithmus auf das in Abbildung 5.33a dargestellte Aktivitätsdiagramm erzeugt beispielsweise u. a. den Pfad:

1, 2, 3, 4, 5, 9, 10, 11, 12, 2, 3, 4, 5, 9, 10, 11, 12, 13

Der Knoten *2* taucht in diesem Pfad zweimal auf (an zweiter und zehnter Stelle). Hieran ist erkennbar, dass der Pfad einen Zyklus (*12* → *2*) enthält. Ohne Vorkehrungen dies zu vermeiden, generiert ein Tiefensuchealgorithmus unendlich viele Pfade, die allerdings zulässig sind, weil sie jeweils gültige Workflows durch das Aktivitätsdiagramm repräsentieren.

Wird ein Tiefensuchealgorithmus allerdings auf das in Abbildung 5.33b dargestellte Aktivitätsdiagramm angewendet, werden unzulässige Pfade erzeugt, die keine gültigen Workflows durch das Aktivitätsdiagramm repräsentieren. Der Pfad

1, 2, 3, 4, 5, 6, 7, 8, 11, 12, 13

ist im Aktivitätsdiagramm in Abbildung 5.33b beispielsweise ungültig, da er eine Fortsetzung des Kontrollflusses an Knoten *11* impliziert, ohne dass zuvor die Knoten *9* und *10* ausgeführt wurden. Aufgrund der Modellierungssemantik von UML-Aktivitätsdiagrammen kann an diesem Knoten der Workflow jedoch erst fortgesetzt werden, wenn auch die Knoten *9* und *10* ausgeführt wurden (vgl. *UML::ForkNode*, OMG [268]).

Die Anwendung eines Breitensuchealgorithmus erzeugt ein ähnliches Problem. Beispielsweise erzeugt er für das in Abbildung 5.33b dargestellte Aktivitätsdiagramm den gültigen Pfad

1, 2, 3, 4, 5, 6, 9, 7, 10, 8, 11, 12, 13.

Im Aktivitätsdiagramm in Abbildung 5.33a ist dieser Pfad hingegen ungültig, weil gemäß der Semantik des Elements *UML::DecisionNode* in diesem Aktivitätsdiagramm die Knoten *6* und *9* keinesfalls hintereinander ausgeführt werden können.

Zur Vermeidung dieser Problematik verfolgt der in dieser Dissertation untersuchte Ansatz zur Generierung von Testfällen aus UML-Aktivitätsdiagrammen den Weg, Aktivitätsdiagramme zunächst in Petri-Netze, konkret Bedingungs-/Ereignisnetze, zu überführen. Petri-Netze basieren auf einem formalen Kalkül, das es erlaubt, Analysen auf Netzen ausführen und hierbei insbesondere Nebenläufigkeit zu berücksichtigen (vgl. Peterson [279]). Die Transformation von Aktivitätsdiagrammen zu Petri-Netzen als Aufbereitung für eine weitere Analyse ist insofern naheliegend, als dass Aktivitätsdiagramme gemäß der Definition im Metamodell der UML bereits in Anlehnung an Petri-Netze als sogenanntes *Token Game* definiert sind, bei dem Kontrollflüsse durch ein Aktivitätsdiagramm als Schaltrelation von Transitionen (d. h. *UML::ActivityEdge*) zwischen markentragenden Stellen (d. h. *UML::ActivityNode*) definiert sind (vgl. OMG [268]).

Es werden B/E-Netze als Spezialfall allgemeiner Petri-Netze verwendet. Sie definieren eine maximale Stellenkapazität von eins sowie ein Kantengewicht von eins. Eine Transition ist schaltfähig, wenn auf allen Stellen im Vorbereich der Transitionen jeweils genau eine Marke liegt und alle Stellen im Nachbereich der Transition markenfrei sind. Stellen im generierten B/E-Netz repräsentieren *UML::ActivityNode*- bzw. *UML::ActivityEdge*-Elemente.

Das Metamodell der UML definiert für Aktivitätsdiagramme, dass Aktionen nebenläufig und ggf. zeitgleich mehrere Instanzen einer Aktion ausgeführt werden können. Die hier untersuchte Testfallgenerierung setzt voraus, dass ein *UML::ActivityNode*-Element (*UML::ActivityEdge*-Element), sofern es durch Anwendung eines der im UML-Profil zur Testfallmodellierung definierten Stereotypen als testrelevant markiert wurde, eine Interaktion des Anwenders mit der App repräsentiert. Zu jedem Zeitpunkt kann genau eine solche Interaktion ausgeführt werden. Das heißt abgebildet auf das Aktivitätsdiagramm, dass zu jedem Zeitpunkt genau ein *UML::ActivityNode*-Element (*UML::ActivityEdge*-Element) aktiv ist. Die nächste Aktion kann erst nach Abschluss der vorhergehenden ausgeführt werden.

Die Semantik von UML-Aktivitätsdiagrammen gibt kein Abstraktionsniveau zur Modellierung von Systemen vor. Weiterhin sind die Möglichkeiten der Modellierung mit Aktivitätsdiagrammen vielfältig. Um eine Kohärenz zwischen modelliertem System, Aktivitätsdiagramm und den hieraus generierten Testfällen zu gewährleisten, werden einige Anforderungen an Aktivitätsdiagramme gestellt. Diese werden in Abschnitt 5.3.3.1.1 diskutiert.

Abschnitt 5.3.3.1.2 diskutiert die eigentliche Transformation von Aktivitätsdiagrammen zu Petri-Netzen zum Zweck der Berechnung testrelevanter Ausführungspfade. Testfälle werden im Anschluss aus der Analyse des Erreichbarkeitsgraphen des Petri-Netzes erzeugt. Die Analyse des Erreichbarkeitsgraphen zur Testfallgenerierung ist Gegenstand des Abschnitts 5.3.3.1.3.

5.3.3.1.1 Anforderungen an Aktivitätsdiagramme zur Testfallgenerierung

Die Erzeugung von Tests aus UML-Aktivitätsdiagrammen wurde bereits mehrfach wissenschaftlich untersucht. Linzhang et al. [239] sowie Mingsong et al. [251] (vgl. Abschnitt 2.4) untersuchen beispielsweise in ihren Arbeiten ebenfalls die Generierung von Tests aus Aktivitätsdiagrammen und entwerfen hierzu jeweils Algorithmen, die auch aus Aktivitätsdiagrammen mit nebenläufigen Strukturen gültige Pfade erzeugen. Allerdings können diese Algorithmen nicht auf die hier untersuchte Problemstellung angewendet werden. Linzhang et al. [239] sowie Mingsong et al. [251] untersuchen nicht die Erzeugung von Tests auf der Basis mit Testda-

ten angereicherter Aktivitätsdiagramme, sondern betrachten Aktivitätsdiagramme ohne Anwendung von Profilen. Insbesondere können die von den Autoren vorgestellten Algorithmen keine *UML::ActivityEdge*-Elemente verarbeiten, die durch Anwendung von Stereotypen um Testdaten angereichert wurden. Die von den Autoren untersuchten Algorithmen behandeln *UML::ActivityEdge*-Elemente nicht als Elemente, mit denen Testdaten modelliert werden.

Weiterhin stellen sowohl Linzhang et al. als auch Mingsong et al. Anforderungen an die Wohlgeformtheit von Aktivitätsdiagrammen. So dürfen im Ansatz der Autoren *UML::ForkNode*- und *UML::JoinNode*-Elemente nur paarweise auftreten, da der vorgestellte Algorithmus anderenfalls nicht in der Lage ist, Ausführungspfade aus dem Aktivitätsdiagramm zu berechnen. Das Metamodell der UML macht hingegen keine Vorgaben zur paarweisen Verwendung von *UML::ForkNode*- und *UML::JoinNode*-Elementen.

UML-Aktivitätsdiagramme bieten vielfältige Möglichkeiten Anwendungsverhalten zu modellieren. Zur Umsetzung der hier untersuchten Methode zur Testfallgenerierung werden an Aktivitätsdiagramme Anforderungen gestellt. An die Wohlgeformtheit von Aktivitätsdiagrammen werden zur Realisierung des hier untersuchten Ansatzes keine Anforderungen gestellt, d. h. es gelten keine Voraussetzungen hinsichtlich der Modellierung von *UML::ForkNode*- und *UML::JoinNode*-Elementen. Allerdings wird die Anforderung gestellt, dass Aktivitätsdiagramme keine *UML::InterruptbileActivityRegion*-Elemente enthalten. Diese Einschränkung ergibt sich aus der Verwendung von B/E-Netzen als Zwischenschritt der Transformation.

Aktivitätsdiagramme bieten die Möglichkeit, sogenannte *UML::InterruptbileActivityRegion*-Elemente zu verwenden, um bei Eintritt von Ereignissen den Abbruch der Verarbeitung der Aktivität zu modellieren. Ein Beispiel für die Verwendung dieses Elements ist in Abbildung 5.34 dargestellt. Modelliert ist ein alternativer Entwurf eines Teilmodells „Standortbestimmung“ der App Mobiler Taxiruf.

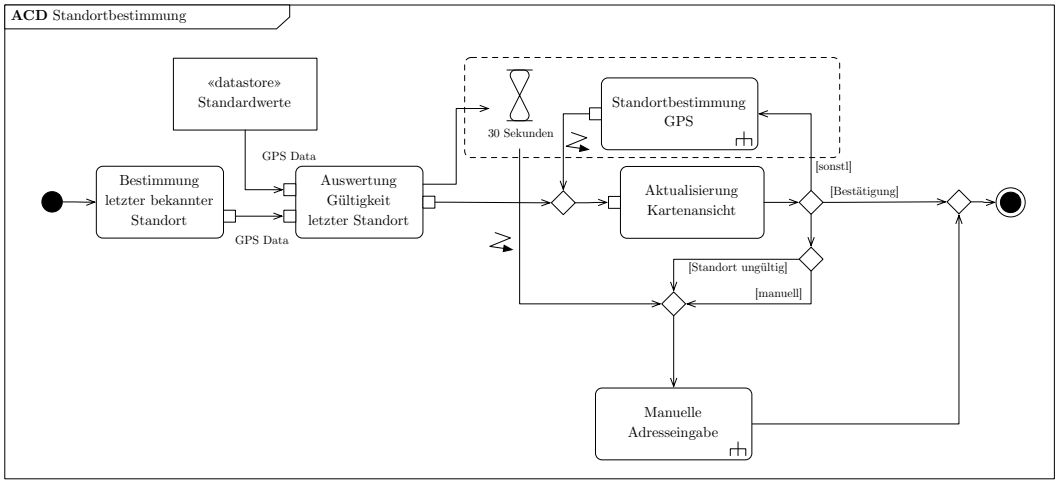


Abbildung 5.34: Verwendung des Elements *UML::InterruptbileActivityRegion* zur Modellierung von Abbrechereignissen im Systemmodell der App Mobiler Taxiruf

In Abbildung 5.34 ist ein Modell dargestellt, in dem ein Teil des Kontrollflusses innerhalb einer *UML::InterruptbileActivityRegion* modelliert ist. Im konkreten Beispiel wird der Versuch modelliert, den Standort des Anwenders zu ermitteln. Gelingt dies innerhalb von 30 Sekunden, werden alle Marken innerhalb der *UML::InterruptbileActivityRegion* gelöscht, wodurch insbe-

sondere die durch das *UML::AcceptTimeEventAction*-Element realisierte Abbruchbedingung deaktiviert wird. Gelingt es nicht innerhalb von 30 Sekunden den Standort zu bestimmen, wird der Versuch abgebrochen und zur manuellen Adresseingabe weitergeleitet.

Die Verwendung des Standardmodells von B/E-Netzen sieht durch einzelne Zustandsübergänge verursachte Löschungen von Marken aus dem Netz nicht vor. Zwar bestünde grundsätzlich die Option, durch Verwendung eines erweiterten Modells von B/E-Netzen mit Inhibitorkanten das Verhalten von *UML::InterruptbileActivityRegion*-Element und insbesondere von *UML::InterruptingEdge*-Elementen nachzubilden. Im Rahmen der hier untersuchten Methode der Generierung von Tests aus Aktivitätsdiagrammen wurde das allerdings nicht umgesetzt.

5.3.3.1.2 Überführung in Petri-Netze

Aktivitätsdiagramme sind bereits nahe an der Semantik von Petri-Netzen definiert, wobei *UML::ActivityEdge*-Elemente die Funktion von Transitionen in Petri-Netzen erfüllen und *UML::ActivityNode*-Elemente markentragende Stellen repräsentieren. Gemäß der semantischen Definition von UML-Aktivitätsdiagrammen kann ein *UML::ActivityNode*-Element ausgeführt werden, sobald an jeder eingehenden Kante (d. h. *UML::ActivityEdge*) eine Marke auf dem *UML::ActivityNode*-Element platziert wird (vgl. OMG [268]). Das UML-Profil zur Testfallmodellierung ermöglicht die Anwendung des Stereotyps *TestFlow* auf *UML::ActivityEdge*-Elemente, wodurch diese selbst zu semantiktragenden Testartefakten werden, die Vor- und Nachbedingungen sowie Interaktionen mit dem SUT modellieren. Deshalb werden zur Transformation von Aktivitätsdiagrammen zu Petri-Netzen neben *UML::ActivityNode*-Elementen ebenfalls *UML::ActivityEdge*-Elemente als markentragende Stellen interpretiert.

Bei der Transformation wird entsprechend den im Folgenden definierten Transformationsregeln *ACD-zu-PN* aus einem Aktivitätsdiagramm ein B/E-Netz erzeugt, wobei für jeden Knoten sowie für jeden Kontroll- bzw. Datenfluss eine Stelle im B/E-Netz erzeugt wird. Die Transitionen werden anhand der Relationen von *UML::ActivityNode*- und *UML::ActivityEdge*-Elementen generiert.

Transformationsregel ACD-zu-PN 1: UML::ActivityNode → B/E-Netz Stelle

> Sei $ACD = (N, A, C, N_I, N_F, R)$ ein Aktivitätsdiagramm. Bei der Transformation von ACD in ein B/E-Netz wird für jeden Knoten $n \in N$ eine Stelle im B/E-Netz erzeugt.

Transformationsregel ACD-zu-PN 2: UML::ActivityEdge → B/E-Netz Stelle

> Sei $ACD = (N, A, C, N_I, N_F, R)$ ein Aktivitätsdiagramm. Bei der Transformation von ACD in ein B/E-Netz wird für jeden Kontrollfluss $r \in R$ eine Stelle im B/E-Netz erzeugt.

Transformationsregel ACD-zu-PN 3: UML::ActivityEdge → B/E-Netz Transitionen

> Sei $ACD = (N, A, C, N_I, N_F, R)$ ein Aktivitätsdiagramm. Bei der Transformation von ACD in ein B/E-Netz werden für jeden Kontrollfluss $r \in R$ Transitionen im B/E-Netz erzeugt.

Für jedes *UML::ActivityEdge*-Element entstehen hierbei höchstens zwei Transitionen, wobei eine Transition diejenigen Stellen des B/E-Netzes miteinander verbinden, die das *UML::ActivityNode*-Element der Quelle des *UML::ActivityEdge*-Elements und das *UML::ActivityEdge*-Element repräsentieren. Die zweite Transition verbindet diejenigen Stellen des B/E-Netzes, die das *UML::ActivityEdge*-Element und das Ziel-*UML::ActivityNode*-Element des *UML::ActivityEdge*-Elements repräsentieren.

Ist die Quelle (das Ziel) des *UML::ActivityEdge*-Elements eine *UML::ForkNode* (*UML::JoinNode*), wird eine bereits existierende Transition wiederverwendet, um die Parallelisierung des Kontrollflusses im B/E-Netz abzubilden.

Transformationsregel ACD-zu-PN 4: Anfangsmarkierung

Die Anfangsmarkierung des B/E-Netzes wird so gewählt, dass auf jeder Stelle eine Marke platziert wird, die ein *UML::InitialNode*-Element oder ein *UML::ActivityNode*-Element ohne eingehende Kanten (ebenfalls als Anfangsknoten qualifiziert, vgl. Metamodell der UML [268]) repräsentiert.

Ein Beispiel einer ACD-zu-PN Transformation ist in Abbildung 5.35 dargestellt. Die Abbildung zeigt den Ausschnitt aus dem Modell der App Mobiler Taxiruf, der die Anwendung auf einem hohen Abstraktionsniveau modelliert.

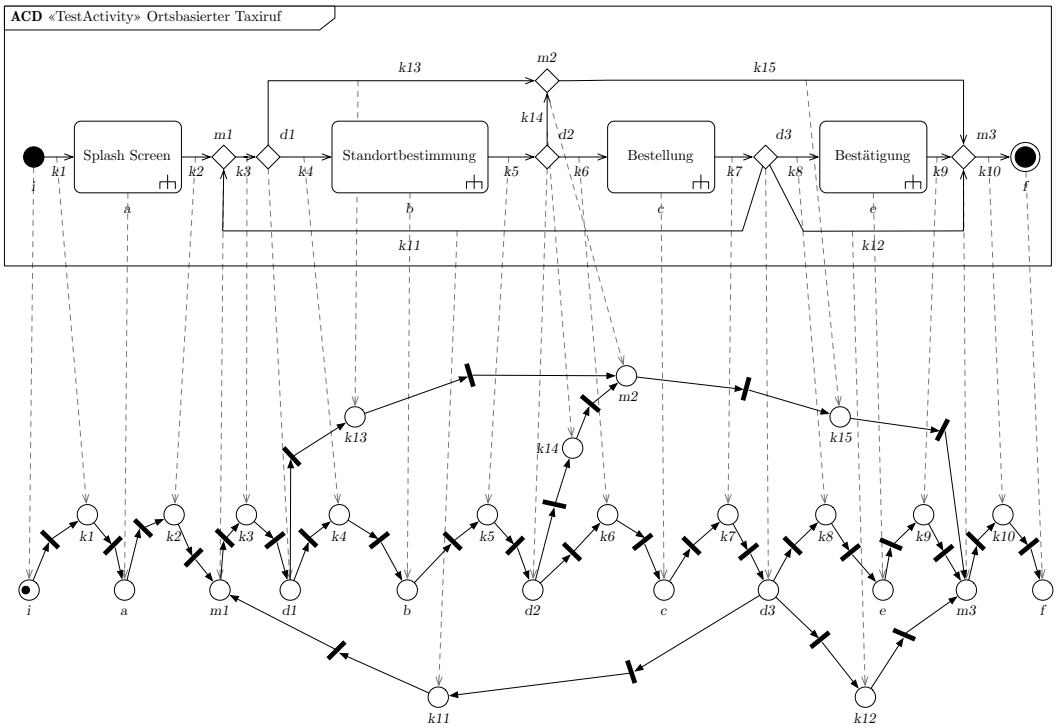


Abbildung 5.35: Beispiel der Transformation eines Aktivitätsdiagramms in ein B/E-Netz. Abgebildet ist ein Teilmodell der App Mobiler Taxiruf mit dem durch die Transformation erzeugten B/E-Netz.

Die Anwendung der Transformationsregeln 1 und 2 manifestieren sich darin, dass für alle *UML::ActivityNode*-Elemente und alle *UML::ActivityEdge*-Elemente (bezeichnet mit *k**)

Stellen im B/E-Netz erzeugt wurden. Transformationsregel 3 manifestiert sich darin, dass für jedes *UML::ActivityEdge*-Element ein Paar von Transitionen erzeugt wurde. Die Anwendung von Transformationsregel 4 platziert eine Marke in Stelle *i*.

Das in Abbildung 5.35 dargestellte B/E-Netz ist ein Abbild des Aktivitätsdiagramms, da es identischer Weise einen Markenfluss durch ein Netz modelliert. Eine auf einer Stelle des B/E-Netzes platzierte Marke impliziert, dass das mit dieser Stelle assoziierte Element des Aktivitätsdiagramms ausgeführt wird. Eine Marke auf der Stelle *i* ist beispielsweise gleichbedeutend mit einer auf dem *UML::InitialNode*-Element *i* platzierten Marke in der Semantik des Aktivitätsdiagramms. Schalten der auf Stelle *i* folgenden Transition löscht die Marke in *i* und platziert eine Marke in Stelle *k1*. Diese wiederum ist assoziiert mit dem Kontrollfluss *k1* im Aktivitätsdiagramm. Eine Marke in Stelle *k1* repräsentiert in der Semantik des Aktivitätsdiagramms den Übergang zwischen dem *UML::InitialNode*-Element und dem *UML::Action*-Element *a*. Die Darstellung des Aktivitätsdiagramms als B/E-Netz bildet die Grundlage für die Berechnung von Ausführungssequenzen der modellierten Anwendung.

5.3.3.1.3 Modellanalyse und Identifikation testrelevanter Ausführungspfade

Testfälle sind geordnete Sequenzen individueller Aktionen, die während der Ausführung von Tests abgearbeitet werden. Da Aktivitätsdiagramme zyklische und nebenläufige Kontrollflussstrukturen enthalten können, ist eine direkte Anwendung von Tiefen- oder Breitensuchealgorithmen auf Aktivitätsdiagramme nicht geeignet, solche geordneten Sequenzen individueller Aktionen zu berechnen.

Basierend auf einem Erreichbarkeitsgraphen des aus einem Aktivitätsdiagramm erzeugten B/E-Netzes können Ausführungssequenzen von *UML::ActivityNode*- und *UML::ActivityEdge*-Elementen berechnet werden, weil der Erreichbarkeitsgraph eines B/E-Netzes zwar nebenläufige Kontrollflüsse im zugehörigen B/E-Netz abbildet, selbst jedoch ausschließlich alternative Verzweigungspunkte enthält. Tiefensuchealgorithmen sind deshalb geeignet, um zur Analyse von Erreichbarkeitsgraphen und damit des zugrundeliegenden B/E-Netzes angewendet zu werden. Die aus dieser Analyse gewonnenen Erkenntnisse können auf das dem B/E-Netz zugrundeliegenden Aktivitätsdiagramm zurückgeführt werden.

Die Erzeugung von Ausführungspfaden aus Aktivitätsdiagrammen erfolgt in mehreren Schritten: (1) Nachdem zu einem Aktivitätsdiagramm der Erreichbarkeitsgraph erstellt wurde, wird dieser durch einen Tiefesuchalgorithmus traversiert, der jede Kante genau einmal besucht. Hierbei entstehen Sequenzen von Markierungen, die Abfolgen von Aktionen im Aktivitätsdiagramm repräsentieren. Liegt eine Marke auf einer Stelle des B/E-Netzes, bedeutet das in der Semantik des Aktivitätsdiagramms, dass die zugehörige Aktion ausgeführt wird. Die Traversierung des Erreichbarkeitsgraphen erzeugt Sequenzen von Markierungen, die solche Ausführungspfade durch das Aktivitätsdiagramm repräsentieren, die bereits das $C_{0_{ACD}}$-Überdeckungskriterium (Aktionsabdeckung) erfüllen, weil sichergestellt ist, dass jede im Aktivitätsdiagramm spezifizierte Aktion in mindestens einer Markierungssequenz enthalten ist.

Der erste Schritt erzeugt Markierungssequenzen, die frei von Zyklen sind und die nicht zwingend mit einer Markierung enden, die einen Endpunkt des Kontrollflusses des Aktivitätsdiagramms repräsentieren. Deshalb werden (2) zur Erfüllung des $C_{2c_{ACD}}$-Überdeckungskriteriums (Basispfadüberdeckung) in einem zweiten Schritt die zuvor erzeugten Markierungsse-

quenzen um Zyklen ergänzt und so vervollständigt, dass jede Sequenz mit einer Markierung endet, die einen Endpunkt des Kontrollflusses des Aktivitätsdiagramms repräsentiert.

Die bis zu diesem Punkt erzeugten Markierungssequenzen enthalten Markierungen, die Marken auf mehreren Stellen des B/E-Netzes platzieren. Hierdurch werden nebenläufige Kontrollflussstrukturen abgebildet. Von der Transformation von Aktivitätsdiagrammen zu Testfällen wird gefordert, das $C_{PS_{ACD}}$-Überdeckungskriterium (Permutationssequenzüberdeckung) zu erfüllen. Deshalb werden (3) Markierungen, die Marken auf mehreren Stellen des B/E-Netzes platzieren, in einem dritten Schritt so sequenzialisiert, dass das in Abschnitt 5.3.2 definierte Kriterium zur Zulässigkeit von Aktionspermutationen nicht verletzt wird.

Aktivitätsdiagramme können durch Verwendung des Modellierungselements *UML::CallBehaviorAction* hierarchisch strukturiert werden, so dass untergeordnete Aktivitätsdiagramme von übergeordneten referenziert werden. Um aus Aktivitätsdiagrammen Testfälle zu generieren, müssen solche Abhängigkeiten bei der Transformation berücksichtigt werden. Deshalb werden (4) hierarchische Beziehungen in einem vierten Schritt aufgelöst, indem Markierungssequenzen untergeordneter Aktivitätsdiagramme in diejenigen Sequenzen eingebettet werden, die für übergeordnete Aktivitätsdiagramme erzeugt wurden.

Die bis zu diesem Zeitpunkt berechneten Markierungssequenzen für B/E-Netze, die ihrerseits aus Aktivitätsdiagrammen generiert wurden, bilden Ausführungspfade durch die zugrundeliegenden Aktivitätsdiagramme ab. Sie enthalten alle *UML::ActivityNode*- und *UML::ActivityEdge*-Elemente, die im Aktivitätsdiagramm modelliert sind, ungeachtet ihrer Kennzeichnung als testrelevant (d. h. Anwendung einer der in Abschnitt 5.2.4 definierten Stereotypen, vgl. Abschnitt 5.3.2). Zur Erzeugung von Ausführungspfaden, die nur testrelevante Modellierungselemente enthalten, werden (5) in einem fünften Schritt alle Elemente aus Ausführungspfaden entfernt, die nicht als testrelevant gekennzeichnet wurden.

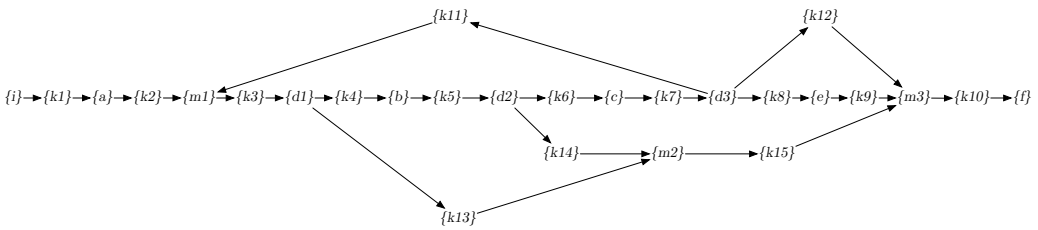


Abbildung 5.36: Erreichbarkeitsgraph des in Abbildung 5.35 dargestellten B/E-Netzes, App Mobiler Taxiruf, Teilmodell Ortsbasierter Taxiruf.

In Abbildung 5.36 ist der zu dem in Abbildung 5.35 dargestellten Aktivitätsdiagramm bzw. B/E-Netz gehörige Erreichbarkeitsgraph abgebildet. Er repräsentiert mögliche Ausführungssequenzen von Aktionen (d. h. Pfade) durch das Aktivitätsdiagramm, das eingebettete Aktivitäten (*UML::CallBehaviorAction*-Elemente) enthält. Das in Abschnitt 5.2.4 vorgestellte UML-Profil zur Integration von Testdaten in Aktivitätsdiagramm definiert den Stereotyp *TestActivity*. Nur solche Aktivitätsdiagramme, auf die dieser Stereotyp angewendet wurde, werden explizit bei der Generierung von Testfällen berücksichtigt, d. h. zu *Test::TestSuite*-Elementen im Testfallmodell transformiert. Aktivitätsdiagramme können allerdings eingebettete Aktivitäten referenzieren, die ihrerseits zur Durchführung von Tests notwendige Informa-

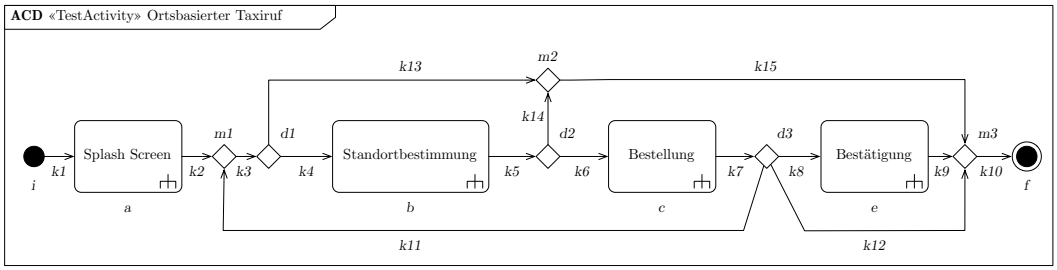
tionen enthalten, d. h. Sequenzen von individuellen Aktionen, die bei der Durchführung eines spezifischen Tests ausgeführt werden.

Aktionsüberdeckung

Die Anwendung eines Tiefensuchealgorithmus auf den Erreichbarkeitsgraphen eines aus einem Aktivitätsdiagramm generierten B/E-Netzes erzeugt Sequenzen von B/E-Netzmarkierungen, die auf Sequenzen der im Aktivitätsdiagramm modellierten Aktionen abgebildet werden können. Ein Tiefensuchealgorithmus, der jede Kante zwischen den Knoten des Erreichbarkeitsgraphen genau einmal besucht, erzeugt Sequenzen von Markierungen (Aktionen), die bereits das $C0_{ACD}$-Überdeckungskriterium (Aktionsüberdeckungskriterium, vgl. Abschnitt 5.3.2) für dieses spezifische Aktivitätsdiagramm unter Abstraktion eingebetteter Aktivitäten erfüllen. Die Knoten des Erreichbarkeitsgraphen repräsentieren Markierungen des B/E-Netzes[5], die jeweils den Ausführungszustand des zugrundeliegenden Aktivitätsdiagramms abbilden.

Für Transitionen im B/E-Netz ist keine natürliche Ordnungsrelation definiert. Deshalb können je nach Auswahl der zuerst schaltenden Transition Erreichbarkeitsgraphen unterschiedlicher Topologie entstehen. Die Erfüllung der in Abschnitt 5.3.2 definierten Überdeckungskriterien wird hierdurch jedoch nicht berührt, da das geforderte Abdeckungskriterium auch für unterschiedliche Topologien des Erreichbarkeitsgraphen erfüllt wird.

Angewendet auf den in Abbildung 5.36 dargestellten Erreichbarkeitsgraphen des Aktivitätsdiagramms Ortsbasierter Taxiruf des Systemmodells der App Mobiler Taxiruf generiert ein Tiefensuchealgorithmus die in Abbildung 5.37b dargestellten Ausführungspfade.



(a) Aktivitätsdiagramm des Teilmodells Ortsbasierter Taxiruf der App Mobiler Taxiruf

$P_1 = i \rightarrow k1 \rightarrow a \rightarrow k2 \rightarrow m1 \rightarrow k3 \rightarrow d1 \rightarrow k4 \rightarrow b \rightarrow k5 \rightarrow d2 \rightarrow k6 \rightarrow c \rightarrow k7 \rightarrow d3 \rightarrow k8 \rightarrow e \rightarrow k9 \rightarrow m3 \rightarrow k10 \rightarrow f$
$P_2 = i \rightarrow k1 \rightarrow a \rightarrow k2 \rightarrow m1 \rightarrow k3 \rightarrow d1 \rightarrow k4 \rightarrow b \rightarrow k5 \rightarrow d2 \rightarrow k6 \rightarrow c \rightarrow k7 \rightarrow d3 \rightarrow k11 \rightarrow m1$
$P_3 = i \rightarrow k1 \rightarrow a \rightarrow k2 \rightarrow m1 \rightarrow k3 \rightarrow d1 \rightarrow k4 \rightarrow b \rightarrow k5 \rightarrow d2 \rightarrow k6 \rightarrow c \rightarrow k7 \rightarrow d3 \rightarrow k12 \rightarrow m3$
$P_4 = i \rightarrow k1 \rightarrow a \rightarrow k2 \rightarrow m1 \rightarrow k3 \rightarrow d1 \rightarrow k4 \rightarrow b \rightarrow k5 \rightarrow d2 \rightarrow k14 \rightarrow m2 \rightarrow k15 \rightarrow m3$
$P_5 = i \rightarrow k1 \rightarrow a \rightarrow k2 \rightarrow m1 \rightarrow k3 \rightarrow d1 \rightarrow k13 \rightarrow m2$

(b) Mögliche Ausführungspfade des Teilmodells Ortsbasierter Taxiruf der App Mobiler Taxiruf unter dem Aktionsabdeckungskriterium

Abbildung 5.37: Ausführungspfade durch das Teilmodell Ortsbasierter Taxiruf der App Mobiler Taxiruf unter dem $C0_{ACD}$-Testabdeckungskriterium (Aktionsabdeckung)

[5]Zur Verkürzung der Darstellung wurde die Schreibweise $\{k\}$ für eine Netzmarkierung gewählt. Sie repräsentiert eine Markierung, in der nur auf der Stelle k eine Marke liegt, während alle anderen Stellen markenfrei sind.

Der Tiefensuchealgorithmus besucht jede Kante des Erreichbarkeitsgraphen genau einmal. Unter den erzeugten Pfaden ist der Pfad P_1 in Abbildung 5.37b der einzige Pfad, der das Teilmodell vollständig vom Anfangsknoten (*UML::InitialNode* i) bis zum Endknoten (*UML::ActivityFinalNode* f) traversiert. Die in Abbildung 5.37b dargestellten Pfade decken dabei in ihrer Gesamtheit alle Aktionen ab, die im Aktivitätsdiagramm modelliert sind.

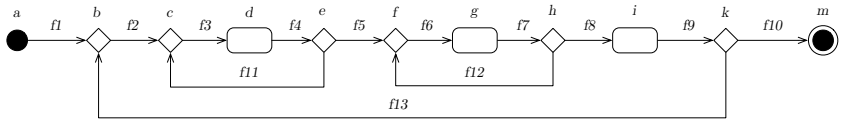
Allerdings enthält das in Abbildung 5.37a abgebildete Aktivitätsdiagramm eingebettete Aktivitäten, so dass die in diesem Schritt der Testfallgenerierung erzeugten Pfade nicht unmittelbar gültige Tests der Anwendung repräsentieren. Einerseits können in den eingebetteten Aktivitäten weitere testrelevante Modellierungselemente enthalten sein (d. h. *UML::Action*-Element und *UML::ActivityEdge*-Element, auf die Stereotypen des UML-Profils zur Integration von Kontext- und Testdaten in die Systemmodellierung, vgl. Abschnitt 5.2.4 angewendet wurden). Die eingebetteten Aktivitäten können weiterhin nebenläufige und alternative Kontrollflussstrukturen enthalten, wodurch sich über die in Abbildung 5.37b dargestellten Ausführungspfade hinaus weitere Pfade ergeben, die zur Erfüllung des Testabdeckungskriteriums bei der Generierung von Testfällen berücksichtigt werden müssen. Andererseits wird von der Transformation von UML-Modellen zu Testfallmodellen gefordert, das $C_{PS_{ACD}}$-Abdeckungskriterium (Permutationssequenzüberdeckung, vgl. Abschnitt 5.3.2, Anforderung REQ-Transformation 4) zu erfüllen. Dieses schließt das $C_{2c_{ACD}}$-Abdeckungskriterium (Basispfadüberdeckungskriterium) mit ein, welches eine mindestens einmalige Traversierung von Zyklen im Kontrollfluss fordert.

Basispfadüberdeckung

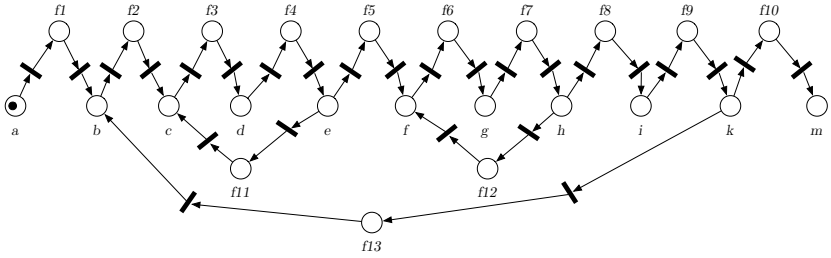
Um das in Abschnitt 5.3.2 definierte $C_{2_{ACD}}$-Kriterium (Basispfadüberdeckung) zu erfüllen, muss die aus dem Erreichbarkeitsgraphen des B/E-Netzes erzeugte Menge von Markierungssequenzen um Zyklen ergänzt werden. Die per Tiefensuche aus dem Erreichbarkeitsgraphen erzeugte Menge von Markierungssequenzen bilden den Erreichbarkeitsgraphen auf einen Erreichbarkeitsbaum ab, dessen Blattknoten entweder solche B/E-Netzmarkierungen repräsentieren, in denen keine weitere Transition mehr schaltfähig ist (d. h. Ende des Kontrollflusses im Aktivitätsdiagramm) oder solche Knoten, die an einer anderen Stelle im Erreichbarkeitsgraphen bereits vorkommen. Aufgrund der strukturellen Eigenschaften der in Abschnitt 5.3.3.1.2 diskutierten Transformation von Aktivitätsdiagrammen zu B/E-Netzen repräsentieren B/E-Netzmarkierungen, in denen keine weitere Transition mehr schaltfähig ist, das Erreichen des Endes des Kontrollflusses im Aktivitätsdiagramm. Blattknoten des Erreichbarkeitsbaums, die an anderer Stelle im Erreichbarkeitsbaum bereits vorkommen, repräsentieren Zustände des zugrundeliegenden Aktivitätsdiagramms, die durch einen anderen Zweig des Erreichbarkeitsbaums bis zum Erreichen des Endes des Kontrollflusses im Aktivitätsdiagramm fortgeführt werden. Insbesondere repräsentieren solche Markierungen potenzielle Zyklen im Kontrollfluss des Aktivitätsdiagramms.

Zyklen im Kontrollfluss eines Aktivitätsdiagramms können basierend auf Markierungssequenzen, die durch Tiefensuche im Erreichbarkeitsgraphen erzeugt wurden, erkannt werden. Hierzu muss überprüft werden, ob eine B/E-Netzmarkierung am Index i im Pfad bereits an einem anderen Index $j < i$ enthalten ist. In Abbildung 5.38a ist beispielhaft ein abstrakter Prozess in einem Aktivitätsdiagramm dargestellt, das mehrere Zyklen enthält. Dieses Beispiel

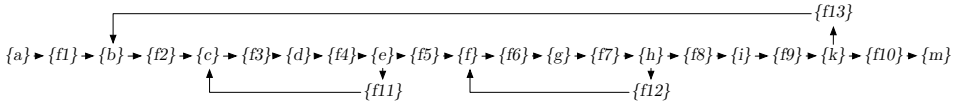
wurde ausgewählt, weil es im Gegensatz zum Systemmodell der App Mobiler Taxiruf, mehrere Zyklen enthält, von denen einer die anderen überdeckt.



(a) Beispiel eines Aktivitätsdiagramms mit mehreren Zyklen



(b) Aus dem Aktivitätsdiagramm in Abbildung 5.38a generiertes B/E-Netz



(c) Aus dem B/E-Netz in Abbildung 5.38b generierter Erreichbarkeitsgraph

$A_1 = a \rightarrow f1 \rightarrow b \rightarrow f2 \rightarrow c \rightarrow f3 \rightarrow d \rightarrow f4 \rightarrow e \rightarrow f5 \rightarrow f \rightarrow f6 \rightarrow g \rightarrow f7 \rightarrow h \rightarrow f8 \rightarrow i \rightarrow f9 \rightarrow k \rightarrow f10 \rightarrow m$

$A_2 = a \rightarrow f1 \rightarrow \mathbf{b}_2 \rightarrow f2 \rightarrow c \rightarrow f3 \rightarrow d \rightarrow f4 \rightarrow e \rightarrow f5 \rightarrow f \rightarrow f6 \rightarrow g \rightarrow f7 \rightarrow h \rightarrow f8 \rightarrow i \rightarrow f9 \rightarrow k \rightarrow f13 \rightarrow \mathbf{b}_{20}$

$A_3 = a \rightarrow f1 \rightarrow b \rightarrow f2 \rightarrow c \rightarrow f3 \rightarrow d \rightarrow f4 \rightarrow e \rightarrow f5 \rightarrow \mathbf{f}_{10} \rightarrow f6 \rightarrow g \rightarrow f7 \rightarrow h \rightarrow f12 \rightarrow \mathbf{f}_{16}$

$A_4 = a \rightarrow f1 \rightarrow b \rightarrow f2 \rightarrow \mathbf{c}_4 \rightarrow f3 \rightarrow d \rightarrow f4 \rightarrow e \rightarrow f11 \rightarrow \mathbf{c}_{10}$

(d) Aus dem Erreichbarkeitsgraph in Abbildung 5.38c durch Tiefensuche generierte Menge von Markierungssequenzen

Abbildung 5.38: Beispiel eines Aktivitätsdiagramms, das mehrere ineinander eingebettete Zyklen enthält.

Die in Abbildung 5.38d dargestellte Menge von Markierungssequenzen wurde durch Anwendung eines Tiefensuchealgorithmus auf den Erreichbarkeitgraphen des zu dem in Abbildung 5.38a dargestellten Aktivitätsdiagramms gehörigen B/E-Netzes (Abbildung 5.38b) berechnet. Die Markierungssequenz A_1 ist eine Traversierung des Erreichbarkeitsgraphen bis zu einer Markierung, die das Ende des Kontrollflusses im Aktivitätsdiagramm repräsentiert. Die Markierungssequenzen A_2, A_3 sowie A_4 enden jeweils mit einer Markierung, die bereits an früherer Stelle im Erreichbarkeitsgraphen vorkommt.

Konkret am Beispiel der Sequenz A_3 tritt die Markierung f an Index zehn und an Index sechzehn auf. In der Semantik des zugrundeliegenden Aktivitätsdiagramms repräsentiert diese Markierungssequenz einen Ausführungspfad, der einen Zyklus am *UML::DecisionNode*-Element h einleitet, der den Kontrollfluss zum *UML::MergeNode*-Element f zurückführt.

Das $C_{2_{ACD}}$-Überdeckungskriterium (Basispfadüberdeckung) fordert, dass solche Testfälle erzeugt werden, die jeden Zyklus im Kontrollfluss einer Anwendungen mindestens einmal durchlaufen. Die durch Tiefensuche im Erreichbarkeitsgraphen des auf dem Aktivitätsdiagramm basierenden B/E-Netzes erzeugte Menge von Markierungssequenzen erfüllt diese An-

forderung nicht, da ein Verzweigungspunkt, der einen Zyklus einleitet, bei der Tiefensuche bereits besucht worden sein kann und die Traversierung an dieser Stelle abgebrochen wird.

Die Erfüllung des $C_{2_{ACD}}$-Überdeckungskriterium kann jedoch durch Expansion derjenigen Markierungssequenzen erfüllt werden, die einen Zyklus im Kontrollfluss des zugrundeliegenden Aktivitätsdiagramms repräsentieren. Zu diesem Zweck wird in jeder Markierungssequenz, die eine Markierung mehrfach enthält, die zwischen den Indices des Auftretens der mehrfach vorkommenden Markierung liegende Teilsequenz identifiziert und hinter dem größeren Index erneut in die Sequenz eingefügt.

Konkret am Beispiel des in Abbildung 5.38d dargestellten Pfads A_3 wird die zwischen Index zehn und Index sechzehn liegende Teilsequenz identifiziert und hinter dem Index sechzehn erneut in die Sequenz eingefügt. Hierbei ist nicht Voraussetzung, dass die Sequenz nach dem größten Index des Mehrfachvorkommens einer Markierung endet. Es ist ebenso zulässig, eine Teilsequenz nicht am Ende einer Sequenz anzufügen, sondern innerhalb der Sequenz einzufügen. Das Schema der Expansion von Markierungssequenzen zur Erfüllung des Basispfadüberdeckungkriteriums ist in Abbildung 5.39 dargestellt.

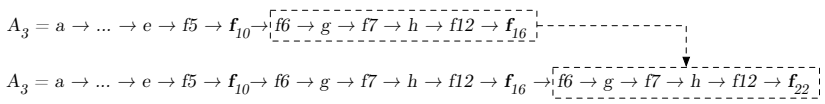


Abbildung 5.39: Schematische Darstellung der Expansion von Markierungssequenzen um Teilsequenzen, die Zyklen im Kontrollfluss repräsentieren.

Dieser Prozess der Zyklusexpansion wird für alle Markierungssequenzen durchgeführt. Hierdurch wird gewährleistet, dass nach der Expansionsphase Markierungssequenzen existieren, so dass jeder Zyklus im Kontrollfluss des zugrundeliegenden Aktivitätsdiagramms mindestens einmal durchlaufen wird. Durch n-fache Wiederholung der Expansionsphase könnte an dieser Stelle das $C_{2_{ACD}}$-Überdeckungskriterium so erweitert werden, dass Zyklen im Kontrollfluss n-fach durchlaufen werden.

Nach der Expansion der im Aktivitätsdiagramm enthaltenen Zyklen ergibt sich für das in Abbildung 5.38 dargestellte Modell die in Abbildung 5.40 dargestellte Menge von Markierungssequenzen.

$A_1 = a \to f1 \to b \to f2 \to c \to f3 \to d \to f4 \to e \to f5 \to f \to f6 \to g \to f7 \to h \to f8 \to i \to f9 \to k \to f10 \to m$

$A_2 = a \to f1 \to b \to f2 \to c \to f3 \to d \to f4 \to e \to f5 \to f \to f6 \to g \to f7 \to h \to f8 \to i \to f9 \to k \to f13 \to b$
$\to f2 \to c \to f3 \to d \to f4 \to e \to f5 \to f \to f6 \to g \to f7 \to h \to f8 \to i \to f9 \to k \to f13 \to b$

$A_3 = a \to f1 \to b \to f2 \to c \to f3 \to d \to f4 \to e \to f5 \to f \to f6 \to g \to f7 \to h \to f12 \to f$
$\to f6 \to g \to f7 \to h \to f12 \to f$

$A_4 = a \to f1 \to b \to f2 \to c_4 \to f3 \to d \to f4 \to e \to f11 \to c$
$\to f3 \to d \to f4 \to e \to f11 \to c$

Abbildung 5.40: Menge von Sequenzen durch das in Abbildung 5.38a dargestellte Aktivitätsdiagramm mit expandierten Zyklen

Die in Abbildung 5.40 dargestellte Menge von Markierungssequenzen (d. h. Pfade durch den Erreichbarkeitsgraphen des B/E-Netzes) enthält nach der Expansion von Zyklen Sequenzen, die nicht mit einer Markierung enden, durch welche das Ende des Kontrollflusses durch

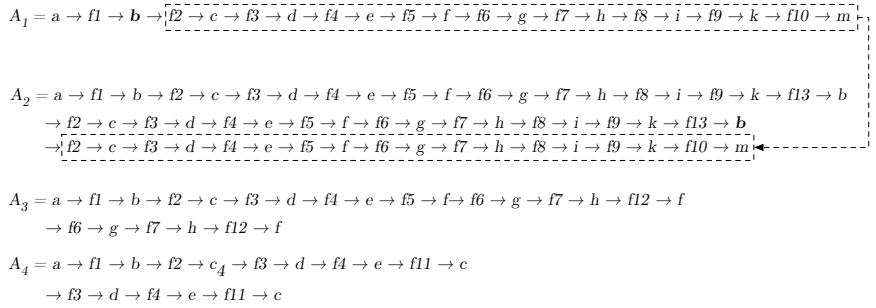
das zugrundeliegende Aktivitätsdiagramm repräsentiert wird. Nur der Pfad A_1 endet mit dem Knoten m, welcher im zugrundeliegenden Aktivitätsdiagramm (Abbildung 5.38a) das *UML::ActivityFinalNode*-Element repräsentiert.

Zur Erfüllung des $C_{2_{ACD}}$-Kriteriums (Basispfadüberdeckung) ist es notwendig, die zur Vervollständigung von Zyklen expandierten Sequenzen von Markierungen so zu erweitern, dass jede Sequenz mit einer Markierung endet, die das Ende des Kontrollflusses des zugrundeliegenden Aktivitätsdiagramms repräsentiert. Ähnlich der Expansion von Zyklen kann diese Erweiterung ebenfalls basierend auf den bisher existierenden Sequenzen erfolgen. Hierzu werden die bisher generierten Sequenzen durch Analyse des jeweils letzten Elements in zwei disjunkte Mengen von Markierungssequenzen eingeteilt, von denen eine diejenigen Markierungssequenzen enthält, die bereits mit einer Markierung enden, die das Ende des Kontrollflusses des Aktivitätsdiagramms repräsentieren, d. h. vollständige Sequenzen. Die zweite Menge enthält alle Markierungssequenzen, deren letztes Element kein Kontrollflussende repräsentiert, d. h. unvollständige Sequenzen.

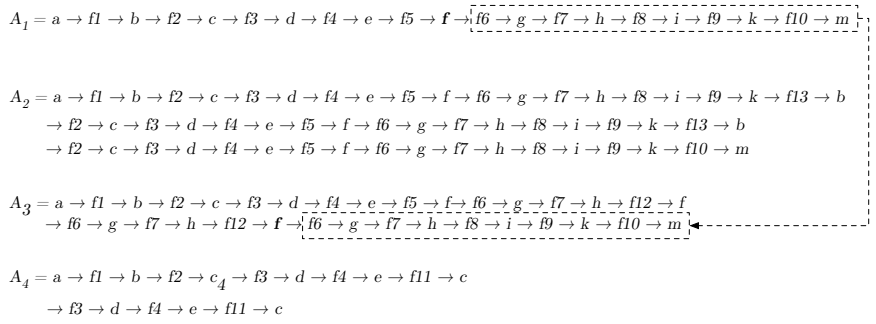
Die Erweiterung der Markierungssequenzen bis zur Erfüllung des $C_{2_{ACD}}$-Kriteriums erfolgt, indem der Menge der unvollständigen Sequenzen einzelne Sequenzen entnommen werden und durch Ergänzung von Postfix-Teilsequenzen aus der Menge der vollständigen Sequenzen ergänzt werden, bis keine weiteren unvollständigen Markierungssequenzen mehr existieren. Zur Ergänzung unvollständiger Sequenzen werden aus der Menge der vollständigen Sequenzen diejenigen ausgewählt, die eine Teilsequenz enthalten, die mit einer Markierung beginnt, die zum letzten Element der bearbeiteten unvollständigen Markierungssequenz identisch ist. Diese Teilsequenz wird am Ende der unvollständigen Sequenz angefügt. In der Menge der vollständigen Sequenzen können mehrere Sequenzen enthalten sein, die eine anzufügende Teilsequenz enthalten. In diesem Fall wird die zu ergänzende unvollständige Sequenz entsprechend der Anzahl möglicher Teilsequenzen dupliziert. Je nach Struktur des zugrundeliegenden Aktivitätsdiagramms kann es hierbei dazu kommen, dass Zyklen im Kontrollfluss erneut in die zu ergänzende Sequenz eingefügt werden, so dass Zyklen mehr als einmal durchlaufen werden. Hierdurch wird das $C_{2_{ACD}}$-Kriterium nicht verletzt.

In Abbildung 5.41 ist die Operation an dem in Abbildung 5.38a eingeführten Beispiel graphisch dargestellt. Im ersten Schritt (Abbildung 5.41a) wird Pfad A_2 am letzten vorkommenden Knoten b um eine ebenfalls mit dem Knoten b beginnende Teilsequenz aus Pfad A_1 ergänzt. Nach der Operation endet auch die Sequenz A_2 mit einer Markierung, die ein *UML::ActivityFinalNode*-Element im zugrundeliegenden Aktivitätsdiagramm repräsentiert. Im zweiten Schritt (Abbildung 5.41b) wird Markierungssequenz A_3 nach dem letzen Vorkommen des Knotens f um eine Teilsequenz aus den Markierungssequenzen A_1 und A_2 ergänzt. Diese sind allerdings identisch, deshalb erfolgt hier kein Duplizieren der Markierungssequenz A_3. Im dritten Schritt (Abbildung 5.41c) wird die Markierungssequenz A_4 nach dem letzten Vorkommen des Knotens c um Teilsequenzen aus bereits vervollständigten Sequenzen ergänzt. In diesem Fall wird Markierungssequenz A_4 dupliziert, weil in mehreren bereits vervollständigten Sequenzen mit c beginnende Teilsequenzen existieren.

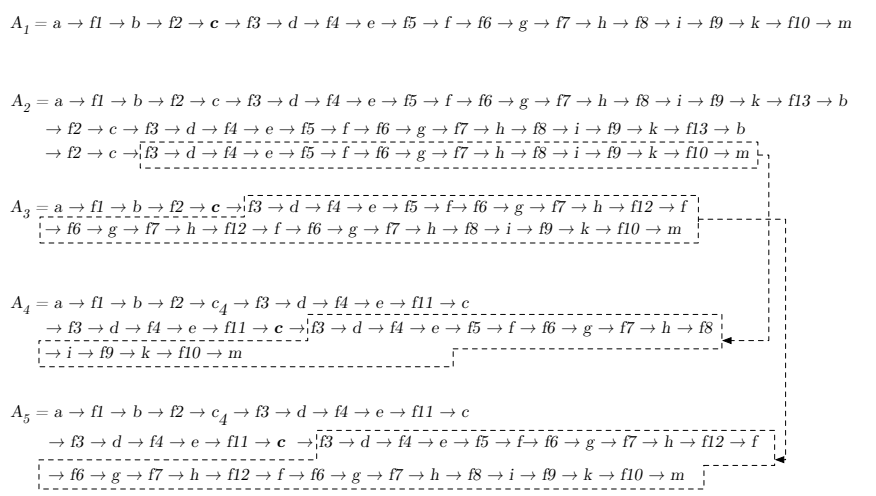
Die Vervollständigungsoperation stellt sicher, dass alle Markierungssequenzen, die zuvor durch Tiefensuche im Erreichbarkeitsgraphen des zu einem Aktivitätsdiagramm gehörenden B/E-Netzes erzeugt wurden, mit einer Markierung enden, die eine Marke auf einer Stelle des



(a) Ergänzung von Markierungssequenz A_2 um eine Teilsequenz aus A_1



(b) Ergänzung von Markierungssequenz A_3 um eine Teilsequenz aus A_2



(c) Ergänzung von Markierungssequenz A_4 um unterschiedliche Teilsequenzen aus A_2 und A_3

Abbildung 5.41: Ergänzung unvollständiger Markierungssequenzen am Beispiel des in Abbildung 5.38a eingeführten Aktivitätsdiagramms zur Erfüllung des $C_{2_{ACD}}$-Kriteriums.

B/E-Netzes trägt, die mit einem *UML::ActivityFinalNode*-Element des Aktivitätsdiagramms assoziiert ist. Die Operation stellt dabei sicher, dass sowohl das $C_{0_{ACD}}$-Überdeckungskriterium (Aktionsüberdeckung) als auch das $C_{2c_{ACD}}$-Kriterium (Basispfadüberdeckung) erfüllt ist.

Durch strukturelle Eigenschaften des zugrundeliegenden Aktivitätsdiagramms können hierbei über die Forderung des $C_{2c_{ACD}}$-Kriteriums Zyklen im Kontrollfluss des Aktivitätsdia-

gramm mehr als nur einmal durchlaufen werden, wodurch das $C_{2c_{ACD}}$-Kriterium allerdings nicht verletzt wird. Ursache hierfür ist die jeweilige Position von Verzweigungsknoten im Aktivitätsdiagramm durch welche Zyklen im Kontrollfluss modelliert sind.

Sequenzialisierung nebenläufiger Kontrollflüsse

UML-Aktivitätsdiagramme können neben zyklischen Kontrollflussstrukturen ebenfalls nebenläufige Kontrollflussstrukturen modellieren. Die Transformation von Aktivitätsdiagrammen zu B/E-Netzen und die darauf basierende Berechnung des Erreichbarkeitsgraphen bildet auch hier die Grundlage zur Sequenzialisierung des Kontrollflusses.

Nebenläufige Modellierung von Kontrollflussstrukturen in Aktivitätsdiagrammen soll abbilden, dass die auf nebenläufigen Kontrollflüssen liegenden Aktionen unabhängig voneinander und ggf. zeitgleich ausgeführt werden können. Im Kontext des Testens mobiler, kontextsensitiver Anwendungen ist das jedoch kein Szenario, das in der tatsächlichen Verwendung einer App durch den Anwender so durchgeführt wird. Ein menschlicher Akteur kann zu jedem Zeitpunkt mit genau einer Komponente des UI einer Anwendung interagieren. In der Konsequenz müssen Aktionen auf nebenläufigen Kontrollflüssen zum Zweck des automatisierten Testens in einer geeigneten Weise sequenzialisiert werden, so dass individuelle Aktionen nur in einer zulässigen Reihenfolge (vgl. Abschnitt 5.3.2) hintereinander ausgeführt werden.

Das in Abschnitt 5.3.2 diskutierte $C_{PS_{ACD}}$-Überdeckungskriterium (Permutationssequenzüberdeckung) fordert, dass zulässige Sequenzialisierungen von Permutationen testrelevanter Elementen auf nebenläufigen Kontrollflussstrukturen durch Testfälle abgedeckt werden.

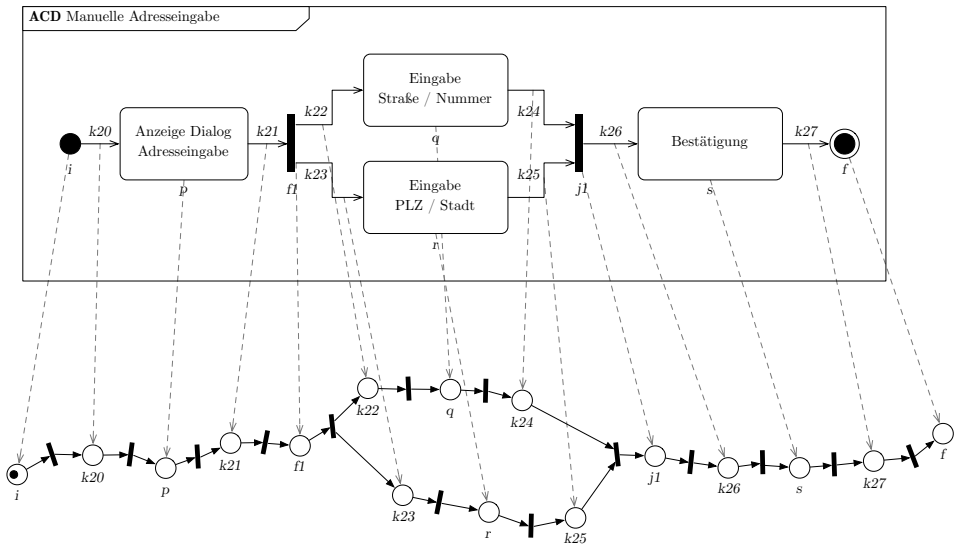


Abbildung 5.42: Beispiel der Transformation eines Aktivitätsdiagramms in ein B/E-Netz. Abgebildet ist das Teilmodell Manuelle Adresseingabe der App Mobiler Taxiruf mit dem durch die Transformation erzeugten B/E-Netz.

Die in Abschnitt 5.3.3.1.2 diskutierte Transformationsregel 3 (*UML::ActivityEdge* → B/E-Netz Transitionen) definiert für Elemente des Typs *UML::ForkNode* (*UML::JoinNode*), wie diese zu B/E-Netzen zu transformieren sind. In Abbildung 5.42 ist die Auswirkung dieser

Transformationsregel an der Transition erkennbar, die auf die Stelle *f1* folgt. Hier wurde nicht für jeden ausgehenden Kontrollfluss aus *UML::ForkNode f1* eine separate Transition erstellt, sondern die Kontrollflüsse *k22* und *k23* sind mit einer gemeinsamen Transition im B/E-Netz verbunden (analog *UML::JoinNode j1*). Die auf Stelle *f1* folgende Transition platziert jeweils eine Marke auf den Stellen *k22* und *k23*, wodurch die Parallelisierung des Kontrollflusses am *UML::ForkNode*-Element *f1* im Aktivitätsdiagramm in Abbildung 5.42 repräsentiert wird.

In Abbildung 5.42 ist ein Teilmodell der App Mobiler Taxiruf dargestellt. Es enthält nebenläufige Kontrollflussstrukturen, die durch die Transformation zu Testfällen zulässig sequenzialisiert werden müssen. Der zu dem in Abbildung 5.42 dargestellten Aktivitätsdiagramm gehörige Erreichbarkeitsgraph ist in Abbildung 5.43 dargestellt.

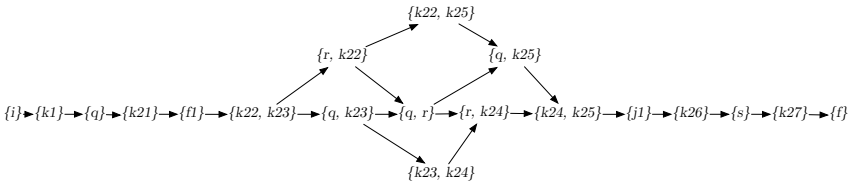


Abbildung 5.43: Erreichbarkeitsgraph des in Abbildung 5.42 dargestellten B/E-Netzes, App Mobiler Taxiruf, Teilmodell Manuelle Adresseingabe.

Eine auf diesen Erreichbarkeitsgraphen angewendete Tiefensuche erzeugt die in Abbildung 5.44 dargestellte Menge von Markierungssequenzen. Im Vergleich zu den im Vorangegangenen diskutierten zyklenbehafteten Erreichbarkeitsgraphen unterscheidet sich die Menge Markierungssequenzen in diesem Fall dadurch, dass einige Markierungen enthalten sind, bei denen Marken auf mehreren Stellen im zugehörigen B/E-Netz liegen.

$$B_1 = i \to k20 \to p \to k21 \to f1 \to k22, k23 \to q, k23 \to q, r \to r, k24 \to k24, k25 \to j1 \to k26 \to s \to k27 \to f$$
$$B_2 = i \to k20 \to p \to k21 \to f1 \to k22, k23 \to q, k23 \to q, r \to q, k25 \to k24, k25 \to j1 \to k26 \to s \to k27 \to f$$
$$B_3 = i \to k20 \to p \to k21 \to f1 \to k22, k23 \to q, k23 \to k23, k24 \to r, k24 \to k24, k25 \to j1 \to k26 \to s \to k27 \to f$$
$$B_4 = i \to k20 \to p \to k21 \to f1 \to k22, k23 \to r, k22 \to q, r \to r, k24 \to k24, k25 \to j1 \to k26 \to s \to k27 \to f$$
$$B_5 = i \to k20 \to p \to k21 \to f1 \to k22, k23 \to r, k22 \to q, r \to q, k25 \to k24, k25 \to j1 \to k26 \to s \to k27 \to f$$
$$B_6 = i \to k20 \to p \to k21 \to f1 \to k22, k23 \to r, k22 \to k22, k25 \to q, k25 \to k24, k25 \to j1 \to k26 \to s \to k27 \to f$$

Abbildung 5.44: Menge von Markierungssequenzen aus dem in Abbildung 5.43 dargestellten Erreichbarkeitsgraphen des Teilmodells Manuelle Adresseingabe der App Mobiler Taxiruf

Diese Markierungssequenzen repräsentieren Workflows durch das in Abbildung 5.42 dargestellte Aktivitätsdiagramm in unterschiedlichen Ausführungsreihenfolgen der Aktionen „Eingabe Straße/Nummer“ und „Eingabe PLZ/Stadt“. Das Modell drückt aus, dass der Anwender die Reihenfolge der Eingaben frei wählen kann. Eine Unterscheidung der Reihenfolge der Eingabe ist für die Durchführung von Tests relevant, weil in Abhängigkeit der Eingabe in ein Textfeld des UI die Eingabe in das jeweils andere plausibilisiert wird.

Obwohl das Modell nur zwei unterschiedliche Ausführungsreihenfolgen zulässt, stellt Abbildung 5.44 sechs unterschiedliche Markierungssequenzen dar. Das ist dem Umstand geschuldet, dass das in Abschnitt 5.2.4 diskutierte UML-Profil zur Testfallmodellierung Kontroll- und

Datenflüsse ebenfalls als testrelevante Artefakte betrachtet und einen Stereotyp bereitstellt, mit dem *UML::ActivityEdge*-Elemente mit Testdaten annotiert werden können. Im konkreten Fall könnten beispielsweise die *UML::ActivityEdge*-Elemente *k22* und *k23* mit Testdaten angereichert werden, etwa mit einer Instanz des Typs *Context::DeviceOrientationPrecondition*, um eine definierte Orientierung des Geräts bei der Testdurchführung zu erzwingen.

Für alle Markierungssequenzen gilt, dass die Sequenzialsierung der nebenläufig modellierten Aktionen gemäß der in Abschnitt 5.3.2 diskutierten Definition zulässig sind, d. h. die Aktionen, die auf einem Kontrollfluss in einer definierten Reihenfolge modelliert sind, werden durch die Sequenzialisierung in ihrer Reihenfolge nicht vertauscht (die mit dem Kontrollfluss *k23* assoziierten Testdaten werden niemals in das SUT eingespeist, nachdem die mit Kontrollfluss *k25* assoziierten Testdaten verarbeitet wurden).

Sequenz B_1 in Abbildung 5.45 repräsentiert beispielsweise eine Ausführungsreihenfolge, in der beginnend mit dem *UML::InitialNode*-Element zunächst die mit dem Kontrollfluss *k20* assoziierten Testdaten verarbeiten werden, dann die mit Aktion p assoziierten, dann die mit Kontrollfluss *k21* assoziierten, dann in beliebiger Reihenfolge *k22* und *k23* usw. Die Assoziation von Modellierungselementen mit Testdaten ist optional und wird durch die Anwendung der Stereotypen des UML-Profils zur Testfallmodellierung gesteuert.

In der vorliegenden Form bedürfen die Markierungssequenzen allerdings einer Aufbereitung vor der eigentlichen Interpretation als Ausführungsreihenfolge von Aktionen beim Testen. In Markierungssequenz B_1 befindet sich an Index fünf beispielsweise eine Markierung, die Marken auf den Stellen *k22* und *k23* platziert. In der unmittelbar folgenden Markierung an Index sechs wurde lediglich die Marke in der Stelle *k22* gelöscht und eine Marke auf Stelle p platziert. Die Marke auf Stelle *k23* ist erhalten geblieben (vgl. Abbildung 5.45). Dies würde übertragen auf die Semantik des UML-Aktivitätsdiagramms implizieren, dass sich entweder die mit der Stelle *k23* assoziierte testrelevante Aktion (in diesem Fall ein Kontrollfluss) weiterhin in der Ausführung befindet oder das diese Aktion zweimal hintereinander ausgeführt wird. Der zweite Fall ist aufgrund struktureller Eigenschaften der Transformation von Aktivitätsdiagramm zu B/E-Netz ausgeschlossen. Diese erzwingt, dass sich zwischen zwei Stellen, die Aktionen (Kontrollflüsse) repräsentieren, mindestens eine Stelle befindet, die einen Kontrollfluss (Aktion) repräsentiert. Hintergrund ist, dass im Syntax von UML-Aktivitätsdiagrammen zwei Elemente des Typs *UML::ActivityNode* nur dann aufeinander folgen, wenn sie mit einem *UML::ActivityEdge*-Element miteinander in Relation gesetzt werden, d. h. ein Aktivitätsdiagramm ist bezüglich *UML::ActivityNode*- und *UML::ActivityEdge*-Elementen bipartit. Der erste Fall ist nicht auf eine Testautomatisierungstechnologie abbildbar, da Interaktionen mit Anwendungen bzw. Manipulationen des Betriebskontext im Rahmen dieser Dissertation als atomar aufgefasst werden. Eine solche Sequenz von Markierungen repräsentiert daher eine unzulässige Aktionsfolge.

Um aus unzulässigen Aktionsfolgen zulässige Aktionsfolgen herzustellen, werden im Rahmen der Transformation von Aktivitätsdiagrammen zu Testfällen als Nachbereitung der Erzeugung von Markierungssequenzen aus dem Erreichbarkeitsgraphen eines B/E-Netzes Marken aus unmittelbar aufeinanderfolgenden Markierungen entfernt, so dass strukturelle Eigenschaften des Workflows durch das zugrundeliegende Aktivitätsdiagramm zwar erhalten bleiben, von individuellen Aktionen aber atomares Verhalten erzwungen wird.

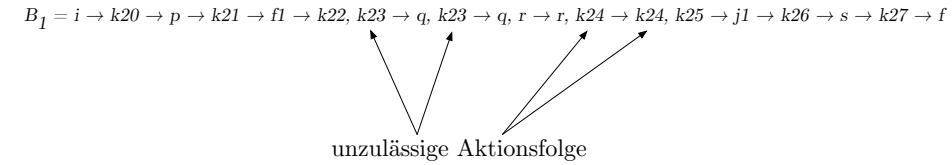


Abbildung 5.45: Beispiel für unzulässige Folgemarkierungen

$B_1 = i \rightarrow k20 \rightarrow p \rightarrow k21 \rightarrow f1 \rightarrow k22,\ k23 \rightarrow q, \sout{k23} \rightarrow \sout{q},\ r \rightarrow \sout{r},\ k24 \rightarrow \sout{k24},\ k25 \rightarrow j1 \rightarrow k26 \rightarrow s \rightarrow k27 \rightarrow f$
$B_2 = i \rightarrow k20 \rightarrow p \rightarrow k21 \rightarrow f1 \rightarrow k22,\ k23 \rightarrow q, \sout{k23} \rightarrow \sout{q},\ r \rightarrow \sout{q},\ k25 \rightarrow k24, \sout{k25} \rightarrow j1 \rightarrow k26 \rightarrow s \rightarrow k27 \rightarrow f$
$B_3 = i \rightarrow k20 \rightarrow p \rightarrow k21 \rightarrow f1 \rightarrow k22,\ k23 \rightarrow q, \sout{k23} \rightarrow \sout{k23},\ k24 \rightarrow r, \sout{k24} \rightarrow \sout{k24},\ k25 \rightarrow j1 \rightarrow k26 \rightarrow s \rightarrow k27 \rightarrow f$
$B_4 = i \rightarrow k20 \rightarrow p \rightarrow k21 \rightarrow f1 \rightarrow k22,\ k23 \rightarrow r, \sout{k22} \rightarrow q, \sout{r} \rightarrow \sout{r},\ k24 \rightarrow \sout{k24},\ k25 \rightarrow j1 \rightarrow k26 \rightarrow s \rightarrow k27 \rightarrow f$
$B_5 = i \rightarrow k20 \rightarrow p \rightarrow k21 \rightarrow f1 \rightarrow k22,\ k23 \rightarrow r, \sout{k22} \rightarrow q, \sout{r} \rightarrow \sout{q},\ k25 \rightarrow k24, \sout{k25} \rightarrow j1 \rightarrow k26 \rightarrow s \rightarrow k27 \rightarrow f$
$B_6 = i \rightarrow k20 \rightarrow p \rightarrow k21 \rightarrow f1 \rightarrow k22,\ k23 \rightarrow r, \sout{k22} \rightarrow \sout{k22},\ k25 \rightarrow q, \sout{k25} \rightarrow k24, \sout{k25} \rightarrow j1 \rightarrow k26 \rightarrow s \rightarrow k27 \rightarrow f$

(a) Identifikation unzulässiger Marken auf Stellen des B/E-Netzes zu Aktivitätsdiagramm in Abbildung 5.42

$B_1 = i \rightarrow k20 \rightarrow p \rightarrow k21 \rightarrow f1 \rightarrow k22,\ k23 \rightarrow q \rightarrow r \rightarrow k24 \rightarrow k25 \rightarrow j1 \rightarrow k26 \rightarrow s \rightarrow k27 \rightarrow f$
$B_2 = i \rightarrow k20 \rightarrow p \rightarrow k21 \rightarrow f1 \rightarrow k22,\ k23 \rightarrow q \rightarrow r \rightarrow k25 \rightarrow k24 \rightarrow j1 \rightarrow k26 \rightarrow s \rightarrow k27 \rightarrow f$
$B_3 = i \rightarrow k20 \rightarrow p \rightarrow k21 \rightarrow f1 \rightarrow k22,\ k23 \rightarrow q \rightarrow k24 \rightarrow r \rightarrow k25 \rightarrow j1 \rightarrow k26 \rightarrow s \rightarrow k27 \rightarrow f$
$B_4 = i \rightarrow k20 \rightarrow p \rightarrow k21 \rightarrow f1 \rightarrow k22,\ k23 \rightarrow r \rightarrow q \rightarrow k24 \rightarrow k25 \rightarrow j1 \rightarrow k26 \rightarrow s \rightarrow k27 \rightarrow f$
$B_5 = i \rightarrow k20 \rightarrow p \rightarrow k21 \rightarrow f1 \rightarrow k22,\ k23 \rightarrow r \rightarrow q \rightarrow k25 \rightarrow k24 \rightarrow j1 \rightarrow k26 \rightarrow s \rightarrow k27 \rightarrow f$
$B_6 = i \rightarrow k20 \rightarrow p \rightarrow k21 \rightarrow f1 \rightarrow k22,\ k23 \rightarrow r \rightarrow k25 \rightarrow q \rightarrow k24 \rightarrow j1 \rightarrow k26 \rightarrow s \rightarrow k27 \rightarrow f$

(b) Entfernen unzulässiger Marken

$B_{1a} = i \rightarrow k20 \rightarrow p \rightarrow k21 \rightarrow f1 \rightarrow k22 \rightarrow k23 \rightarrow q \rightarrow r \rightarrow k24 \rightarrow k25 \rightarrow j1 \rightarrow k26 \rightarrow s \rightarrow k27 \rightarrow f$
$B_{1b} = i \rightarrow k20 \rightarrow p \rightarrow k21 \rightarrow f1 \rightarrow k23 \rightarrow k22 \rightarrow q \rightarrow r \rightarrow k24 \rightarrow k25 \rightarrow j1 \rightarrow k26 \rightarrow s \rightarrow k27 \rightarrow f$
$B_{2a} = i \rightarrow k20 \rightarrow p \rightarrow k21 \rightarrow f1 \rightarrow k22 \rightarrow k23 \rightarrow q \rightarrow r \rightarrow k25 \rightarrow k24 \rightarrow j1 \rightarrow k26 \rightarrow s \rightarrow k27 \rightarrow f$
$B_{2b} = i \rightarrow k20 \rightarrow p \rightarrow k21 \rightarrow f1 \rightarrow k23 \rightarrow k22 \rightarrow q \rightarrow r \rightarrow k25 \rightarrow k24 \rightarrow j1 \rightarrow k26 \rightarrow s \rightarrow k27 \rightarrow f$
$B_{3a} = i \rightarrow k20 \rightarrow p \rightarrow k21 \rightarrow f1 \rightarrow k22 \rightarrow k23 \rightarrow q \rightarrow k24 \rightarrow r \rightarrow k25 \rightarrow j1 \rightarrow k26 \rightarrow s \rightarrow k27 \rightarrow f$
$B_{3b} = i \rightarrow k20 \rightarrow p \rightarrow k21 \rightarrow f1 \rightarrow k23 \rightarrow k22 \rightarrow q \rightarrow k24 \rightarrow r \rightarrow k25 \rightarrow j1 \rightarrow k26 \rightarrow s \rightarrow k27 \rightarrow f$
$B_{4a} = i \rightarrow k20 \rightarrow p \rightarrow k21 \rightarrow f1 \rightarrow k22 \rightarrow k23 \rightarrow r \rightarrow q \rightarrow k24 \rightarrow k25 \rightarrow j1 \rightarrow k26 \rightarrow s \rightarrow k27 \rightarrow f$
$B_{4b} = i \rightarrow k20 \rightarrow p \rightarrow k21 \rightarrow f1 \rightarrow k23 \rightarrow k22 \rightarrow r \rightarrow q \rightarrow k24 \rightarrow k25 \rightarrow j1 \rightarrow k26 \rightarrow s \rightarrow k27 \rightarrow f$
$B_{5a} = i \rightarrow k20 \rightarrow p \rightarrow k21 \rightarrow f1 \rightarrow k22 \rightarrow k23 \rightarrow r \rightarrow q \rightarrow k25 \rightarrow k24 \rightarrow j1 \rightarrow k26 \rightarrow s \rightarrow k27 \rightarrow f$
$B_{5b} = i \rightarrow k20 \rightarrow p \rightarrow k21 \rightarrow f1 \rightarrow k23 \rightarrow k22 \rightarrow r \rightarrow q \rightarrow k25 \rightarrow k24 \rightarrow j1 \rightarrow k26 \rightarrow s \rightarrow k27 \rightarrow f$
$B_{6a} = i \rightarrow k20 \rightarrow p \rightarrow k21 \rightarrow f1 \rightarrow k22 \rightarrow k23 \rightarrow r \rightarrow k25 \rightarrow q \rightarrow k24 \rightarrow j1 \rightarrow k26 \rightarrow s \rightarrow k27 \rightarrow f$
$B_{6b} = i \rightarrow k20 \rightarrow p \rightarrow k21 \rightarrow f1 \rightarrow k23 \rightarrow k22 \rightarrow r \rightarrow k25 \rightarrow q \rightarrow k24 \rightarrow j1 \rightarrow k26 \rightarrow s \rightarrow k27 \rightarrow f$

(c) Bildung von Permutationen zulässiger Markierungen

Abbildung 5.46: Herstellung zulässiger Markierungssequenzen durch Entfernen unzulässiger Marken von Stellen des B/E-Netzes am Beispiel des in Abbildung 5.42 dargestellten Aktivitätsdiagramm

In Abbildung 5.46 ist die Bereinigung unzulässiger Markierungen abgebildet. Es existieren nach dem Entfernen von Mehrfachmarkierungen solche Markierungen, in denen legitim Marken auf mehreren Stellen im B/E-Netz liegen. Solche Markierungen in der Sequenz drücken eine beliebige Abfolge der assoziierten Elemente im Aktivitätsdiagramm aus. Dem $C_{PS_{ACD}}$-Überdeckungskriterium wird genügt, wenn beliebige Permutationen aus der Menge herausgegriffen werden, z. B. B_{na} oder B_{nb}. Die Verwendung aller generierten Sequenzen für die weitere Erzeugung von Testfälle kann zu einem State-Explosion-Problem führen.

Hierarchische Einbettungen von Aktivitäten

Das UML-Profil zur Testfallmodellierung (Abschnitt 5.2.4) definiert den Stereotyp *TestActivity*. Bei der Generierung von Testfällen aus UML-Modellen werden nur aus solchen Aktivitätsdiagrammen direkt Testfälle erzeugt, auf welche dieser Stereotyp angewendet wurde.

Da allerdings Aktivitätsdiagramme durch Verwendung des Elements *UML::CallBehaviorAction* andere Aktivitätsdiagramme referenzieren können, in denen Kontext- und Testdaten modelliert wurden, ist es notwendig, bei der Generierung von Testfällen in die Hierarchie modellierter Aktivitätsdiagramme hinabzusteigen und testrelevante Sequenzen von Aktionen (erzeugt aus der Transformation zu B/E-Netzen und anschließender Analyse deren Erreichbarkeitsgraphen) auch für eingebettete Aktivitäten zu erzeugen.

Zur Generierung von Testfällen werden die im Vorangegangenen diskutierten Schritte zunächst auf alle im UML-Modell spezifizierten Aktivitätsdiagramme angewendet, so dass vor der Analyse der hierarchischen Zusammenhänge zwischen mehreren Aktivitätsdiagrammen für einzelne Aktivitätsdiagramme bereits aufbereitete Markierungssequenzen für jedes Aktivitätsdiagramm vorliegen.

Für die Generierung von Testfällen aus Aktivitätsdiagrammen der obersten Hierarchieebene müssen die in Aktivitätsdiagrammen auf untergeordneten Hierarchieebenen modellierten Workflows berücksichtigt werden. Um das zu gewährleisten, werden alle Markierungssequenzen, die in den vorhergehenden Schritten aus dem Aktivitätsdiagramm der höchsten Hierarchieebene erzeugt wurden, auf Markierungen des zugehörigen B/E-Netzes durchsucht, die ein *UML::CallBehaviorAction*-Element repräsentieren. Jedes Vorkommen einer solchen Markierung wird anschließend durch Ausführungspfade desjenigen Aktivitätsdiagramms ersetzt, welches durch das *UML::CallBehaviorAction*-Element referenziert wird.

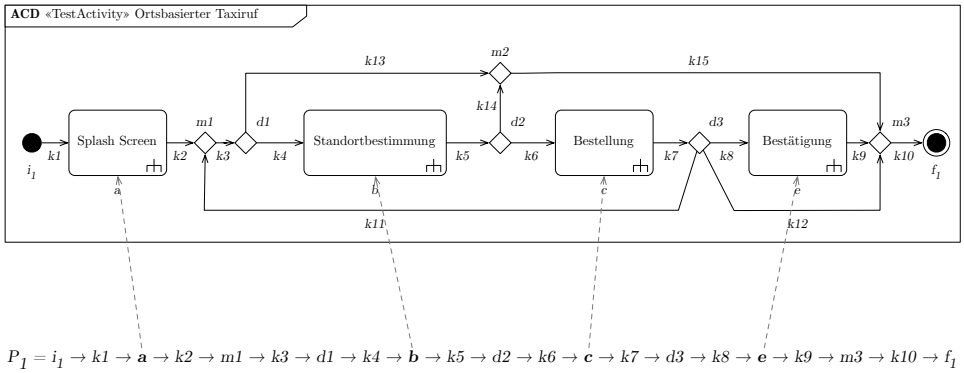


Abbildung 5.47: Identifikation von B/E-Netzmarkierungen, die *UML::CallBehaviorAction*-Elemente repräsentieren

Die in Abbildung 5.47 dargestellte Markierungssequenz P_1 wurde aus dem zum abgebildeten Aktivitätsdiagramm gehörigen B/E-Netz berechnet. Die Markierungen a, b, c und e platzieren jeweils eine Marke auf denjenigen Stellen des B/E-Netzes, die mit den *UML::CallBehaviorAction*-Elementen „Splash Screen“, „Standortbestimmung“, „Bestellung“ und „Bestätigung“ assoziiert sind. Bei der Generierung vollständiger Ausführungspfade für das übergeordnete Aktivitätsdiagramm werden Vorkommen von *UML::CallBehaviorAction*-Elementen durch die für das referenzierte Aktivitätsdiagramm generierten Pfade ersetzt.

Der Prozess ist in Abbildung 5.48 am Beispiel des Modells der App Mobiler Taxiruf exemplarisch dargestellt. Der in Abbildung 5.48a dargestellte Pfad P_{1a} wurde erzeugt, indem der aus dem übergeordneten Aktivitätsdiagramm Ortsbasierter Taxiruf generierte Pfad P_1 auf Vorkommen von *UML::CallBehaviorAction*-Elementen untersucht wurde und diese durch die Pfade ersetzt wurden, die aus den referenzierten Aktivitätsdiagrammen generiert wurden.

$P_{1a} =$

$i \rightarrow k1 \rightarrow$

a $i_6 \rightarrow f61 \rightarrow splash \rightarrow f62 \rightarrow f_6 \rightarrow$

$k2 \rightarrow m1 \rightarrow k3 \rightarrow d1 \rightarrow k4 \rightarrow$

b $i_5 \rightarrow data \rightarrow b1 \rightarrow letzter\ standort \rightarrow b2 \rightarrow b3 \rightarrow gültigkeit\ standort \rightarrow b4 \rightarrow$
$kartenansicht \rightarrow b5 \rightarrow d1 \rightarrow b6 \rightarrow d2 \rightarrow b7 \rightarrow d3 \rightarrow b8 \rightarrow m1 \rightarrow b9 \rightarrow f_5 \rightarrow$

$k5 \rightarrow d2 \rightarrow k6 \rightarrow$

c $i_7 \rightarrow f71 \rightarrow bestellung \rightarrow f72 \rightarrow f_7 \rightarrow$

$k7 \rightarrow d3 \rightarrow k8 \rightarrow$

e $i_8 \rightarrow f81 \rightarrow bestätigung \rightarrow f82 \rightarrow f_8 \rightarrow$

$k9 \rightarrow m3 \rightarrow k10 \rightarrow f$

(a) In dem in Abbildung 5.47 dargestellten Ausführungspfad wurden die Aktionen a, b, c und e jeweils durch die Ausführungspfade der referenzierten Aktivitätsdiagramme ersetzt. Aktion b referenziert ein Aktivitätsdiagramm, zu welchem mehrere Ausführungspfade erzeugt wurden. Eines hiervon wurde zur Erzeugung des Ausführungspfads P_{1a} ausgewählt.

$P_{1b} =$

$i \rightarrow k1 \rightarrow$

a $i_6 \rightarrow f61 \rightarrow splash \rightarrow f62 \rightarrow f_6 \rightarrow$

$k2 \rightarrow m1 \rightarrow k3 \rightarrow d1 \rightarrow k4 \rightarrow$

b $i_5 \rightarrow data \rightarrow b1 \rightarrow letzter\ standort \rightarrow b2 \rightarrow b3 \rightarrow gültigkeit\ standort \rightarrow b4 \rightarrow$
$kartenansicht \rightarrow b5 \rightarrow d1 \rightarrow b14 \rightarrow m2 \rightarrow b11 \rightarrow$ **manuelle adresseingabe** $\rightarrow b12 \rightarrow m1 \rightarrow b9 \rightarrow f_5 \rightarrow$

$k5 \rightarrow d2 \rightarrow k6 \rightarrow$

c $i_7 \rightarrow f71 \rightarrow bestellung \rightarrow f72 \rightarrow f_7 \rightarrow$

$k7 \rightarrow d3 \rightarrow k8 \rightarrow$

e $i_8 \rightarrow f81 \rightarrow bestätigung \rightarrow f82 \rightarrow f_8 \rightarrow$

$k9 \rightarrow m3 \rightarrow k10 \rightarrow f$

(b) Im Unterschied zu Abbildung 5.48a wurde hier für die Erzeugung von P_{1b} ein anderer Pfad aus dem in b referenzierten Aktivitätsdiagramm ausgewählt. Es enthält selbst eine Referenz zu dem Aktivitätsdiagramm „Manuelle Adresseingabe“.

Abbildung 5.48: Beispiel des Einfügens von Markierungssequenzen untergeordneter Aktivitätsdiagramme in die Markierungssequenz eines übergeordneten Aktivitätsdiagramms

Der Transformationsalgorithmus kann zu den aus einem Aktivitätsdiagramm erzeugten B/E-Netz mehrere Markierungssequenzen erzeugen (z. B. nebenläufige Kontrollflussstrukturen, Zyklen). Die übergeordnete Sequenz wird dann entsprechend der Anzahl der untergeordneten Sequenzen dupliziert. Da eine einzubettende Sequenz ebenfalls Markierungen enthalten kann, die *UML::CallBehaviorAction*-Elemente referenzieren, wird der Ersetzungsprozess rekursiv fortgeführt, bis die aus dem Aktivitätsdiagramm auf der obersten Ebene erzeugten Sequenzen keine Markierungen mehr enthalten, die *UML::CallBehaviorAction*-Elemente referenzieren. Dieser Fall ist in Abbildung 5.48b angedeutet. Die Ersetzungsoperation hat in P_{1b} eine Aktionssequenz erzeugt, die selbst eine Referenz zum Aktivitätsdiagramm Manuelle Adresseingabe enthält. Deshalb wird erneutes Einfügen von Aktionssequenzen aus dem referenzierten Aktivitätsdiagramm anstelle des *UML::CallBehaviorAction*-Elements notwendig.

Reduktion auf testrelevante Elemente

Für Aktivitätsdiagramme, die durch Anwendung des Stereotyps *TestActivity* aus dem in Abschnitt 5.2.4 entworfenen UML-Profil als testrelevant gekennzeichnet wurden, enthält die Menge der durch die vorhergehenden Schritte Ausführungspfaden alle Pfade, die durch Referenzierung untergeordneter Aktivitätsdiagramm generiert wurden. Ebenfalls ist gewährleistet, dass solche Pfade existieren, die alle vorkommenden Zyklen mindestens einmal durchlaufen und nebenläufige Kontrollflussstrukturen zulässig sequenzialisiert wurden.

In diesen Ausführungspfaden sind zunächst alle *UML::ActivityNode*- und *UML::ActivityEdge*-Elemente enthalten, insbesondere auch solche Elemente, die für einen Tester auf der Ebene von Akzeptanztests nicht erreichbar sind. Akzeptanztests adressieren jedoch nur solche Elemente in Aktivitätsdiagrammen, die eine Interaktion des Anwenders mit der App zulassen. Die übrigen in einem Aktivitätsdiagramm modellierten Elemente werden nur indirekt getestet.

Das in Abschnitt 5.2.4 diskutierte UML-Profil erlaubt die Kennzeichnung solcher Modellierungselement in Aktivitätsdiagrammen, die beim Durchführen von Tests explizit adressiert werden sollen. Deshalb werden zum Abschluss der Transformation von Aktivitätsdiagrammen zu Ausführungspfaden alle *UML::ActivityNode*- und *UML::ActivityEdge*-Elemente aus den bisher generierten Ausführungspfaden gelöscht, die nicht durch Anwendung einer der Stereotypen des UML-Profils als testrelevant gekennzeichnet wurden.

Die in den bisherigen Schritten generierten Ausführungspfade repräsentieren *UML::ActivityNode*- und *UML::ActivityEdge*-Elemente in ihrer Ausführungsreihenfolge zur Realisierung von Testfällen. Um irrelevante Elemente zu löschen, werden die Ausführungspfade vom ersten bis zum letzten Element traviersiert und solche Elemente entfernt, auf die keiner der Stereotypen *TestActivity* oder *TestFlow* (vgl. Abschnitt 5.2.4) angewendet wurde.

Es kann hierbei der Fall eintreten, dass Instanzen des Typs *UML::DecisionNode* aufgefunden werden. Charakteristisch für Verzweigungsknoten ist, dass sie alternative Kontrollflussverläufe modellieren. Sowohl einem menschlichen Tester als auch einer Testautomatisierungstechnologie erschließt sich nicht intuitiv, welche Kriterien zur Auswahl einer der verfügbaren Alternativen führen. Insbesondere im Umfeld mobiler, kontextsensitiver Anwendungen können alternative Kontrollflussverläufe abseits von Interaktionen des Anwenders durch Kontextparameter gesteuert werden. Die Transformation von Aktivitätsdiagrammen zu Testfällen erfordert deshalb, dass alle testrelevanten Kontrollflussverläufe durch Anwendung des Stereotyps auf *UML::ActivityEdge*-Elemente gekennzeichnet werden, die ausgehende Kanten aus *UML::DecisionNode*-Elementen modellieren. Wird in einem Ausführungspfad folgend auf ein *UML::DecisionNode*-Element ein *UML::ActivityEdge*-Element, auf das nicht der Stereotyp *TestFlow* angewendet wurde, dann wird der gesamte Pfad gelöscht. Eine Automatisierungstechnologie könnte diesen Pfad nicht ausführen, da unbekannt ist, unter welchen Bedingungen einer Verzweigungsalternative gefolgt wird.

Nach der Ausführung dieses letzten Schritts der Transformation von Aktivitätsdiagrammen zu Ausführungspfaden erfolgt die eigentliche Transformation von Aktivitätsdiagrammen zu Testfallmodellen wie in Abschnitt 5.3.3.2 beschrieben.

5.3.3.2 Abbildung von UML-Elementen auf Testfallmodellelemente

Die hier untersuchte Methode der modellbasierten Generierung von Tests für mobile, kontextsensitive Anwendungen verfolgt den Ansatz, vor der Generierung technologiespezifischer Tests zunächst ein plattformunabhängiges Testfallmodell zu erzeugen. Dieses bildet dann die Grundlage für die Generierung technologiespezifischer Tests.

Ausführungspfade durch Aktivitätsdiagramme bilden die eigentlichen Testfälle ab. Sie sind eine geordnete Menge individueller Aktionen (*UML::ActivityNode*- und *UML::ActivityEdge*-Elemente), die einen Workflow durch das SUT abbilden. Um von UML-Modellen zu Testfallmodellen zu gelangen, gilt es relevante Elemente aus UML-Modellen zu Instanzen von Typen des in Abschnitt 5.2.3.2 diskutierten Metamodells zur Testfallmodellierung zu transformieren. Die prinzipielle Funktionsweise der Modelltransformation wurde Abschnitt 5.3.1 erläutert. In diesem Abschnitt erfolgt nun die Diskussion konkreter Transformationsregeln, durch welche die Abbildung von UML-Elementen auf Testfallmodellelemente beschrieben wird.

In der technischen Implementierung sind UML-Aktivitätsdiagramme in einem übergeordneten UML-Modell eingebettet. Dieses dient i. d. R. nicht nur als Container für Aktivitätsdiagramme, sondern kann darüber hinaus auch weitere UML-Modelle und zugehörige Diagramme, wie z. B. Klassendiagramme und Sequenzdiagramme, enthalten. Insbesondere ist dieses übergeordnete UML-Modell ebenfalls Container für ein Objektmodell, in welchem die im Rahmen der Annotation eines UML-Aktivitätsdiagramms mit Kontextparametern und mit Testdaten durch Verwendung des Metamodells zur Kontextmodellierung (vgl. Abschnitt 5.2.2.2) und des UML-Profils zur Integration der Testfallmodellierung in die Systemmodellierung (vgl. Abschnitt 5.2.4) spezifizierten Entitäten enthalten sind.

Transformationsregel UML-zu-Testfallmodell 1: Element *UML::Model*

> *UML::Model*-Elemente werden durch die Transformation auf Instanzen des Typs *Test::TestModel* des Metamodells zur Testfallmodellierung (vgl. Abschnitt 5.2.3.2.2) abgebildet. Die im UML-Modell enthaltenen Metainformationen werden hierbei auf die durch die Superklasse *Test::TestEntity* definierten Attribute übertragen. Alle im UML-Modell enthaltenen Informationen werden dieser *Test::TestModel*-Instanz zugeordnet.
>
> Durch die Transformation gehen keine für die Testdurchführung relevanten Informationen verloren. Allerdings erzeugt die Transformation nicht zwingend für alle im UML-Model enthaltenen Elemente 1:1-Abbildungen. Insbesondere für individuelle Aktionen innerhalb von Aktivitätsdiagrammen und durch die Stereotypen *TestStep* referenzierten Objekte werden ggf. 1:n-Abbildungen erzeugt, da Aktionen in Aktivitätsdiagrammen zu mehreren *Test::Step*-Instanzen transformiert werden können, die wiederum in mehreren *Test::TestCase*-Instanzen vorkommen können.

Transformationsregel UML-zu-Testfallmodell 2: Element *UML::Activity*

> Die Transformation bildet *UML::Activity*-Elemente auf *Test::TestSuite*-Elemente ab. Metainformationen zur *UML::Activity*, wie z. B. Bezeichnung oder Autor werden dabei auf die durch die Superklasse *Test::TestEntity* bereitgestellten Attribute

abgebildet. Alle in der *UML::Activity* enthaltenen *UML::Action*-Elemente werden bei der Transformation genau dieser *Test::TestSuite* zugeordnet.

Die Modelltransformation überprüft jedes *UML::Activity*-Element auf die Anwendung des Stereotyps *TestActivity*. Bei positiver Auswertung wird das *UML::Activity*-Element durch die Transformation verarbeitet. Anderenfalls wird für dieses Element kein *Test::TestSuite*-Element erzeugt.

Transformationsregel UML-zu-Testfallmodell 3: Element *UML::ActivityNode*

Die UML definiert Knotentypen für Aktivitätsdiagramme, die sich in Knoten zur Kontrollfluss-/Objektflusssteuerung (*UML::ControlNode*-Elemente) und solche Knoten, durch welche Aktionen repräsentiert werden, einteilen lassen.

Für Knoten, die Kontroll- und Objektflüsse durch Aktivitätsdiagramme steuern, gilt Transformationsregel 5. Für Knoten, durch welche Aktionen repräsentiert werden gilt, Transformationsregel 4.

Transformationsregel UML-zu-Testfallmodell 4: Element *UML::ActionNode*

Knotentypen zur Modellierung von Aktionen werden durch Elemente des Typs *UML::Action* generalisiert. Hierzu gehören beispielsweise die Typen *UML::OpaqueAction*, *UML::CallBehaviorAction* oder *UML::CallOperationAction*. Obwohl die UML-Superstructure [268] für diese Knotentypen jeweils individuelles Verhalten definiert, verhalten sie sich aus der Perspektive des Testens gleichartig. Sie spezifizieren eine funktionale Komponente eines Softwaresystems. Für die Durchführung funktionaler Testfälle sind Details der technischen Implementierung nicht relevant. Relevant ist hingegen, ob diese Aktionen direkt durch den Anwender erreichbar sind. Das ist z. B. dann der Fall, wenn sie eine Schnittstelle zur Benutzungsoberfläche besitzen. Im hier betrachteten Fall sind diese Knotentypen ebenfalls relevant, wenn sie Kontextparameter verarbeiten.

Sofern auf Knoten des Typs *UML::ActionNode* der Stereotyp *TestStep* angewendet wurde, werden sie durch die Transformation auf Instanzen von *Test::Step* abgebildet (vgl. Abschnitt 5.2.3.2). Testdaten und Kontextinformationen, die durch die Anwendung des Stereotyp *TestStep* modelliert sind, werden hierbei auf die Attribute „action“, „precondition“ und „postcondition“ übertragen. Sie dienen bei der Durchführung von Tests zur Parametrisierung der Automatisierungstechnologie.

Transformationsregel UML-zu-Testfallmodell 5: Element *UML::ControlNode*

Das UML-Metamodell definiert für Aktivitätsdiagramme Knoten zur Kontrollflusssteuerung. Hierunter fallen neben solchen Knoten, die Anfang und Ende eines Kontrollflusses modellieren ebenfalls Knoten, die Kontrollflussstrukturen parallelisieren oder synchronisieren oder alternative Kontrollflussverläufe modellieren.

Das Metamodell zur Testfallmodellierung definiert mit Ausnahme der Elemente *TestModel::InitialNode* und *TestModel::FinalNode* keine Knoten zur Kontrollflusssteuerung. Ebenfalls lässt es das UML-Profil zur Testfallmodellierung nicht zu, auf Elemente des Typs *UML::ControlNode* (z. B. *UML::InitialNode*, *UML::ActivityFinalNode*, *UML::DecisionNode*, *UML::ForkNode*) einen der definierten Stereotypen anzuwenden.

Elemente das Typs *UML::ControlNode* werden zwar bei der Generierung des zum Aktivitätsdiagramm gehörigen B/E-Netzes verarbeitet (vgl. Abschnitt 5.3.3.1.2), werden darüber hinaus bei der Erstellung von Testfallmodellen aber nicht weiter berücksichtigt. Eine Ausnahme ist in Transformationsregel 6 definiert.

Transformationsregel UML-zu-Testfallmodell 6: UML::InitialNode und FinalNode

Das UML-Metamodell (vgl. OMG [268]) definiert das Element *UML::InitialNode* als Startknoten des Kontrollflusses einer *UML::Activity*. Es zeichnet sich dadurch aus, dass es keine eingehenden Kontrollflüsse hat. Eine *UML::Activity* kann mehrere *UML::InitialNode*-Elemente spezifizieren. Für diesen Fall werden beim Ausführen der *UML::Activity* mehrere nebenläufige Kontrollflüsse gestartet. Weiterhin definiert die UML-Superstructure (OMG [268]), dass implizit Kontrollflüsse bei allen *UML::ActionNode*-Elementen starten.

Das Element UML::*ActivityFinalNode* beendet alle Kontrollflüsse einer *UML::Activity*. Insbesondere heißt das, dass nach dem Erreichen eines *UML::ActivityFinalNode*-Elements die Ausführung aller *UML::Action*-Elemente abgebrochen wird. *UML::Activity*-Elemente können mehrere *UML::ActivityFinalNode*-Elemente spezifizieren, die Ausführung ist mit dem Erreichen des ersten *UML::ActivityFinalNode*-Elements beendet.

Die Transformation von Aktivitätsdiagrammen zu Ausführungspfaden entfernt im letzten Schritt alle Elemente aus dem Ausführungpfad, auf die keiner der Stereotypen *TestStep* oder *TestFlow* angewendet wurde. Ausnahme hiervon sind das erste Element bzw. das letzte Element, sofern es vom Typ *UML::InitialNode* bzw. *UML::ActivityFinalNode* ist.

Die Transformation von UML-Modellen zu Testfallmodellen erzeugt für das erste (letzte) Element in einem Ausführungspfad ein *Test::InitialNode*-Element (*Test::FinalNode*-Element), wenn dieses Element vom Typ *UML::InitialNode* bzw. *UML::ActivityFinalNode* ist. Anderenfalls werden *Test::InitialNode* bzw. *Test::FinalNode*-Elemente erzeugt und dem ersten Element vorangestellt bzw. nach dem letzten Element eingefügt.

Transformationsregel UML-zu-Testfallmodell 7: Element *UML::ActivityEdge*

Bei der Transformation von Aktivitätsdiagrammen zu Ausführungspfaden werden Elemente vom Typ *UML::ActivityEdge* ebenfalls zu Stellen im B/E-Netz transformiert und anschließend im Ausführungspfad wie Aktionen behandelt. *UML::ActivityEdge*-Elemente, auf die der Stereotyp *TestFlow* angewendet wurde, werden

analog zu *UML::ActionNode*-Elementen (vgl. Transformationsregel 4) zu *Test::Step*-Elementen im Testfallmodell transformiert.

Transformationsregel UML-zu-Testfallmodell 8: Ausführungspfade

In Abschnitt 5.3.3.1.2 sowie Abschnitt 5.3.3.1.3 wurde diskutiert, wie aus Aktivitätsdiagrammen Ausführungspfade generiert werden.

Bei der Transformation zu Testfallmodellen wird für jeden aus einem Aktivitätsdiagramm berechneten Ausführungspfad ein *Test::TestCase*-Element erzeugt. Diesem werden die aus den individuellen Elementen des Pfads generierten *Test::Step*-Elemente zugeordnet.

In Abbildung 5.49 ist das Schema der Modelltransformation von UML-Aktivitätsdiagramm zu Testfallmodell dargestellt. Das Resultat der Transformation bildet die Grundlage zur Erzeugung technologiespezifischer Testfälle, welche in Abschnitt 5.4 am Beispiel der Testautomatisierungstechnologie Calabash diskutiert wird.

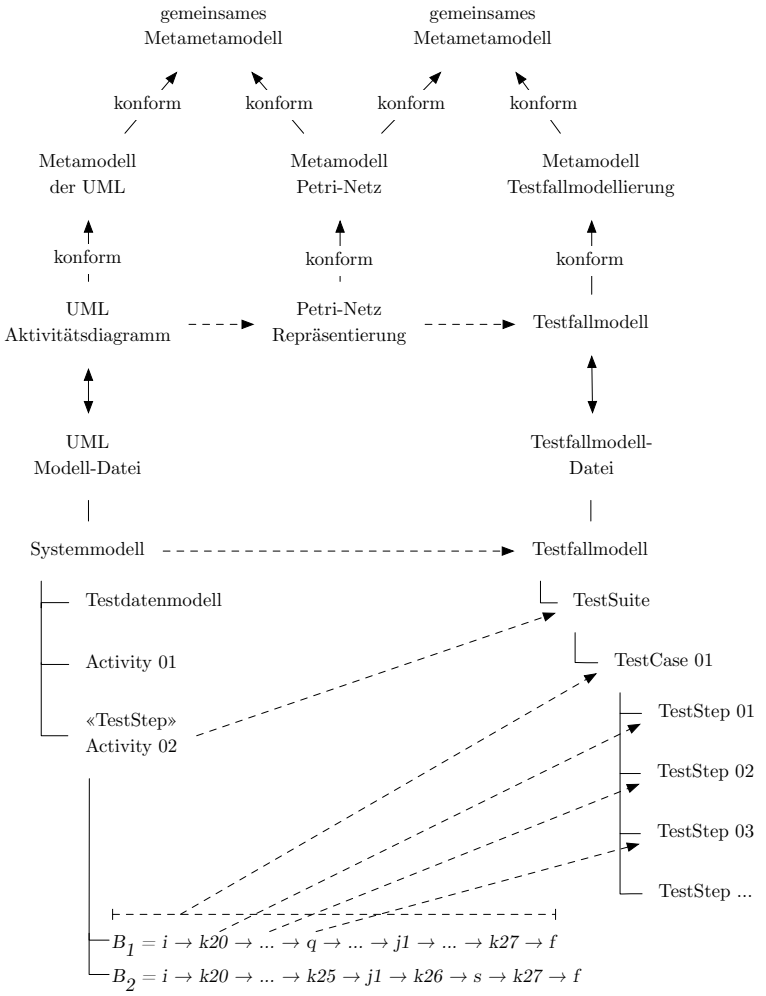


Abbildung 5.49: Schema der Transformation von Aktivitätsdiagramm zu Testfallmodell

5.3.4 Zusammenfassung

In den vorangegangen Abschnitten 5.3.1, 5.3.2 sowie 5.3.3 wurde die Transformation von UML-Aktivitätsdiagrammen zu Testfallmodellen diskutiert. Hierzu wurde zunächst in Abschnitt 5.3.1 Modelltransformation als Werkzeug modellgetriebener Methoden der Softwaretechnologie im Allgemeinen und ihre Anwendung zur Transformation von UML-Aktivitätsdiagrammen zu Testfallmodellen diskutiert.

Weiterhin wurden in Abschnitt 5.3.2 Testabdeckungskriterien und ihre Anwendung auf die Generierung von Testfällen aus Aktivitätsdiagrammen diskutiert. Insbesondere wurde das $C_{PS_{ACD}}$-Überdeckungskriterium (Permutationssequenzüberdeckung) definiert, welches von den generierten Testfällen fordert, das SUT so zu überdecken, dass alle testrelevanten Elemente des Aktivitätsdiagramms durch mindestens einen Testfall adressiert werden, alle Zyklen im Kontrollfluss des modellierten Prozesses mindestens einmal durchlaufen werden und dass testrelevante Elemente auf nebenläufigen Kontrollflussstrukturen zulässig sequenzialisiert werden, so dass sie ohne Verletzung ihrer ursprünglichen Reihenfolge durch eine Testautomatisierungstechnolgie sequenziell, d. h. hintereinander, ausgeführt werden.

Der eigentliche Transformationsprozess von UML-Aktivitätsdiagramm zu Testfallmodell wurde in Abschnitt 5.3.3 im Detail diskutiert. Aktivitätsdiagramme werden hierzu zunächst zu B/E-Netzen transformiert. Auf der Basis der Erreichbarkeitsgraphen dieser B/E-Netze werden testrelevante Ausführungspfade durch die modellierten Aktivitäten berechnet. Hierbei werden sowohl Zyklen, nebenläufige Kontrollflüsse als auch hierarchisch modellierte Aktivitäten berücksichtigt. Weiterhin wurden Transformationsregeln definiert, durch welche die Abbildung von Instanzen individueller UML-Modellelemente in Aktivitätsdiagrammen auf Elemente von Testfallmodellen erfolgt.

5.4 Generierung technologiespezifischer Testfälle am Beispiel der Automatisierungstechnologie Calabash für Android

Im vorangegangenen Abschnitt 5.3 wurde diskutiert, wie durch Verwendung von Modelltransformation aus Aktivitätsdiagrammen, auf welche das in Abschnitt 5.2.4 definierte UML-Profil zur Integration von Kontext- und Testdaten in die Systemmodellierung angewendet wurde, plattform- und technologieunabhängige Testfallmodelle generiert werden können. Dieses Modell enthält alle zur Durchführung von Testfällen notwendigen Informationen in Form von Vorbedingungen für einzelne Testschritte, Parameter zur Ausführung individueller Testschritte und die zur Beurteilung der Ergebnisse notwendigen Nachbedingungen.

Der Durchführung von Tests für eine konkrete Anwendung liegt neben der Testfallbeschreibung, hier gegeben durch das plattformunabhängige Testfallmodell, eine technologiespezifische Implementierung dieser Anwendung zugrunde, z. B. für die Plattform Android. Mit der Einführung einer Technologiespezialisierung der zu testenden Anwendung geht zugleich die Spezialisierung der zu verwendenden Testwerkzeuge einher. Für eine manuelle Testdurchführung ist eine Beschreibung von Tests nicht technologiegebunden, da ein menschlicher Akteur spezifisches Plattformwissen während der Testdurchführung ad-hoc ergänzen kann. Für eine automatisierte Durchführung von Tests ist der Technologiebezug hingegen stark

ausgeprägt, da spezifische Automatisierungswerkzeuge i. d. R. eine spezifische Zieltechnologie bedienen. Sofern Technologie zur Testautomatisierung existiert, ist diese regelmäßig eng an die Zieltechnologie gebunden und kann nicht auf andere Zielplattformen übertragen werden.

Um ein plattformunabhängiges Testmodell, wie es nach Anwendung des in Abschnitt 5.3 vorgestellten Verfahrens vorliegt, auf eine spezifische Zieltechnologie anzuwenden, ist es daher notwendig, das plattformunabhängige Testfallmodell in plattformspezifische Testfälle zu überführen, d. h. in die von der Technologie verwendete Repräsentierung zu transformieren.

In den folgenden Abschnitten wird am Beispiel der Testautomatisierungstechnologie Calabash diskutiert, wie ein plattformunabhängiges Testfallmodell durch Anwendung einer M2T-Transformation in technologiespezifische Tests überführt wird, die im Anschluss unter Verwendung der in Abschnitt 5.5 diskutierten Werkzeugimplementierungen automatisiert ausgeführt werden können. Das Schema der Transformation ist in Abbildung 5.50 dargestellt.

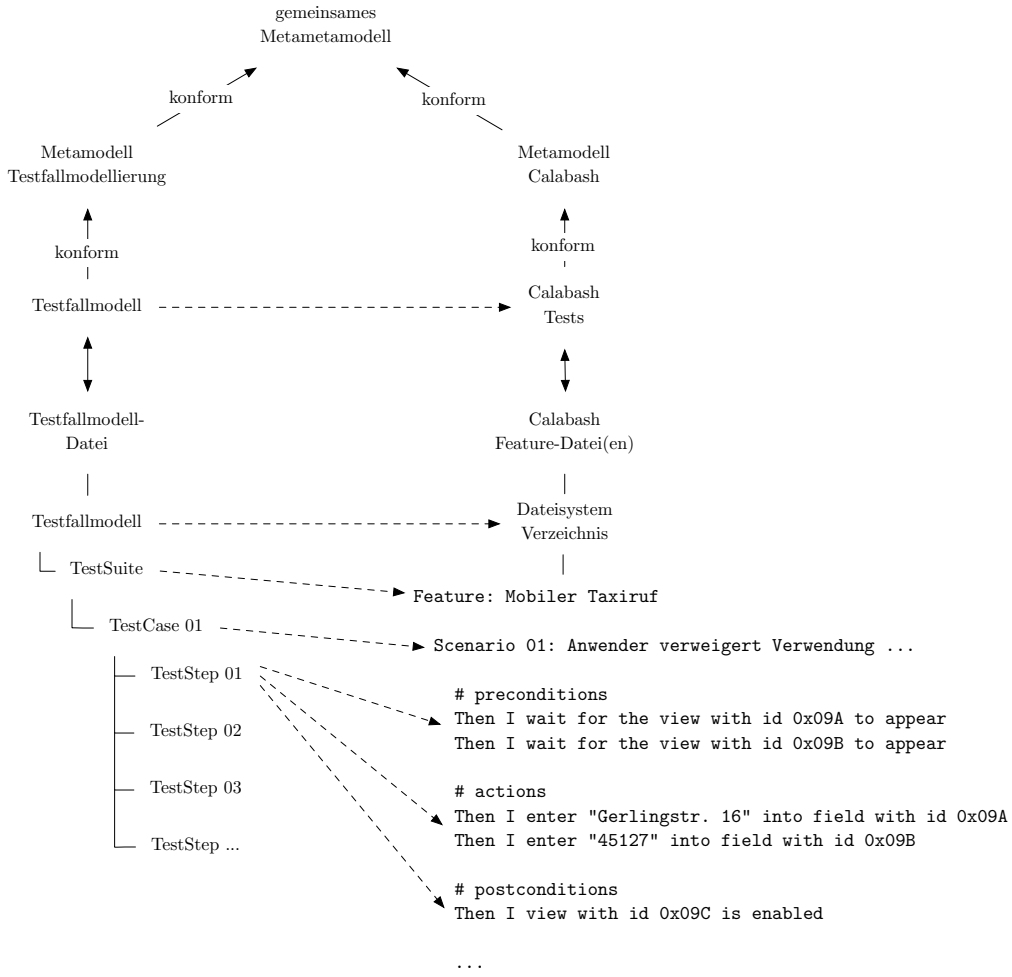


Abbildung 5.50: Schema der Transformation von Testfallmodell zu Calabash-Test-Code

Die M2T-Transformation analysiert das im vorherigen Schritt aus dem Aktivitätsdiagramm generierte Testfallmodell und identifiziert Elemente des Typs *Test::TestSuite*. Für jedes dieser Elemente wird eine Calabash-Feature-Datei (vgl. Abschnitt 2.5.2.6) im Dateisystem angelegt. Gemäß der Definition des Metamodells zur Testfallmodellierung (vgl. Ab-

schnitt 5.2.3.2) kann ein Testfallmodell mehrere *Test::TestSuite*-Elemente enthalten. Diese repräsentieren jeweils die als testrelevant gekennzeichneten Aktivitätsdiagramme des Systemmodells. Während der Transformation von Aktivitätsdiagramm zu Testfallmodell wurden durch das in Abschnitt 5.3.3 diskutierte Verfahren eine Menge von Ausführungspfaden erzeugt. Diese werden im Testfallmodell als *Test::TestCase*-Elemente abgebildet und bei der Transformation zu Calabash-Tests in Calabash-Scenarios überführt. *Test::TestCase*-Elemente ihrerseits sind aus *Test::Step*-Elementen aufgebaut, die beginnend mit einem *Test::InitialNode*-Element bis zum *Test::FinalNode*-Element durch *Test::Edge*-Elemente strukturiert werden. Die Transformation identifiziert das *Test::InitialNode*-Element innerhalb des *Test::TestCase*-Elements und traversiert das Modell bis zum Erreichen des *Test::FinalNode*-Elements. Jedes *Test::Step*-Element wird auf das Vorhandensein von *Test::Precondition*, *Test::Action* und *Test::Postcondition*-Elementen überprüft. Sofern vorhanden wird für jedes dieser Elemente ein Calabash-Testschritt erzeugt, der sich in einer individuellen Calabash-Code-Zeile manifestiert.

Bei der Transformation werden Komponenten (d. h. *Test::Precondition*, *Test::Action* und *Test::Postcondition*) aus den *Test::Step*-Elementen extrahiert und anhand ihres durch das Metamodell zur Testfallmodellierung bzw. durch das Metamodell zur Kontextmodellierung definierten Typs identifiziert und korrespondierend auf Calabash-Testschritte abgebildet.

Die Standardimplementierung des Calabash-Frameworks bietet mit Ausnahme eines Testschritts zur eingeschränkten Simulation des Standorts keine Testschritte zur Simulation von Kontextparametern an. Die Anpassungen des Calabash-Frameworks zur Herstellung der Fähigkeit zur Simulation weiterer Kontextparameter wird in Abschnitt 5.5.3 im Detail erläutert.

Nach der Anwendung der M2T-Transformation zur Erzeugung von Calabash-Tests aus Testfallmodellen liegen im Dateisystem eine Anzahl von Calabash-Feature-Dateien vor, die durch die in Abschnitt 5.5.3 diskutierte Implementierung des Calabash-Frameworks in Zusammenwirken mit der in Abschnitt 5.5.2 vorgestellten angepassten Implementierung des Android-Betriebssystems verwendet werden können, um mobile, kontextsensitive Apps sowohl in einer emulierten Umgebung als auch auf mobilen Geräten automatisiert zu testen.

5.5 Automatisierte Durchführung von Testfällen für kontextsensitive mobile Anwendungen

In Abschnitt 1.2.4 wurde die Forschungshypothese aufgestellt, dass das Testen mobiler, kontextsensitiver Anwendungen besonders dann effektiv und effizient durchgeführt werden kann, wenn Kontextinformationen durchgängig vom Systementwurf bis zur Testdurchführung berücksichtigt werden und der Testprozess durch geeignete Konzepte, Methoden und Werkzeuge automatisiert durchführbar ist

In den im vorangegangenen diskutierten Abschnitten wurden durch die Bereitstellung von Metamodellen zur Kontext- und Testfallmodellierung (Abschnitt 5.2.2 und Abschnitt 5.2.3) sowie durch ein UML-Profil zur Testfallmodellierung (Abschnitt 5.2.4) die Voraussetzungen geschaffen, Kontextinformationen und Testdaten durchgängig beim Systementwurf und bei der Modellierung zu berücksichtigen. In Abschnitt 5.3 wurde dargelegt, wie durch Modelltransformation aus UML-Aktivitätsdiagrammen plattformunabhängige Testfallmodelle generiert werden können. Zum Nachweis der Forschungshypothese ist nun noch zu zeigen, dass

Tests, die durch diese Technologie aus Systemmodellen generiert wurden, durch eine Testautomatisierungstechnologie verarbeitet werden können.

Abgrenzungsmerkmal des automatisierten Testens mobiler, kontextsensitiver Anwendungen gegenüber traditionellen Anwendungen ist, dass beim Testen neben fachlichen Testdaten ebenfalls Kontextparameter (vgl. Abschnitt 3.1.2) als zusätzliche Testdaten bereitgestellt werden müssen. Die im Vorangegangenen diskutierte Erweiterung der UML ermöglicht die Abbildung dieser Daten im Testfallmodell. Zur ihrer tatsächlichen Verwendung bei der Testdurchführung muss ein Automatisierungswerkzeug in der Lage sein, Kontextparameter in das SUT einzuspeisen. Unterschiedliche Optionen zur Simulation von Kontext in Testfällen werden in Abschnitt 5.5.1 im Detail thematisiert. In Anlehnung an das Paradigma *Designed for Testability* [281] stützt sich die hier untersuchte Methode zur Testautomatisierung zur Kontextsimulation auf eine Modifikation des Android-Betriebssystems. Diese wird in Abschnitt 5.5.2 detailliert diskutiert.

Zur Testautomatisierung kommen grundsätzlich unterschiedliche Technologien wie beispielsweise JUnit, UIAutomator oder Calabash in Frage. In dem in dieser Dissertation untersuchten Ansatz zur Testautomatisierung stehen Akzeptanztests im Fokus. Diese zielen auf fachliche Funktionalität des SUT ab, nicht auf Details der technischen Realisierung. Während bei JUnit für Android, UIAutomator und verwandten Technologien (vgl. Abschnitt 2.5) keine klare Abgrenzung zwischen Unit-Tests und Akzeptanztests existiert, ist die Technologie Calabash klar auf die Durchführung von Akzeptanztests unter Abstraktion technischer Details der Implementierung ausgerichtet. Deshalb wurde Calabash als Zieltechnologie der hier untersuchten Methode zur Testautomatisierung ausgewählt. Die grundsätzliche Funktionsweise des Testautomatisierungs-Frameworks wurde bereits in Abschnitt 2.5.2.6 erläutert. Zur Unterstützung des Testens mobiler, kontextsensitiver Anwendungen sind allerdings Anpassungen am Calabash-Framework erforderlich, weil Funktionen zur Simulation von Kontextparametern in der Standardimplementierung nicht existieren.

5.5.1 Simulation von Kontextparametern bei der Testdurchführung

Aufgrund der Vielfalt von Kontextparametern (vgl. Abschnitt 3.1.2) erschöpft sich das Testen kontextsensitiver Anwendungen nicht in der Simulation physikalischer Kontextparameter. Insbesondere Kontext auf hohen Abstraktionsebenen wird i. d. R. durch Aggregation mehrerer Information aus unterschiedlichen Quellen berechnet. Hierzu zählen auch Informationen, die aus anderen Quellen als den Sensoren mobiler Geräte gewonnen werden. Diese bei der Durchführung von Tests zu berücksichtigen, erfordert neben der lokalen Bereitstellung von Testdaten im SUT ebenfalls die Manipulation externer Kontextquellen. Um das zu realisieren, wird eine Testumgebung benötigt, die nicht nur lokale Eigenschaften des SUT simulieren kann, sondern zugleich auch Eigenschaften und Verhalten externer Systeme. Aufgrund der hohen Komplexität der Simulation von Kontextinformationen über Systemgrenzen hinweg ist der hier untersuchte Ansatz auf die Simulation physikalischer Kontextparameter beschränkt, da bereits diese Aufgabe durch existierende Methoden und Technologien nicht adäquat und bislang ohne Integration in Automatisierungswerkzeuge adressiert wird (vgl. Abschnitt 4.2).

Zur Simulation von Kontext bei der Durchführung von Tests kommen vom Grundsatz her mehrere Ansätze in Frage. Die erste Option zur Simulation von Kontextparametern ist eine

Simulation basierend auf der Bereitstellung einer Laborumgebung, die in der Lage ist, physikalische Kontextparameter (vgl. Abschnitt 3.1.2.1) herzustellen und geeignet auf physikalische mobile Geräte einzuwirken. Es ist jedoch davon auszugehen, dass dieser Ansatz insgesamt unwirtschaftlich und für einige Kontextparameter technisch unmöglich ist (z. B. aufgrund von räumlichen Restriktionen bei Bewegung eines mobilen Geräts bei hohen Geschwindigkeiten, etwa Simulation einer Bahnfahrt). Darüber hinaus wäre dieser Ansatz für Tests mit einem Geräteemulator nicht verfügbar, da dieser nicht über physikalische Sensoren verfügt.

Die zweite Option ist die automatisierte Steuerung von Werkzeugen des SDKs der jeweiligen Plattform (vgl. Abschnitt 4.2.1.2), beispielsweise durch Integration von Schnittstellen zu SDK-Werkzeugen (z. B. Telnet-Verbindung zum Emulator) in eine Testautomatisierungstechnologie. Diese Option ist allerdings Tests vorbehalten, die auf Emulatoren ausgeführt werden, da physikalische Geräte nicht über diese Schnittstellen verfügen. Aufgrund der heterogenen Landschaft mobiler Geräte (vgl. Abschnitt 4.2.1) sind Tests jedoch nur dann wirksam, wenn sie auf einer repräsentativen Menge von Geräten ausgeführt werden. Deshalb ist diese Option insgesamt nicht zur Automatisierung von Tests im größeren Maßstab geeignet.

Die dritte Option ist die Simulation von Kontextparametern direkt innerhalb des SUT durch Ersetzen derjenigen Anwendungskomponenten innerhalb einer Anwendung, die durch Verwendung von Systemkomponenten auf Sensoren oder das GPS-Modul zugreifen. Diesen Weg beschreitet beispielsweise das Produkt SensorSimulator™ des Herstellers OpenIntents [269]. Es bietet eine Programmierbibliothek an, die anstelle der Programmierbibliothek des Android-SDK zum Zugriff auf die Sensoren eines mobilen Geräts verwendet werden kann. Vor- und Nachteile dieser Lösung werden in Abschnitt 5.5.1.1 im Detail diskutiert.

Die vierte Option ist die Simulation von Kontextparametern auf Ebene des Betriebssystems. Diese Option hat den Vorteil, dass das SUT zum Testen nicht modifiziert werden muss, sondern die Spezifikationskonformität einer App durch Black-Box-Tests überprüft werden kann. Diese Option wird in Abschnitt 5.5.1.2 im Detail betrachtet.

5.5.1.1 Testschnittstelle in der zu testenden Anwendung

Die Erfassung physikalischer Kontextparameter in Anwendungen für mobile Geräte erfolgt über Sensoren, die je nach Sensortyp und Bauart auf unterschiedliche Weise physikalische Parameter ihrer Umwelt messen. Unterschieden werden aus der Sicht der Systemarchitektur zwei verschiedene Arten von Sensoren: solche, die direkt als Hardware-Baustein auf der Platine des mobilen Geräts vorhanden sind und solche, die in Software realisiert sind. Sensoren der zweiten Kategorie basieren i. d. R. auf physikalischen Sensoren, bereiten deren Signale oder Messwerte jedoch für einen spezifischen Anwendungszweck auf, um die Verwendung des Sensors für den Softwareentwickler zu vereinfachen. Als Beispiel kann hier der Schrittsensor angeführt werden, der dem Softwareentwickler im API des SDK mobiler Plattformen als eigenständiger Sensor präsentiert wird, tatsächlich aber eine in Software realisierte Abstraktion des Beschleunigungssensors ist.

Die SDKs mobiler Plattformen präsentieren dem Entwickler alle Sensoren homogen, i. d. R. durch ein Subscriber-Entwurfsmuster, bei dem der Entwickler eine Callback-Methode registriert, die aus tiefer liegenden Systemschichten heraus aufgerufen werden, wenn sich Sensorwerte innerhalb bestimmter Rahmenbedingungen verändern. In Normalfall steht es dem

Entwickler frei zu bestimmen, mit welcher Häufigkeit die Callback-Methode aufgerufen wird, etwa in von SDK spezifizierten Konstanten, die ihrerseits Zeitintervalle repräsentieren oder wenn eine signifikante Änderung der Messwerte eintritt. Einige Sensoren werden abweichend hiervon in einem vom System bestimmten Intervall aktualisiert, wie etwa der Schrittzähler auf der Plattform Android.

Quellcodefragment 5.2: Verarbeitung von Sensordaten in Android

```
1  package de.paluno.accelerometer;
2
3  import android.app.Activity;
4  import android.content.Context;
5  import android.hardware.Sensor;
6  import android.hardware.SensorEvent;
7  import android.hardware.SensorEventListener;
8  import android.hardware.SensorManager;
9  import android.os.Bundle;
10
11 public class MainActivity extends Activity implements ↙
       ↳ SensorEventListener {
12
13     private SensorManager mSensorManager;
14     private Sensor mAccelerometer;
15
16     @Override
17     protected void onCreate(Bundle savedInstanceState) {
18         super.onCreate(savedInstanceState);
19         setContentView(R.layout.activity_main);
20         mSensorManager = (SensorManager) getSystemService(Context.↙
               ↳ SENSOR_SERVICE);
21         mAccelerometer = mSensorManager.getDefaultSensor(Sensor.↙
               ↳ TYPE_ACCELEROMETER);
22     }
23
24     @Override
25     protected void onResume() {
26         super.onResume();
27         mSensorManager.registerListener(this, mAccelerometer, ↙
               ↳ SensorManager.SENSOR_DELAY_NORMAL);
28     }
29
30     @Override
31     protected void onPause() {
32         mSensorManager.unregisterListener(this);
33         super.onPause();
34     }
35
36     @Override
37     public void onSensorChanged(SensorEvent event) {
38         // TODO handle sensor data
39     }
40
41     @Override
42     public void onAccuracyChanged(Sensor sensor, int accuracy) {
43
44     }
45
46 }
```

In Quellcodefragment 5.2 ist exemplarisch der Zugriff auf den Beschleunigungssensor in einer Android-App dargestellt. Abgebildet ist in Zeile 27 die Registrierung einer Callback-Methode, die in Zeile 37 zyklisch vom System aufgerufen wird. Die Häufigkeit des Aufrufs kann vom Entwickler als systemdefinierte Konstante in groben Einteilungen vorgeben werden. Der tatsächliche Zeitabstand aufeinanderfolgender Aufrufe ist allerdings von mehreren Faktoren abhängig. Einerseits obliegt es dem Gerätehersteller, die Aktualisierungsfrequenz für Sensoren individuell festzulegen. Das heißt, die Konstante *SENSOR_DELAY_NORMAL* (vgl. Zeile 27 in Quellcodefragment 5.2) kann je nach konkretem Android-Gerät unterschiedliche Werte

annehmen[6]. Auf mobilen Plattformen können mehrere Apps gleichzeitig auf einen Sensor zugreifen und hierbei jeweils unterschiedliche Aktualisierungsintervalle spezifizieren. In diesem Fall versorgt das System alle Apps, die diesen Sensor verwenden mit der höchsten von einer der Apps angeforderten Aktualisierungsfrequenz. Das heißt, dass andere Apps, die diesen Sensor verwenden, in kürzeren Intervallen aktualisiert werden, als vom System angefordert.

Interessant für die Simulation von Kontextparametern bei der Testdurchführung sind die vom SDK der Plattform bereitgestellten Klassen zur Implementierung des Zugriffs auf Sensoren (vgl. Quellcodefragment 5.2, Zeilen 5-8). Hier ergibt sich ein potenzieller Ansatzpunkt, Sensordaten während des Testens durch reproduzierbare Testdaten zu substituieren.

Beispielsweise kann ein Anbieter einer Simulationstechnologie eine Programmierbibliothek erstellen, die das Verhalten und die Struktur der Systembibliotheken bis hin zu Klassennamen imitieren, der verwendenden Anwendung aber nicht die tatsächlich durch die Sensoren des Geräts erzeugten Daten liefern, sondern Daten aus alternativen Quellen. Als alternative Quelle kommt beispielsweise eine Netzwerkschnittstelle in Frage, über welche Sensordaten als Testdaten von einer externen Quelle in das SUT eingespeist werden. Eine schematische Abbildung dieser Simulationstechnologie ist in Abbildung 5.51 dargestellt.

[6]In der im Rahmen dieser Dissertation betreuten Masterarbeit von Qui Don Ho [288] wurden zur Performanzanalyse von Sensoren auf der Plattform Android unterschiedliche Werte der SDK-Konstanten zur Spezifikation des Aktualisierungsintervalls in Abhängigkeit von Gerät und Sensor nachgewiesen.

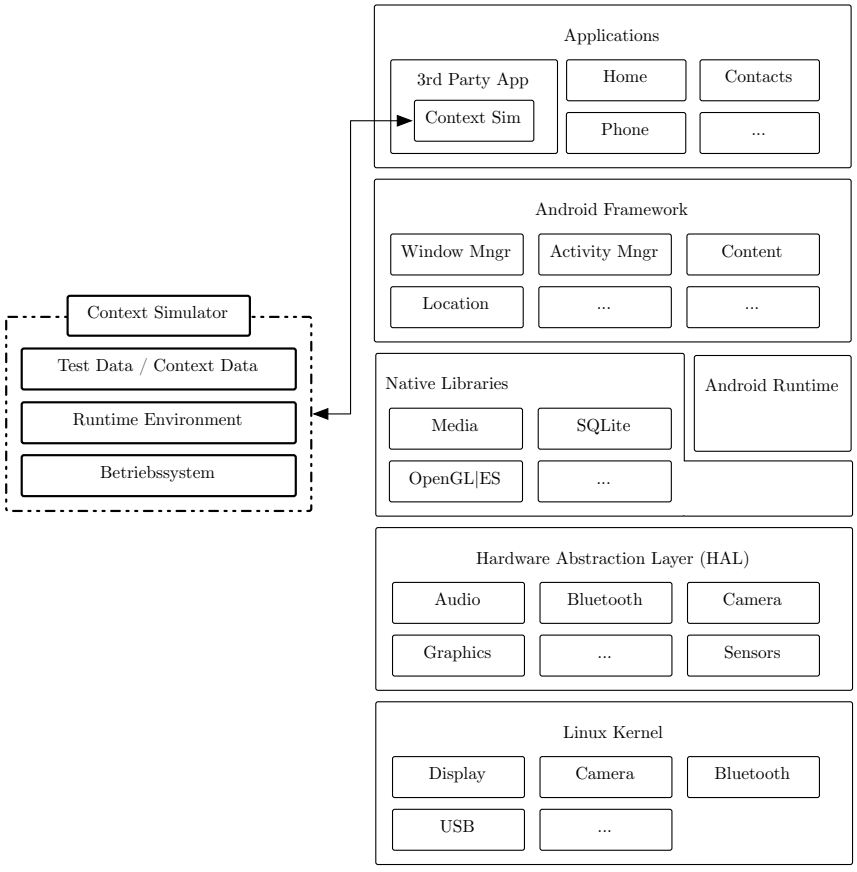


Abbildung 5.51: Schema der Simulation von Sensordaten innerhalb des SUT, Testschnittstelle innerhalb der App

Das Produkt SensorSimulator™ des Herstellers OpenIntents [269] implementiert eine Technologie zur Simulation von Sensordaten basierend auf diesem Schema. Es wird eine Programmierbibliothek bereitgestellt, durch welche die Funktion der Systembibliotheken zum Zugriff auf Sensoren imitiert wird. Die Verwendung ist in Quellcodefragment 5.3 abgebildet. Die Modifikation am Quellcode manifestiert sich zunächst nur im Import eines von den Systembibliotheken abweichenden Java-Pakets (vgl. Quellcodefragment 5.3, Zeilen 5-8), so dass dem Entwickler hier der Eindruck entsteht, nur eine geringfügige Änderung am Quellcode der App vorgenommen zu haben.

Quellcodefragment 5.3: Verarbeitung von Sensordaten in Android mit dem OpenIntents SensorSimulator™

```
1  package de.paluno.accelerometer;
2
3  import android.app.Activity;
4  import android.content.Context;
5  import org.openintents.sensorsimulator.hardware.Sensor;
6  import org.openintents.sensorsimulator.hardware.SensorEvent;
7  import org.openintents.sensorsimulator.hardware.SensorEventListener;
8  import org.openintents.sensorsimulator.hardware.↙
       ↳ SensorManagerSimulator;
9  import android.hardware.SensorManager;
10 import android.os.Bundle;
11
12 public class MainActivity extends Activity implements ↙
       ↳ SensorEventListener {
13
14     @Override
15     protected void onCreate(Bundle savedInstanceState) {
16         super.onCreate(savedInstanceState);
17         setContentView(R.layout.activity_main);
18         mSensorManager = (SensorManager) getSystemService(Context.↙
               ↳ SENSOR_SERVICE);
19         mSensorManager.connectSimulator();
20         mAccelerometer = mSensorManager.getDefaultSensor(Sensor.↙
               ↳ TYPE_ACCELEROMETER);
21     }
22
23     @Override
24     public void onSensorChanged(SensorEvent event) {
25         // TODO handle sensor data
26     }
27 }
```

Die Bibliothek implementiert Substitute für die durch die Systembibliotheken bereitgestellten Klassen zur Umsetzung des Subscriber-Entwurfsmusters zur Verarbeitung von Sensordaten. Um Testdaten in das SUT einzuspeisen, öffnet die OpenIntents-Bibliothek eine Netzwerkschnittstelle, an welcher sie Testdaten akzeptiert und in das SUT anstelle der tatsächlich durch die Sensoren des Geräts gemessen Daten an die verarbeitenden Klassen übergibt. Zur Bereitstellung simulierter Sensordaten stellt OpenIntents eine Desktop-Anwendung bereit (Abbildung 5.52). Mit dieser ist es dem Entwickler oder Tester möglich im Rahmen manueller Tests Sensordaten in das SUT einzuspeisen.

Ingesamt ist dieser Ansatz grundsätzlich technologisch sinnvoll und nützlich für manuelle Tests, hat aber einige Nachteile, die diesen Lösungsweg insgesamt vom Einsatz in Automatisierungslösungen ausschließen. Zunächst einmal ist die OpenIntents-Programmierbibliothek dazu ausgelegt, nach dem Testen in der Produktivversion der App zu verbleiben. Im Sinne des effektiven Testens ist dies sogar eine zwingende Anforderung, da eine Entfernung der Bibliothek aus der App und eine Rückkehr zu den Systembibliotheken eine Modifikation der zu testenden Anwendung ist, durch welche alle bisherigen Testergebnisse u. U. obsolet werden.

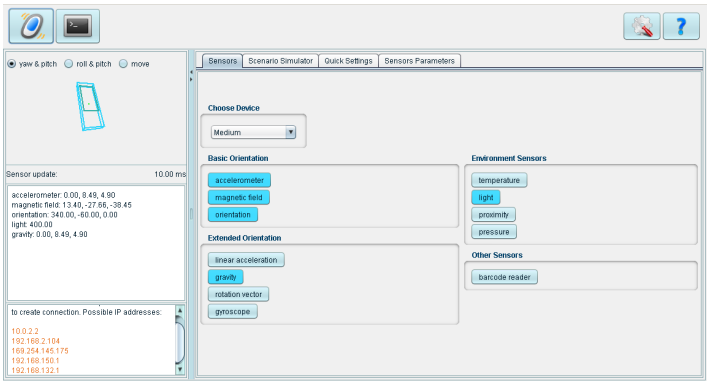


Abbildung 5.52: Desktop-App des Produkts SensorSimulator™ des Herstellers OpenIntents [269], [270]

Ein Verbleiben der OpenIntents-Programmierbibliothek in der App repräsentiert andererseits einen Angriffsvektor auf die App, da die Technologie eine Netzwerkverbindung ohne Zugriffsschutz öffnet. Hier besteht theoretisch die Möglichkeit, ohne Wissen des Anwenders auch im regelmäßigen Betrieb der App Daten in die Schnittstelle einzuspeisen und so einen auf Sensordaten basierenden Anwendungsfall zu kompromittieren.

Ein weiterer Nachteil dieser Technologie ist die Abhängigkeit von einem Produkt eines Drittanbieters. Es ist unbekannt, ob und mit welcher Zeitverzögerung mögliche Änderungen im SDK der Zielplattform in der OpenIntents-Programmierbibliothek nachgepflegt werden, so dass zusätzlich eine weitere Zeitverzögerung bei der Anpassung einer App an eine neuere Version des Betriebssystems der Android-Plattform entsteht.

Ebenfalls kann diese Technologie ausschließlich für White-Box-Tests angewendet werden, da nicht nur Kenntnisse des Quellcode erforderlich sind, sondern darüber hinaus Änderungen am Quellcode notwendig sind. Hierdurch wird die Nutzbarkeit dieses Ansatzes auf manuelle Tests während der Entwicklung beschränkt. Testen der App durch andere Stakeholder, beispielsweise im Rahmen von Akzeptanztests, sind hingegen nicht möglich.

Eine Änderung der OpenIntents-Programmierbibliothek durch den Hersteller erfordert zudem eine Anpassung aller Apps, die diese Bibliothek verwenden, wenn sie ihren Lebenszyklus noch nicht vollendet haben (d. h. Außerdienststellung). Selbst wenn es keine inhaltlichen Änderungen an Komponenten einer App gibt, die Sensoren verwenden, müssen diese angepasst werden, um die Funktionstüchtigkeit der Testschnittstellen zu erhalten.

Der für die hier untersuchte Methode der Testautomatisierung gravierendste Nachteil ist jedoch, dass die OpenIntents-Programmierbibliothek nicht über Schnittstellen zu Testautomatisierungstechnologien verfügt. Das Produkt ist dediziert auf die Verwendung der zugehörigen Desktop-Anwendung (Abbildung 5.52) ausgelegt und kann nur durch Anpassungen in Technologien wie JUnit, UIAutomator oder Calabash integriert werden. Es werden zur Realisierung einer technischen Implementierung der hier untersuchten Methode zur Testautomatisierung Änderungen am Calabash-Framework zur Bereitstellung von Funktionen zur Integration von Kontextinformationen in Tests notwendig. Zur Simulation von Sensordaten ist zusätzlich eine Technologie zur Einspeisung von Sensordaten als Testdaten in das SUT eine zwingende Voraussetzung. Es bestünde hier die Möglichkeit, die OpenIntents-Programmierbibliothek um Schnittstellen zum Testautomatisierungswerkzeug Calabash zu erweitern. Der Realisierungs-

aufwand ist allerdings hoch und die Bereitstellung einer solchen Schnittstelle ist abseits der Integrationsfähigkeit in Calabash nicht geeignet, die oben genannten Nachteile einer Simulation von Sensordaten innerhalb des SUT auszugleichen. Deshalb wird stattdessen der in Abschnitt 5.5.1.2 vorgestellte Lösungsansatz verfolgt, die Schnittstellen zur Simulation von Sensordaten direkt in die Zielplattform zu integrieren.

5.5.1.2 Testschnittstelle auf Betriebssystemebene

Gegenüber der Einspeisung von Sensordaten als reproduzierbare Testdaten direkt in das SUT besteht ebenfalls die Möglichkeit, Testdaten direkt in die Zielplattform einzuspeisen. Hierdurch entsteht einerseits Aufwand zur Anpassung der Zielplattform, im konkreten Beispiel der Plattform Android, andererseits ergeben sich aber eine Reihe von Vorteilen gegenüber der in Abschnitt 5.5.1.1 diskutierten Simulation von Sensordaten direkt im SUT.

Die in dieser Dissertation vorgestellte Methode zur Testautomatisierung mobiler, kontextsensitiver Anwendungen verfolgt den Ansatz, Kontextinformationen, insbesondere Sensor- und Standortinformationen, nicht direkt in das SUT einzuspeisen, sondern stattdessen eine Schnittstelle zur Simulation von Kontextdaten in der Zielplattform zu implementieren. Am konkreten Beispiel der Plattform Android ist diese Lösung umsetzbar, weil es sich bei Android um eine OSS-Plattform handelt, die ohne besondere Zugangshürden auf individuelle Anforderungen angepasst werden kann. In der Regel machen Hersteller von mobilen Geräten hiervon Gebrauch, um Android auf die Anforderungen der Hardware spezifischer Geräte anzupassen. Das Android-OSS-Projekt [147] stellt hierzu den Quellcode der Plattform öffentlich bereit.

Das Verfahren der Einspeisung von Sensordaten ist schematisch in Abbildung 5.53 dargestellt. Es unterscheidet sich von der in Abschnitt 5.5.1.1 diskutierten Einspeisung von Kontextdaten in das SUT in erster Linie darin, dass die Schnittstelle zur Einspeisung nicht Teil des SUT, sondern Teil der Zielplattform ist. Hierdurch kann ein SUT mit künstlichen, reproduzierbaren Sensordaten getestet werden, ohne dass hierzu Modifikationen am SUT vorgenommen werden müssen. Das Verfahren ist somit eine Überführung des Paradigmas *Designed for Testability* [281] vom SUT auf die Zielplattform, mit dem unmittelbaren Vorteil, dass Black-Box-Test-Verfahren auf Apps angewendet werden können, also das für die Auslieferung bestimmte Artefakt einer Software getestet werden kann, ohne dass Kenntnisse von Quellcode notwendig sind oder dieser zum Testen angepasst werden muss.

Das in Abbildung 5.53 dargestellte Schema zur Einspeisung von Sensordaten in die Plattform basiert auf zwei Komponenten, die in ihrem Zusammenwirken das automatisierte Testen mobiler, kontextsensitiver Anwendungen ermöglichen. Die erste Komponente ist eine Anpassung am Android-Betriebssystem, die eine Netzwerkschnittstelle auf einer geeigneten Ebene der Android-Systemarchitektur implementiert. Diese ist ausreichend tief im System angesiedelt, dass zu testende Anwendungen nicht mehr in der Lage sind, zwischen künstlichen und originären Quellen von Sensordaten zu unterscheiden. Diese Eigenschaft ist wünschenswert um zu verhindern, dass ein verarbeitender Algorithmus in Kenntnis von Eigenschaften von Testdaten entwickelt wird. Diese Modifikation des Android-Betriebssystems wird in Abschnitt 5.5.2 im Detail erläutert. Die zweite Komponente ist ein angepasstes Calabash-Framework, das in der Lage ist, mit der neu geschaffenen Testschnittstelle im Android-Betriebssystem zu kommunizieren und die im Testfallmodell (vgl. Abschnitt 5.2.3) enthaltenen Kontextinformationen

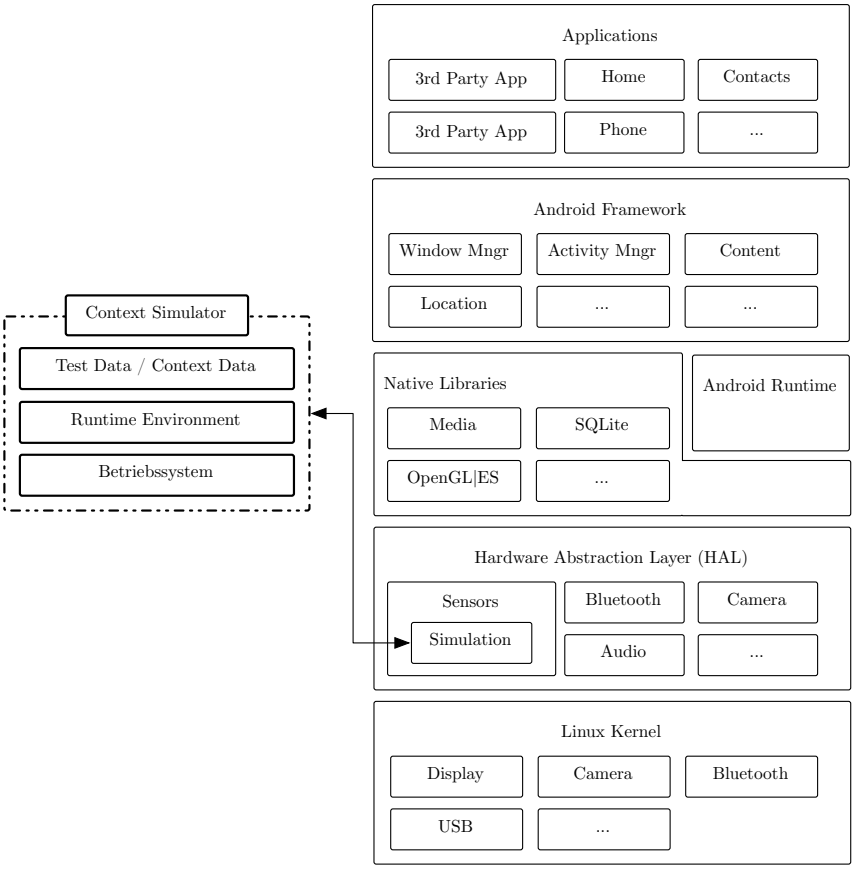


Abbildung 5.53: Schema der Simulation von Sensordaten, Testschnittstelle innerhalb der Plattform

in die Schnittstelle einzuspeisen. Die Modifikation des Calabash-Frameworks wird in Abschnitt 5.5.3 im Detail diskutiert.

Gegenüber der in Abschnitt 5.5.1.1 diskutierten Option (Integration Testschnittstelle in die zu testenden Anwendung) ergeben sich hierdurch eine Reihe von Vorteilen. Zunächst können Tests als Black-Box-Tests durchgeführt werden. Weder ist eine Modifikation des SUT notwendig, noch werden Kenntnisse des Quellcode[s] des SUT benötigt. Insbesondere für Softwareprozessmodelle, in denen bestimmte Tests nicht direkt durch Entwickler vorgenommen werden, eröffnen sich hier Möglichkeiten Teile des Testens durch Outsourcing an Drittanbieter zu vergeben, ohne diesen hierzu den Quellcode des Softwareprodukts offenlegen zu müssen.

Da durch das in Abbildung 5.53 dargestellte Verfahren keine testspezifischen Änderungen am SUT notwendig werden, ergibt sich der besonders signifikante Vorteil, dass das für die Auslieferung bestimmte Softwareartefakt getestet werden kann, ohne dass nachträglich Testschnittstellen aus dem SUT entfernt werden müssen. Somit ist einerseits gewährleistet, dass zwischen dem getesteten und dem ausgelieferten Artefakt keine Differenzen existieren, durch welche Testergebnisse kompromittiert würden und andererseits exponiert das SUT keine Angriffsvektoren durch im Produkt verbliebene Testschnittstellen.

Durch die Integration der Testschnittstelle in Komponenten des Betriebssystems wird zudem im Vergleich zur Integration in das SUT eine Unabhängigkeit von der verwendeten Technologie zur App-Entwicklung erreicht. Die Lokalisation der Testschnittstelle im Betriebs-

system eröffnet Möglichkeiten, simulierte Sensordaten als Testdaten nicht nur in solchen Anwendungen zu verwenden, die unter Verwendung des nativen SDK der Plattform entwickelt wurden, sondern steht darüber hinaus auch webbasierten Anwendungen oder Cross-Platform-Technologien zur Verfügung.

Weiterhin kann durch dieses Vorgehen eine Abhängigkeit von einer Programmierbibliothek eines Drittanbieters vermieden werden. Da ein Softwarehersteller i. d. R. mehr als ein Produkt betreut, kann der Anpassungsaufwand bei einer Evolution der Plattform durch eine Verlagerung der Testschnittstelle in die Plattform anstelle des SUT wesentlich herabgesetzt werden. Es muss dann nicht jede App, die diese Testtechnologie verwendet, einzeln angepasst werden, sondern der Anpassungsaufwand fällt nur einmal für das verwendete Plattformkompilat an.

Im Anpassungsaufwand an die Evolution der Plattform liegt auch der Nachteil der Integration der Testschnittstelle in das Betriebssystem. Im Umfeld mobiler Plattformen ist i. d. R. in Intervallen von sechs Monaten mit einer größeren Aktualisierung der Plattform zu rechnen. Findet eine solche Aktualisierung statt, wird ggf. eine erneute Anpassung der Zielplattform zur Integration der Testschnittstelle erforderlich. Ob eine solche Anpassung tatsächlich erforderlich wird, muss im konkreten Einzelfall überprüft werden. Der Aufwand zur Anpassung wird hierbei wesentlich durch die Lokalisation der Testschnittstelle innerhalb der Plattform bestimmt. Von einer Aktualisierung der Android-Plattform durch Google sind i. d. R. unterschiedliche Teile des Systems betroffen. Eine Änderung des verwendeten Linux-Kernels findet im Vergleich zur Häufigkeit der Systemaktualisierungen hingegen nur vergleichsweise selten statt[7]. Durch eine nahe dem Linux-Kernel liegende Lokalisation der Testschnittstelle kann der Anpassungsaufwand beispielsweise weiter reduziert werden, weil die Anpassung nur vergleichsweise selten tatsächlich erforderlich wird.

Anpassungen an der Testschnittstelle sind jedoch weder bei einer Lokalisation der Testschnittstelle im SUT noch bei einer Lokalisation in der Plattform vermeidbar, wenn durch eine Evolution der Plattform Komponenten berührt werden, die den Zugriff auf Sensoren, GPS oder andere Quellen physikalischer Kontextparameter implementieren. Eine einmalige Anpassung der Plattform stellt deshalb die effizientere Alternative dar als die Anpassung aller Apps im Portfolio eines Anbieters oder Herstellers. Dies gilt insbesondere, weil eine evolutionsbedingte Anpassung der Sensorschnittstellen eine Anpassung an Apps erforderlich macht, die eine Testschnittstelle nach dem in Abschnitt 5.5.1.1 diskutierten Schema implementieren, selbst wenn aus fachlicher Perspektive eigentlich keine Anpassung erforderlich wäre. Dies kann bei einer Integration der Testschnittstelle in die Plattform vermieden werden.

Aufgrund der in Abschnitt 5.5.1.1 und Abschnitt 5.5.1.2 diskutierten Vor- und Nachteile der Integration einer Testschnittstelle in das SUT oder in die Plattform wird die zweite Alternative ingesamt als die technisch vorteilhaftere und wirtschaftlich günstigere Alternative bewertet. Zur Umsetzung des in dieser Dissertation vorgestellten Ansatzes zur Testautomatisierung mobiler, kontextsensitiver Anwendungen wird deshalb die Variante der Schnittstellenintegration in die Plattform umgesetzt.

[7] Die Plattform Android hat bis zum Jahr 2016 insgesamt 13 große Aktualisierungen (*Major Upgrades*) und 47 kleine Aktualisierungen (*Minor Upgrades*) erfahren. In nur drei dieser Upgrades wurde der verwendete Linux-Kernel ebenfalls durch eine neuere Version ersetzt [140].

5.5.2 Anpassung des Android-OS zur Durchführung von Tests für kontextsensitive Anwendungen

Um die in Abschnitt 5.5.1.2 diskutierte Methode zur Simulation von Sensordaten auf Betriebssystemebene zur realisieren, müssen Änderungen am Android-Betriebssystem vorgenommen werden, da in der Standardimplementierung keine der notwendigen Schnittstellen enthalten sind. Sensoren sind Hardware-Bausteine, für die jeweils vom Hersteller spezifische Gerätetreiber bereitgestellt werden. Diese werden im Linux-Kernel des Android-*System-Stacks* ausgeführt. Hersteller von mobilen Geräten sind frei in der Wahl des Sensor-Herstellers, so dass für unterschiedliche Geräte jeweils individuelle Gerätetreiber notwendig sind. Gerätetreiber sind herstellerspezifische Softwareartefakte, die nicht dem Android-OSS-Projekt zuzuordnen sind.

Mobile Geräte verhalten sich im Normalfall wie eingebettet Geräte, die dem Anwender neben dem Standard-UI i. d. R. keine weiteren Schnittstellen anbieten. Aus diesem Grund ist es für den Anwender irrelevant, welche konkreten Sensor-Module in einem Gerät verwendet werden. Eventuelle Aktualisierungen von Gerätetreibern, wie sie im Desktop-Umfeld nicht unüblich sind, erlebt der Android-Anwender als Aktualisierung des Gesamtsystems, ohne weitergehenden Einfluss nehmen zu können.

Bei der Integration von Testschnittstellen auf Betriebssystemebene ist Wissen über die technische Einbettung von Sensoren in den Android-*System-Stack* (engl. Stapel aufeinander aufbauender funktional differenzierter Systemkomponenten) erforderlich, damit einerseits Testschnittstellen so im System lokalisiert werden können, dass Apps im Black-Box-Test-Verfahren geprüft werden können, andererseits das System insgesamt nicht durch die Testschnittstellen kompromittiert wird.

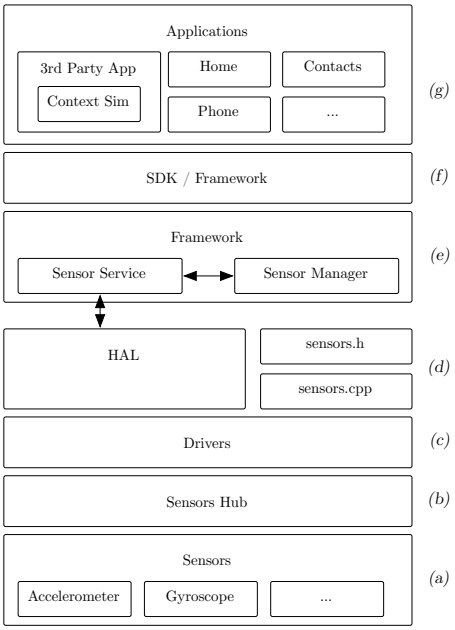


Abbildung 5.54: Android-System-Stack mit Fokus auf Verarbeitung von Sensordaten [142]

Der Android-*Sensor-Stack* als Bestandteil des Gesamtsystems ist in Abbildung 5.54 mit einem besonderen Fokus auf die Verarbeitung von Sensordaten dargestellt. Messdaten werden

auf der untersten Ebene *a* mit der Hardware der Sensoren erfasst und von dort in einen zentralen Broker, den *Sensors-Hub* (Ebene *b*) weitergeleitet. Die Steuerung individueller Sensoren erfolgt durch die Gerätetreiber in der darüberliegenden Ebene *c*. Die Abstraktion von konkreten Hardware-Modulen erfolgt in Ebene *d*, dem *Hardware-Abstraction-Layer* (HAL). Oberhalb des HAL beginnen bereits die Schichten des Android-Frameworks (Ebene *e*, verkürzt abgebildet) mit dem Android-SDK in Ebene *f*. Dieses beinhaltet alle Architekturkomponenten, die Entwicklern zur Programmierung von Apps (Ebene *g*) zur Verfügung stehen. Innerhalb der obersten Ebene des Android-System-Stacks existieren Apps hierarchielos nebeneinander.

Android-Apps können entweder in der Programmiersprache Java (Verwendung des SDK) oder in den Programmiersprachen C/C++ durch Verwendung des *Native Development Kits* (NDK) programmiert werden. In beiden Fällen erfolgt die Ausführung jeder Anwendung in einer separaten Instanz der *Ahead-of-Time-Compiler Android Runtime* (ART), wodurch eine Kapselung jeder App gegenüber dem restlichen System gewährleistet ist. Dies ist einerseits die zentrale Philosophie der Android-Systemarchitektur und zugleich wesentlicher Aspekt der Sicherheitsarchitektur des Systems, da hierdurch Anwendungen vor gegenseitigem unberechtigtem Zugriff geschützt werden.

Diese Systemarchitektur bedingt unmittelbar, dass Apps nicht direkt auf Sensor-Hardware zugreifen können. Statt eines direkten Zugriffs auf die Sensor-Hardware, wird der Transport von Sensordaten aus der Hardware bis zur verwendenden App vollständig durch Komponenten des Systems realisiert, auf die reguläre Android-Apps keinen direkten Zugriff haben. Hierin manifestiert sich das Subscriber-Entwurfsmuster der Sensorverwendung in Android-Apps.

Soll eine App (Ebene *g*, Abbildung 5.54) Sensor-Ereignisse empfangen, muss zunächst durch Verwendung des Android SDKs (Ebene *f*, Abbildung 5.54) eine Instanz des vom Android-Framework bereitgestellten Sensor-*Managers* angefordert werden. Am Sensor-Manager, lokalisiert in Ebene *e* (Abbildung 5.54), wird ein *Event-Handler* (Objekt zur Behandlung von Sensor-Ereignissen) registriert. Dies veranlasst den Sensor-*Service* (d. h. zentrale Komponente zur Verwaltung von Sensoren, ebenfalls Ebene *e* in Abbildung 5.54), den zu verwendenden Sensor zu aktivieren, sofern er nicht bereits von einer anderen App verwendet wird. Die Aktivierung eines Sensors wird an das HAL (Ebene *d*, Abbildung 5.54) kommuniziert, welches die Aktivierung des Hardware-Bausteins veranlasst (Ebenen *c* bis *a*, Abbildung 5.54). Dieser Prozess ist in einem UML-Sequenzdiagramm in Abbildung 5.55 dargestellt.

Nach erfolgter Aktivierung des Sensors überprüft der Sensor-Manager kontinuierlich, ob aktualisierte Daten vorliegen. Der Sensor-Manager leitet diese Anfrage an den Sensor-Service weiter, welcher wiederum das Sensor-HAL veranlasst, durch *Polling* (engl. wiederholtes Anfragen von Daten in einem festgesetzten Intervall) Daten vom Sensor abzufragen. Liegen neue Daten vor, werden sie durch den Sensor-Service an den Sensor-Manager weitergegeben, welcher wiederum die Callback-Methode in der App aufruft. So ist gewährleistet, dass einzelne Apps durch das Subscriber-Entwurfsmuster, d. h. im *Push*-Betrieb (die App wird vom System über Änderungen benachrichtigt), auf Sensoren zugreifen und Apps keine Polling-Algorithmen implementieren. Die Transition vom Polling von Sensordaten zum Push-Betrieb erfolgt im Sensor-Service. Der Prozess ist in Abbildung 5.56 dargestellt.

Ziel der in dieser Dissertation vorgestellten Simulationstechnologie für Kontextparameter und insbesondere Sensordaten ist es, Testschnittstellen so zu implementieren, dass keine Mo-

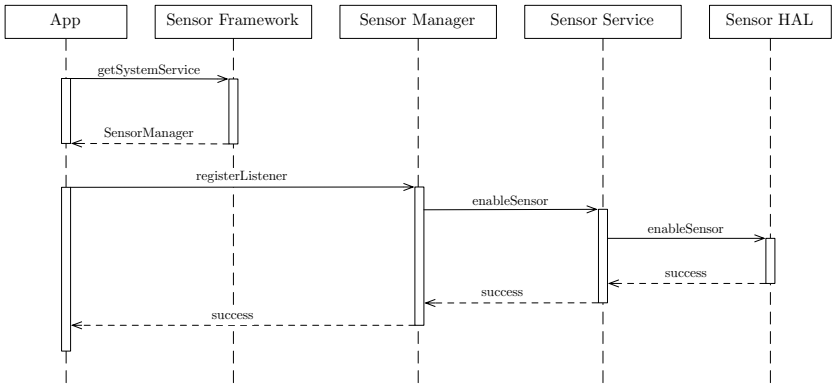


Abbildung 5.55: UML-Sequenzdiagramm der Aktivierung eines Sensors in der Plattform Android

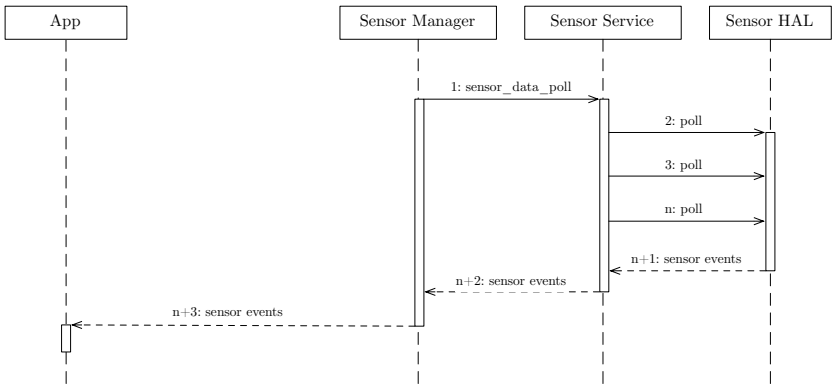


Abbildung 5.56: UML-Sequenzdiagramm der Übertragung von Sensordaten aus der Hardware in die verwendende App

difikation an der zu testenden Anwendung notwendig ist. Eine Integration der Testschnittstelle in die Applikationsschicht (Ebene h, Abbildung 5.54) kommt daher nicht in Frage. Eine Anpassung des Android-SDK wäre eine Option zur Integration einer Testschnittstelle durch Bereitstellung zusätzlicher Sensortypen, die innerhalb der App genutzt werden könnten, um anstelle der originär von der Sensor-Hardware erzeugten Daten Testdaten aus anderen Quellen in die zu testende App zu propagieren. Denkbar wäre hier eine Implementierung analog dem vom Android Testing-Framework bereitgestellten *Mock-Location-Provider*[8]. Eine Anpassung des SDK hat aber zur Folge, dass die zum Zweck des Testens hinzugefügten Sensoren nur solchen Apps verfügbar sind, die mit dem angepassten SDK kompiliert wurden. In der Praxis wird hierdurch die Möglichkeit der Durchführung von Black-Box-Tests eingeschränkt, insbesondere könnten Apps, die mit dem modifizierten SDK kompiliert wurden nicht in den App Stores vertrieben werden, weil das Kompilat nicht kompatibel zur Infrastruktur des Plattformbetreibers ist.

Zur Realisierung der Testschnittstelle zur Einspeisung von Testdaten als Sensordaten wurde in dieser Dissertation daher eine Anpassung des Android-Betriebssystems an der Schnittstelle zwischen Sensor-HAL und Sensor-Service auf der Ebene der Systembibliotheken gewählt.

[8]Diese Variante wurde in der im Rahmen dieser Dissertation betreuten Masterarbeit von Chinh Truc Dao [79] untersucht.

Diese Lokalisierung der Testschnittstelle gewährleistet einerseits Unabhängigkeit von der im Gerät verwendeten Hardware und ermöglich ebenfalls die Simulation auf mobilen Geräten und in emulierten Umgebungen, ohne hierzu auf die in Abschnitt 4.2.4.2 diskutierte manuelle Simulation von Sensordaten zurückgreifen zu müssen.

Zur Simulation von Sensordaten wurde das Sensor-HAL an der Schnittstelle zum Sensor-Service um eine Netzwerkschnittstelle ergänzt (vgl. Abbildung 5.57). Im Testbetrieb werden an dieser Netzwerkschnittstelle Testdaten empfangen und anstelle der originär von der Sensor-Hardware erzeugten Daten an den Sensor-Service des Android-Systems übergeben. Dieser Prozess in Abbildung 5.58 dargestellt.

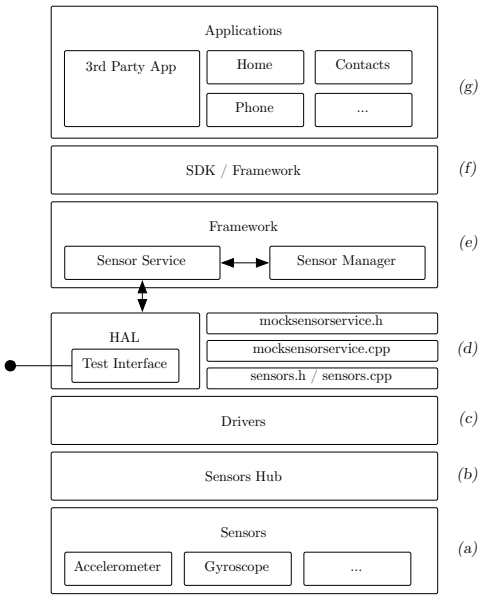


Abbildung 5.57: Angepasster Android-System-Stack. Auf HAL-Ebene wird ein Interface zur Bereitstellung von Testdaten exponiert.

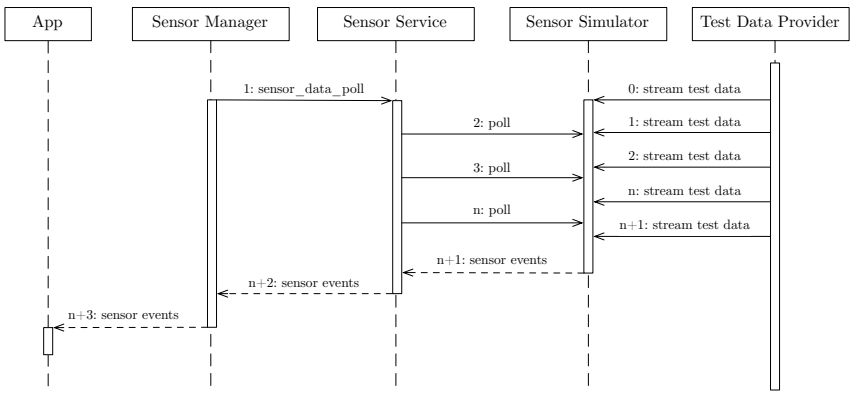


Abbildung 5.58: UML-Sequenzdiagramm der Bereitstellung von Sensortestdaten

Zur Herstellung einer möglichst umfassenden Fähigkeit zum Testen sensorbasierter Anwendungen bietet die Schnittstelle zur Einspeisung von Sensordaten als Testdaten alle Sensoren an, die im SDK des Android-Betriebssystems in der Version 5.x Lollipop enthalten

sind (vgl. Android Sensor Dokumentation [143]). Hierdurch haben Entwickler und Tester die Möglichkeit zur Implementierung und zum Testen von sensorbasierteren Apps sowohl in Emulatoren als auch auf mobilen Geräten. Insbesondere können Apps auf mobilen Geräten getestet werden, deren Hardware einige Sensoren nicht enthalten.

Die Bereitstellung von Testdaten kann wahlweise über eine Desktop-Anwendung in Anlehnung an den OpenIntents SensorSimulator™ oder durch ein Automatisierungswerkzeug erfolgen. Insbesondere die Bereitstellung durch ein Testautomatisierungswerkzeug kann wesentlich zur Steigerung der Effizienz des Testens mobiler, kontextsensitiver Anwendungen beitragen und ist ein Kernbeitrag dieser Dissertation. Die Integration des Testautomatisierungswerkzeugs Calabash mit der hier vorgestellten Modifikation des Android Betriebssystems wird in Abschnitt 5.5.3 diskutiert.

5.5.3 Anpassung des Calabash-Frameworks zur Durchführung von Tests für kontextsensitive Anwendungen

Das Testautomatisierungswerkzeug Calabash (vgl. Abschnitt 2.5.2.6) stellt in seiner Standardimplementierung nur eingeschränkte Funktionen zum Testen kontextsensitiver Apps zur Verfügung (vgl. Abschnitt 2.5.2.6.4). Konkret ist lediglich ein Testschritt zur Simulation von Standortdaten verfügbar, der es allerdings nicht zulässt, Metainformationen wie Präzision und Alter der Standortinformation im Testfall zu spezifizieren. Hierdurch werden die Möglichkeiten zum Testen anderer Kriterien als dem *Happy-Path*-Kriterium (vgl. Vieira et al. [351], Testkriterium, das Tests umfasst, bei denen nur solche Daten verwendet werden, die einen Workflow erfolgreich absolvieren), signifikant eingeschränkt, z. B. die Verwendung hinsichtlich eines gewählten Qualitätskriteriums ungültiger Standortinformationen.

Zur Simulation von Sensordaten sind in der Calabash-Standardimplementierung keine Funktionen verfügbar. Ebenfalls enthält die Calabash-Standardimplementierung keine Funktionen, um in Tests Datum und Uhrzeit im SUT zu manipulieren. Dies liegt darin begründet, dass Calabash in der Zielplattform zusätzlich zum SUT ein weiteres Paket auf Anwendungsebene installiert, in welchem die zur Interaktion und Inspektion (d. h. Instrumentierung) des SUT sowie die zur Kommunikation mit dem Entwicklungscomputer notwendigen Komponenten implementiert sind. Dieses Paket verfügt aus Gründen der Systemsicherheit nicht über die erforderliche Berechtigung, die Systemzeit zu verändern. Hierdurch wird die Nutzbarkeit zum Testen kontextsensitiver Apps eingeschränkt, wenn diese die Kontextparameter Datum und Uhrzeit (vgl. Abschnitt 3.1.2.1.2) verwenden.

Im Rahmen dieser Dissertation wurde die Implementierung des Werkzeugs Calabash deshalb um drei Funktionen erweitert: (1) Funktion zur Simulation von Sensordaten in Tests, (2) Funktion zur Verwendung von Standortinformationen mit Metadaten in Tests sowie (3) einer Funktion zur Spezifikation des Kontextparameters Zeit in Tests. In Verbindung mit der in Abschnitt 5.5.2 diskutierten Anpassung am Android-Betriebssystem kann das modifizierte Calabash-Framework verwendet werden, um mobile, kontextsensitive Apps automatisiert zu testen. Die Architektur des Testwerkzeugs ist schematisch in Abbildung 5.59 dargestellt.

Links in Abbildung 5.59 ist der Entwicklungscomputer mit dem modifizierten Calabash-Framework abgebildet. Hierbei kann es sich unmittelbar um den von einem Entwickler bei

der Programmierung des SUT verwendeten Computer oder um einen Build-Server in einer CI-Umgebung handeln. Im ersten Fall kann ein Entwickler Tests als Teil des Entwicklungsprozesses ausführen, im zweiten Fall können Tests beim Einpflegen von Quellcode in ein zentrales SCM oder bei der Durchführung von Akzeptanztests in Vorbereitung der Inbetriebnahme des SUT ausgeführt werden.

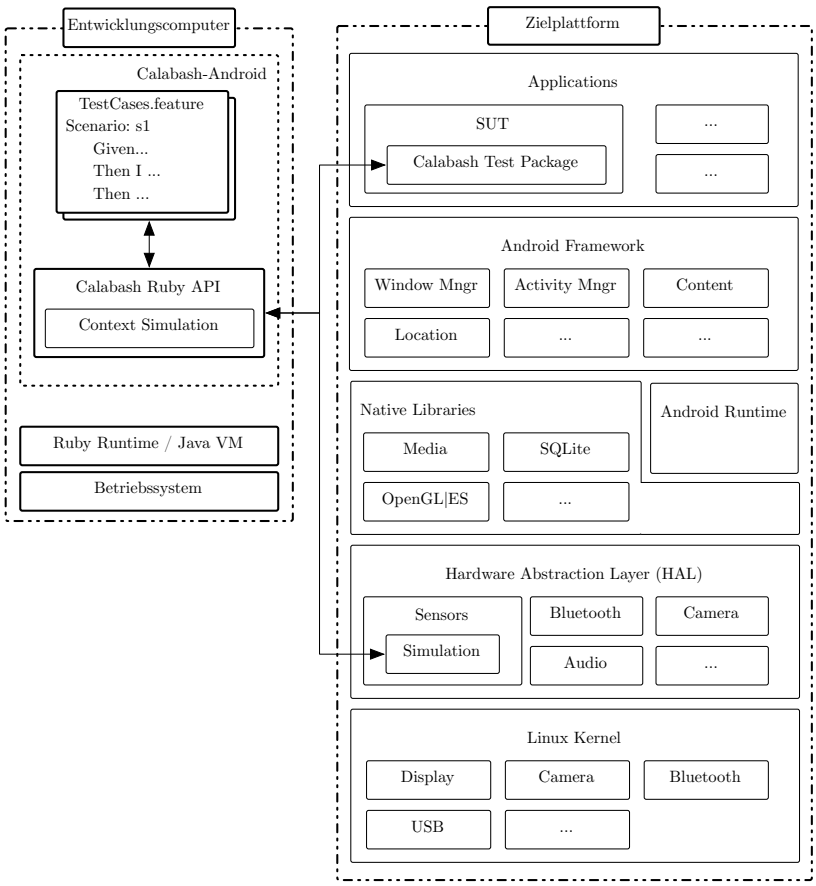


Abbildung 5.59: Übersicht über die Architektur des modifizierten Calabash-Frameworks im Zusammenwirken mit dem Android *Designed for Testability*

Im modifizierten Calabash-Framework ist insbesondere die Kontextsimulation enthalten. Diese Komponente interpretiert die Testdaten aus dem Calabash-Feature (d. h. dem Testfall) und ermittelt, welche Sensoren zur Simulation des im Testfall spezifizierten Kontext zu verwenden sind. Im einfachen Fall spezifiziert der Testfall konkrete Daten für einen spezifischen Sensor, z. B. den Magnetfeldsensor. In diesem Fall übermittelt der Kontextsimulator die Testdaten direkt an die Schnittstelle des modifizierten Android-Systems. In komplexeren Fällen, beispielsweise wenn eine Spezifikation der Geräteorientierung Teil des Testfalls ist, werden die zur Bestimmung der Geräteorientierung verwendeten Sensoren ermittelt und aus den Testdaten die konkreten Daten für diese Sensoren berechnet. Im Beispiel der Geräteorientierung werden Daten für den Beschleunigungssensor und den Magnetfeldsensor berechnet, die der spezifizierten Geräteorientierung entsprechen[9].

[9]Diese Funktion ist bereits Teil des OpenIntents SensorSimulator™. Zur Realisierung der in dieser Dissertation erarbeiteten Anpassung des Calabash-Frameworks wurden Komponenten zur mathematischen Berechnung individueller Sensordaten für die Geräteorientierung wiederverwendet.

Calabash verwendet zur Steuerung des SUT und zur Übertragung von Testergebnissen zwischen dem Entwicklungscomputer und der Zielplattform eine Netzwerkverbindung. Die Netzwerkverbindung wird ebenfalls zur Kommunikation von Sensordaten zwischen dem Kontextsimulator und der Testschnittstelle verwendet. Um eine aufwändige Konfiguration (d. h. Ermittlung von IP-Adresse und Portnummer) des mobilen Geräts bzw. des Emulators zu vermeiden, wird der Netzwerktunnel verwendet, der durch die Werkzeuge des Android-SDK, insbesondere der *Android Debug Bridge* (ADB), bereitgestellt wird.

Rechts in Abbildung 5.59 ist das Zielsystem schematisch abgebildet. Hierbei kann es sich um ein mobiles Gerät oder um einen Emulator handeln. Insbesondere bei Verwendung einer CI-Umgebung können hier mehrere Geräte oder mehrere Emulatorkonfigurationen zur Gewährleistung einer adäquaten Geräte- oder Konfigurationsabdeckung (d. h. Geräte unterschiedlicher Hersteller mit unterschiedlichen Eigenschaften) verwendet werden.

Bei der Testausführung wird das SUT auf der Zielplattform installiert und durch das Calabash-Framework ferngesteuert. Diese Fernsteuerung entspricht der Simulation der Verwendung des SUT durch einen menschlichen Akteur (d. h. dem zukünftigen Anwender). Zusätzlich zur Standardimplementierung des Calabash-Frameworks ist die im Rahmen dieser Dissertation erstellte modifizierte Implementierung in der Lage, die in Abschnitt 5.5.1.2 diskutierte Schnittstelle zur Simulation von Sensordaten mit Testdaten zu versorgen. Durch die Verwendung von Automatisierungstechnologie ist gegenüber einer manuellen Testausführung gewährleistet, dass Testdaten zuverlässig in allen Zielsystemen reproduziert werden.

Zur Realisierung dieser Funktionalität wurde das Calabash-Framework in zweierlei Hinsicht angepasst. Zum einen wurde eine zusätzliche Komponente, der Sensorsimulator, in das Werkzeug eingefügt. Diese erweitert die Schnittstelle zwischen derjenigen Komponente, die Calabash-typische textuelle Repräsentation von Testfällen interpretiert und derjenigen Komponente, die aus einzelnen Testschritten Steuerbefehle generiert und an das SUT sendet. Tritt in einem Calabash-Feature eine Codezeile auf, die einer Simulation von Kontextparametern zugeordnet werden kann, wird zusätzlich zu den inhaltlichen Steuerbefehlen zur Interaktion mit dem SUT der Sensorsimulator instruiert, simulierte Sensordaten an die Schnittstelle im Android-System zu übermitteln. Der Sensorsimulator ist ebenfalls für die Simulation von Sensorrauschen verantwortlich und wird in Abschnitt 5.5.3.1 im Detail thematisiert.

Die zweite Anpassung am Calabash-Framework ist die Bereitstellung zusätzlicher Definitionen von Testschritten (Calabash-*Step-Definitions*) im Paradigma des BDD. Der BDD-Ansatz erlaubt grundsätzlich die Definition von Calabash-Testschritten in einem an eine natürliche Sprache (d. h. Englisch, Deutsch) angelehnten Syntax, wodurch das Framework auf den spezifischen Projektkontext angepasst werden kann. Die Definition zusätzlicher Calabash-Step-Definitions wird in Abschnitt 5.5.3.2 thematisiert.

5.5.3.1 Sensorsimulator

Aufgabe des Sensorsimulators ist die Bereitstellung von Testdaten. Diese können auf unterschiedlichen Abstraktionsebenen (z. B. spezifisch für einen Sensor oder abstrakt, z. B. als diskrete Geräteorientierung) im Testfallmodell hinterlegt sein.

Der Sensorsimulator erfüllt drei Funktionen. Erstens extrahiert er Testdaten aus dem Calabash-Feature und überträgt diese Daten an die Testschnittstelle im Android-Betriebssystem.

Zweitens ist es Aufgabe des Sensorsimulators, aus komplexen Testdaten, die mehrere Sensoren des SUT umfassen, die konkreten Sollwerte für jeden betroffenen Sensor zu berechnen. Drittens addiert der Sensorsimulator im Bedarfsfall künstliches Sensorrauschen zu den Testdaten hinzu, um realistische Sensormesswerte im SUT zu generieren.

Die Extraktion der zu simulierenden Daten erfolgt durch Inspektion des Calabash-Steps. Die Transformation von plattformunabhängigem Testfallmodell zu Calabash-Testfällen hat aus individuellen *Test::Precondition*-, *Test::Action*- oder *Test::Postcondition*-Elementen im Testfallmodell jeweils Calabash-Steps (d. h. Gherkin-Statements) generiert (vgl. Abschnitt 5.4). In diesen Gherkin-Statements sind die an die Testschnittstelle zu übertragenden Kontextparameter im Klartext enthalten. Calabash untersucht die Gherkin-Statements auf das Vorkommen bestimmter Muster (d. h. Step-Defintions). Jeder Calabash-Step wird hierbei in zwei Komponenten zerlegt: die auszuführende Testinstruktion (d. h. Interaktion mit dem SUT) und die hierbei zu verwendenden Testdaten.

Der Calabash-Parser wurde zur Implementierung des Sensorsimulators so angepasst, dass er in der Lage ist, zwischen solchen Calabash-Steps, die eine Interaktion mit dem SUT repräsentieren und solchen, die eine Parametrisierung des Kontext der Testausführung repräsentieren, zu unterscheiden. Identifiziert der Parser einen Calabash-Step, der einen zu simulierenden Kontextparameter beschreibt, werden die Testdaten aus diesem Calabash-Step extrahiert, in ein Datenpaket verpackt und in den Datenstrom der an die Testschnittstelle im Android-Betriebssystems (vgl. Abschnitt 5.5.2) eingeleitet. Die modifizierten Komponenten des Android-Betriebssystems hinter der Testschnittstelle propagieren diese Daten dann anstelle der originär von der Sensor-Hardware erzeugten Daten an alle Anwendungen, die Datenaktualisierungen dieses spezifischen Sensors abonniert haben. Der Inhalt des Datenpakets ist eine für den Transport aufbereitete Repräsentierung der aus dem Testfallmodell extrahierten Daten, konkret diejenigen Werte, die im Rahmen des Testfalls im SUT erzeugt werden sollen.

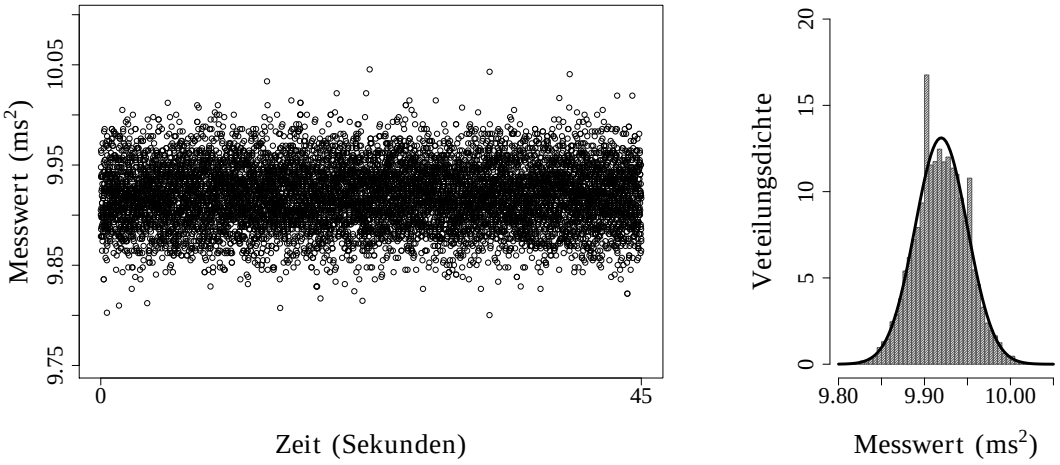
Werden in einem Testfallmodell Kontextparameter spezifiziert, die mehrere Sensoren betreffen, z. B. Geräteorientierung (vgl. Abschnitt 3.1.2.2.1), werden zunächst die beteiligten Sensoren identifiziert und anschließend die pro Sensor zu simulierenden Daten berechnet. Zur Realisierung dieser Funktion wurde auf existierende Komponenten aus dem Projekt Open-Intents SensorSimulator™ [269] zurückgegriffen, in welchem die Berechnungsformel für diesen Anwendungsfall bereits realisiert wurde. Die berechneten Daten werden ebenfalls an die Schnittstelle im Android-Betriebssystem übertragen.

Die dritte Aufgabe des Sensorsimulators ist das Hinzufügen künstlichen Sensorrauschens zu den Testdaten, um eine realitätsnahe Simulation von Sensordaten in Tests zu gewährleisten. Die Qualität einer mobilen Anwendung, die Sensordaten verwertet, wird wesentlich dadurch geprägt, wie gut Zufallsfehler in den Sensordaten kompensiert werden, beispielsweise durch Verwendung von Filtern (z. B. *High-Pass-Filter*, Milette und Stroud [249]). Bei der Simulation von Sensordaten im Rahmen des Testens ist es daher notwendig, neben den eigentlichen Werten für Sensoren ebenfalls Sensorrauschen in die Testdaten einführen.

Die Messung physikalischer Größen unterliegt i. d. R. Störungen. Diese können in solche Störungen unterschieden werden, die durch äußere Einflüsse auftreten, wie beispielsweise Temperaturschwankungen, und solche, die durch Eigenschaften des Messgeräts verursacht werden.

Hierzu gehören z. B. Messverfahren und die spezifische Bauart des Sensors. Durch Eigenschaften des Messgerätes verursachte Abweichungen des Messwerts vom tatsächlichen Wert der gemessenen Größe werden als Sensorrauschen bezeichnet (vgl. Milette und Stroud [249]). Im Gegensatz zu systematischen Messfehlern, die immer unter gleichen Bedingungen reproduzierbare Abweichungen erzeugen, groben Messfehlern, die anhand einzelner signifikant abweichender Messwerte gut identifizierbar sind und methodischen Messfehlern, die aufgrund des Messprinzips des Sensors unvermeidbar sind, handelt es sich bei Sensorrauschen um Zufallsfehler, die Abweichungen zwischen Messwert und tatsächlichem Wert der Messgröße verursachen.

Sensorrauschen repräsentiert einen zufälligen Messfehler, der durch eine Gaußsche Verteilung (d. h. Normalverteilung) beschrieben wird (vgl. Hering und Schönfelder [180]). Zur Verdeutlichung ist in Abbildung 5.60 eine Messreihe des Beschleunigungssensors eines sich in Ruhe befindlichen Geräts, konkret eines Nexus 7 Geräts des Herstellers Asus der Baureihe 2013 (ASUS-1A019A), dargestellt. Abgebildet sind die Messwerte der z-Achse im Referenzsystem des Geräts, welches zur Messung flach auf einer Tischoberfläche platziert wurde.



(a) Die Messwerte des Beschleunigungssensors streuen um den Mittelwert $9.919\,495\,\mathrm{m\,s^{-2}}$

(b) Die Verteilung des Zufallsfehlers (d. h. Sensorrauschen)

Abbildung 5.60: Messwerte der z-Achse des Beschleunigungssensors eines flach auf einer Tischoberfläche platzierten Geräts

In Abbildung 5.60a ist zu erkennen, dass der Sensor keine konstanten Messwerte generiert, sondern die Messwerte um einen Mittelwert herum verteilt sind. Im konkreten Fall beträgt der Mittelwert der im Messzeitraum von ca. 45 Sekunden erfassten 10001 Messpunkten $9.919\,495\,\mathrm{m\,s^{-2}}$ mit Standardabweichung 0,03041076. Bei einem flach auf einer Tischoberfläche platzierten, sich in Ruhe befindlichen Gerät ist die Gravitation die einzige entlang der z-Achse des Referenzsystems des Geräts einwirkende Beschleunigungskraft. Auf der Erde beträgt ihr Wert im Mittel $9.81\,\mathrm{m\,s^{-2}}$. Die tatsächlich wirksame Gravitationsbeschleunigung ist ortsabhängig und beträgt zwischen $9.801\,\mathrm{m\,s^{-2}}$ am Äquator und $9.867\,\mathrm{m\,s^{-2}}$ an den Polen [175].

Bei einem flach auf einer Tischoberfläche platzierten, sich in Ruhe befindlichen Gerät ist zu erwarten, dass der vom Beschleunigungssensor des Geräts entlang der z-Achse des Referenzsystems des Geräts gemessene Wert gleich der Gravitationskonstante am Ort der Messung ist. Die im konkreten Fall zu beobachtende Abweichung vom Mittelwert der Messung

und dem erwarten Wert entsteht einerseits daraus, dass der nominelle Wert $9.81\,\mathrm{m\,s^{-2}}$ nicht pauschal für alle Orte auf der Erde tatsächlich gültig ist und andererseits aus bauartbedingter Messungenauigkeit des konkret im Gerät vorhandenen Sensors. Dasselbe Experiment wurde ebenfalls mit einem Samsung Galaxy S6 Gerät durchgeführt. Hierbei wurde ebenfalls eine Abweichung vom Mittelwert der Messung und dem nominellen Wert der Erdgravitation in vergleichbarer Größenordnung festgestellt.

In Abbildung 5.60b ist die Verteilung der Messwerte in einem Histogramm dargestellt. Erkennbar ist in der Abbildung, dass die Mehrheit der Messwerte in der Nähe des Mittelwertes liegt und der Zufallsfehler in den Randbereichen der Messwerte stark abnimmt. Das Histogramm in Abbildung 5.60b wurde mit einer für die Gaußsche Normalverteilung typischen Dichtefunktion überlagert. Die Abbildung lässt vermuten, dass eine Normalverteilung des Zufallsfehlers in den Messwerten vorliegt. Die Gültigkeit der Annahme, dass Sensorrauschen (d. h. Zufallsfehler in den Messdaten) normalverteilt ist, wird durch wissenschaftlich-technische Literatur (z. B. Hering und Schönfelder [180] sowie Milette und Stroud [249]) bekräftigt.

Eine Anwendung statistischer Tests zur Überprüfung einer Stichprobe auf Normalverteilung, konkret Shapiro-Wilk-Test [320] und Anderson-Darling-Test [9], auf die im Experiment erhobenen Messwerte konnte eine Normalverteilung jedoch nicht sicher nachweisen. Für die Ursache der Nichtnachweisbarkeit der Normalverteilung mit dem Shapiro-Wilk-Test können mehrere Gründe angenommen werden. Zum einen reagiert der Shapiro-Wilk-Test sensibel auf Ausreißer, d. h. Messwerte die stark vom erwartetem Wert abweichen. Durch eine Analyse der Stichprobe mit dem Grubbs-Test [160] sowie dem Dixon-Test [98] (beides Tests zum Nachweis von Ausreißern in Stichproben) konnten Ausreißer nachgewiesen werden, so dass eine Korrumpierung des Shapiro-Wilk-Tests plausibilisiert wird. Zum anderen erzeugen Sensoren technisch bedingt Messwerte mit einer begrenzten Auflösung (d. h. die kleinste unterscheidbare Änderung der Messgröße [180]). Hierdurch ergibt sich zwangsläufig, dass der Messbereich eines Sensors nicht das gesamte Intervall der reellen Zahlen innerhalb des Messbereichs abdeckt. Hieraus folgt, dass sowohl Messwerte als auch Zufallsfehler nicht mathematisch korrekt normverteilt in der Stichprobe auftreten.

Für die Simulation von Rauschen in Sensordaten wird im Sensorsimulator trotz dieser Erkenntnisse in Übereinstimmung mit der wissenschaftlich-technischen Literatur [180, 249] Normalverteilung angenommen. Die zu einer Normalverteilung gehörige Dichtefunktion ist gegeben durch (vgl. Henze [179, S. 297]):

$$p(x) = \frac{1}{\sigma\sqrt{2\pi}} \cdot e^{-\frac{1}{2}(\frac{x-\mu}{\sigma})^2}, x \in \mathbb{R}, \mu \in \mathbb{R}, \sigma > 0$$

Die Dichtefunktion zu einer idealen Messreihe ist nur vom Erwartungswert μ des Experiments und der Streuung σ der Messmethode abhängig. Ihre Wendepunkte werden durch die Standardabweichung σ bestimmt, die Sensormesswerte sind glockenförmig um den tatsächlichen Wert μ verteilt. Für Sensormesswerte mit Standardnormalverteilung ist der tatsächliche Wert mit einer Wahrscheinlichkeit von 99,7% der arithmetische Mittelwert $\overline{x}$ aller Messwerte im Bereich $[-3\sigma; +3\sigma]$. Die Berechnung einer Pseudozufallszahle x kann durch die Funktion

$$P(\mu + \sigma X \leq x) = P(X \leq \frac{x-\mu}{\sigma}) = \phi(\frac{x-\mu}{\sigma}), x \in \mathbb{R}$$

nach der Transformation $x \rightarrow \mu + \sigma x$ durch Angabe einer Standardabweichung σ und einem Erwartungswert μ erzeugt werden [179, S. 298]. Unter der Voraussetzung, dass μ und σ für Sensoren in Android Geräten bekannt sind, kann hierdurch den in einem Testfall spezifizierten Testdaten künstlich Sensorrauschen hinzugefügt werden. Der Erwartungswert μ ist durch die Spezifikation im Testfallmodell direkt gegeben. Die Standardabweichung σ ist hingegen sensorspezifisch und von der konkret in einem mobilen Gerät verwendeten Hardware abhängig.

Zur Erhebung realitätsnaher Daten für die Standardabweichung wurden experimentell Daten erhoben. Hierzu wurden für jeden Sensor in ausgewählten Geräten[10] Messreihen durchgeführt und anschließend zur Berechnung der Standardabweichung herangezogen[11]. Die so ermittelten Sensorprofile enthalten für alle im spezifischen Gerät verfügbaren Sensoren typische Werte für die Standardabweichung, die bei der Generierung von Testdaten unter Angabe eines Profils zur Erzeugung künstlichen Sensorrauschens verwendet werden.

Im Testbetrieb erlauben die in Abschnitt 5.5.3.2 diskutierten Calabash-Step-Definitions Sensorrauschen bedarfsweise hinzuzufügen oder zu deaktivieren. So haben Entwickler die Möglichkeit, Algorithmen zur Verarbeitung von Sensordaten zunächst ohne Sensorrauschen zu implementieren, um eine grundsätzliche Funktionalität zu erarbeiten und später durch Hinzufügen künstliche Rauschens gegenüber realitätsnahen Bedingungen zu härten (z. B. durch Implementierung geeigneter Filter).

Wird der Sensorsimulator im rauschfreien Modus betrieben, ist gewährleistet, dass die im Testfall spezifizierten Daten genau so in das SUT eingespeist werden, wie sie im Testfall (respektive im Testfallmodell) spezifiziert wurden. Die Reproduzierbarkeit der Testdaten ist hierdurch explizit gegeben. Wird der Sensorsimulator hingegen mit künstlichem Rauschen betrieben, wird der Zufallsfehler jedes Datums (d. h. Rauschen) so berechnet, dass der Mittelwert der in das SUT eingespeisten Daten den im Testfall spezifizierten Daten entspricht. Reproduzierbarkeit von Testdaten ist in diesem Fall nur noch hinsichtlich des Mittelwerts der Testdaten gegeben, nicht mehr hinsichtlich individueller Messpunkte.

Auch für den Anwendungsfall, dass Daten mit Rauschen zuverlässig reproduziert werden müssen, wurde im Sensorsimulator ein C'n'R-Modus implementiert. Dieser erlaubt das Aufzeichnen von Sensordaten auf einem mobilen Gerät und deren Verwendung zur Reproduktion von Testdaten im SUT.

5.5.3.2 Calabash-Step-Definitions

Das Calabash-Framework zur Testautomatisierung ist konzipiert, je nach spezifischen Projektkontext auf die Bedürfnisse der Stakeholder angepasst zu werden. Mit dem Framework werden eine Reihe von Step-Definitions für Standardaufgaben ausgeliefert (vgl. Abschnitt 2.5.2.6). Diese erschöpfen sich allerdings in Step-Definitions zur Interaktion mit dem SUT und einer einzelnen Step-Definition zur Simulation von Standortinformationen. Step-Definitions, die eine Manipulation anderer Kontextparameter ermöglichen, sind in der Standardimplementierung des Calabash-Frameworks nicht enthalten.

[10] Nexus 7, Asus, Baureihe 2013 (ASUS-1A019A); Nexus 5, LG Electronics, LG D821; Nexus 10, Samsung, GT-P8110

[11] Die Messreihen und Berechnung sensortypischer Standardabweichungen wurden in einer im Rahmen dieser Dissertation betreuten Bachelorarbeit von Marc Gesthüsen [242] erstellt.

Durch die in Abschnitt 5.5.2 diskutierte Anpassung am Android-Betriebssystem werden Schnittstellen bereitgestellt, die eine Manipulation von Kontextparameter ermöglichen, die auf Sensordaten basieren. Um diese mit dem Calabash-Framework nutzen zu können, wurden im Rahmen der Anpassung des Frameworks neben der Integration des Sensorsimulators (vgl. Abschnitt 5.5.3.1) zusätzlich neue Step-Definitions erstellt, die es erlauben, Kontextparameter als Testdaten in Tests zu spezifizieren.

Die in das Android-Betriebssystem integrierte Testschnittstelle erlaubt die Einspeisung von Sensordaten als Testdaten in das System und stellt zusätzlich eine Funktion bereit, mit welcher Datum und Uhrzeit des SUT als Vorbedingung von Tests eingestellt werden können. Hierzu korrespondierend wurden Calabash Step-Definitions erstellt, mit denen sowohl Datum und Uhrzeit in Testfällen spezifiziert als auch die Werte für Sensoren vorgegeben werden können. Konkret wurden für diese Sensoren Step-Definitions erstellt:

Beschleunigungssensor Sensor zur Messung der auf ein Gerät einwirkenden Beschleunigungskräfte, einschließlich der Gravitation.

Magnetfeldsensor (kalibriert und unkalibriert): Sensor zur Messung der Magnetfeldstärke im Umfeld des Geräts.

Orientierungssensor Sensor zur Messung der Orientierung des Geräts (Roll-, Nick- und Gierwinkel). Dieser Sensor wurde mit der Version 2.2 der Android-Plattform abgekündigt und wird lediglich aus Gründen der Abwärtskompatibilität unterstützt. Seine Funktion wird durch eine Kombination aus Magnetfeldsensor und Beschleunigungssensor ersetzt.

Gyroskop (kalibriert und unkalibriert): Sensor zur Messung der Rotationsrate des Geräts.

Luftdruck: Sensor zur Messung des Umgebungsluftdrucks.

Gravitation: Sensor zur Messung der Richtung des Betrags der auf das Gerät einwirkenden Gravitationskraft. Dieser Sensor dient als Ergänzung zum Beschleunigungssensor. Befindet sich das Gerät in Ruhe, liefern diese beide Sensoren identische Werte.

Linearbeschleunigungssensor: Sensor zur Messung der auf ein Gerät einwirkenden Beschleunigungskräfte ohne Berücksichtigung der Gravitation.

Rotationsvektor (auch für Spiele): Sensor zur Messung der Rotation des Geräts. Dieser Sensor ist eine sogenannter *Fusion Sensor*, der als Abstraktion der Hardware in Software realisiert ist und in Abhängigkeit von der konkreten Implementierung unterschiedliche Sensoren zur Berechnung der Orientierung des Geräts verwendet. Die Variante für Spiele bestimmt die Orientierung des Geräts in einem relativen Koordinatensystem. Sie dient der Ermittlung der Drehung des Geräts um die Achsen des Koordinatensystems des Geräts unabhängig von geographischen Referenzsystemen.

Geomagnetischer Rotationssensor: Erfüllt dieselbe Aufgabe wie der Rotationssensor, verwendet aber den Magnetfeldsensor anstelle des Gryroskops.

Viele der in dieser Auflistung enthaltenen Sensoren erfüllen eine ähnliche oder gar identische Funktion. Dem Entwickler steht es frei, zur Realisierung einer spezifischen Anforderung einen

beliebigen Sensor zu verwenden, der funktional zur Erbringung der Anforderung geeignet ist. Zur Erreichung einer möglichst großen Abdeckung von Geräten bietet es sich jedoch an, entweder Sensoren zu verwenden, die in vielen unterschiedlichen Geräten verfügbar sind oder mehrere Varianten der Implementierung zu erstellen und hiervon zur Laufzeit anhand der im spezifischen Gerät vorhandenen Sensoren die optimale auszuwählen. Aus diesem Grund wurden sowohl in der Implementierung des Sensorsimulators (vgl. Abschnitt 5.5.3.1) als auch in der Calabash-Implementierung alle in der Version 5.1.x Lollipop des Android-SDKs verfügbaren Sensoren berücksichtigt.

Um den Tester davon zu entlasten, für die Verwendung der Geräteorientierung in Testfälle manuell Sensordaten zu berechnen, die eine spezifische Geräteorientierung repräsentiert, wurden ebenfalls Makro-Step-Definitions erstellt, die es dem Tester erlauben, die Geräteorientierung in Roll-, Nick- und Gierwinkel im Testfall zu spezifizieren. Der Sensorsimulator berechnet hieraus die Werte für die einzelnen Sensoren. Analog stellt das Metamodell zur Kontextmodellierung (vgl. Abschnitt 5.2.2.2) ein Modellierungselement zur Verfügung, über welches die Geräteorientierung auf einem abstrakteren Niveau als individuellen Sensordaten modelliert werden kann (vgl. Abschnitt 5.2.2.1).

Quellcodefragment 5.4: Calabash-Step-Definitions zur Abbildung der Geräteorientierung (spezifisch in Zeilen 2-5) und als Makro (Zeilen 7-10) zur Spezifikation einer diskreten Geräteorientierung.

```
1  # device orientation
2  Given /^device orientation is ([-+]?\d+):([-+]?\d+):([-+]?\d+):(\d+)↙
       ↳ $/ do |azimuth,pitch,roll,noise|
3    sensorsimulator_connect()
4    sensorsimulator_setDeviceOrientation(azimuth, pitch, roll, noise)
5  end
6
7  Given /^device orientation is landscape right$/ do
8    sensorsimulator_connect()
9    sensorsimulator_setDeviceOrientation('0', '0', '-90', '1')
10 end
```

In Quellcodefragment 5.4 sind beispielhaft zwei der Step-Definitions abgebildet. In Zeilen 2-5 ist eine Step-Definition dargestellt, die eine beliebige Spezifikation der Geräteorientierung in Roll-, Gier- und Nickwinkel erlaubt. Konkret sind die vier numerischen Parameter Roll-, Nick- und Gierwinkel sowie ein Wert für das durch den Sensorsimulator künstlich zu erzeugende Rauschen zu spezifizieren. Der Wert für das Sensorrauschen ist aus technischen Gründen als numerischer Wert ausgeführt, hat aber bool'sche Semantik (d. h. kein künstliches Rauschen wenn null, sonst künstliches Rauschen). Die Rahmenbedingungen des künstlichen Rauschens werden aus den zuvor erstellten Profilen von Sensoren aus Geräten abgeleitet.

In den Zeilen 7-10 wird ein Makro definiert, das es erlaubt eine diskrete Geräteorientierung in einer Testfallspezifikation vorzugeben. Korrespondierend definiert das Metamodell zur Kontextmodellierung Modellierungselemente, um die Geräteorientierung im UML-Aktivitätsdiagramm des SUT abzubilden. Bei der Generierung von plattformspezifischen Testfälle (vgl. Abschnitt 5.4) werden aus dem Testfallmodell entsprechende Calabash-Testschritte erzeugt. Eine Auflistung der im Rahmen dieser Dissertation erstellten Calabash-Step-Definitions zur Abbildung von Kontextparametern in Testfällen ist in Anhang A dokumentiert.

5.6 Zusammenfassung

Ein modellbasierter Ansatz zu Testautomatisierung setzt voraus, dass alle zur Testdurchführung notwendigen Testdaten im Modell enthalten sind. Besonderheit der hier untersuchten Methode ist, dass neben traditionellen Testdaten (d. h. Eingabedaten, erwartete Ausgabedaten, Interkationsbeschreibungen) zusätzlich Kontextparameter, insbesondere physikalische Parameter (vgl. Abschnitt 3.1.2.1), bei der Testdurchführung verwendet werden. Hierzu ist es notwendig, dass auch solche Testdaten als Teil der Systemmodellierung zusammen mit dem SUT modelliert werden. Abschnitt 5.2 thematisiert die Modellierung von Testfällen für mobile, kontextsensitive Anwendungen. Insbesondere wird in Abschnitt 5.2.2 ein Metamodell zur Modellierung von Kontext vorgestellt, welches zusammen mit dem in Abschnitt 5.2.3 diskutierten Metamodell zur Testfallmodellierung die Grundlage zur Integration von Kontext- und Testdaten in die Systemmodellierung bildet. Die Integration der Testfallmodellierung in die Systemmodellierung wird durch das in Abschnitt 5.2.4 vorgestellte UML-Profil realisiert.

Die Generierung plattformunabhängiger Testfallmodelle wird in Abschnitt 5.3 im Detail diskutiert. Bei UML-Aktivitätsdiagrammen handelt es sich um gerichtete Graphen, die sich durch die Besonderheit auszeichnen, dass sie zwei unterschiedliche Arten Verzweigungsknoten enthalten, nämlich solche zur Modellierung von Alternativen und solche zur Modellierung von Parallelisierungen des Kontrollflusses. Hierdurch können zur Generierung von Testfällen (d. h. die Menge der Pfade vom Initialelement bis zum Ende des Kontrollflusses) aus Aktivitätsdiagrammen Tiefen- und Breitensuchealgorithmen nicht angewendet werden (vgl. Abschnitt 5.3.3.1). Aus diesem Grund wird hier der Ansatz verfolgt, UML-Aktivitätsdiagramme zunächst in Petri-Netze zu transformieren. Der Erreichbarkeitsgraph der möglichen Markierungen eines Petri-Netzes ist ein gerichteter Graph, der zwar noch alternative Verzweigungen enthält aber keine parellelisierenden Verzweigungsknoten. Aus diesem Grund können Tiefen- und Breitensuchealgorithmen auf den Erreichbarkeitsgraphen angewendet werden, um Pfade durch das zugrundeliegende Aktivitätsdiagramm zu berechnen.

Der Prozess der Transformation von UML-Aktivitätsdiagramm zu Petri-Netz wurde in Abschnitt 5.3.3.1.2 im Detail diskutiert. Dieser Vorgang generiert zunächst eine Menge von Pfaden aus dem Aktivitätsdiagramm. In einem zweiten Schritt (vgl. Abschnitt 5.3.3.2) werden individuelle Elemente des Aktivitätsdiagramms zu Modellierungselementen des Metamodells zur Testfallmodellierung (vgl. Abschnitt 5.2.3.2) transformiert.

In Abschnitt 5.4 wurde die Transformation von plattformunabhängigen zu plattformspezifischen Tests am Beispiel der Technologie Calabash (vgl. Abschnitt 2.5.2.6) thematisiert. Diese M2T-Transformation überführt den Inhalt des Testfallmodells in die Syntax des Calabash-Frameworks. Hierzu werden alle *Test::TestCase*-Elemente aus dem Testfallmodell von Anfang bis Ende traversiert und für jedes auf dem Pfad liegende Element eine Menge Code-Zeilen in Gherkin-Syntax erstellt. Durch diesen Schritt wird das Testfallmodell in ein Softwareartefakte überführt, welches direkt von der angepassten Calabash-Implementierung ausgeführt werden kann. Alternativ könnte hier eine M2T-Transformation in den Syntax einer anderen Testautomatisierungstechnologie, beispielsweise JUnit, implementiert werden. Hierdurch wäre es aber erforderlich, eine Implementierung des JUnit-Frameworks zu erstellen, die in der Lage ist, Kontextparameter bei der Testdurchführung zu simulieren.

Die automatisierte Ausführung von Tests wurde in Abschnitt 5.5 erörtert. Das Testautomatisierungs-Framework Calabash für die Plattform Android stellt in seiner Standardimplementierung keine Step-Definitions zur Verwendung physikalischer Kontextparameter in Tests bereit. Auch existieren keine Schnittstellen zur Injektion künstlicher Sensordaten als Testdaten in der Standardimplementierung des Android-Betriebssystems, die mit Testautomatisierungswerkzeugen adressiert werden können. In Abschnitt 5.5.2 wurde erläutert, wie das Android-Betriebssystem um solche Schnittstellen erweitert wurde. Konkret wurden oberhalb des Sensor-HAL Netzwerkschnittstellen geschaffen, die es zulassen, Sensordaten als Testdaten von einer externen Quelle in das System einzuspeisen, wo sie anstelle der tatsächlich von den Sensoren des Geräts erzeugen Daten an individuelle Apps propagiert werden. In Abschnitt 5.5.3 wurde diskutiert, welche zusätzlichen Step-Definitions in einer modifizierten Implementierung des Calabash-Frameworks erstellt wurden. Diese ermöglichen die Integration physikalischer Kontextparameter in Calabash-Tests.

Im Zusammenwirken der in dieser Dissertation untersuchten Technologie zur Generierung von Tests aus Systemmodellen, dem um Testschnittstellen erweiterten Android-Betriebssystem und dem modifizierten Calabash-Framework entsteht ein Ansatz zur modellbasierten Testautomatisierung, bei dem Tests aus angereicherten UML-Aktivitätsdiagrammen erzeugt werden und anschließend automatisch ausgeführt werden.

Kapitel 6

Evaluierung

In Kapitel 5 wurden eine Methode und ein Konzept für ein Werkzeug zur Testautomatisierung für mobile, kontextsensitive Anwendungen vorgestellt. Die Algorithmen zur Generierung von Tests aus UML-Aktivitätsdiagrammen (Abschnitt 5.3), zur Generierung plattformspezifischer Tests (Abschnitt 5.4) und zur automatisierten Durchführung von Tests (Abschnitt 5.5) wurden in den vorangegangenen Abschnitten im Detail diskutiert und wird im Kapitel 6 evaluiert.

Der in dieser Dissertation vorgestellte Ansatz zur Testautomatisierung für mobile, kontextsensitive Anwendungen umfasst mehrere Bausteine. Die Erbringung des Nachweises, dass der Ansatz korrekt, wirksam und nützlich ist, erfordert die Bestätigung der in Abschnitt 1.2.4 aufgestellten Forschungshypothese, dass die Berücksichtigung von Kontextinformationen im Systementwurf durch Bereitstellung von Modellierungsmethoden und Werkzeugen dabei hilft, Abhängigkeiten individueller Softwareartefakte von Kontextinformationen herauszustellen, deren Bedeutung für die Funktion einer Anwendung zu verdeutlichen und so die Grundlage zum automatisierten Erstellen und Durchführen von kontextsensitiven Testfälle bildet.

Die Validierung der Methode erfordert, für jeden individuellen Baustein des Lösungskonzeptes (vgl. Abschnitt 5.1) den Nachweis der Korrektheit zu erbringen. Das heißt, es ist zu zeigen, dass die Generierung plattformunabhängiger Testmodelle solche Tests erzeugt, die für das Zielsystem gültig sind und es adäquat abdecken (vgl. Abschnitt 5.3.2). Für die Generierung plattformspezifischer Test ist zu zeigen, dass die erzeugten Artefakte der Zieltechnologie, d. h. Calabash, gültige Repräsentierungen der zugrundeliegenden plattformunabhängigen Testmodelle sind. Letztlich ist zu zeigen, dass die Methode nützlich ist, d. h. dass im Vergleich zur Anwendung traditionellen Vorgehens beim Testen ein objektiv messbarer Nutzen entsteht. Die Nützlichkeit des Ansatzes im Sinne eines Vorteils ihrer Verwendung gegenüber traditionellen Methoden zur Erstellung und Durchführung von Tests kann nicht basierend auf den einzelnen Bausteinen erbracht werden, da diese zwar vom Grundsatz her einzeln innerhalb der Softwaretechnik anwendbar sind, zum Ziel der Steigerung der Effizienz des Testens durch Automatisierung aber nur wirksam sind, wenn sie in der in Abschnitt 5.1 vorgestellten Weise aufeinander aufbauend angewendet werden.

Weiterhin soll in Kapitel 6 untersucht werden, ob die in Abschnitt 5.5 diskutierte Technologie aus einem um Testschnittstellen erweiterten Android-Betriebssystem und einer angepassten Implementierung des Calabash-Frameworks geeignet ist, mobile, kontextsensitive Anwendungen effektiv und effizient zu testen.

Mit der Bewertung der Nützlichkeit einer Innovation befasst sich Rogers [301] in seiner Diffusionstheorie, die konkret den Prozess der Schöpfung, der Diffusion und der Akzeptanz von Innovationen zum Gegenstand hat. Rogers fokussiert hierbei nicht spezifisch auf das Themengebiet Softwaretechnik, sondern betrachtet Innovationen in einem ganzheitlichen sozioökologischen Zusammenhang, der zwar nicht in seiner Gesamtheit, aber doch in einigen Aspekten auf den Kontext dieser Dissertation übertragbar ist.

Rogers bezeichnet eine *Innovation* als eine Idee, eine Praxis (i. S. v. Anwendung, Gebrauch, Gepflogenheit, hier insb. Verfahren bzw. Methode) oder ein Objekt, die bzw. das von einem sozialen System als neu wahrgenommen wird. Die *Neuheit* der Innovation wird durch Rogers hierbei nicht als objektive Neuheit im Sinn einer kurzen Existenzdauer interpretiert, sondern als wahrgenommene Neuheit i. S. v. Kenntnis, Überzeugung oder Rezeption. Die grundsätzliche Anwendbarkeit des Begriffs *Innovation* auf den Kontext dieser Dissertation ergibt sich daraus, dass Rogers seine Argumentation primär auf technologische Innovationen stützt und die Begriffe *Innovation* und *Technologie* synonym verwendet. Als eine Technologie bezeichnet Rogers ein

> „*[. . .] design for instrumental action that reduces the uncertainty in the cause-effect relationships involved in achieving a desired outcome.*“ [301]

Die Terminologie *instrumental action* (engl. *instrumental*, als Mittel dienend; *action*, Handlung, Tat, Wirksamkeit) zielt hier primär auf den Prozess einer Handlung ab, deckt aber vom Grundsatz her sowohl eine Handlung i. S. v. Methode als auch ein gegenständliches Artefakt (d. h. Werkzeug) ab. Die in dieser Dissertation erarbeitete Methode sowie das implementierende Werkzeug sind Technologien (d. h. *instrumental action*) in diesem Sinne. Durch den Ansatz zur Testautomatisierung wird sowohl eine Methode, d. h. ein planmäßiges, systematisches Vorgehen zur Erreichung eines konkreten Ziels, definiert (vgl. Methode zur modellbasierten Testautomatisierung, Abschnitt 5.1) als auch ein Werkzeug zur Unterstützung der Testdurchführung implementiert (vgl. Abschnitt 5.5, Automatisierte Durchführung von Testfällen für kontextsensitive mobile Anwendungen).

Methode und Werkzeug dienen zur Reduzierung von Unsicherheit im Ursache-Wirkung-Zusammenhang beim Erreichen eines erwünschten Ergebnisses. Das erwünschte Ergebnis ist die Steigerung der Effizienz und der Wirksamkeit von Softwaretests für mobile, kontextsensitive Anwendungen durch die Verwendung von Automatisierung (Ursache) bei der Erstellung und Durchführung von Tests, bei gleichzeitiger Reduzierung von Aufwand und Fehleranfälligkeit (Wirkung) gegenüber manueller Testausführung.

Die in dieser Dissertation vorgestellte Methode zur Testautomatisierung ist in diesem Sinne eine *Technologie*. Sie wird zur *Innovation*, weil sie in einem sozialen System, konkret im Umfeld der Softwaretechnik, speziell im Bereich Testen mobiler Anwendungen, eine neue Praxis gemäß der von Rogers [301] zugrunde gelegten Definition ist. Die einzelnen Lösungselemente sind nicht neu i. S. einer kurzen Existenzdauer. Tatsächlich werden Technologien zur Testautomatisierung mit Ausnahme der technologiegestützten Simulation von Kontextparametern, insb. Sensordaten, bereits seit vielen Jahren und mit unterschiedlichen spezifischen Zielsetzungen erforscht (vgl. Stand der Wissenschaft und Technik, Kapitel 2). Technologien, die mobile Anwendungen im Kontext der Smartphone-Ära adressieren, können hingegen objektiv als

neu beurteilt werden. Im Jahr 2016 existieren mobile Anwendungen auf einem industriellen Niveau außerhalb hochspezialisierter Nischen für einen Zeitraum, der ungefähr auf eine Dekade beziffert werden kann. Vorherige wissenschaftliche Forschungen im Umfeld mobiler, kontextsensitiver Anwendungen sind deshalb nicht oder nur mit Einschränkungen auf die Anforderungen mobiler Systeme der Smartphone-Ära übertragbar.

Die gemeinsame, aufeinander aufbauende Verwendung der in dieser Dissertation erarbeiteten Lösungselemente, d. h. Modellierung von Verhaltensaspekten als Bestandteil der Systemmodellierung, Verfeinerung von Systemmodellen mit testspezifischen Details, Generierung plattformunabhängiger Testmodelle aus Systemmodellen, Transformation von Testmodellen in technologiespezifische Tests sowie die automatisierte Ausführung dieser Tests mit Simulation des relevanten Kontext (vgl. Abschnitt 5.1), wurde in dieser Form und in diesem Umfang in der wissenschaftlichen Literatur bislang nicht adressiert (vgl. Kapitel 2). Die von Rogers definierten Kriterien an Innovationen (d. h. objektive und, bezüglich eines sozialen Systems, subjektive Neuigkeit) werden deshalb bereits pro forma erfüllt.

Rogers definiert darüber hinaus weitere Charakteristiken von Innovationen, nämlich (1) relativer Vorteil, (2) Kompatibilität, (3) Komplexität, (4) Erprobbarkeit sowie (5) Beobachtbarkeit.[1] Die Validierung des in dieser Dissertation vorgestellten Vorgehens zur Testautomatisierung mobiler, kontextsensitiver Anwendungen umfasst folglich neben der technischen Evaluierung der Korrektheit der einzelnen Lösungselemente und deren Zusammenwirken ebenfalls eine Beurteilung hinsichtlich dieser Kriterien.

Der relative Vorteil einer Innovation ergibt sich aus einem Vorteil bzw. einer überlegenen Nützlichkeit einer Technologie gegenüber einer bereits existierenden bzw. sich in Gebrauch befindlichen Technologie. Nach Rogers muss sich dieser Vorteil nicht zwingend in objektiven ökonomischen Metriken manifestieren. Es ist ausreichend, wenn die Innovation im Kontext des adressierten sozialen Systems gegenüber existierenden Technologien als vorteilhaft wahrgenommen wird. Im konkreten Bezug auf die hier untersuchte Technologie zur Testautomatisierung für mobile Anwendungen kann ein relativer Vorteil gegenüber traditionellen Methoden vorausgesetzt werden. Vergleichbare Technologien, die Aktivitäten des Testens von Software vom Entwurf bis hin zur Implementierung durch Automatisierung unterstützten, sind im Jahr 2016 nicht existent. Existierende Technologien (vgl. Technologisches Umfeld, Abschnitt 2.5) decken Testprozesse i. d. R. nur anteilig ab, beispielsweise mit Werkzeugen zur automatisierten Testausführung (z. B. JUnit, UIAutomator, Robotium), die keine Unterstützung beim Erstellen von Tests geben oder Technologien zur Generierung von Tests auf Basis von Softwaremodellen, Quellcode oder anderen Artefakten (z. B. *GUI-Crawler* [6] basierend auf der Analyse des UI), die keine Unterstützung beim Ausführen von Tests geben.

Die Simulation von Sensordaten beim Testen mobiler Anwendungen der Smartphone-Ära nimmt hierbei eine herausgehobene Stellung ein. Einerseits ist eine manuelle Reproduktion von Testdaten quasi ausgeschlossen (vgl. Abschnitte 1.2.3 und 4.2.4), andererseits existieren nur wenige Werkzeuge mit wesentlichen Einschränkungen zur automatisierten Reproduktion von Testdaten. Insbesondere im Kontext mobiler Anwendungen existieren im Jahr 2016 keine Technologien, die Testen mit künstlichen Sensordaten ermöglichen, ohne dass hierzu massive

[1] Übersetzt a. d. Engl. *relative advantage*, *compatiblity*, *complexity*, *triability*, *observability*, Rogers [301, S. 14, 15, 210ff]

Eingriffe am SUT (d. h. Modifikation des Quellcode, die nach dem Testen vor der Auslieferung rückgängig gemacht werden) notwendig werden (vgl. SensorSimulator™ [269]). In dieser Hinsicht ist der hier vorgestellte Ansatz zur Testautomatisierung nicht nur in der Wahrnehmung des soziökologischen Umfelds vorteilhaft, sondern er ist objektiv i. S. v. Rogers vorteilhaft, weil er eine bislang nicht existierende Technologie implementiert, die das Bedürfnis nach einer Methode und einem Werkzeug zur effizienten Unterstützung von Testaktivitäten adressiert.

Der in dieser Dissertation vorgestellte Ansatz zur Testautomatisierung vereinigt Technologien zur modellbasierten Generierung von Tests aus Systemmodellen, zur automatisierten Ausführung von Tests und zur Simulation von Kontextparametern in einem gemeinsamen Konzept. Er repräsentiert deshalb einen relativen Vorteil gegenüber existierenden Technologien, die diese Funktionsumfang nicht ganzheitlich abdecken. Hinsichtlich des Funktionsumfangs ist dieser Vorteil objektiv messbar: die hier vorgestellte Technologie ermöglicht eine Automatisierung der Testfallerstellung, deren automatisierte Ausführung und die Simulation von Kontextparametern innerhalb eines konsistenten und kohärent aufeinander abgestimmten Werkzeugkastens (d. h. UML-Profil zur Modellierung von Kontext- und Testdaten, Transformationswerkzeug zur Testfallgenerierung, angepasstes Android-Betriebssystem). Ein vergleichbares Konzept existiert im Jahr 2016 nicht. Die Vorteile gegenüber der bislang vorherrschenden manuellen Testausführung sind objektiv messbar, da durch das hier untersuchte Konzept erstmals ein Werkzeug verfügbar ist, das es erlaubt, Kontextinformationen, insb. Sensordaten, zur Testausführung zu reproduzieren, ohne dass hierzu Modifikationen am SUT vorgenommen werden müssen. Konkret ergibt sich hier zum einen der Vorteil, dass durch die Technologie zur Simulation von Sensordaten eine zuverlässige Reproduktion von Testdaten möglich wird, wodurch insbesondere für Regressionstests aussagefähige Testergebnisse erzeugt werden. Zum anderen ergibt sich der Vorteil, dass Tests in kürzerer Zeit absolviert werden können, da eine maschinelle Testausführung im Vergleich mit einer manuellen Testausführung i. d. R. schneller arbeitet, keine Erholungsphasen benötigt und durch Parallelisierung skaliert werden kann.

Rogers definiert weiterhin das Kriterium Kompatibilität als Metrik, mit der die Konsistenz einer Innovation mit existierenden Werten, Erfahrungen und Bedürfnissen potenzieller Anwender quantifiziert wird. Innovationen, die in besonderem Maße von existierenden Werten, Denkmodellen und Erwartungen eines sozialen Systems divergieren, haben eine geringe Aussicht auf Akzeptanz. Eine Innovation, die von der intendierten Zielgruppe nicht akzeptiert wird, findet keine Verbreitung in der Zielgruppe, wodurch wiederum kein Anreiz zur Integration mit anderen Technologien entsteht. Selbst wenn die Innovation ihren Bestimmungszweck in korrekter Weise erfüllt, ist sie dann nicht nützlich und wird letztlich aufgrund des Fehlens technologischer Einbettung obsolet.

Der in dieser Dissertation vorgestellte Ansatz zur Testautomatisierung ist gemäß Rogers Definition kompatibel zu Werten und Denkmodellen der intendierten Zielgruppe. Die in Kapitel 2 dargelegte Entwicklung themenverwandter Methoden und Technologien sind in ein Umfeld einzuordnen, in dem das grundlegende Denkmodell, dass die Generierung und automatisierte Ausführung von Tests gegenüber manuellem Erstellen und Ausführen von Tests vorteilhaft ist, fundamental vorausgesetzt wird. Der hier untersuchte Ansatz fügt sich in den wissenschaftlich erforschten und technologisch erschlossenen Kontext der Methoden und Tech-

nologien zur Testautomatisierung in Form einer konsequenten Weiterentwicklung, Optimierung und Anpassung auf eine spezifische Zieltechnologie (d. h. mobile Anwendungen) ein. Hierdurch wird ebenfalls der von Rogers postulierten Kompatibilität mit vorhergehenden Innovationen und Technologien Rechnung getragen, beispielsweise durch Verwendung der UML und UML-Profile als Basis der Testmodellierung, die zum etablierten Kanon der Werkzeuge der Softwaretechnik gehören.

Rogers argumentiert weiterhin, dass eine Innovation besonders dann als kompatibel akzeptiert wird, wenn sie eine Weiterentwicklung existierender Innovationen repräsentiert. Die in dieser Dissertation untersuchte Methode zur Testautomatisierung erfüllt dieses Kriterium, weil sie existierende Methoden der Testautomatisierung mit einem Fokus auf einen Bereich der Softwaretechnik (d. h. mobile Anwendungen) ausdehnt, in welchem aufgrund seiner relativen Neuheit bisher trotz hoher Innovationskraft und signifikantem Marktpotenzial bislang nur geringer praxisrelevanter Fortschritt erzielt wurde. Als Ursache kann hier der hohe Wettbewerbsdruck vermutet werden, der Plattformbetreiber und Gerätehersteller veranlasst, sich anstelle einer intensiven Unterstützung aller Aktivitäten in Softwareprozessen, insb. von Testaktivitäten, auf eine sprunghafte Weiterentwicklung technologischer Geräte-Features zu fokussieren, um sich von Wettbewerbern abzusetzen.

Weiterhin ist der hier untersuchte Ansatz zur Testautomatisierung kompatibel mit den Bedürfnissen einer Zielgruppe (vgl. Rogers [301]). Als Zielgruppe wird hierbei die Allgemeinheit aller Entwickler und Tester angenommen, die sich im Rahmen ihrer Tätigkeit mit der Entwicklung mobiler Anwendungen befassen. Das Bedürfnis manifestiert sich im Wunsch nach Methoden und Werkzeugen, die Aktivitäten des Testens von menschlicher Arbeitskraft entkoppeln – einerseits zur Steigerung der Effizienz, anderseits zur Sicherstellung der Qualität von Testaktivitäten.

Rogers definiert Komplexität als dritte geforderte Charakteristik einer Innovation. Sie beschreibt, ob eine Innovation als schwierig bzw. kompliziert zu verstehen und zu verwenden ist. Die Verbreitung einer Innovation innerhalb einer Zielgruppe wird wesentlich durch ihre Komplexität beeinflusst. Rogers argumentiert, dass Innovation mit geringer Komplexität, d. h. einfach verständliche Innovation mit steiler Lernkurve, mit größerem Erfolg von der Zielgruppe akzeptiert werden, wohingegen kompliziert zu verstehende Innovation mit geringer Lerneffizienz im Vergleich eher abgelehnt werden.

Der in dieser Dissertation untersuchte Ansatz zur modellbasierten Testautomatisierung für mobile Anwendungen hat zum Ziel, die Komplexität von Testaktivitäten in den Dimensionen Aufwand und Technologiebeherrschung zu reduzieren. Dem hier vorgestellten ganzheitlichen Ansatz der Testautomatisierung von der Modellierung des SUT, der Modellierung von Testdaten, der Testfallerstellung bis zur Testdurchführung mit Simulation von Kontextparametern, stehen nach dem Stand der Technologie im Jahr 2016 mehrheitlich manuelle Methoden und Technologien gegenüber, sofern sie überhaupt existent sind. In Abschnitt 2.3 sowie in Abschnitt 2.4 wurden einige Methoden und Technologien zur Generierung von Softwaretests und deren automatisierte Ausführung diskutiert. Die dort untersuchten Methoden unterscheiden sich entweder fundamental in der technologischen Basis (z. B. Testgenerierung durch UI-Crawling, Amalfitano et al. [6], Abschnitt 4.3.2.1.2) oder sind aufgrund ihrer technologischen Basis (d. h. Verwendung der UML, Modellierung von Testdaten als UML-

Objektdiagramm) in einer ähnliche Komplexitätsklasse einzuordnen. Der hier untersuchte Ansatz repräsentiert jedoch eine Reduzierung der Komplexität bei der Simulation von Kontextparametern. Zum einen existieren neben den Werkzeugen der SDKs mobiler Plattformen und dem SensorSimulator™ [269] gegenwärtig keine Simulationswerkzeuge, zum anderen repräsentieren diese Werkzeuge lediglich einen technischen Zugang zum SUT, ohne jedoch selbst realitätsnahe Testdaten (vgl. Abschnitt 5.5.1, Abschnitt 5.5.2 sowie Abschnitt 5.5.3) erzeugen zu können. In dieser Hinsicht repräsentiert der hier vorgestellte Ansatz eine Reduzierung der Komplexität der Bereitstellung und Reproduktion von Testdaten durch die Simulation von Sensordaten (vgl. Abschnitt 5.5.3.1). Demgegenüber steht gegenwärtig lediglich der Versuch, Sensordaten manuell zu reproduzieren, z. B. durch grobe Orientierung des Geräts anhand zuvor definierter Kriterien, ggfs. unter Verwendung von Hilfmitteln (z. B. klassischer Kompass, Winkelmesser), wobei Abweichungen durch externe Einflüsse hingenommen werden müssen.

Als viertes Kriterium nennt Rogers die Erprobbarkeit einer Innovation. Sie beschreibt, in welchem Umfang und mit welchem Aufwand es einem interessierten Anwender möglich ist, mit der Innovation zu experimentieren, um sie hinsichtlich ihrer Eignung zur Erreichung eines bestimmten Ziels zu überprüfen. Erprobbarkeit ist ein Kriterium, das von unterschiedlichen Anwendern unterschiedlich gewichtet wird. Sogenannte *Early Adopter* (engl. frühzeitige Anwender) bewerten die Erprobbarkeit einer Innovation i. d. R. mit höherem Gewicht als sogenannte *Laggards* (engl. Nachzügler) [301].

Der hier untersuchte Ansatz zur Testautomatisierung für mobile Anwendungen stellt verhältnismäßig hohe Hürden an Erprobbarkeit durch Dritte, da es sich um eine Familie von Technologien handelt, die bereits jeweils einzeln beim Anwender einen hohen Durchdringungsgrad und Einbettung in Werkzeuge und Methoden zur Entwicklung mobiler Anwendungen erfordern. Ein Early Adopter muss zur Erprobung des hier untersuchten Ansatzes zur Testautomatisierung über Kenntnisse der Anwendungsentwicklung für die Plattform Android verfügen, muss über Kenntnisse in der Modellierung von Softwaresystemen mit UML und zusätzlich über Erfahrungen beim Testen kontextsensitiver Anwendungen verfügen. Darüber hinaus ist ein versierter Umgang mit den Werkzeugen der SDKs der Plattform Android erforderlich, beispielsweise zum Aufspielen der in Abschnitt 5.5.2 vorgestellten angepassten Implementierung des Android-Betriebssystems auf ein Gerät oder eine Emulator-Konfiguration. Einem Anwender, der diese Kriterien erfüllt, ist es möglich, den hier vorgestellten Ansatz praktisch zu erproben. Eine tatsächliche Quantifizierung der Erprobbarkeit des Ansatzes könnte durch eine repräsentative Studie mit Anwendern der Kategorie Early Adopter erfolgen, die im Rahmen eines kontrollierten Versuchs mit der hier untersuchten Testautomatisierungstechnologie experimentieren und anschließend zu objektiven und subjektiven Eindrücken befragt werden. Aufgrund der hohen Anforderungen an eine solche Studie konnte eine entsprechendes Experiment im Rahmen dieser Dissertation jedoch nicht durchgeführt werden.

Das fünfte von Rogers diskutierte Kriterium an Innovationen ist Beobachtbarkeit. Sie beschreibt, in welchem Ausmaß die Auswirkungen einer Innovation für Dritte erkennbar werden. Dieses Kriterium ist insbesondere für eine Diffusion der Innovation von Early Adopters hin zu Laggards relevant, da eine Innovation, die bereits bei Early Adopters keine Akzeptanz findet, kaum eine Chance zur weiteren Durchdringung des sozioökonomischen Umfelds bis hin zu den Laggards hat.

Die Auswirkungen des hier untersuchten Ansatzes zur modellbasierten Testautomatisierung sind für Dritte unmittelbar erkennbar. Für den Komplex Testfallgenerierung muss zum Nachweis der Beobachtbarkeit gezeigt werden, dass durch die Testfallgenerierung eine höhere Abdeckung des SUT erzielt wird, als bislang durch eine manuelle Testfallerstellung erzielt werden konnte. Zur Erbringung dieses Nachweises wird im Rahmen der Evaluierung eine Fallstudie an der App Mobiler Taxiruf durchgeführt. Für den Komplex Kontextsimulation werden ebenfalls Fallstudien durchgeführt (vgl. Abschnitt 6.5.2 bis Abschnitt 6.5.3), wobei automatisierte Tests an existierenden Apps durchgeführt werden, die Sensordaten zur Realisierung eines spezifischen Anwendungsfalls verwenden. Gezeigt wird, dass die in den Fallstudien untersuchten Apps die durch die Testautomatisierungstechnologie vorgegebenen Testdaten für Sensoren tatsächlich korrekt verarbeiten.

Das Kapitel Evaluierung ist in die Abschnitte Methode und Kriterien zur Evaluierung, prototypische Implementierung, Validierung der Testfallgenerierung, Fallstudien und schließlich Bewertung im Vergleich zu alternativen Technologien unterteilt. Das Vorgehen, die Methode und die zu bewertenden Kriterien bei der Durchführung der Evaluierung werden in Abschnitt 6.1 vorgestellt. Die Durchführung der Evaluierung basiert auf der prototypischen Implementierung der Methode zur Generierung von Testfällen sowie einer prototypischen Implementierung der Testschnittstelle im Android-Betriebssystem sowie des angepassten Calabash-Frameworks. Diese werden in Abschnitt 6.2 im Detail diskutiert. Die Evaluierung der Generierung von Testfällen ist Gegenstand von Abschnitt 6.3. In Abschnitt 6.5 werden Fallstudien zur Anwendung des Ansatzes zur Testautomatisierung durchgeführt. In Abschnitt 6.6 wird der Ansatz ingesamt bewertet.

6.1 Methode und Kriterien zur Evaluierung

Die Evaluierung des in dieser Dissertation erarbeiteten Ansatzes zur Testautomatisierung für mobile, kontextsensitive Anwendungen erfolgt auf der Basis einer Implementierung der in Kapitel 5 vorgestellten Algorithmen, Methoden und Werkzeuge zur Modellierung von Testfällen, zur Generierung plattformunabhängiger Testfällen, zur Generierung plattformspezifischer Testfällen und zur automatisierten Durchführung dieser Testfälle mit Simulation von Sensordaten als Kontextparameter.

Die erste und zweite Phase der Testfallgenerierung, d. h. die Modellierung von Verhaltensaspekten als Bestandteil der Systemmodellierung sowie die Verfeinerung von Systemmodellen mit testspezifischen Details sind keine individuellen Ziele der Validierung. Die grundsätzliche Möglichkeit, Testdaten durch Anwendung von UML-Profilen in Systemmodellen zu integrieren (vgl. Abschnitt 5.1), wurden bereits in anderen wissenschaftlichen Arbeiten zum Thema Testautomatisierung (vgl. Abschnitt 2.2.2 sowie Abschnitt 2.4) untersucht. Die Validierung der Verwendbarkeit des in dieser Dissertation erstellten UML-Profils (vgl. Abschnitt 5.2.4) in Verbindung mit den Metamodellen zur Kontext- und Testdatenmodellierung (vgl. Abschnitt 5.2.2.2 sowie Abschnitt 5.2.3.2) erfolgt implizit durch die Validierung der dritten Phase der Testfallgenerierung, der Generierung plattformunabhängiger Testmodelle.

Gegenstand der Validierung der dritten Phase, der Testfallgenerierung plattformunabhängiger Testfälle, ist die Transformation von UML-Aktivitätsdiagrammen zu plattformun-

abhängigen Tests. Zur Durchführung der Validierung wurden Fragmente von UML-Aktivitätsdiagrammen mit geringer Komplexität erstellt und sukzessive zu komplexen Modellen zusammengesetzt. Auf diese Fragmente und Modelle wurde zunächst der Algorithmus zur Generierung plattformunabhängiger Testfallmodelle angewendet und die Zwischenergebnisse der Transformation (d. h. das Petri-Netz, dessen Erreichbarkeitsgraph sowie das Testfallmodell) auf Vollständigkeit und Korrektheit überprüft. Das Ergebnis der Transformation, d. h. das Testfallmodell, wurde zusätzlich auf Konformität zum Metamodell zur Testfallmodellierung (vgl. Abschnitt 5.2.3.2) überprüft und die im Modell enthaltenen Ausführungspfade im Hinblick auf das in Abschnitt 5.3.3.1.3 diskutierte $C_{PS_{ACD}}$-Abdeckungskriterium (Permutationssequenzüberdeckung) überprüft. Ziel dieses Vorgehens ist der Nachweis, dass die Transformation von UML-Aktivitätsdiagrammen unter Berücksichtigung der in Abschnitt 5.3.3.1.1 diskutierten Anforderungen an Aktivitätsdiagramme stets im Sinne des Metamodells zur Testfallmodellierung (vgl. Abschnitt 5.2.3.2) korrekte Testfallmodelle erzeugt. Kriterien zur Bewertung der Transformation zu plattformunabhängigen Testfallmodellen ist einerseits die Erfüllung des geforderten Abdeckungskriteriums und andererseits Vollständigkeit, Korrektheit und Metamodellkonformität der resultierenden Testfallmodelle.

Die Generierung technologiespezifischer Testfälle aus dem Testfallmodell, konkret Tests in Calabash-Syntax, ist Teil der Phase vier des Generierungsprozesses. Im Rahmen der Validierung wird hier ebenfalls für die im vorherigen Schritt der Validierung erstellten Modelle überprüft, ob i. S. der in dieser Dissertation erstellten Implementierung des Calabash-Frameworks gültige und korrekte Calabash-Tests aus dem Testfallmodell generiert werden. Im Fokus steht hierbei Fokus die technische Korrektheit der M2T-Transformation. Hierin manifestiert sich zugleich das Bewertungskriterium der Validierung. Die Generierung der Calabash-Tests ist genau dann valide, wenn syntaktisch gültige Calabash-Step-Definitions erzeugt werden, diese das Testfallmodell inhaltlich korrekt repräsentieren und durch das modifizierte Calabash-Framework ausgeführt werden können.

Zusätzlich zur Validierung der Testfallgenerierung wird die Implementierung des modifizierten Calabash-Frameworks dahingehend validiert, ob die durch die Testfallgenerierung erstellten Tests korrekt durch das Werkzeug zur Testautomatisierung interpretiert werden. Im Rahmen der Validierung wurde die Anwendung Mobiler Taxiruf vollständig modelliert und prototypisch implementiert. Das Modell der App wurde unter Verwendung der Metamodelle zur Kontext- und Testfallmodellierung und mit Hilfe des UML-Profils zur Integration der Testfallmodellierung in die Systemmodellierung um Testdaten angereichert. Nach der Testfallgenerierung wurden die erzeugten Testfälle zunächst manuell auf Vollständigkeit, Korrektheit und Erfüllung des $C_{PS_{ACD}}$-Abdeckungskriteriums überprüft. Im Anschluss wurde die Anwendung Mobiler Taxiruf mit der generierten Testfällen getestet. Kriterium für die Bewertung der Validität des Ansatzes ist die Feststellung, dass die prototypische Implementierung der Anwendung Mobiler Taxiruf durch die automatisch generierten Testfälle getestet wird, ohne dass hierzu über die Modellierung der Anwendung und der Testdaten hinaus weitere manuelle Interaktionen bei der Testausführung notwendig sind.

Der hier untersuchte Ansatz zur Testautomatisierung umfasst neben der Generierung von Tests aus UML-Aktivitätsdiagrammen ebenfalls die automatisierte Durchführung von Tests mit Simulation von Kontextparametern, insb. Sensordaten. Es ist deshalb ebenfalls

Gegenstand der Validierung zu überprüfen, ob die durch die Simulationstechnologie (vgl. Abschnitt 5.5.1) erzeugten Testdaten reale Sensordaten in solcher Weise adäquat repräsentieren, dass durch die Simulation von Sensordaten keine Artefakte in den Testprozess eingeführt werden, durch welche die Gültigkeit von Testergebnissen kompromittiert werden. Kriterium für die Bewertung der Validität der Simulationstechnologie ist deshalb der Nachweis, dass die simulierten Sensordaten nicht nur technisch korrekt in das SUT übertragen werden, sondern zudem eine adäquate Ähnlichkeit zu realen Sensordaten aufweisen. Der Nachweis der Korrektheit der Simulation von Sensordaten wird erbracht, indem zum einen durch ein Experiment sichergestellt wird, dass einerseits bei der Übertragung der simulierten Sensordaten von Calabash ins SUT keine Daten korrumpiert werden und andererseits die im SUT gemessenen Daten dieselben mathematischen und technischen Eigenschaften aufweisen wie reale Sensordaten.

6.2 Prototypische Werkzeugimplementierung

Die in Kapitel 6 durchgeführte Validierung des Ansatzes zur Generierung und Durchführung von Tests für mobile, kontextsensitive Anwendungen erfolgt auf Basis einer prototypischen Implementierung der in den Abschnitten 5.3 und 5.4 vorgestellten Algorithmen zur Transformation von UML-Aktivitätsdiagrammen zu plattformunabhängigen Testfallmodellen und anschließender M2T-Transformation zu plattformspezifischen Tests für die Automatisierungtechnologie Calabash.

Konkret wurden mehrere Softwareartefakte erarbeitet: (1) eine Implementierung des in Abschnitt 5.2.2.2 diskutierten Metamodells zur Kontextmodellierung, (2) eine Implementierung des in Abschnitt 5.2.3.2 diskutierten Metamodells zur Testfallmodellierung, (3) eine Implementierung des UML-Profils zur Integration der Testfallmodellierung in die Systemmodellierung mit der UML (vgl. Abschnitt 5.2.4), (4) eine Implementierung eines Werkzeugs zur Transformation von UML-Aktivitätsdiagrammen zu Testfallmodellen, (5) eine Implementierung eines Werkzeugs zur Transformation von Testfallmodellen zu Calabash-Tests, (6) eine angepasste Implementierungen des Android-Betriebssystems und (7) eine modifizierte Implementierung des Calabash-Frameworks.

Um im Einklang mit Rogers [301] Anforderung an Innovationen kompatibel mit existierenden Denkmustern, Praktiken, Technologien und Werkzeugen zu sein, wurde die prototypische Implementierung unter Verwendung etablierter Werkzeuge und Programmiersprachen durchgeführt. Diese Kompatibilität manifestiert sich einerseits in der Auswahl der UML als technologische Basis für die Erstellung der verwendeten Metamodelle und andererseits in der Implementierung der Metamodelle sowie des UML-Profils auf Basis des *Eclipse Modeling Framework* (EMF). Die Eclipse-Plattform ist ein etabliertes Werkzeug in der Software Technik, das nicht nur als IDE vielfältig in der Praxis verwendet wird [120], sondern zudem zu einer verbreiteten *Rich-Client-Platform* (RCP) avanciert ist, die insbesondere durch eine Vielzahl von Werkzeugen und *Plug-Ins* (engl. Einschub, Zusatzmodul) im Umfeld der Modellierung mit UML etabliert ist.

Im Rahmen dieser Dissertation wurden die Metamodelle zur Kontext- und Testfallmodellierung sowie das UML-Profil zur Integration der Testfallmodellierung in die Systemmodellierung unter Verwendung des *Eclipse Modeling Framework* (EMF) modelliert. Das EMF ist ein

Werkzeug zur modellgetriebenen Softwareentwicklung auf Basis der Eclipse-Plattform. Mit ihm ist es möglich, Modelle und Metamodelle in einem graphischen UI zu modellieren und aus diesen Modellen Quellcode zu generieren, der anschließend in Software als Programmierbibliothek verwendet werden kann. Wichtigster Bestandteil des EMF ist das *Ecore*-Metamodell (vgl. Steinberg et al. [335]), welches die Beschreibung, Verarbeitung und Persistierung von Modellen ermöglicht. Es basiert auf der *Essential Meta Object Facility* (EMOF), welche wiederum eine Untermenge der durch die OMG eingeführten *Meta Object Facility* (MOF) ist. Die MOF spezifiziert ihrerseits den XMI-Standard, ein XML-Dialekt zum Austausch von Modellen zwischen Werkzeugen [43]. Die EMOF enthält alle Elemente der UML-*Infrastructure* [267] und ist somit geeignet, die UML innerhalb des EMF zu repräsentieren [161].

In den folgenden Abschnitten werden die einzelnen Softwareartefakte, die in ihrer Gesamtheit den in Kapitel 5 beschriebenen Prozess der Testfallgenerierung aus UML-Aktivitätsdiagrammen abbilden, beschrieben.

6.2.1 Implementierung der Metamodelle

Basierend auf dem EMF wurden im Rahmen der Implementierung des Werkzeugprototypen das in Abschnitt 5.2.2.2 beschriebene Metamodell zur Kontextmodellierung sowie das in Abschnitt 5.2.3.2 beschriebene Metamodell zur Testfallmodellierung unter Verwendung des Ecore-Metamodells modelliert.

Dargestellt ist in Abbildung 6.1 ein Bildschirmabdruck der Eclipse-IDE, in welcher der graphische Editor des EMF dazu verwendet wurde, das in Abschnitt 5.2.2.2 diskutierte Metamodell zur Kontextmodellierung in einem maschinenlesbaren Format zu manifestieren. Aus dem im graphischen Editor erstellten Metamodell zur Kontextmodellierung wurde durch Verwendung des EMF Quellcode generiert, welcher durch das Eclipse-Plug-In zur Testfallgenerierung (vgl. Abschnitt 6.2.3) verwendet wird, um während des Transformationsprozesses individuelle Kontextartefakte (z. B. Instanzen vom Typ *Context::WGS84Location*) abzubilden.

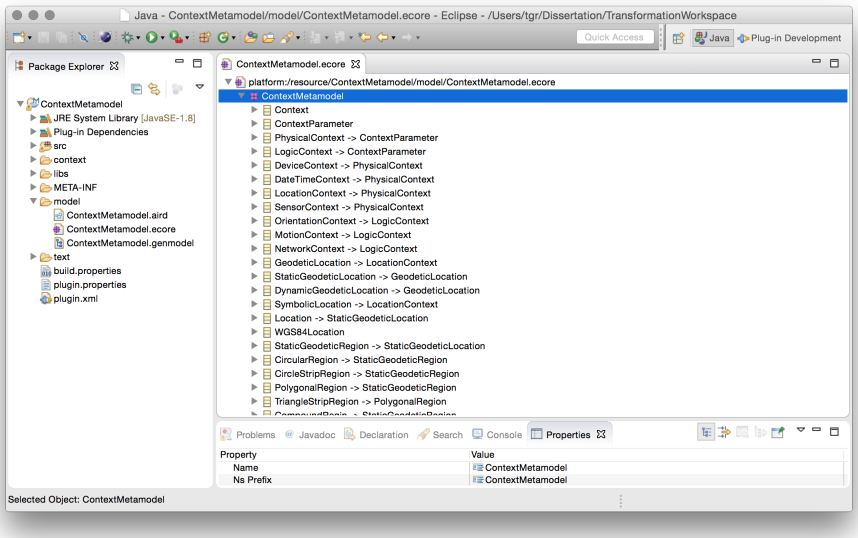


Abbildung 6.1: Bildschirmabdruck des im EMF modellierten Metamodells zur Kontextmodellierung

Analog zum Metamodell zur Kontextmodellierung wurde in identischer Weise das in Abschnitt 5.2.3.2 entworfene Metamodell zur Testfallmodellierung implementiert (vgl. Abbildung 6.2). Auch hier wurde der graphische Editor dazu verwendet, die individuellen Klassen des Metamodells zur Testfallmodellierung zu modellieren und aus diesem Modell im Anschluss durch Verwendung den EMF Quellcode zu generieren.

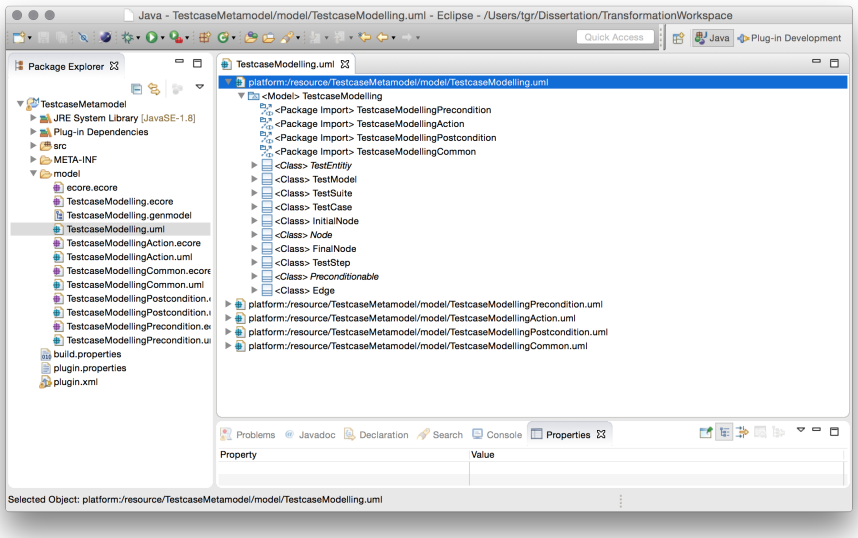


Abbildung 6.2: Bildschirmabdruck des mit dem EMF modellierten Metamodells zur Testfallmodellierung

Zur Generierung der Klassen des Metamodells zur Kontextmodellierung und des Metamodells zur Testfallmodellierung wurde auf Funktionalität des EMF zurückgegriffen. Das EMF-Plug-In zur Generierung von Quellcode aus Modellen erlaubt es, Fachklassen des Modells ohne manuelle Programmierung in Quellcode zu überführen, wobei die Kongruenz von Modell und Code gewährleistet ist. Der auf diese Weise generierte Quellcode kann beispielsweise in weiteren Plug-Ins für das Eclipse-Framework wiederverwendet werden. Im konkreten Anwendungsfall des Werkzeugprototypen wurden die so erzeugen Fachklassen der Metamodelle sowohl bei der Modellierung des UML-Profils zur Integration der Testfallmodellierung in die UML als auch im Eclipse-Plug-In zur Testfallgenerierung wiederverwendet.

6.2.2 Implementierung des UML-Profils zur Integration der Testfallmodellierung in die Systemmodellierung mit der UML

In einem ähnlichen Vorgehen wie bei der Implementierung der Metamodelle zur Kontext- bzw. Testfallmodellierung wurde das in Abschnitt 5.2.4 diskutierte UML-Profil zur Testfallmodellierung in einem maschinenlesbaren Format abgebildet. Konkret wurde hier ein UML-Profilmodell erstellt, in welchem die Stereotypen *TestModel*, *TestActivity*, *TestStep* sowie *TestFlow* implementiert wurden (vgl. Abbildung 6.3). UML-Modelle, die zur Generierung von Testfällen aus Aktivitätsdiagrammen verwendet werden sollen, müssen dieses UML-Profil referenzieren. Das UML-Profil stellt alle benötigten Stereotypen und Referenzen zu den Metamodellen zur Kontext- und zur Testfallmodellierung bereit, die benötigt werden, um ein

Modell einer mobilen, kontextsensitiven Anwendungen gemäß der in Kapitel 5 vorgestellten Methode um Testdaten anzureichern und zur Testfallgenerierung zu verwenden.

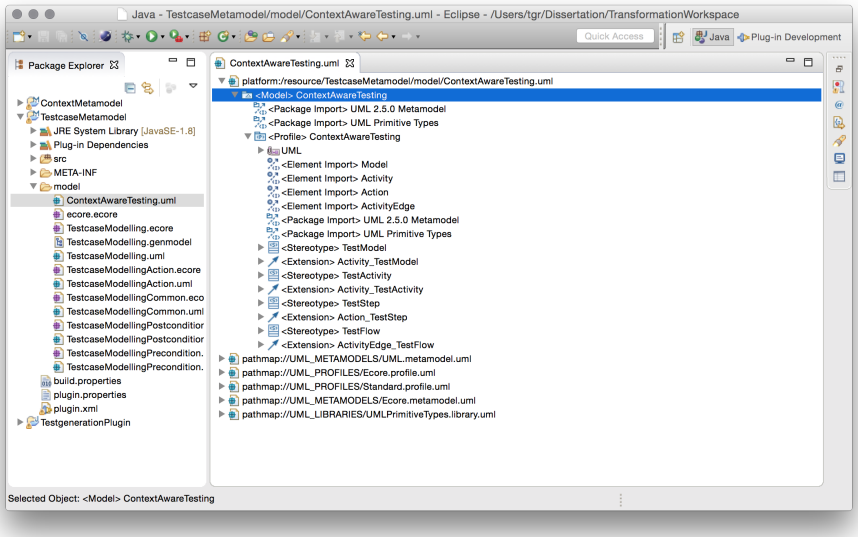


Abbildung 6.3: Bildschirmabdruck des im EMF modellierten UML-Profils zur Integration der Testfallmodellierung in die Systemmodellierung

Vom Grundsatz her ist diese Implementierung des UML-Profils unabhängig vom verwendeten Editor. Das heißt, zur Erstellung von Modellen mobiler Anwendungen, aus denen Testfälle generiert werden sollen, kann ein beliebiger UML-Editor verwendet werden. Im Rahmen dieser Dissertation wurde jedoch für alle Modellierungsaufgaben die Eclipse-IDE verwendet.

6.2.3 Implementierung der Testfallgenerierung

Aufbauend auf der Implementierung der Metamodelle zur Kontext- und Testfallmodellierung wurden die Algorithmen zur Transformation von UML-Aktivitätsdiagramm zu Testfallmodell und zur M2T-Transformation von Testfallmodell zu Calabash als Plug-In für die Eclipse-Plattform implementiert.

Die Eclipse-Plattform basiert auf einem OSGi-Framework, welches es erlaubt, eine Software aus lose gekoppelten Komponenten mit gegenseitigen Abhängigkeiten zu implementieren. Zur Realisierung der Modelltransformation wurde eine solche Komponente in Form eines Eclipse-Plug-Ins erstellt. Dieses kann in eine Instanz der Eclipse-Rich-Client-Platform integriert werden und ermöglicht – basierend auf den durch die Eclipse-Plattform mitgelieferten Plug-Ins zur Modellierung mit UML – die Abbildung des gesamten Prozesses von der Modellierung einer mobilen Anwendung bis hin zur Generierung plattformspezifischer Tests.

Der schematische Aufbau des Eclipse-Plug-Ins zur Testfallgenerierung ist in Abbildung 6.4 verkürzt auf die wesentlichen Komponenten dargestellt. Die Basis bildet die Eclipse-Rich-Client-Platform, die bereits eine Laufzeitumgebung mitliefert, innerhalb welcher individuelle Plug-Ins ausgeführt werden. Das Plug-In zur Testfallgenerierung basiert auf den Metamodellen zur Kontext- bzw. Testfallmodellierung, welche ebenfalls als Eclipse-Plug-Ins realisiert

sind und die durch das EMF aus den Metamodellen generierte Klassen für das Plug-In zur Testfallgenerierung verfügbar machen.

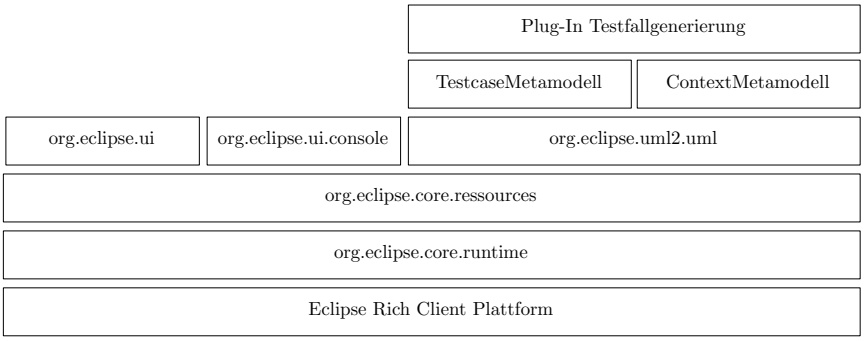


Abbildung 6.4: Schematischer Aufbau der Eclipse-Plug-Ins zur Testfallgenerierung

Das Plug-In zur Testfallgenerierung definiert zwei *Extension Points* (engl. Erweiterungspunkte) in Form von Kontextmenüs im *Eclipse Project Explorer*. Der erste Erweiterungspunkt erzeugt ein Kontextmenü „Generate Test Cases“ (vgl. Abbildung 6.5), welches dem Anwender zur Verfügung steht, wenn auf einer UML-Modelldatei ein Rechtsklick ausgeführt wird. Wählt der Anwender diese Option, wird der in Kapitel 5 vorgestellte Algorithmus zur Generierung plattformunabhängiger Testfälle ausgeführt und aus dem im Editor ausgewählten UML-Modell ein Testfallmodell erzeugt.

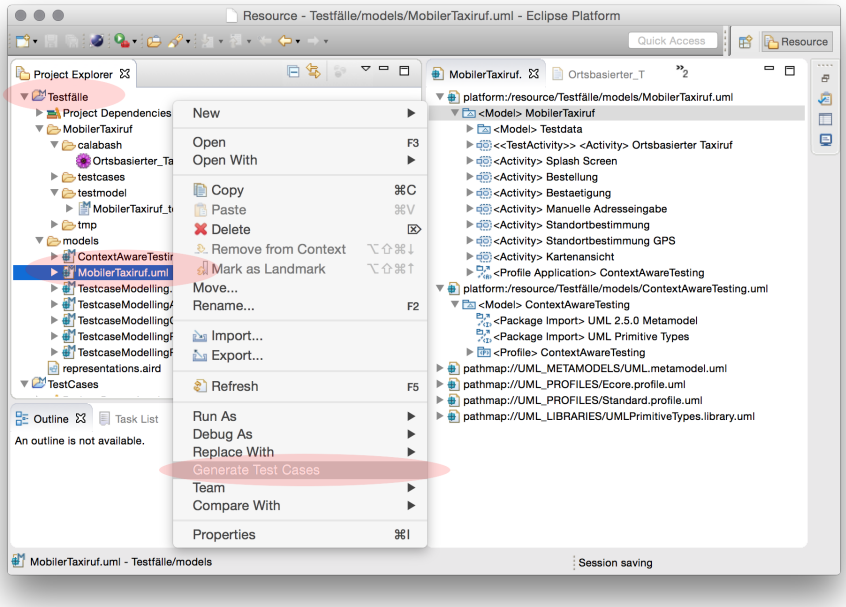


Abbildung 6.5: Bildschirmabdruck des Eclipse Plug-Ins zur Testfallgenerierung

Rechts in Abbildung 6.5 ist das Modell der Anwendung Mobiler Taxiruf im Eclipse-UML Editor, konkret im Project Explorer[2] dargestellt. In der Ansicht ist die hierarchische Baum-

[2]Eclipse bietet einen modularen Aufbau des Editorfenster, mit dem unterschiedliche Perspektiven auf ein Softwareprojekt abgebildet werden können. Der Project Explorer ist eine dieser Perspektiven.

struktur des Modells zu erkennen. Das Modell enthält eine Reihe *Activities* (d. h. UML-Aktivitätsdiagramme), von denen lediglich eines mit dem Stereotyp *TestActivity* annotiert ist. Zu erkennen ist in der Baumansicht ebenfalls die Anwendung des UML-Profils sowie ein eingebettetes Modell „Testdata“, in welchem das zu diesen UML-Aktivitätsdiagrammen gehörende UML-Objektmodell mit den Testdaten hinterlegt ist.

Links in Abbildung 6.5 ist der Projekt Explorer zu erkennen, in dem zur aktuell selektierten Datei „MobilerTaxiruf.uml“ das Kontextmenü geöffnet wurde. Das Plug-In zur Testfallgenerierung stellt die Option „Generate Test Cases“ bereit. Wird diese Option ausgewählt, wird der Algorithmus zur Berechnung des plattformunabhängigen Testfallmodells ausgeführt. Als Ergebnis entsteht das Testfallmodell.

Neben dem Algorithmus zur Generierung des plattformunabhängigen Testfallmodells implementiert das Plug-In einen zweiten Extension Point zur Generierung technologiespezifischer Testfälle, konkret für die Automatisierungstechnologie Calabash. Diese Funktion ist ebenfalls im Projekt Explorer über ein Kontextmenü verfügbar (vgl. Abbildung 6.6).

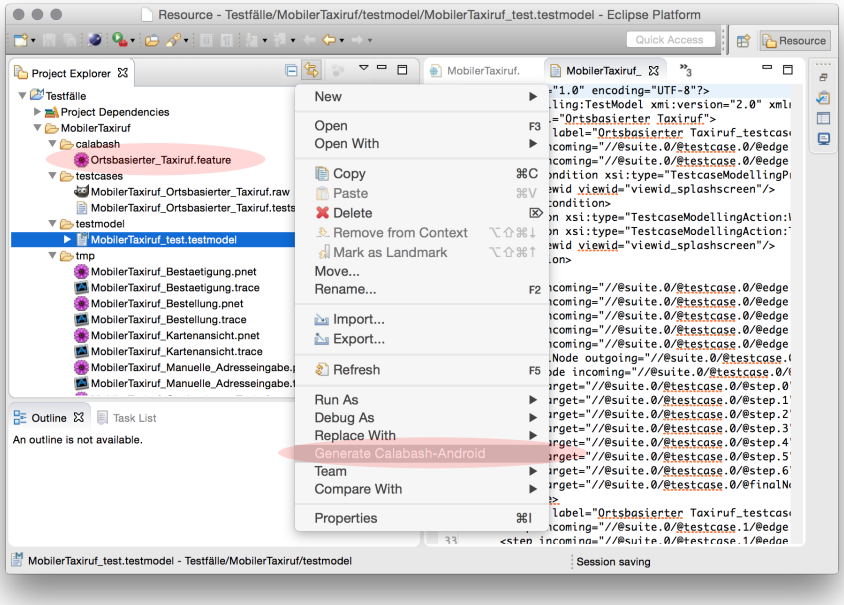


Abbildung 6.6: Bildschirmabdruck des Eclipse Plug-Ins zur Generierung von Calabash-Tests

Eine weitere Integration der Eclipse-Rich-Client-Platform und der Ausführung der generierten Calabash-Tests wäre technisch zwar möglich, wurde aber im Rahmen dieser Dissertation nicht durchgeführt. Bis zum Ende des Jahres 2015 war die Eclipse-IDE das offizielle Entwicklungswerkzeug für die Plattform Android, wurde dann aber durch die IDE Android Studio abgelöst. Vor diesem Hintergrund ist eine weitere Integration der Testautomatisierungstechnologie in die Eclipse IDE nicht sinnvoll.

Diese Implementierung bildet die technologische Basis für die in Abschnitt 6.3 diskutierte Validierung der Testfallgenerierung sowie für die Durchführung der im Rahmen der Evaluierung durchgeführten Fallstudien (vgl. Abschnitt 6.5).

6.2.4 Implementierung des modifizierten Android-Betriebssystems

Der in dieser Dissertation untersuchte Ansatz zur modellbasierten Generierung und automatisierten Durchführung von Tests für mobile, kontextsensitive Anwendungen basiert neben dem Algorithmus zur Testfallgenerierung auf UML-Aktivitätsdiagrammen aus einer angepassten Implementierung des Android-Betriebssystems und einer ebenfalls angepassten Implementierung des Calabash-Testautomatisierungs-Frameworks.

Die theoretischen Grundlagen der notwendigen Anpassungen am Android-Betriebssystem wurden bereits in Abschnitt 5.5.2 diskutiert. Im Rahmen der Evaluierung der Methode und des Werkzeugs zur Testautomatisierung wurden diese Modifikationen im Android-Betriebssystem implementiert. Hierzu wurde der Quellcode des Android-Betriebssystems aus dem Google Quellcodeverwaltungssystem [147], konkret der Codezweig mit der Identifikation LMY48I [139], heruntergeladen. Es handelt sich dabei um Android in der Version 5.1.1 Lollipop, Release Candidate 9. Zum Zeitpunkt der Umsetzung handelte es sich hierbei um das neueste verfügbare Release (engl. Version, Veröffentlichung), das neben dem Emulator die gesamte Familie der Nexus Geräte unterstützt[3]. Neben den Kriterien Aktualität zum Zeitpunkt der Implementierung und Unterstützung für die spezifische Auswahl der Geräte der Nexus Familie waren keine weiteren Kriterien für die Auswahl des spezifischen Code-Zweigs LMY48I ausschlaggebend. Die Modifikationen können analog in andere Code-Zweige integriert werden.

In diesem Codezweig wurden die in Abschnitt 5.5.2 diskutierten Modifikationen implementiert. Konkret wurde das Android Sensor-Subsystem auf der Ebene des *Hardware-Abstraction-Layer* (HAL) so erweitert, dass das System an neu geschaffenen Netzwerkschnittstellen Testdaten akzeptiert, die vom modifizierten Sensor-HAL als Sensordaten im System zu konsumierenden Apps propagiert werden. In Abbildung 6.7 ist eine konkrete Realisierung der in Abschnitt 5.5.2, Abbildung 5.54 diskutierten Modifikation des Android-Betriebssystem, insb. des Sensor-Stacks dargestellt. Der originale Android-Sensor-Stack wurde um die Klassen *MockDateTimeService*, *MockSensorDevice*, *MockSensorService*, *MockSensorThread* ergänzt, die Klassen *SensorDevice*, *SensorService* wurde modifiziert.

Die Klasse *MockDateTimeService* implementiert eine Schnittstelle, an welcher das System Testdaten zur Anpassung von Datum und Uhrzeit im System entgegen nimmt. Diese Funktion dient der Verwendung von Datum und Uhrzeit als Kontextparameter (vgl. Abschnitt 3.1.2.1.2) in Tests. Die Implementierung dieser Schnittstelle ist notwendig, weil eine Anpassung von Datum und Uhrzeit des Systems aus der Domäne Anwendungen (*User Space* in der Terminologie der Linux-Basis des Android-Systsms) heraus nicht verändert werden kann. Die Schnittstelle *(1)* in Abbildung 6.7 kommuniziert über eine Socket-Netzwerkverbindung mit dem Sensorsimulator (vgl. Abschnitt 5.5.3.1), der beim Testen Datum und Uhrzeit gemäß der Testfallspezifikation in das SUT einspeist.

Die Klassen *MockSensorDevice*, *MockSensorService* sowie *MockSensorThread* implementieren die Schnittstelle, an welcher das System Testdaten als Sensordaten entgegen nimmt. Zur Realisierung dieser Funktion im Zusammenwirken mit dem Sensorsimulator (vgl. Abschnitt 5.5.3.1) kommuniziert das System über zwei separate Netzwerk-Ports mit dem Sen-

[3]Die Geräte der Nexus-Serie sind Smartphones, die von Google speziell für Entwickler herausgegeben werden. Sie zeichnen sich insb. dadurch aus, dass Treiber-Software für Hardware-Bausteine frei verfügbar ist, so dass Entwickler in der Lage sind, Anpassungen am Android-Betriebssystem auf Hardware zu testen.

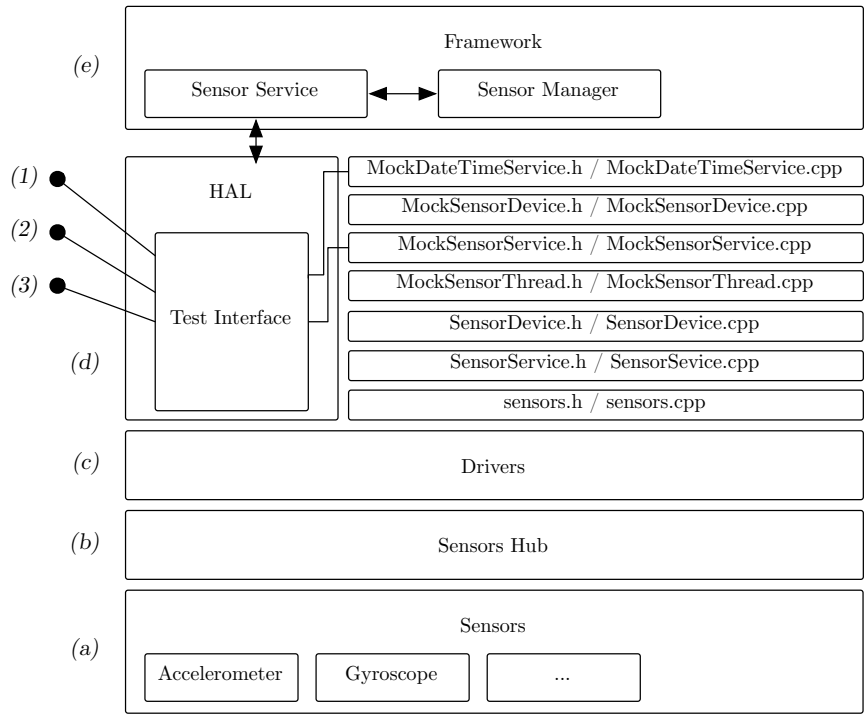
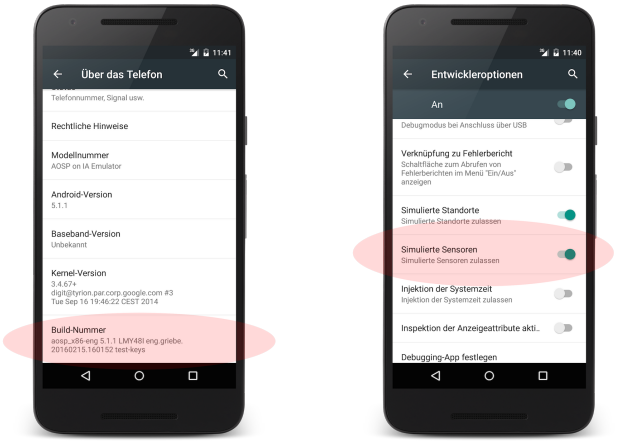


Abbildung 6.7: Schematische Darstellung des modifizierten Sensor-Stacks des Android-Betriebssystems

sorsimulator über jeweils eine Socket-Netzwerkverbindung, *(2)* und *(3)* in Abbildung 6.7. Wird eine Anwendung gestartet, die einen oder mehrere Sensoren verwendet, werden über die Schnittstelle *(2)* Metadaten zu den verwendeten Sensoren an den Sensorsimulator übertragen (d. h. Sensortyp, Aktualisierungsintervall). Der Sensorsimulator verwendet diese Information zur Berechnung künstlicher Sensordaten gemäß der Testfallspezifikation. Diese Daten werden dann über Schnittstelle *(3)* vom Sensorsimulator in das SUT übertragen. Künstliche Sensordaten werden von dem in Abschnitt 5.5.3.1 diskutieren Sensorsimulator erzeugt.

Das modifizierte Android-Betriebssystem wurde für den Emulator und für die Geräte der Nexus Familie kompiliert. Mit Ausnahme der Testschnittstellen unterscheidet es sich nicht von einer Standardimplementierung und kann regulär auf einem Gerät verwendet werden. In Abbildung 6.8a ist ein Bildschirmabdruck der Informationsseite „Über das Telefon“ dargestellt. Diese Informationsseite gehört zur Standardfunktionalität des Betriebssystems und gibt unter anderem Auskunft über die Version und die Build-Identifikation des auf dem Gerät installierten Systems.

Um unbeabsichtigtes Einspeisen von Sensordaten außerhalb der Verwendung des Geräts zum Testen mobiler Anwendungen zu verhindern, wurden der Dialog „Entwickleroptionen“ des Betriebssystems um die Option „Simulierte Sensoren“ (vgl. Abbildung 6.8b) ergänzt. Über einen Schalter kann zwischen simulierten Sensordaten und der Verwendung der originären Sensoren des Geräts gewechselt werden. Der Tester hat so die Möglichkeit, eine Anwendung zusätzlich zur automatischen Testdurchführung ebenfalls manuell zu testen.



(a) Bildschirmabdruck der Ansicht „Über das Telefon“, Abschnitt Build-Number. Hervorgehoben ist die Build-Identifikation LMY48I. Das Postfix „eng.griebe.20160215.160152 test-keys“ gibt Auskunft über den Entwickler des Android-Codezweigs und den Zeitpunkt des Erstellens des Codezweigs.

(b) Bildschirmabdruck der Ansicht „Entwickleroptionen“. Hervorgehoben ist Abschnitt Simulierte Sensoren, der durch die Modifikation des Android-Betriebssystems ergänzt wurde. Wird diese Option aktiviert, akzeptiert das System Testdaten als Sensordaten an einer Netzwerkschnittstelle.

Abbildung 6.8: Bildschirmabdrucke des angepassten Android-Betriebssystems. Hervorgehoben ist in 6.8a der Abschnitt Build-Number der Ansicht „Über das Telefon“ sowie in 6.8b die Ansicht „Entwickleroptionen“.

6.2.5 Implementierung des modifizierten Calabash-Frameworks

Die Ausführung von Testfällen kann vom Grundsatz her durch unterschiedliche Technologien automatisiert werden (vgl. Abschnitt 2.5). In dieser Dissertation wurde Calabash als Zieltechnologie für die Generierung technologiespezifischer Tests gewählt. Es handelt sich bei Calabash um Open-Source-Software, die durch Integration in die Cross-Platform-Technologie Xamarin einen großen Anwenderkreis erreicht.

Die Calabash-Standardimplementierung stellt neben einer rudimentären Funktion zur Simulation von Standortdaten keine Mittel zur Verfügung, um Kontextparameter beim Testen zu berücksichtigen. Im Rahmen der Implementierung des Werkzeugprototypen wurde diese Funktionalität ergänzt.

Calabash basiert auf einer komplexen Integration einer Vielzahl Technologien (vgl. Abbildung 2.2), konkret einer Laufzeitumgebung für die Programmiersprache Ruby, die ihrerseits in einer Java Virtual-Machine ausgeführt wird. Zur Kommunikation mit dem SUT verwendet Calabash eine Netzwerkverbindung, über welche das zu testende System durch Calabash ferngesteuert wird. Das heißt, während der Testausführung werden Interaktionen mit dem SUT schrittweise aus der Testfallbeschreibung gelesen, an das SUT übertragen und dort interpretiert und mit Hilfe von Instrumentierungstechnologie in konkrete Steuerbefehle umgewandelt.

Enthält ein Testfall beispielsweise die Anweisung, ein konkretes Element einer grafischen Benutzungsoberfläche durch Berühren mit dem Finger zu manipulieren, wird diese Anweisung in ein Übertragungsformat konvertiert und auf dem Zielgerät in ein UI-*Event* übersetzt, das in den Event-Strom des SUT eingeleitet wird. Das SUT ist nicht in der Lage, eine manuelle Interaktion von auf diese Weise simulierten Interaktion zu unterscheiden.

Zur Bereitstellung der zur Simulation von Sensordaten benötigen Funktionalität wurde die Calabash-Ruby-API erweitert. Konkret wurden die in Abschnitt 5.5.3.2 sowie Anhang A vorgestellten Step-Definitions implementiert. Zusätzlich zu der von Calabash bereitgestellten Netzwerkverbindung zwischen dem Entwicklungscomputer und dem SUT wurden drei weitere Netzwerkverbiundungen erzeugt, nämlich die zur Anbindung der in Abbildung 6.7 definierten Schnittstellen benötigten Schnittstellen zur Übertragung der simulierten Sensordaten an das SUT, der Metadaten zu verwendeten Sensoren vom SUT an Calabash sowie zur Übertragung von simulierten Datumsinformationen an das SUT.

Teil des Werkzeugprototypen ist ebenfalls eine Implementierung des in Abschnitt 5.5.3.1 diskutierten Sensorsimulators. Dieser wurde als Java-Bibliothek unter Verwendung einer Ruby-Java-Bridge in die Calabash-Implementierung integriert[4]. Aufgabe des Sensorsimulators ist die Erzeugung realitätsnaher Sensordaten, d. h. solcher Daten, die in ausreichender Weise Ähnlichkeiten zu den Charakteristiken realer Sensordaten aufweisen, insb. im Hinblick auf Aktualisierungsintervall und Sensorrauschen.

In der Praxis stellt die Bibliothek eine Schnittstelle bereit, über welche der Sensorsimulator für spezifische Sensoren mit dem gewünschten Mittelwert parametrisiert werden kann. Dieser Wert wird zur Laufzeit aus der Testfallspezifikation extrahiert. Die Addition künstlichen Rauschens kann ebenfalls durch die Testfallspezifikation gesteuert werden.

6.3 Validierung der Testfallgenerierung

Das in dieser Dissertation untersuchte Verfahren zur Testautomatisierung für mobile, kontextsensitive Anwendungen umfasst die fünf in Abschnitt 5.1 vorgestellten Schritte, die sich grob in die Phase der Testfallgenerierung und die Phase der Testfallausführung unterteilen. Abschnitt 6.3 hat die Validierung der ersten Phase zum Gegenstand, d. h. die automatisierte Generierung des Testfallmodells aus einem UML-Modell. In diesem Abschnitt soll zum einen der Nachweis erbracht werden, dass der Algorithmus zu Generierung des technologieunspezifischen Testfallmodells gültige Testfälle erzeugt, also Testfälle in denen simulierte Interaktionen mit dem SUT nur in einer gültigen Reihenfolge erfolgt und die zum anderen dem in Abschnitt 5.3.2 definierten Permutationssequenzüberdeckungskriterium $C_{PS_{ACD}}$ genügen.

Die Generierung von Testfallmodellen aus UML-Aktivitätsdiagrammen erzeugt aufgrund nebenläufiger und zyklischer Strukturen im Kontrollfluss bereits für Aktivitätsdiagramme geringer Komplexität Testfallmodelle, deren Komplexität es unmöglich macht, die Korrektheit der Transformation durch manuelle Inspektion der Transformationsergebnisse zu überprüfen. Es kann zwar auch für komplexe Modelle manuell überprüft werden, ob die generierten Ausführungspfade gültige Sequenzen im zugrundeliegenden Aktivitätsdiagramm sind. Hier-

[4] Teile der Implementierung der Integration des Sensorsimulators wurden in einer im Rahmen dieser Dissertation betreuten Masterarbeit von Marc Gesthüsen [243] erarbeitet.

zu muss für jeden Ausführungspfad überprüft werden, ob die enthaltenen Testschritte in der im Testfall spezifizierten Reihenfolge im Aktivitätsdiagramm ebenfalls in genau dieser Reihenfolge auftreten können. Für komplexe Modelle ist es hingegen nicht mehr möglich, durch manuelle Inspektion der Transformationsergebnisse zu überprüfen, ob tatsächlich alle möglichen Ausführungspfade im zugrundeliegenden Aktivitätsdiagramm durch eine Sequenz im Testfallmodell abgedeckt sind (vgl. State-Explosion-Problem, Valmari [348]).

Eine Strategie zum Umgang mit der Herausforderung der Validierung der Transformation komplexer Modelle ist die Identifikation von Modellierungsmustern geringer Komplexität, für die der Nachweis der Korrektheit durch manuelle Inspektion der Transformationsergebnisse erbracht werden kann. Hierzu wird vorausgesetzt, dass komplexe Modelle aus Modellfragmenten geringerer Komplexität zusammengesetzt sind, die jeweils einzelnen Modellierungsmustern zugeordnet werden können. Umgekehrt gilt dann, dass komplexe Modelle in Komponenten geringerer Komplexität zerlegt werden können. Wenn gezeigt werden kann, dass Modellfragmente (Muster) geringer Komplexität durch den Algorithmus zur Transformation vom Aktivitätsdiagrammen zu Testfallmodellen korrekt ist und weiterhin gezeigt werden kann, dass auch die Kombination atomarer Mustern zu komplexeren (Teil-)Modellen korrekt transformiert wird, so kann gefolgert werden, dass die Transformation auch für Modelle beliebiger Komplexität korrekte Ergebnisse erzeugt. Diese Folgerung basiert auf der Eigenschaft der verwendeten Modellierungstechnologien (d. h. UML-Aktivitätsdiagramme und Petri-Netze), Teilmodelle durch Ersetzen mit atomaren Modellierungselementen zu maskieren (Vergröberung) oder ein atomares Modellierungselement durch ein Teilmodell zu ersetzen (Verfeinerung).

Die Validierung der Transformation stützt sich darauf, zunächst den Nachweis darüber zu erbringen, dass rudimentäre Fragmente von UML-Aktivitätsdiagrammen mit geringer Komplexität durch den Algorithmus und dessen Implementierung im Sinne des Abdeckungskriteriums korrekt in Fragmente von Testfallmodellen überführt werden. Diese Untersuchung erfolgt durch Inspektion der Zwischenergebnisse des Transformationsalgorithmus, konkret des Petri-Netzes und dem zugehörigen Erreichbarkeitsgraphen. In einem zweiten Schritt der Validierung werden nach der Art des Beweises durch Induktion die rudimentären Modellfragmente sukzessive zu komplexeren Modellen kombiniert (d. h. verfeinert) und gezeigt, dass auch diese komplexeren Modelle korrekt zu Testfallmodellen transformiert werden.

Ziel dieses Vorgehens ist es, den Nachweis zu erbringen, dass die Transformation nur zulässige Sequenzen einzelner Testschritte erzeugt, d. h. dass im Testfallmodell keine zwei Testschritte vertauscht gegenüber ihrer ursprünglichen Reihenfolge im UML-Aktivitätsdiagramm auftreten, parallele Abläufe gemäß dem in Abschnitt 5.3.2 definierten Abdeckungskriterium auf nicht-nebenläufige Sequenzen abgebildet werden und dass Zyklen im Kontrollfluss des UML-Aktivitätsdiagramms entsprechend dem Abdeckungskriterium durchlaufen werden.

6.3.1 Identifikation und Validierung der Transformation atomarer Modellierungsmuster

Die Interpretation komplexer Modelle als sukzessive Verfeinerungsoperationen auf Modellierungsmustern geringer Komplexität wurde bereits in früheren Forschungsarbeiten untersucht. Staines [330] argumentiert in seiner Arbeit zur Transformation von UML-Aktivitätsdiagram-

men zu gefärbten Petri-Netzen, dass eine Transformation von Aktivitätsdiagrammen zu Petri-Netzen gegenüber einer direkten Verwendung von Aktivitätsdiagrammen bei der maschinellen Verarbeitung (d. h. Simulation, Analyse) vielfältige Vorteile bieten, z. B. das formale Petri-Netz-Kalül, darüber hinaus jedoch auch eine Reihe von Nachteilen hat, wie etwa die erhöhte Komplexität eines Petri-Netzes gegenüber dem Aktivitätsdiagramm (d. h. höhere Anzahl Knoten und Kanten) oder weniger intuitives Verständnis der visuellen Darstellung. Insbesondere der Aspekt der erhöhten Komplexität des Petri-Netzes gegenüber einem UML-Aktivitätsdiagramm tritt bei dem in dieser Dissertation vorgestellten Algorithmus zur Testfallgenerierung in den Vordergrund, da sowohl für jeden Knoten des Aktivitätsdiagramms als auch für jede Kante eine Stelle im Petri-Netz erzeugt wird, wodurch die Anzahl der Knoten und Kanten im Petri-Netz substanziell größer ist als im UML-Aktivitätsdiagramm (vgl. Abschnitt 5.3.3).

Wenngleich für eine Anwendung in einer anderen Domäne identifiziert Staines das Problem der erhöhten Komplexität von Petri-Netzen im Vergleich zu Aktivitätsdiagrammen und argumentiert, dass eine Identifikation atomarer Modellierungsmuster zur Komplexitätsreduktion beiträgt. Die Arbeit von Staines ist für diese Dissertation als Anhaltspunkt relevant, mögliche atomare Modellierungsmuster zu identifizieren, die zur Validierung des Transformationsalgorithmus verwendet werden können. Konkret identifiziert Staines für UML-Aktivitätsdiagramme die Modellierungsmuster (1) Aktivitätssequenz, (2) Aktivitätsparallelisierung/-sychronisierung, (3) alternative Verzweigung, (4) Zusammenführung alternativer Zweige und (5) Iteration. Weiterhin unterstützt Staines die These, dass sich komplexe UML-Aktivitätsdiagramme durch wiederholte Kombination (d. h. Verfeinerung) dieser Muster erstellen lassen.

Ebenfalls mit der Identifikation rudimentärer Modellierungsmuster befassen sich Wohed et al. [367, 366]. Im Fokus der Autoren steht eine wissenschaftliche Beurteilung der Modellierungsmächtigkeit von UML-Aktivitätsdiagrammen. Wohed et al. identifizieren in ihrer Arbeit diese Modellierungsmuster: (1) Sequenz, (2) explizite *und*-Verzweigung, (3) explizite *und*-Synchronisation, (4) explizite *entweder-oder*-Verzweigung, (5) *entweder-oder*-Zusammenführung und (6) *oder*-Verzweigung.

In Anlehnung an die von Staines und Wohed et al. identifizierten Modellierungsmuster wurden zum Zweck der Validierung der Transformation von UML-Aktivitätsdiagramm zu Testfallmodell die in den folgenden Abschnitten diskutierten atomaren Muster herangezogen. Die Auswahl der konkret betrachteten Muster basiert auf der Auswertung wissenschaftlicher Literatur und orientiert sich zudem an den Syntaxelementen, die durch das UML-Metamodell für Aktivitätsdiagramme vorgegeben werden.

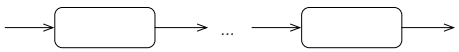
Zu jedem identifizierten Modellierungsmuster wurde im Rahmen der Validierung des Transformationsalgorithmus jeweils ein minimales UML-Aktivitätsdiagramm gebildet, auf welches die Testfallgenerierung angewendet wurde und das Transformationsergebnis durch manuelle Inspektion aller Zwischenergebnisse auf Konsistenz gegenüber dem in Abschnitt 5.3.2 diskutierten Abdeckungskriterium überprüft wurde.

6.3.1.1 Aktionssequenz

Das einfachste Modellierungsmuster, das mit den Mitteln des Metamodells der UML für Aktivitätsdiagramme identifiziert werden kann, ist eine Sequenz von Aktivitäten. Dieses Muster wurde ebenfalls von Wohed et al. [366, 367] und von Staines [330] in unterschiedlichen Vari-

anten identifiziert. Bei Staines [330] wird eine Aktivitätssequenz jeweils am Anfang und am Ende durch einen Kontrollfluss begrenzt, während bei Wohed et al. eine Sequenz jeweils am Anfang und am Ende durch eine Aktivität begrenzt wird.

Abweichend von diesen beiden Autoren definiert diese Dissertation zum Zweck der Validierung der Modelltransformation zwei unterschiedliche Varianten des Modellierungsmusters Aktivitätssequenz, die beide jeweils am Anfang und am Ende durch einen Kontrollfluss begrenzt werden. In Abbildung 6.9 ist das Modellierungsmuster Aktionssequenz in einer generischen Ausprägung beliebiger Länge (Abbildung 6.9a) und in einer spezifischen Ausprägung der Länge eins (Abbildung 6.9b) dargestellt.



(a) Aktionssequenz beliebiger Länge

(b) Aktionssequenz der Länge eins

Abbildung 6.9: Das Modellierungsmuster Aktionssequenz in der generischen Ausprägung beliebiger Länge (Abbildung 6.9a) und der Länge eins (Abbildung 6.9b)

Die Ausprägung der Länge eins definiert bei Einbettung in das leere Aktivitätsdiagramm das minimale syntaktisch korrekte, nicht-leere Aktivitätsdiagramm (Abbildung 6.10), das neben Anfangs- und Endknoten nur eine einzige Aktivität spezifiziert. Dieses minimale Modell bildet den Ausgangspunkt der Kombination komplexer Aktivitätsdiagramme durch wiederholtes Anwenden von Ersetzungsoperationen des Modellierungsmusters Aktivitätssequenz der Länge eins durch höherwertige Modellierungsmuster. Eine solche Ersetzung ist zulässig, da das Metamodell der UML die Spezifikation einer Aktivität als *UML::CallBehaviorAction* zulässt, das eine Vergröberung eines Modellierungsinhalts als Mittel der Abstraktion repräsentiert.

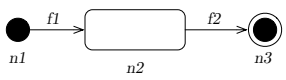
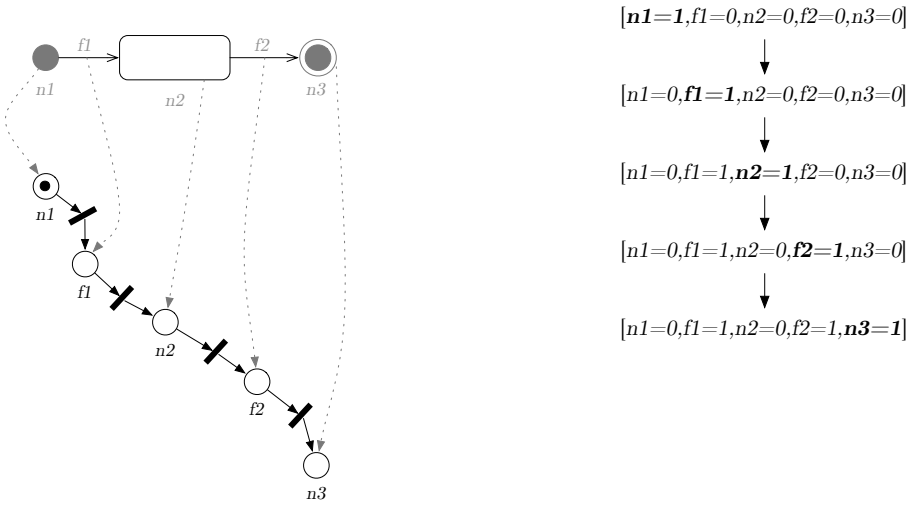


Abbildung 6.10: Minimales syntaktisch korrektes Aktivitätsdiagramm zum Muster Aktivitätssequenz der Länge eins

Unter der Voraussetzung, dass auf alle im UML-Aktivitätsdiagramm enthaltenen Kontrollflüsse und Aktivitäten die Stereotypen *TestFlow* bzw. *TestStep* angewendet wurde (vgl. Abschnitt 5.2.4), muss der in Abschnitt 5.3.3 entworfene Algorithmus zur Transformation eine Instanz von *Test::TestModel* generieren, die genau ein *Test::TestSuite*-Element mit genau einem *Test::TestCase*-Element enthält. In diesem *Test::TestCase*-Element müssen drei *Test::Step*-Elemente enthalten sein, die Elemente $f1$, $n2$ und $f3$ in genau dieser Ausführungsreihenfolge

repräsentieren. Als Zwischenergebnis der Transformation wird das in Abbildung 6.11a dargestellte Petri-Netz erzeugt. Es entspricht strukturell dem zugrundeliegenden Aktivitätsdiagramm, wodurch der erste Schritt der Transformation, d. h. die Überführung des Aktivitätsdiagramms in ein strukturgleiches Petri-Netz, für das Modellierungsmuster Aktivitätssequenz validiert wurde.



(a) Petri-Netz zum minimalen, syntaktisch korrekten, nicht-leeren Aktivitätsdiagramm

(b) zu Abbildung 6.11a gehöriger Erreichbarkeitsgraph

Abbildung 6.11: Petri-Netz und zugehöriger Erreichbarkeitsgraph zu dem in Abbildung 6.10 dargestellten minimalen UML-Aktivitätsdiagramm

Die Erzeugung des Testfallmodells basiert auf der Traversierung des Erreichbarkeitsgraphen des als Zwischenergebnis erzeugten Petri-Netzes. Für das minimale UML-Aktivitätsdiagramm wurde der in Abbildung 6.11b dargestellte Erreichbarkeitsgraph erzeugt. Aufgrund der Abwesenheit von Verzweigungen degeneriert der Graph zu einem Erreichbarkeitsbaum mit genau einem Blattkonten, für den genau eine Traviersierungsreihenfolge vom Wurzelknoten zur Blattkonten existiert, so dass die Abfolge der Markierungen des Petri-Netzes bei jeder gegebenen Anfangsmarkierung eindeutig definiert ist. Diese Traversierungsreihenfolge des Erreichbarkeitsgraphen entspricht der Reihenfolge der Modellierungselemente im Aktivitätsdiagramm und zugleich der Ausführungsreihenfolge atomarer Testschritte, da diese aus den Markierungen des Petri-Netzes abgeleitet werden (vgl. Abschnitt 5.3.3.1.3).

Die Korrektheit der Transformation des minimalen UML-Aktivitätsdiagramms zu einem Ausführungspfad ist somit validiert, da nachgewiesen ist, dass der Transformationsalgorithmus zu diesem Aktivitätsdiagramm genau die durch das Abdeckungskriterium (vgl. Abschnitt 5.3.2) geforderte Ausführungssequenz von Testschritten generiert. Durch manuelle Inspektion des aus dem Ausführungspfad generierten Testfallmodells konnte ebenfalls nachgewiesen werden, dass auch die Erzeugung des Testfallmodells gemäß des in Abschnitt 5.2.3.2 entworfenen Metamodells zur Testfallmodellierung korrekt ist.

Analog zur Ableitung des minimalen Aktivitätsdiagramms aus dem Modellierungsmuster Aktivitätssequenz der Länge eins können aus dem Modellierungsmuster Aktivitätssequenz beliebiger Länge ebenfalls Aktivitätsdiagramme abgeleitet werden, die mehrere Aktivitäten in einer eindeutig definierten Ausführungsreihenfolge modellieren. In Abbildung 6.12 ist ein Aktivitätsdiagramm dargestellt, welches das minimale Aktivitätsdiagramm um genau eine weitere Aktivität ergänzt.

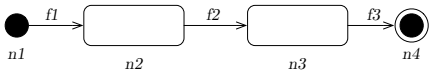
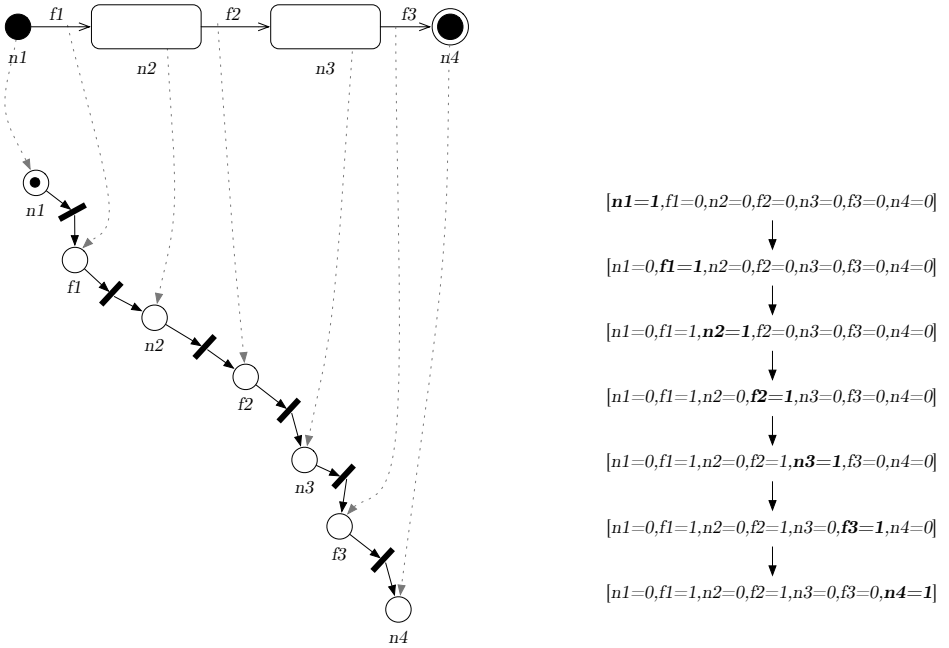


Abbildung 6.12: Syntaktisch korrektes, nicht-leeres Aktivitätsdiagramm zum Modellierungsmuster Aktivitätssequenz der Länge größer eins

Wird der Transformationsalgorithmus auf dieses Aktivitätsdiagramm angewendet, entsteht das in Abbildung 6.13a dargestellte Petri-Netz mit dem zugehörigen in Abbildung 6.13b dargestellten Erreichbarkeitgraphen. Analog zur Argumentation der Korrektheit der Transformation des minimalen UML-Aktivitätsdiagramms gilt ebenfalls, dass für Markierungen des Petri-Netzes mit Markenanzahl eins genau eine Folgemarkierung existiert. Bei der Erzeugung des Testfallmodells ist also auch für dieses Aktivitätsdiagramm ausschließlich die korrekte Reihenfolge der generierten *Test::Step*-Elemente möglich, wodurch die Korrektheit der Transformation für Aktivitätssequenzen beliebiger Länge nachgewiesen ist.



(a) Petri-Netz zum nicht-leeren Aktivitätsdiagramm zum Muster Aktivitätssequenz

(b) zu Abbildung 6.13a gehöriger Erreichbarkeitsgraph

Abbildung 6.13: Petri-Netz und zugehöriger Erreichbarkeitsgraph zu dem in Abbildung 6.12 dargestellten UML-Aktivitätsdiagramm

6.3.1.2 Disjunktive Verzweigung

Ein weiteres Modellierungsmuster in UML-Aktivitätsdiagrammen ist die disjunktive Kontrollflussverzweigung. Dieses Muster wurde ebenfalls von Staines [330] und Wohed et al. [366, 367] identifiziert. Das Metamodell der UML sieht für diese Funktion das Element *UML::DecisionNode* vor (vgl. Abbildung 6.14). Es bildet die Grundlage für die höheren Modellierungsmuster Zyklus (Abschnitt 6.3.2.3) und Bypass (Abschnitt 6.3.2.2).

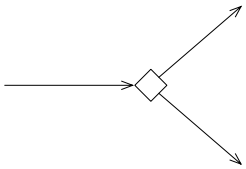
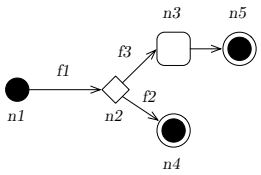
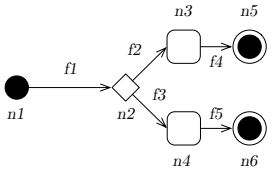


Abbildung 6.14: Modellierungsmuster disjunkive Kontrollflussverzweigung

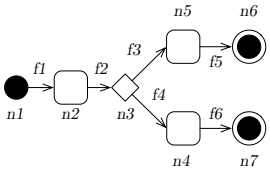
Analog zum Modellierungsmuster Aktivitätssequenz wurden zum Zweck der Validierung der Modelltransformation minimale Aktivitätsdiagramme unter Verwendung des Modellierungsmusters disjunktive Kontrollflussverzweigung erstellt und der Algorithmus zur Generierung des Testfallmodells darauf angewendet. Anders als für das Modellierungsmuster Aktivitätssequenz existieren für das Muster disjunktive Kontrollflussverzweigung drei nicht-leere minimale Modelle. Sie sind in Abbildung 6.15 dargestellt.



(a) Muster Aktivitätssequenz nach der Fallunterscheidung



(b) Alternative Muster Aktivitätssequenzen nach der Fallunterscheidung



(c) Muster Aktivitätssequenzen, alternative Muster Aktivitätssequenzen nach der Fallunterscheidung

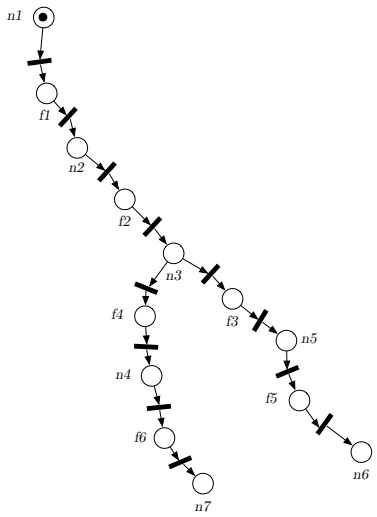
Abbildung 6.15: Minimale Aktivitätsdiagramme mit dem Modellierungsmuster disjunktive Kontrollflussverzeigung. Die Verzweigungsbedingungen wurden der Übersichtlichkeit halber weggelassen.

Der in Abschnitt 5.3.1 entworfene Algorithmus zur Modelltransformation wurde auf die in Abbildung 6.15 dargestellten minimalen, aus dem Modellierungsmuster disjunktive Kontrollflussverzweigung erzeugbaren Aktivitätsdiagramme, angewendet. Analog zum Modellierungsmuster Aktivitätssequenz wurde die Korrektheit der Transformation durch manuelle Inspektion der Zwischenergebnisse der Transformation (d. h. Petri-Netz, dessen Erreichbarkeitsgraph, die daraus erzeugten Ausführungspfade und die resultierenden Testfallmodelle) überprüft. Die experimentelle Anwendung des Transformationsalgorithmus erfolgte hierbei unter Ausschöpfung sinnhafter Varianten der Annotation der Modellierungselemente mit den UML-Stereotypen *TestStep* respektive *TestFlow*. Das heißt beispielsweise für das Modell in

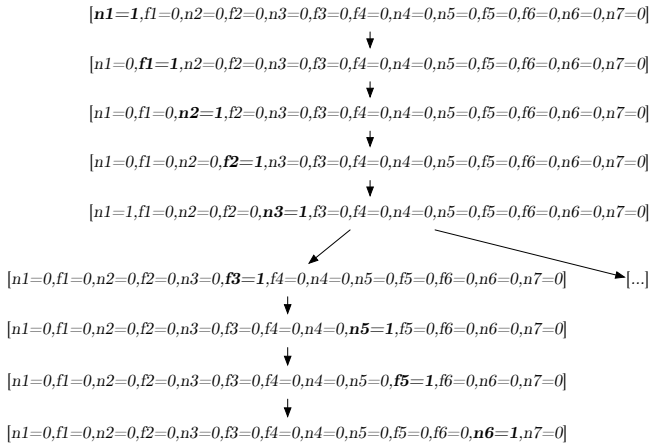
Abbildung 6.15c, dass Varianten überprüft wurden, in denen die Kontrollflüsse $f3$, $f4$, $f5$ und $f6$ in allen möglichen nicht-leeren Kombinationen überprüft wurden. Diese Varianten wurden deshalb ausgewählt, weil insb. die Annotation der Kontrollflüsse $f3$ und $f4$ in Abbildung 6.15c darüber entscheiden, ob für die auf Knoten $n6$ bzw. $n7$ endenden Ausführungspfade korrekte Sequenzen von Testschritten erzeugt werden.

Diese besondere Analyse der Transformationsergebnisse für unterschiedlich annotierte Teilmodelle ist deshalb ausschlaggebend für die Korrektheit der Transformation, weil der in Abschnitt 5.3.3.1.3 entworfene Algorithmus zur Reduktion der Ausführungspfade auf testrelevante Elemente solche Pfade entfernt, die aus einem Entscheidungsknoten (d. h. *UML::DecisionNode*) resultieren, dessen ausgehende Kanten nicht mit dem Stereotyp *TestFlow* annotiert sind. Bei der Testausführung ist dann nicht entscheidbar, welche Vorbedingungen oder Interaktionen zur Auswahl einer der Alternativen führt.

In Abbildung 6.16 ist das durch Modelltransformation zu dem Aktivitätsdiagramm in Abbildung 6.15c gehörige Petri-Netz dargestellt. Die manuelle Überprüfung hat ergeben, dass dieser Schritt der Modelltransformation für dieses Aktivitätsdiagramm ein korrektes Petri-Netz, den zugehörigen Erreichbarkeitsgraphen, korrekte Ausführungspfade und ein gegenüber dem Abdeckungskriterium konsistentes Testfallmodell erzeugt.



(a) Petri-Netz zum minimalen, syntaktisch korrekten, nicht-leeren Aktivitätsdiagramm aus Modellierungsmuster disjunktive Kontrollflussverzweigung



(b) zu Abbildung 6.16a gehöriger Erreichbarkeitsgraph, Darstellung verkürzt auf den in $n6$ endenden Teilbaum

Abbildung 6.16: Petri-Netz und zugehöriger Erreichbarkeitsgraph zum rechten Zweig des in Abbildung 6.15c dargestellten UML-Aktivitätsdiagramm

In Abbildung 6.16b ist ein Ausschnitt aus dem zum Modell in Abbildung 6.15c gehörigen Erreichbarkeitsgraph dargestellt. Im nächsten Transformationsschritt wurden aus diesem Erreichbarkeitgraphen die möglichen Ausführungspfade errechnet. Auch hier hat eine manuelle

Inspektion die Korrektheit der Transformation bestätigt. Insbesondere wurde überprüft, ob die Reduktion auf testrelevante Elemente ebenfalls im Sinne des in Abschnitt 5.3.2 diskutierten Abdeckungskriteriums korrekte Testfallmodelle generiert. Hierbei konnte festgestellt werden, dass für alle in Abbildung 6.15 dargestellten, minimalen Aktivitätsdiagramme jeweils die korrekten Ausführungspfade erstellt wurden. Inbesondere wurden für unterschiedliche Annotationsvarianten der Kontrollflüsse $f3$ und $f4$ in Abbildung 6.15c korrekt diejenigen Pfade entfernt, für die keine Entscheidungsbedingung an den Kanten $f3$ und $f4$ modelliert wurden.

Beispiel des Aktivitätsdiagramms aus Abbildung 6.15c wurden die Varianten 1 bis 3 in Abbildung 6.17 evaluiert. Gemäß dem Abdeckungskriterium hat die Transformation für Variante 1 den Aufführungspfad $P_1 = n1 \rightarrow n2 \rightarrow f3 \rightarrow f5 \rightarrow n6$ ergeben, da in dieser Variante auf den Kontrollfluss $f4$ nicht der Stereotyp *TestFlow* angewendet wurde. Für die Variante 2 hingegen wurde nur der Pfad $P_2 = n1 \rightarrow n2 \rightarrow f4 \rightarrow n4 \rightarrow n7$ erzeugt, da in dieser Variante der Kontrollfluss $f4$ annotiert wurde, nicht aber $f3$. Für die Variante 3 wurden beide Pfade P_1 und P_2 erzeugt.

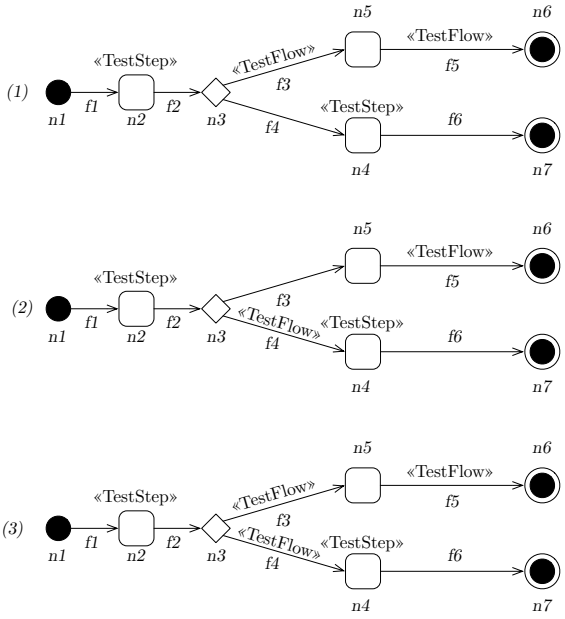
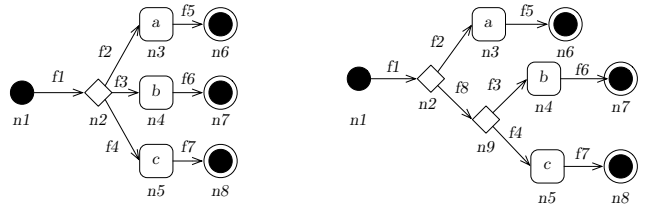


Abbildung 6.17: Minimales annotiertes Aktivitätsdiagramm aus dem Modellierungsmuster disjunktive Verzweigung

Die in Abbildung 6.15 dargestellten minimalen Aktivitätsdiagramme aus dem Modellierungsmuster disjunktive Kontrollflussverzweigung sind auf zwei alternative Kontrollflussverläufe nach dem Entscheidungsknoten beschränkt. Das Metamodell der UML erlaubt beliebig viele disjunktive, alternative Kontrollflussverläufe. Ein Modell, das mehr als zwei Kontrollflussalternativen an einem Entscheidungsknoten abbildet, kann auf eine Hintereinanderschaltung mehrerer Modellierungsmuster des Typs disjunktive Verzweigung zurückgeführt werden. Aufgrund des bereits erbrachten Nachweises der Korrektheit der Transformation des Modellierungsmusters Sequenz (vgl. Abschnitt 6.3.1.1), ist die Korrektheit der Transformation hintereinandergeschalteter disjunktiver Verzweigungen indirekt nachgewiesen.

Ein Beispiel für hintereinandergeschaltete disjunktive Verzweigungen ist in Abbildung 6.18 dargestellt. Die in Abbildung 6.18a und Abbildung 6.18b dargestellten Aktivitätsdiagramme



(a) Multiple Kontrollflussalternativen an einer *UML::DecisionNode*

(b) Verkettung von *UML::DecisionNode*-Elementen mit jeweils zwei Alternativen zu einer multiplen Kontrollflussverzweigung

Abbildung 6.18: Minimale Aktivitätsdiagramme mit einem erweiterten Modellierungsmuster disjunktive Kontrollflussverzeigung.

sind im Sinne des Metamodells semantisch äquivalent, wenngleich sie syntaktisch verschieden voneinander strukturiert sind. Für das Modellierungselement *UML::DecisionNode* definiert das UML-Profil zur Testfallmodellierung keinen Stereotyp, so dass dieses Element per-se nicht als testrelevant annotierbar ist. Der in Abschnitt 5.3.3.1 vorgestellte Transformationsalgorithmus entfernt nicht testrelevante Elemente bei der Generierung von Ausführungspfaden, so dass die semantische Äquivalenz der Modellfragmente in Abbildung 6.18a und Abbildung 6.18b auch bei der Transformation von Aktivitätsdiagramm zu Testfallmodell erhalten bleibt.

Die Transformationsergebnisse der Aktivitätsdiagramme in Abbildung 6.18a und Abbildung 6.18b sind in Quellcodefragment 6.1 sowie Quellcodefragment 6.2 abgedruckt.

Quellcodefragment 6.1: Textuelle Repräsentierung der aus dem Aktivitätsdiagramm in Abbildung 6.18a generierten Ausführungspfade

```
1 !n1->f1->f2->n3->f5->!n6
2 !n1->f1->f3->n4->f6->!n7
3 !n1->f1->f4->n5->f7->!n8
```

Quellcodefragment 6.2: Textuelle Repräsentierung der aus dem Aktivitätsdiagramm in Abbildung 6.18b generierten Ausführungspfade

```
1 !n1->f1->f2->n3->f5->!n6
2 !n1->f1->f8->f3->n4->f6->!n7
3 !n1->f1->f8->f4->n5->f7->!n8
```

Mit Ausnahme des Kontrollflusses $f8$ sind die Mengen der Ausführungspfade in Quellcodefragment 6.1 sowie Quellcodefragment 6.2 identisch. Kontrollfluss $f8$ verbindet die beiden *UML::DecisionNode*-Elemente $n2$ und $n9$ in dem in Abbildung 6.18b dargestellten Aktivitätsdiagramm. Vom Grundsatz her ist es nicht notwendig, diesen Kontrollfluss mit dem Stereotyp *TestFlow* zu annotieren, da die aus dem Knoten $n9$ entspringenden Kontrollflüsse jeweils mit dem Stereotyp *TestFlow* annotiert werden müssen, um die effektive Vorbedingung der Verzweigung abzubilden. Die im Rahmen der Validierung erarbeitete prototypische Implementierung der Testfallgenerierung (vgl. Abschnitt 6.2.3) verarbeitet diesen Kontrollfluss jedoch, weshalb der Fluss $f8$ in Quellcodefragment 6.2 enthalten ist.

6.3.1.3 Disjunktive Vereinigung

Das ebenfalls von Wohed et al., nicht aber von Staines identifizierte Pendant zum Modellierungsmuster disjunktive Verzweigung ist die disjunktive Vereinigung. Sie setzt unterschiedliche Kontrollflüsse auf einem gemeinsamen Strang des Kontrollflusses fort. Ziel der Validierung der Modelltransformation ist es, den Nachweis zu erbringen, dass auch für minimale, nicht-leere UML-Aktivitätsdiagramme, die aus dem Muster disjunktive Vereinigung gebildet werden können, im Sinn des Abdeckungskriteriums korrekte Testfallmodelle gebildet werden.

In Abbildung 6.19 ist das Modellierungsmuster dargestellt. Analog zum Modellierungsmuster disjunktive Verzweigung ist es auch hier wieder möglich, mehr als nur zwei Kontrollflussabschnitte durch ein *UML::MergeNode*-Element zu vereinigen.

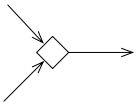
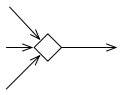
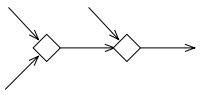


Abbildung 6.19: Muster disjunkive Kontrollflussvereinigung

Eine Vereinigung von mehr als zwei Kontrollflussabschnitten kann auch hier als Hintereinanderschaltung mehrerer disjunktiver Vereinigungen ausgedrückt werden. Dieser Fall ist in Abbildung 6.20 dargestellt. Die beiden Modellfragmente in Abbildung 6.20a und Abbildung 6.20b modellieren semantisch dieselbe Situation, unterscheiden sich aber strukturell voneinander.



(a) Modellierungsmuster disjunkive Kontrollflussvereinigung mit mehreren eingehenden Kontrollflüssen



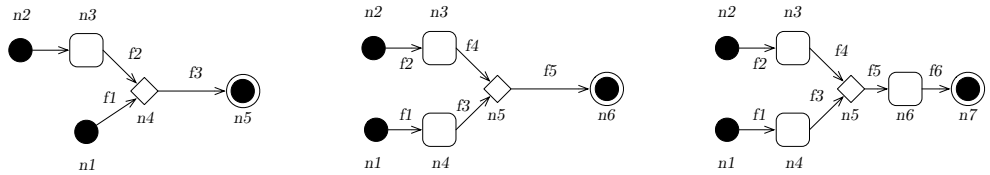
(b) Hintereinanderschatung des Modellierungsmusters disjunkive Kontrollflussvereinigung mit jeweils zwei eingehenden Kontrollflüssen

Abbildung 6.20: Minimale Aktivitätsdiagramme mit dem Modellierungsmuster disjunktive Kontrollflussvereinigung

Analog zum Modellierungsmuster disjunktive Verzweigung (vgl. Abschnitt 6.3.1.2) ist es auch hier wieder ausreichend, die Korrektheit der Transformation für zwei eingehende Kontrollflüsse in ein *UML::MergeNode*-Element zu zeigen, weil die Korrektheit der Transformation für hintereinander auftretende Modellierungselemente durch den Nachweis der Korrektheit der Transformation für das Muster Sequenz (vgl.Abschnitt 6.3.1.1) bereits erbracht ist.

Zum Muster disjunktive Vereinigung existieren mehrere minimale, nichtleere Aktivitätsdiagramme, auf die im Rahmen der Validierung jeweils individuell der Transformationsalgorithmus angewendet wurde. Diese Aktivitätsdiagramme sind in Abbildung 6.21 dargestellt.

Gegenüber dem im vorherigen Abschnitt 6.3.1.2 diskutierten Modellierungsmuster ergibt sich hier die Besonderheit, dass mehrere Anfangskonten vorhanden sind, wodurch nebenläufige Kontrollflüsse entstehen.



(a) Muster Aktivitätssequenz vor der Vereinigung

(b) Alternative Muster Aktivitätssequenzen vor der Vereinigung

(c) Alternative Muster Aktivitätssequenzen vor der Vereinigung, Muster Aktivitätssequenzen nach der Vereinigung

Abbildung 6.21: Minimale Aktivitätsdiagramme mit dem Modellierungsmuster disjunktive Kontrollflussvereinigung.

Ausschlaggebend für die Transformation des Modellierungsmusters disjunktive Vereinigung ist, ob alternative Kontrollflüsse im Vorbereich des Vereinigungsknotens nebenläufig sind oder durch einen vorhergehenden Verzweigungsknoten exklusive Alternativen sind. Das Modellierungslement *UML::MergeNode* zeichnet sich dadurch aus, dass es im Gegensatz zum Modellierungselement *UML::JoinNode* (vgl. Abschnitt 6.3.1.5) den Kontrollfluss nicht blockiert, bis auf allen eingehenden Kanten Marken anliegen. Unter der Voraussetzung, dass Kontrollflüsse im Vorbereich der Vereinigung nebenläufig sind, ist zu erwarten, dass die Transformation Ausführungspfade erzeugt, in denen Knoten aus dem Nachbereich der Vereinigung mehrfach vorkommen.

In Abbildung 6.22 ist das aus dem Aktivitätsdiagramm in Abbildung 6.21c generierte Petri-Netz dargestellt. Der Bereich des Netzes, der mehrfaches Vorkommen von Knoten (d. h. mehrfaches Vorkommen von Testschritten im Testmodell) im Nachbereich der Vereinigung verursacht, ist in der Abbildung hervorgehoben.

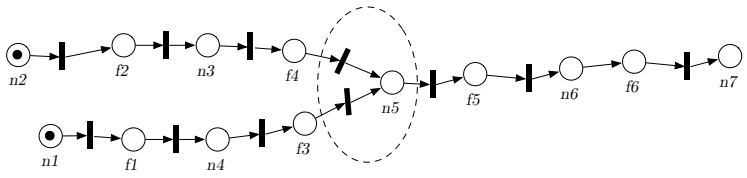


Abbildung 6.22: Aus dem Aktivitätsdiagramm in Abbildung 6.21c generiertes Petri-Netz.

Aufgrund der nebenläufigen Abschnitte vor dem Vereinigungsknoten $n5$ erzeugt die Transformation bereits für die minimalen UML-Aktivitätsdiagramme in Abbildung 6.21 eine große Anzahl Ausführungspfade. Konkret für das Aktivitätsdiagramm in Abbildung 6.21c werden bei vollständiger Annotation (d. h. auf jedes Element im Modell wird der Stereotyp *TestFlow* bzw. *TestStep* angewendet) bereits 700 unterschiedliche Ausführungspfade generiert. Die Validierung der Korrektheit der Transformation erfolgt deshalb nicht über manuelle Inspektion der einzelnen Ausführungspfade. Stattdessen wurde die Menge der generierten Pfade auf Enthaltensein von i. S. des Abdeckungskriteriums unzulässige Ausführungssequenzen untersucht.

Im Aktivitätsdiagramm in Abbildung 6.21c sind in der resultierenden Testfallmenge Ausführungssequenzen unzulässig, in denen die Knoten oder Kanten (d. h. mit diesen assoziierte Vor- oder Nachbedingungen bzw. Interaktionen) im Kontrollflussabschnitt zwischen $n2$ und $n5$ sowie zwischen $n1$ und $n5$ in vertauschter Reihenfolge auftreten. Ausführungspfade mit der Anfangssequenz $P = n3 \rightarrow f2 \rightarrow n2 \rightarrow f4 \rightarrow n5$ sind beispielsweise unzulässig, weil im Aktivitätsdiagramm das Element $n2$ vor dem Element $n3$ spezifiziert ist, eine Ausführung in anderer Reihenfolge repräsentiert daher keinen gültigen Testfall.

Die Modelltransformation erzeugt als Zwischenergebnis eine textuelle Repräsentation der Ausführungssequenzen. Ein Ausschnitt der Menge der aus dem Aktivitätsdiagramm in Abbildung 6.21c generierten Ausführungspfade ist in Quellcodefragment 6.3 dargestellt.

Quellcodefragment 6.3: Textuelle Repräsentierung eines Ausschnitts der Menge der aus dem Aktivitätsdiagramm in Abbildung 6.21c generierten Ausführungspfade

```
1  f1-n4-f2-n3-f3-!n5-f4-f5-!n5-n6-f5-f6-n6
2  f1-n4-f2-n3-f3-!n5-f4-f5-!n5-n6-f5-f6
3  f1-n4-f2-n3-f3-!n5-f4-f5-!n5-n6-f6-f5-n6
4  f1-n4-f2-n3-f3-!n5-f4-f5-!n5-n6-f6-f5
5  f1-n4-f2-n3-f3-!n5-f4-f5-!n5-n6-f6
6  f1-n4-f2-n3-f3-!n5-f4-f5-n6-!n5-f5-f6-n6
7  f1-n4-f2-n3-f3-!n5-f4-f5-n6-!n5-f5-f6
8  f1-n4-f2-n3-f3-!n5-f4-f5-n6-!n5-f6-f5-n6
9  f1-n4-f2-n3-f3-!n5-f4-f5-n6-!n5-f6-f5
10 f1-n4-f2-n3-f3-!n5-f4-f5-n6-!n5-f6
11 ...
```

Das Vorkommen eines Knotens oder einer Kante (z. B. $f1$) in einem Ausführungspfad bedeutet, dass die mit diesem Knoten (Kante) assoziierte Information zur Testausführung an dieser Position ausgewertet wird (d. h. beim Testen ins SUT eingespielt wird). Der Knoten $n5$ ist im Quellcodefragment 6.3 mit vorausgehendem Ausrufezeichen als „!$n5$“ dargestellt. Der Knoten ist im Zwischenergebnis der Transformation auf diese Weise hervorgehoben dargestellt, weil das UML-Profil zur Testfallmodellierung keinen Stereotyp zur Annotation dieses Knotentyps definiert. Im finalen Testfallmodell ist der Knoten nicht mehr enthalten.

Zur indirekten Validierung des Transformationsalgorithmus wurde auf die textuelle Darstellung des Zwischenergebnisses der Transformation eine Textsuche mit regulären Ausdrücken angewendet, die unzulässige Teilsequenzen identifizieren. Eine Suche nach dem regulären Ausdruck `!n5.*n2` identifiziert beispielsweise unzulässige Teilsequenzen, in denen der Knoten $n5$ vor dem Knoten $n2$ ausgeführt wird.

Diese Analyse wurde für alle in der Menge der aus Aktivitätsdiagramm in Abbildung 6.21c generierten Ausführungspfade möglichen Kontrollflussabschnitte zwischen $n2$ und $n5$ sowie zwischen $n1$ und $n5$ durchgeführt. Hierbei konnte keine unzulässige Teilsequenz identifiziert werden, wodurch die Korrektheit der Transformation für das Modellierungsmuster disjunktive Vereinigung nebenläufiger Kontrollflussstrukturen indirekt nachgewiesen ist.

6.3.1.4 Konjunktive Verzweigung

Ein weiteres sowohl von Wohed et al. als auch von Staines identifiziertes Modellierungsmuster ist die konjunktive Verzweigung, die im Syntax der UML als *UML::ForkNode* modelliert wird. In der Semantik des Markenflusses platziert dieses Modellierungselement Marken an allen ausgehenden Kontrollflüssen, wodurch der Kontrollfluss explizit parallelisiert wird.

In der Semantik der Testausführung bedeutet die Verwendung dieses Modellierungsmusters, dass testrelevante, auf den nebenläufigen Kontrollflüssen verlaufenden Aktivitäten gemischt, aber nicht in ihrer Reihenfolge vertauscht ausgeführt werden. Hierbei gilt weiterhin die Annahme, dass ein Anwender nicht in der Lage ist, im Sinne einer Parallelisierung zeitgleich mit mehreren Komponenten einer App zu interagieren. Beim Durchführen des Testens werden sie gemäß des in Abschnitt 5.3.2 definierten Abdeckungskriteriums sequenzialisiert.

Hierin manifestiert sich der Gegenstand der Validierung der Transformation von UML-Aktivitätsdiagrammen, die auf dem Muster konjunktive Verzweigung basieren. Es muss überprüft werden, ob die Transformation ausschließlich i. S. des Abdeckungskriteriums zulässige Ausführungssequenzen generiert. Um dies zu überprüfen, wurden analog zu den vorangegangenen Modellierungsmustern auch für die konjunktive Verzweigung minimale Aktivitätsdiagramme gebildet, auf welche die Transformation angewendet wurde. Das Muster ist in Abbildung 6.23 dargestellt, die zugehörigen minimalen Aktivitätsdiagramme in Abbildung 6.24.

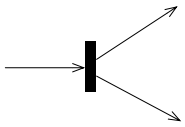
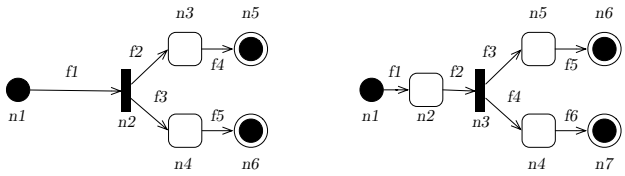


Abbildung 6.23: Muster konjunktive Kontrollflussverzweigung



(a) Muster Aktivitätssequenzen nach der konjunktiver Verzweigung

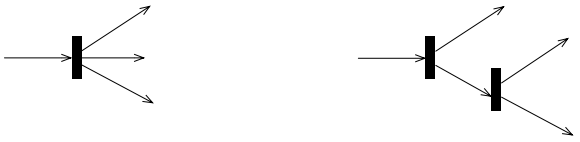
(b) Aktivitätssequenzen vor und nach der Verzweigung

Abbildung 6.24: Minimale Aktivitätsdiagramme mit dem Modellierungsmuster konjunkive Kontrollflussverzeigung

Analog zu den anderen Mustern gilt, dass ein Element des Typs *UML::ForkNode* mehr als zwei ausgehende Kontrollflüsse haben kann. Diese Situation kann wie bei Muster disjunktive Verzweigung durch Hintereinanderschalten mehrerer *UML::ForkNode*-Elemente mit jeweils zwei ausgehenden Kanten modelliert werden. Dieser Fall ist in Abbildung 6.25 dargestellt.

Für das Muster konjunktive Verzweigung existieren nur zwei minimale Aktivitätsdiagramme, da eine Variante, in der nur auf einem der konjunktiven Kontrollflüsse eine Aktivität modelliert ist, keine Nebenläufigkeit modelliert. Es ist aber sinnvoll, beide in Abbildung 6.24 dargestellten minimalen Aktivitätsdiagramme zu definieren, damit das Muster so zur Bildung komplexerer Modelle verwendet werden kann, die Ersetzungsoperation auf der Aktivität im Vorbereich des *UML::ForkNode*-Elements durch andere, ggf. komplexere Muster ermöglicht.

Der Transformationsalgorithmus hat das in Abbildung 6.26 dargestellte Petri-Netz erzeugt. Es unterscheidet sich von dem in Abbildung 6.16a dargestellten Netz im graphisch



(a) Multiple Kontrollflussparallelisierung an einer *UML::DecisionNode*

(b) Verkettung mehrerer *UML::ForkNode*-Elemente mit jeweils zwei ausgehenden Kontrollflüssen zu einer multiplen Kontrollflussparallelisierung

Abbildung 6.25: Semantisch äquivalente Modellierungsmuster konjunktive Kontrollflussverzeigung

hervorgehobenen Bereich, konkret folgt hier auf den Knoten $n3$ nur eine einzelne Transition (zwei Transitionen in Abbildung 6.16a), allerdings mit zwei Stellen im Nachbereich, wodurch die im Aktivitätsdiagramm modellierte Situation, d. h. eine explizite Parallelisierung des Kontrollflusses, durch das Petri-Netz abgebildet wird.

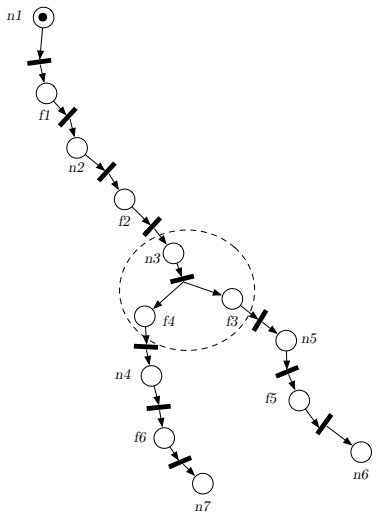


Abbildung 6.26: Aus dem Aktivitätsdiagramm in Abbildung 6.24b generiertes Petri-Netz

Die Anwendung des Transformationsalgorithmus hat neben dem in Abbildung 6.26 dargestellten Petri-Netz die 20 in Quellcodefragment 6.4 abgedruckten Ausführungssequenzen erzeugt. Auffällig ist, dass nicht in allen Ausführungssequenzen alle im Aktivitätsdiagramm aus Abbildung 6.24b modellierten Knoten enthalten sind. Basierend auf der im Metamodell der UML [268] definierten Semantik des Elements *UML::ActivityFinalNode* ist dieses Transformationsergebnis korrekt, da nach dem Erreichen des ersten *UML::ActivityFinalNode*-Elements auf einem beliebigen Kontrollfluss in einem Aktivitätsdiagramm alle Marken aus dem Netz gelöscht werden und die Ausführung unmittelbar terminiert.

Dass die Transformation dieser Anforderung genügt ist daran erkennbar, dass in keiner der in Quellcodefragment 6.4 abgebildeten Ausführungssequenzen nach den Knoten $!n6$ oder $!n7$ (beide repräsentieren Enknoten im Aktivitätsdiagramm) ein weiterer Knoten folgt. Die manuelle Inspektion der Transformationsergebnisse hat ebenfalls ergeben, dass in allen gene-

rierten Ausführungssequenzen bis enschließlich zur Stelle $f3 \rightarrow f4$ die Knoten in genau der im Aktivitätsdiagramm modellierten Reihenfolge auftreten.

Allen Ausführungspfaden ist die Anfangssequenz $!n1 \rightarrow \ldots f4$ gemeinsam. Auf Knoten $f4$ folgen variierende Ausführungspfade, die jeweils unterschiedliche Markenbelegungen des zugehörigen Petri-Netzes repräsentieren. Auffällig ist, dass in den generierten Pfaden lediglich die Teilsequenz $f3 \rightarrow f4$ vertreten ist, nicht aber $f4 \rightarrow f3$, was ebenfalls eine gültige Permuationssequenz nebenläufiger Abschnitte des zum Petri-Netz in Abbildung 6.26 gehörigen Erreichbarkeitsgraphen wäre. Das in Abschnitt 5.3.2 definierte Permutationssequenzüberdeckungskriterium fordert jedoch zur Vermeidung des State-Explosion-Problems lediglich, dass mindenstens eine zulässige Sequenzialisierung durch einen Test abgedeckt wird. Dieses Kriterium wird durch die in Quellcodefragment 6.4 dargestellte Menge von Ausführungspfaden erfüllt.

Zu Sicherstellung, dass in der Menge der generierten Ausführungspfade ausschließlich zulässige Permutationen nebenläufiger Elemente enthalten sind, wurde analog zur Evaluierung der disjunktiven Vereinigung (Abschnitt 6.3.1.3) mit Hilfe regulärer Ausdrücke überprüft, dass keine unzulässigen Teilsequenzen enthalten sind.

Quellcodefragment 6.4: Textuelle Repräsentierung eines Ausschnitts der Menge der aus dem Aktivitätsdiagramm in Abbildung 6.21c generierten Ausführungspfade

```
1  !n1->f1->n2->f2->f3->f4->n5->n4->f5->!n6
2  !n1->f1->n2->f2->f3->f4->n5->n4->f5->f6->!n7
3  !n1->f1->n2->f2->f3->f4->n5->n4->f5->f6->!n6
4  !n1->f1->n2->f2->f3->f4->n5->n4->f6->f5->!n7
5  !n1->f1->n2->f2->f3->f4->n5->n4->f6->f5->!n6
6  !n1->f1->n2->f2->f3->f4->n5->n4->f6->!n7
7  !n1->f1->n2->f2->f3->f4->n5->f5->n4->!n6
8  !n1->f1->n2->f2->f3->f4->n5->f5->n4->f6->!n7
9  !n1->f1->n2->f2->f3->f4->n5->f5->n4->f6->!n6
10 !n1->f1->n2->f2->f3->f4->n5->f5->!n6
11 !n1->f1->n2->f2->f3->f4->n4->n5->f5->!n6
12 !n1->f1->n2->f2->f3->f4->n4->n5->f5->f6->!n7
13 !n1->f1->n2->f2->f3->f4->n4->n5->f5->f6->!n6
14 !n1->f1->n2->f2->f3->f4->n4->n5->f6->f5->!n7
15 !n1->f1->n2->f2->f3->f4->n4->n5->f6->f5->!n6
16 !n1->f1->n2->f2->f3->f4->n4->n5->f6->!n7
17 !n1->f1->n2->f2->f3->f4->n4->f6->n5->f5->!n7
18 !n1->f1->n2->f2->f3->f4->n4->f6->n5->f5->!n6
19 !n1->f1->n2->f2->f3->f4->n4->f6->n5->!n7
20 !n1->f1->n2->f2->f3->f4->n4->f6->!n7
```

6.3.1.5 Konjunktive Synchronisation

Die konjunktive Vereinigung von Kontrollflüssen ist ein weiteres elementares Modellierungsmuster, das auch bereits von Wohed et al. sowie von Staines identifiziert wurde. Zur Modellierung dieses Musters stellt die UML das Element *UML::JoinNode* bereit. In der Semantik des Markenflusses durch ein Aktivitätsdiagramm repräsentiert dieses Muster eine Synchronisation nebenläufiger Kontrollflüsse, wobei die Fortführung des Kontrollflusses so lange blockiert, bis an allen eingehenden Kontrollflüssen eine Marke anliegt.

In der Semantik des Testens einer Anwendung bedeutet die Verwendung dieses Modellierungsmusters in der Abbildung des Kontrollflusses, dass bis zu einem *UML::JoinNode*-Element nebenläufig modellierte Aktivitäten gemischt, aber nicht in ihrer Reihenfolge vertauscht vor-

kommen dürfen und der Kontrollfluss nach dem *UML::ForkNode*-Element in einem einzelnen Strang ohne lokale Nebenläufigkeit vorgesetzt wird.

Gegenstand der Validierung der Transformation von UML-Aktivitätsdiagrammen, die auf dem Modellierungsmuster konjunktive Vereinigung basieren, ist es, den Nachweis zu erbringen, dass nur solche Ausführungssequenzen testrelevanter Aktivitäten erzeugt werden, die gemäß des in Abschnitt 5.3.2 definierten Abdeckungskriteriums zulässig sind. Insbesondere muss gezeigt werden, dass keine Ausführungssequenzen erzeugt werden, in denen Aktivitäten, die ursprünglich auf nebenläufigen Kontrollflüssen vor der konjunktiven Vereinigung stehen, in den durch die Transformation generierten Ausführungspfad nach der Vereinigung vorkommen.

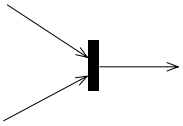


Abbildung 6.27: Muster konjunktive Kontrollflussvereinigung

In Abbildung 6.27 ist das Muster konjunktive Kontrollflussvereinigung dargestellt. Analog zu den bereits diskutieren Mustern lässt es die Syntax der UML zu, dass mehrere eingehende Kanten an einem *UML::JoinNode*-Element modelliert werden. Auch diese Situation kann durch Hintereinanderschalten mehrerer *UML::JoinNode*-Elemente mit jeweils zwei eingehenden Kanten modelliert werden (vgl. Abbildung 6.28).

(a) Multiple Kontrollflussparallelisierung an einer *UML::DecisionNode*

(b) Verkettung mehrerer *UML::ForkNode*-Elemente mit jeweils zwei ausgehenden Kontrollflüssen zu einer multiplen Kontrollflussparallelisierung

Abbildung 6.28: Semantisch äquivalente erweiterte Modellierungsmuster konjunktive Kontrollflussvereinigung.

Zu diesem Muster existieren ebenfalls minimale UML-Aktivitätsdiagramme. Für die Validierung wurde lediglich das in Abbildung 6.27 dargestellte minimale Aktivitätsdiagramm ausgewählt, in welchem sowohl vor dem Vereinigungsknoten auf allen nebenläufigen Kontrollflüssen Aktivitäten vorkommen als auch eine Aktivität nach dem Vereinigungsknoten. Andere Varianten analog der disjunktiven Vereinigung (vgl. Abschnitt 6.3.1.3) wurden nicht berücksichtigt, da sie das Muster auf triviale Fälle reduzieren, in denen tatsächlich keine Nebenläufigkeit existiert (z. B. nur Aktivitäten auf einem der nebenläufigen Kontrollflüsse) oder in denen die Synchronisation für den weiteren Kontrollflussverlauf folgenlos ist, d. h. keine weiteren Aktivitäten nach der Vereinigung modelliert sind.

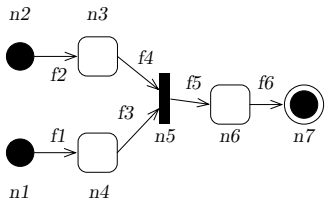


Abbildung 6.29: Minimales Aktivitätsdiagramm aus dem Modellierungsmuster konjunktive Kontrollflussvereinigung

Als Zwischenergebnis der Transformation wurde das in Abbildung 6.30 dargestellte Petri-Netz erzeugt. Es unterscheidet sich vom Petri-Netz zum vergleichbaren Modellierungsmuster disjunktive Vereinigung durch den graphisch hervorghobenen Bereich (vgl. Abschnitt 6.3.1.3, Abbildung 6.22).

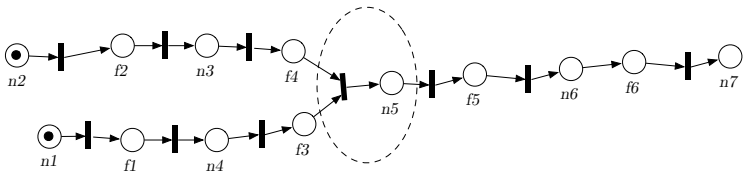


Abbildung 6.30: Aus dem Aktivitätsdiagramm in Abbildung 6.29 generiertes Petri-Netz.

Durch die Anwendung des Transformationsalgorithmus auf das in Abbildung 6.29 dargestellte Aktivitätsdiagramm wurden 20 unterschiedliche Ausführungssequenzen generiert. Diese sind in Quellcodefragment 6.5 abgebildet. Zu berücksichtigen ist, dass die mit ! beginnenden Knoten (z. B. $!n1$, $!n2$) für das im nächsten Schritt generierte Testfallmodell nicht relevant sind, weil sie Elemente des UML-Aktivitätsdiagramms repräsentieren, für welches das UML-Profil zur Testfallmodellierung (vgl. Abschnitt 5.2.4) keine Stereotypen zur Annotation definiert, d. h. sie können nicht Träger von Informationen zu Vor- oder Nachbedingungen sein.

Durch manuelle Überprüfung wurde sichergestellt, dass aus dem in Abbildung 6.29 dargestellten UML-Aktivitätsdiagramm nur im Sinne des in Abschnitt 5.3.2 definierten Permutationssequenzüberdeckungskriterium $C_{PS_{ACD}}$ zulässige Ausführungssequenzen erzeugt wurden.

Quellcodefragment 6.5: Textuelle Repräsentierung der aus dem Aktivitätsdiagramm in Abbildung 6.29 generierten Ausführungspfade

```
1  !n1->!n2->f1->n4->f2->n3->f3->f4->!n5->f5->n6->f6->!n7
2  !n1->!n2->f1->n4->f2->n3->f4->f3->!n5->f5->n6->f6->!n7
3  !n1->!n2->f1->n4->f2->f3->n3->f4->!n5->f5->n6->f6->!n7
4  !n1->!n2->f1->n4->f3->f2->n3->f4->!n5->f5->n6->f6->!n7
5  !n1->!n2->f1->f2->n4->n3->f3->f4->!n5->f5->n6->f6->!n7
6  !n1->!n2->f1->f2->n4->n3->f4->f3->!n5->f5->n6->f6->!n7
7  !n1->!n2->f1->f2->n4->f3->n3->f4->!n5->f5->n6->f6->!n7
8  !n1->!n2->f1->f2->n3->n4->f3->f4->!n5->f5->n6->f6->!n7
9  !n1->!n2->f1->f2->n3->n4->f4->f3->!n5->f5->n6->f6->!n7
10 !n1->!n2->f1->f2->n3->f4->n4->f3->!n5->f5->n6->f6->!n7
11 !n1->!n2->f2->f1->n4->n3->f3->f4->!n5->f5->n6->f6->!n7
12 !n1->!n2->f2->f1->n4->n3->f4->f3->!n5->f5->n6->f6->!n7
13 !n1->!n2->f2->f1->n4->f3->n3->f4->!n5->f5->n6->f6->!n7
14 !n1->!n2->f2->f1->n3->n4->f3->f4->!n5->f5->n6->f6->!n7
15 !n1->!n2->f2->f1->n3->n4->f4->f3->!n5->f5->n6->f6->!n7
16 !n1->!n2->f2->f1->n3->f4->n4->f3->!n5->f5->n6->f6->!n7
17 !n1->!n2->f2->n3->f1->n4->f3->f4->!n5->f5->n6->f6->!n7
18 !n1->!n2->f2->n3->f1->n4->f4->f3->!n5->f5->n6->f6->!n7
```

```
19  !n1->!n2->f2->n3->f1->f4->n4->f3->!n5->f5->n6->f6->!n7
20  !n1->!n2->f2->n3->f4->f1->n4->f3->!n5->f5->n6->f6->!n7
```

Durch Anwendung eines regulären Ausdrucks wurde analog zur Validierung der disjunktiven Vereinigung (vgl. Abschnitt 6.3.1.3) sichergestellt, dass die generierten Ausführungssequenzen keine unzulässigen Teilsequenzen enthalten, in denen Aktivitäten der nebenläufigen Kontrollflüsse vor der Vereinigung in der Reihenfolge vertauscht vorkommen. Weiterhin ist gewährleistet, dass nach der Stelle der Vereinigung, repräsentiert durch den Knoten $!n5$, die Ausführung vor der Fortsetzung mit $f5$ blockiert, bis alle Aktivitäten auf nebenläufigen Kontrollflüssen abgearbeitet sind. Als besonderes Beispiel kann hierfür Zeile 20 in Quellcodefragment 6.5 herangezogen werden, in dem zunächst einer der nebenläufigen Kontrollflüsse aus dem Aktivitätsdiagramm in Abbildung 6.29 (jeweils der untere Abschnitt in Abbildung 6.29 und Abbildung 6.30) vollständig abgearbeitet wird, bevor der zweite nebenläufige Kontrollfluss und anschließend die Teilsequenz nach der Vereinigung abgearbeitet wird.

6.3.1.6 Zusammenfassung Validierung der Transformation atomarer Modellierungsmuster

In den vorangegangenen Abschnitten wurden fünf atomare Modellierungsmuster definiert, die bei der Modellierung von UML-Aktivitätsdiagrammen verwendet werden. Die Auswahl der untersuchten Muster stützte sich hierbei auf eine Analyse existierender wissenschaftlicher Literatur mit dem Ziel, im nächsten Schritt der Validierung nachzuweisen, dass komplexe Aktivitätsdiagramme auf Kombinationen atomarer Muster zurückgeführt werden können.

Für die in den vorangegangen Abschnitten untersuchten Modellierungsmuster wurden jeweils minimale UML-Aktivitätsdiagramme gebildet, d. h. solche, die durch Entfernen weiterer Modellierungselemente zu trivialen Aktionssequenzen oder dem leeren Modell degenerieren. Auf diese Muster wurde jeweils der in Abschnitt 5.3.3 vorgestellte Transformationsalgorithmus angewendet und durch Inspektion der resultierenden Testfallmodelle und insb. der Zwischenergebnisse die Korrektheit der Transformation bestätigt.

6.3.2 Validierung der Transformation zusammengesetzter Modelle

Die Strategie zur Validierung des Transformationsalgorithmus basiert auf der Voraussetzung, dass komplexe Modelle stets als Kombinationen von Modellen geringerer Komplexität interpretiert werden können. Diese Voraussetzung stützt sich auf das Metamodell der UML, welches zur Modellierung von Aktivitätsdiagrammen das Element *UML::CallBehaviorAction* definiert, durch welches individuelle Teilmodelle in Aktivitätsdiagrammen in beliebigem Detailgrad vergröbert oder verfeinert werden können.

In den folgenden Abschnitten wird zum einen gezeigt, dass höhere Modelle durch Kombinationen der im vorangegangenen Abschnitt definierten Modellierungsmuster erzeugt werden können. Zum anderen wird gezeigt, dass der in Abschnitt 5.3.3 vorgestellte Transformationsalgorithmus auch höhere Modelle korrekt in Testfallmodelle überführt.

Die Kombination atomarer Modellierungsmuster zu höheren Modellen stützt sich hierbei auf Ersetzungsoperationen, die individuelle Aktivitäten innerhalb eines Aktivitätsdiagramms durch ein Teilmodell ersetzen oder umgekehrt ein Teilmodell eines Aktivitätsdia-

gramms durch eine Aktivität vom Typ *UML::CallBehaviorAction* ersetzt. Ersetzungsoperationen können hierbei auf knotenberandeten Teilmodellen oder auf kantenberandeten Teilmodellen ausgeführt werden, d. h. ein knotenberandetes Teilmodell kann durch ein knotenberandetes Teilmodell geringerer Komplexität ersetzt werden (z. B. ein einzelnes Element vom Typ *UML::CallBehaviorAction*), nicht aber durch ein kantenberandetes Teilmodell (analog für kantenberandetes Teilmodell).

Definition 9 Sei $ACD = (N, A, C, N_I, N_F, R)$ ein UML-Aktivitätsdiagramm (vgl. Definition Aktivitätsdiagramm, Abschnitt 5.3.2). Ein Teilmodell eines Aktivitätsdiagramms ist definiert als Tupel $Teil(ACD) = (N_{Teil}, A_{Teil}, C_{Teil}, R_{Teil})$ mit:

$N_{Teil} \subseteq N$;

$N_{Teil} \subseteq N$;

$A_{Teil} \subseteq A$;

$C_{Teil} \subseteq C$;

$R_{Teil} \subseteq R$.

Definition 10 Der Rand eines Teilnetzes ist definiert als Vereinigungsmenge von Knoten und Kanten $Rand(ACD) = N_{Teil} \cup R_{Teil}$, die das Teilmodell $Teil(ACD)$ und das Modell ACD gemeinsam haben, so dass:

für $k \in Rand(ACD)$ gilt $k \in N_{Teil}$ und k ist durch einen Kontrollfluss $r \in R \wedge r \notin R_{Teil}$ mit einem Knoten $l \in N \wedge l \notin Rand(ACD)$ verbunden

– oder –

für $e \in Rand(ACD)$ gilt $e \in R_{Teil}$ und e verbindet zwei Konten a, b mit $a \in Rand(ACD)$ und $b \in N \wedge b \notin Rand(ACD)$

Definition 11 Ein *knotenberandetes Teilmodell eines Aktivitätsdiagramms* ist definiert als Teilmodell $Teil(ACD)$ eines Aktivitätsdiagramms ACD dessen Randelemente Knoten sind, also:

$\forall k \in Rand(Teil(ACD))$ ist $k \in N_{Teil}$

Definition 12 Ein *kantenberandetes Teilmodell eines Aktivitätsdiagramms* ist definiert als Teilmodell $Teil(ACD)$ eines Aktivitätsdiagramms ACD dessen Randelemente Kontrollflüsse sind, also:

$\forall e \in Rand(Teil(ACD))$ ist $e \in R_{Teil}$

In Abbildung 6.31 ist ein UML-Aktivitätsdiagramm dargestellt, in welchem ein Teilmodell graphisch hervorgehoben ist. Das Teilmodell wird durch den Knoten $n3$ und die Kontrollflüsse $f5, f6$ berandet. Das Teilmodell ist weder kantenberandet noch knotenberandet.

Die Validierung der Modelltransformation basiert auf der Voraussetzung, dass komplexe Modelle aus niederen Teilmodellen zusammengesetzt sind. Teilmodelle können wiederum in niedere Teilmodelle zerlegt werden, die in der letzten Zerlegungsstufe nur noch aus den in

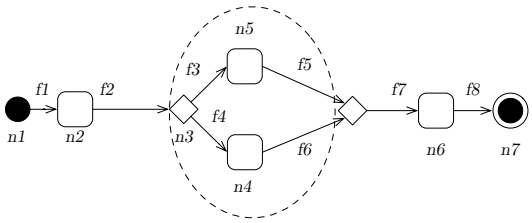


Abbildung 6.31: Beispiel eines Teilmodells

Abschnitt 6.3.1 diskutierten atomaren Modellierungsmustern bestehen, deren korrekte Transformation bereits in Abschnitt 6.3.1 nachgewiesen wurde.

Das kleinste, nichtleere knotenberandete Teilmodell ist das in Abschnitt 6.3.1.1 eingeführte Modellierungsmuster Sequenz der Länge eins. Wird es durch ein höheres Teilmodell ersetzt, wird hierdurch eine Verfeinerung des Modells repräsentiert. Ein Beispiel einer Ersetzungsoperation ist die in Abbildung 6.32 dargestellte Verfeinerung/Vergröberung, die das Modellierungselement *UML::CallBehaviorAction* verwendet, um ein knotenberandetes Teilmodell durch eine einzelne Aktivität zu ersetzen.

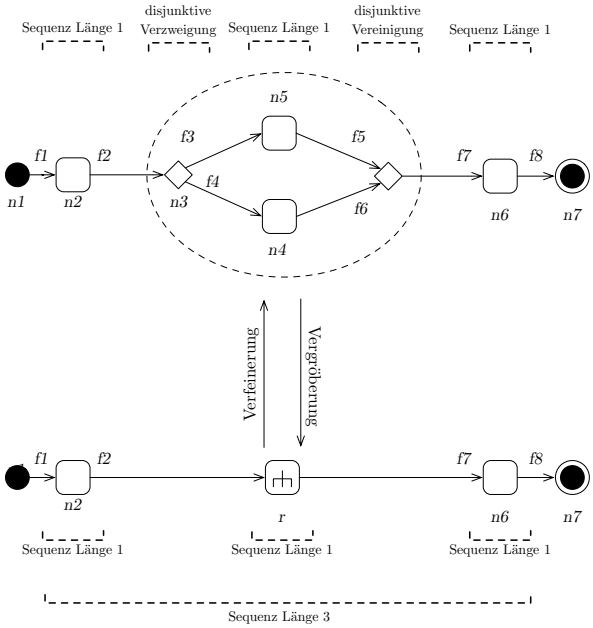


Abbildung 6.32: Beispiel einer Verfeinerung/Vergröberung

In Abbildung 6.32 ist im oberen Abschnitt ein Modell dargestellt, das aus den atomaren Modellierungsmustern Sequenz (vgl. Abschnitt 6.3.1.1), disjunktive Verzweigung (vgl. Abschnitt 6.3.1.2) sowie disjunktive Vereinigung (vgl. Abschnitt 6.3.1.3) kombiniert wurde. Zu erkennen ist, wie die Verfeinerung (bzw. Vergröberung) das knotenberandete Teilmodell im oberen Abschnitt der Abbildung auf eine einzelne Aktivität reduziert (bzw. erweitert). Für alle in Abbildung 6.32 identifizierbaren Modellierungsmuster ist die Korrektheit des Algorithmus zur Transformation von UML-Aktivitätsdiagramm zu Testfallmodell bereits nachgewiesen (vgl. Abschnitt 6.3.1). Die Vergröberung in Abbildung 6.32 zeigt zusätzlich, wie ein komplexes, aus mehreren Modellierungsmustern zusammengesetztes Modell, für das die Kor-

rektheit der Transformation noch nicht nachgewiesen wurde, auf ein Modell vergröbert wurde, in welchem ausschließlich Modellierungsmuster identifizierbar sind, deren korrekte Transformation bereits erwiesen ist.

In den folgenden Abschnitten werden analog zur Validierung atomarer Modellierungmuster (vgl. Abschnitt 6.3.1) komplexe Muster durch Anwendung der Transformation und manuelle Inspektion der Ergebnisse validiert.

6.3.2.1 Sequenz

Aktivitätsdiagramme, die ausschließlich durch Verwendung des Modellierungsmusters Aktivitätssequenz (vgl. Abschnitt 6.3.1.1) beliebiger Länge gebildet werden, können durch Vergröberung stets auf Aktivitätsdiagramme basierend auf dem Modellierungsmusters Aktivitätssequenz der Länge eins zurückgeführt werden. Ausgenutzt wird hierzu das Modellierungselement *UML::CallBehaviorAction* (vgl. OMG [268, S. 494]), welches es erlaubt, ein knotenberandetes Teilmodell eines Aktivitätsdiagramms durch ein einzelnes Modellierungselement zu ersetzen.

In Abbildung 6.33 ist dargestellt, wie ein Aktivitätsdiagramm, welches auf dem Modellierungsmuster Aktivitätssequenz der Länge n basiert (Stufe 1 in Abbildung 6.33), durch wiederholte Vergröberung auf ein Aktivitätsdiagramm reduziert wurde, welches nur noch aus dem Modellierungsmuster Aktivitätssequenz der Länge eins besteht (Stufe 4 in Abbildung 6.33). Die Vergröberung erfolgt jeweils durch Identifikation eines knotenberandeten Teilmodells, das auf dem Modellierungsmuster Aktivitätssequenz einer geringeren Länge basiert. Dieses kann durch eine Instanz von *UML::CallBehaviorAction* ersetzt werden.

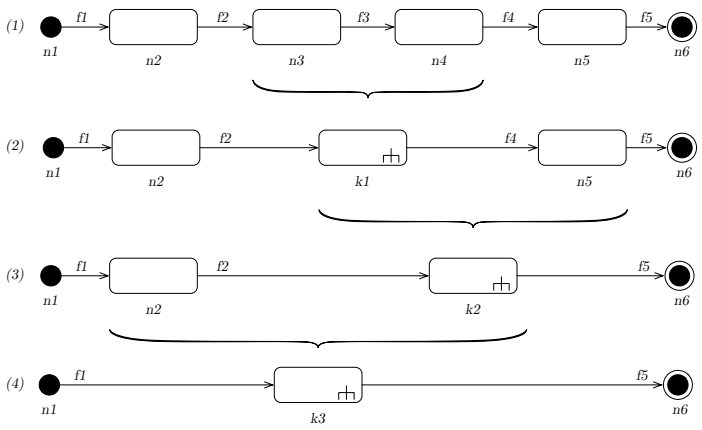


Abbildung 6.33: Reduktion eines Aktivitätsdiagramms durch Identifikation und Ersetzung des Modellierungsmusters Aktivitätssequenz beliebiger Länge

Für Stufe 4 und Stufe 3 in Abbildung 6.33 wurde die Korrektheit der Transformation des Modellierungsmusters Aktivitätssequenz in Abschnitt 6.3.1.1 direkt nachgewiesen. Für Stufe 2 gilt, dass der Nachweis der Korrektheit der Transformation implizit gegeben ist, da sich das Modell auf dieser Stufe durch Vergröberung in ein Modell überführen lässt, für das die Korrektheit der Transformation bereits bewiesen ist, konkret das Modell in Stufe 3; analog für Stufe 1 in Abbildung 6.33.

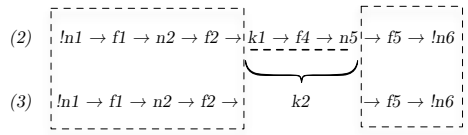


Abbildung 6.34: Ausführungspfade der Verfeinerungsstufen 2 und 3 des ACDs in Abbildung 6.33

In Abbildung 6.34 sind die generierten Ausführungspfade für die Stufen 2 und 3 exemplarisch dargestellt. Stufe 2 in Abbildung 6.33 repräsentiert eine Verfeinerung der Aktivität $k2$ in Stufe 3 durch ein knotenberandetes Teilmodell bestehend aus den Aktivitäten $k1$ und $n5$, die ihrerseits durch das Modellierungsmuster Sequenz abgebildet werden. Das Abdeckungskriterium fordert von der Transformation, dass der Abschnitt vor Knoten $k2$ sowohl in Stufe 2 und in Stufe 3 im generierten Ausführungspfad existiert (analog Abschnitt nach $k2$). Die Transformation hat für Stufe 2 in Abbildung 6.33 tatsächlich einen Ausführungspfad erzeugt, in dem diese Bedingung erfüllt ist. Durch die Verfeinerung wurde $k2$ durch ein Teilmodell ersetzt, für welches die Korrektheit der Transformation bereits bewiesen ist. Gemäß des Abdeckungskriteriums ist zu erwarten, dass der zu diesem Teilmodell gehörige Ausführungspfad an der richtigen Stelle im Ausführungspfad für Stufe 2 enthalten ist. Eine Analyse der Ausführungspfade für die 4 in Abbildung 6.33 dargestellten Aktivitätsdiagrammen hat ergeben, dass diese Forderung an die Transformation erfüllt ist.

Hierdurch ist nachgewiesen, dass Modelle, die auf dem Modellierungsmuster Aktivitätssequenz beliebiger Länge basieren, durch den Transformationsalgorithmus korrekt in Testfallmodelle überführt werden, da sie stets durch Vergröberung in Modellierungsmuster zerlegt werden können, für die die Korrektheit der Transformation bereits erwiesen ist.

6.3.2.2 Bypass und Alternative

Die im vorangegangenen Abschnitt 6.3.1 identifizierten Modellierungsmuster bilden die Grundlage für eine Reihe höherer Modellierungsmuster, zu denen die Muster Bypass und Alternative gehören. Sie verwenden die atomaren Modellierungsmuster disjunktive Verzweigung (vgl. Abschnitt 6.3.1.2) und disjunktive Vereinigung (vgl. Abschnitt 6.3.1.3), um in Abhängigkeit von einer Bedingung (repräsentiert durch ein Guard-Condition-Element an einer *UML::DecisionNode*) alternative Kontrollflüsse in einem UML-Aktivitätsdiagramm zu aktivieren, wobei alternative Kontrollflüsse zu einem späteren Zeitpunkt optional wieder zusammengeführt werden. Der Bypass ist eine besondere Form der Alternative, bei welcher eine Kontrollflussalternative verwendet wird, ein Teilmodell ersatzlos zu umgehen.

Die Muster Bypass und Alternative sind Gegenstand der Validierung der Transformation UML-Aktivitätsdiagramm zu Testfallmodell, weil sichergestellt werden muss, dass der Transformationsalgorithmus im Sinne des Permutationssequenzüberdeckungskriteriums (vgl. Abschnitt 5.3.2) korrekte Ausführungspfade generiert.

In Abbildung 6.35 ist ein Aktivitätsdiagramm dargestellt, das die Muster Bypass und Alternative in einem gemeinsamen Modell realisiert. Kontrollfluss $f11$ in Abbildung 6.35 repräsentiert einen Bypass, der in Abhängigkeit einer Verzweigungsbedingung den Knoten $n4$

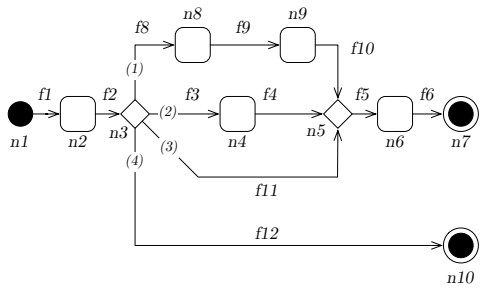


Abbildung 6.35: Beispiel eines Aktivitätsdiagramms, welches die Modellierungsmuster Bypass und Alternative realisiert. Die Nummerierung der aus Knoten $n3$ ausgehenden Kontrollflüsse referenziert die Zeilen der in Quellcodefragment 6.6 dargestellten generierten Ausführungspfade.

überspringt und die Ausführung des Prozesses an Knoten $n5$, einer disjunktiven Vereinigung, fortsetzt. Kontrollfluss $f8$ hingegen leitet eine Alternative ein, die Knoten $n4$ überpringt und stattdessen die Knotensequenz (vgl. Abschnitt 6.3.1.1) $n8 \rightarrow f9 \rightarrow n9$ ausführt. Kontrollfluss $f12$ entspringt ebenfalls als Alternative an Knoten $n3$, weist aber die Besonderheit auf, dass er nicht wieder mit anderen Kontrollflüssen vereinigt wird, sondern den Prozess unmittelbar terminiert. Dieses Modell wurde ausgewählt, weil es die Implikationen der Verwendung der atomaren Muster disjunktive Verzweigung und disjunktive Vereinigung an einem nicht-trivialen Modell demonstriert und eine Komplexität aufweist, deren zu erwartendes Transformationsergebnis von einem menschlichen Akteur noch interpretiert werden kann. Aus Gründen der Übersichtlichkeit sind in der Abbildung die Stereotypen ausgeblendet, es wurden aber alle Knoten mit dem Stereotyp *TestStep* und alle Kontrollflüsse mit dem Stereotyp *TestFlow* annotiert.

Das Permutationssequenzüberdeckungskriterium fordert für dieses Modell, dass vier Ausführungspfade generiert werden, die jeweils die durch $f3$, $f8$, $f11$ sowie $f12$ eingeleiteten Alternativen repräsentieren. Gegenstand der Validierung ist der Nachweis, dass auf diesen vier Ausführungspfaden jeweils alle Knoten in der modellierten Reihenfolge enthalten sind, kein Knoten eines alternativen Kontrollflusses enthalten ist und drei Alternativen mit der Sequenz $n6 \rightarrow f6 \rightarrow !n7$ enden, während Alternative $f12$ unmittelbar terminiert.

Zur Erbringung des Nachweises der Korrektheit der Transformation gegenüber dem Permutationssequenzüberdeckungskriterium wurde das in Abbildung 6.35 dargestellte Modell mit der in Abschnitt 6.2.3 vorgestellten prototypischen Implementierung des Transformationswerkzeugs in ein Testfallmodell überführt. Die Zwischenergebnisse wurden hierbei jeweils manuell auf Konformität zum Abdeckungskriterium überprüft. Auf die Darstellung des zu diesem Modell gehörigen Petri-Netz wird aus Platzgründen verzichtet, die resultierenden Ausführungspfade sind jedoch in Quellcodefragment 6.6 abgebildet.

Quellcodefragment 6.6: Textuelle Repräsentierung der aus dem Aktivitätsdiagramm in Abbildung 6.35 generierten Ausführungspfade

```
1 !n1->f1->n2->f2->f8->n8->f9->n9->f10->f5->n6->f6->!n7
2 !n1->f1->n2->f2->f3->n4->f4->f5->n6->f6->!n7
3 !n1->f1->n2->f2->f11->f5->n6->f6->!n7
4 !n1->f1->n2->f2->f12->!n10
```

Die Zeilen in Quellcodefragment 6.6 repräsentieren jeweils die zu den korrespondierenden Kontrollflüssen in Abbildung 6.35 gehörigen Ausführungspfade. Die Knoten $n3$ und $n5$ sind in den Ausführungspfaden nicht enthalten, da sie Typen repräsentieren (*UML::DecisionNode* und *UML::MergeNode*), für die das UML-Profil zur Testfallmodellierung (vgl. Abschnitt 5.2.4) keine Stereotypen definiert, die also nicht als testrelevant annotiert werden können.

Die manuelle Inspektion der Transformationsergebnisse (d. h. der Ausführungspfade in Quellcodefragment 6.6) hat ergeben, dass das erzeugte Testfallmodell dem Permutationssequenzüberdeckungskriterium (vgl. Abschnitt 5.3.2) genügt, d. h. dass auf keinem der generierten Ausführungspfade Knoten vorkommen, die im Aktivitätsdiagramm in Abbildung 6.35 nicht bzw. auf einem alternativen Kontrollfluss spezifiziert wurden. Weiterhin hat die Inspektion der Ergebnisse ergeben, dass alle als testrelevant annotierten Knoten jeweils auf den einzelnen Ausführungspfaden enthalten sind.

Die Herausforderung der Validierung der Transformation ist es, die Korrektheit der Transformation auch für Modelle zu plausibilisieren, deren Komplexität ein menschlicher Akteur nicht mehr zu überblicken in der Lage ist. Aus diesem Grund wird der Ansatz verfolgt, zunächst die Korrektheit der Transformation von Modellierungsmustern geringer Komplexität zu validieren und anschließend zu zeigen, dass durch eine Verfeinerung von Teilmodellen (d. h. ersetzen knotenberandeter Teilmodelle durch knotenberandete Teilmodelle höherer Komplexität) das Transformationsergebnis weiterhin korrekt ist und lediglich lokal an der Stelle der Verfeinerung Abweichungen in Form erweiterter Ausführungspfade aufweist. Diese Erweiterung manifestiert sich dadurch, dass der verfeinerte Modellabschnitt im Ausführungspfad durch die aus dem verfeinernden Teilmodell generierten Ausführungspfade ersetzt wird.

Am Beispiel der Modellierungsmuster Alternative und Bypass kann nach diesem Vorgehen die Korrektheit der Transformation für Modelle höherer Komplexität gezeigt werden, indem einzelne Aktivitäten im Aktivitätsdiagramm in Abbildung 6.35 durch ein knotenberandetes Teilmodell höherer Komplexität verfeinert werden.

Beispielhaft wurde nach diesem Ansatz der Knoten $n4$ in Abbildung 6.35 durch eine vollständige Kopie des in Abbildung 6.35 dargestellten Aktivitätsdiagramms ersetzt, d. h. das Aktivitätsdiagramm wurde an Aktivität $n4$ mit sich selbst verfeinert. Die Bezeichnung der Knoten und Kanten wurde hierbei strukturell beibehalten. Es wurde lediglich das Präfix angepasst, um die Knoten in Abbildung 6.36 differenzieren zu können. Knoten $n1$ wird dabei zu $m1$, $n2$ zu $m2$ usw.; analog Kante $f1$ zu $g1$.

Gemäß dem Permutationssequenzüberdeckungskriterium ist zu erwarten, das diejenigen Ausführungspfade in Quellcodefragment 6.6 (Zeile 1, 3 und 4) weiterhin unverändert in der Menge der generierten Ausführungspfade enthalten sind, die Knoten $n4$ nicht enthalten. Wenn der Knoten $n4$ durch das Aktivitätsdiagramm in Abbildung 6.35 ersetzt wird, ist zu erwarten dass die Ergebnismenge der generierten Ausführungspfade insgesamt sieben Pfade enthält, von denen vier den Pfad in Zeile 2 in Quellcodefragment 6.6 ersetzen, da er den Knoten $n4$ enthält. Von diesen Pfaden wird erwartet, jeweils mit der Sequenz $!n1 \rightarrow f1 \rightarrow n2 \rightarrow f2 \rightarrow f3$ zu beginnen und mit der Sequenz $f4 \rightarrow f5 \rightarrow n6 \rightarrow f6 \rightarrow !n7$ zu enden.

In Abbildung 6.36 ist das aus der Verfeinerung des Knotens $n4$ mit dem in Abbildung 6.35 dargestellten Aktivitätsdiagramm resultierende Aktivitätsdiagramm dargestellt. Die Knoten vom Typ *UML::InitialNode* und *UML::ActivityFinalNode* existieren im inneren Aktivitäts-

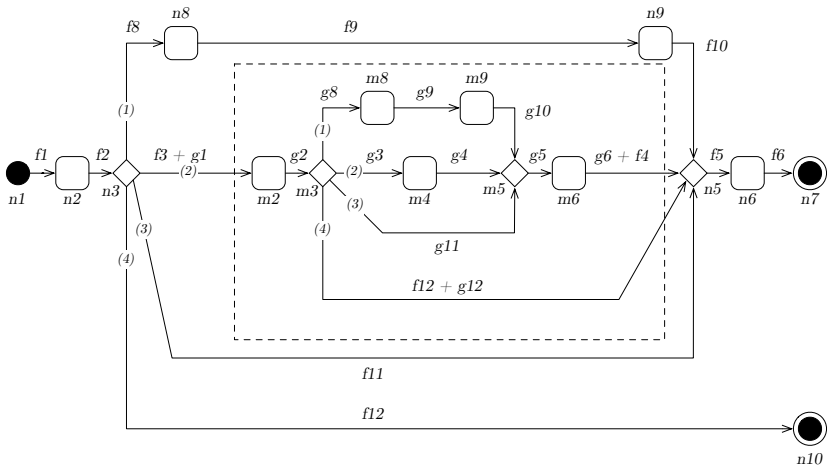


Abbildung 6.36: Verfeinerte Variante des in Abbildung 6.35 dargestellten Aktivitätsdiagramms, wobei der Knoten $n4$ mit einer weiteren Instanz von des in Abbildung 6.35 dargestellten Aktivitätsdiagramms verfeinert wurde.

diagramm (graphisch hervorgehoben) nicht mehr. Hintergrund hierfür ist, dass Elemente vom Typ *UML::InitialNode* und *UML::ActivityFinalNode* mit eingehenden bzw. ausgehenden Knoten zu modellieren nicht zulässig sind (vgl. Metamodell der UML, OMG [268]). Ausschlaggebend ist, dass der Kontrollfluss am ersten Knoten des inneren Aktivitätsdiagramms fortgesetzt wird, wenn im ursprünglichen Aktivitätsdiagramm eine Marke die Aktivität $n4$ erreicht bzw. an Knoten $n5$ fortgesetzt wird, wenn im inneren Aktivitätsdiagramm eine Marke ein Element vom Typ *UML::ActivityFinalNode* erreicht.

Die Bezeichnungen der Kontrollflüsse $f3{+}g1$, $g6{+}f4$ sowie $f12{+}g12$ deuten an, dass durch die Verfeinerung des Knotens $n4$ im Aktivitätsdiagramm in Abbildung 6.36 mit einer weiteren Instanz des Aktivitätsdiagramms in Abbildung 6.36 die durch die Annotation mit dem Stereotyp *TestFlow* definierten Vor- und Nachbedingungen bei der Verfeinerung auf diesem Kontrollfluss vereinigt werden.

Quellcodefragment 6.7: Textuelle Repräsentierung der aus dem verfeinerten Aktivitätsdiagramm in Abbildung 6.36 generierten Ausführungspfade

```
1 !n1->f1->n2->f2->f12->!n10
2 !n1->f1->n2->f2->f8->n8->f9->n9->f10->f5->n6->f6->!n7
3 !n1->f1->n2->f2->f11->f5->n6->f6->!n7
4 !n1->f1->n2->f2->f3->g1->m2->g2->g3->m4->g4->g5->m6->g6->f4->f5->n6↙
    ↳ ->f6->!n7
5 !n1->f1->n2->f2->f3->g1->m2->g2->g12->f4->f5->n6->f6->!n7
6 !n1->f1->n2->f2->f3->g1->m2->g2->g8->m8->g9->m9->g10->g5->m6->g6->f4↙
    ↳ ->f5->n6->f6->!n7
7 !n1->f1->n2->f2->f3->g1->m2->g2->g11->g5->m6->g6->f4->f5->n6->f6->!↙
    ↳ n7
```

In Quellcodefragment 6.7 sind die generierten Ausführungspfade des verfeinerten Aktivitätsdiagramms aus Abbildung 6.36 abgebildet. Konform zum Permutationssequenzüberdeckungskriterium wurden insgesamt sieben Ausführungspfade generiert. Von diesen sind drei bereits in der Menge der aus dem Aktivitätsdiagramm in Abbildung 6.35 generierten Ausführungspfade identisch enthalten (Zeilen 1-3), da der Knoten $n4$ auf den korrespon-

dierenden Kontrollflüssen nicht existiert (ersetzt durch Verfeinerung). Zeilen 4-7 in Quellcodefragment 6.7 repräsentieren die durch Verfeinerung des Knotens $n4$ hinzugekommenen Ausführungspfade. Konform mit dem Abdeckungskriterium beginnen sie mit der Sequenz $!n1 \rightarrow f1 \rightarrow n2 \rightarrow f2 \rightarrow f3$ und enden mit der Sequenz $f4 \rightarrow f5 \rightarrow n6 \rightarrow f6 \rightarrow !n7$. Zwischen Anfangs- und Abschlusssequenz sind die generierten Ausführungspfade strukturell identisch zu den aus dem Aktivitätsdiagramm in Abbildung 6.35 generierten Ausführungspfaden.

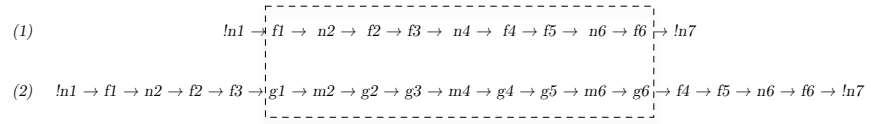


Abbildung 6.37: Zeile 2 aus Quellcodefragment 6.6 ist strukturell identisch in Zeile 4 in Quellcodefragment 6.7 enthalten.

In Abbildung 6.37 sind exemplarisch Zeile 2 aus Quellcodefragment 6.6, basierend auf dem Aktivitätsdiagramm in Abbildung 6.35, und Zeile 4 aus Quellcodefragment 6.7, basierend auf dem Aktivitätsdiagramm in Abbildung 6.36, gegenüber gestellt. Unter Berücksichtigung der verwendeten Namenskonvention ist erkennbar, dass der im verfeinerten Modell eingebettete Abschnitt des Ausführungspfads strukturell identisch mit dem ursprünglichen Ausführungspfad ist. Die Korrektheit im Sinne des Permutationssequenzüberdeckungskriteriums ist somit auch für die Muster Alternative und Bypass für die Verfeinerung nachgewiesen.

Eine erneute Verfeinerung des Knotens $m4$ in Aktivitätsdiagramm in Abbildung 6.36 mit einer weiteren Instanz des Aktivitätsdiagramms aus Abbildung 6.35 und eine Anwendung der Transformation auf diesem Modell hat ebensfall im Sinne des Permutationssequenzüberdeckungskriteriums korrekte Ergebnisse erzeugt.

Quellcodefragment 6.8: Textuelle Repräsentierung der generierten Ausführungspfade nach erneuter Verfeinerung des Aktivitätsdiagramms aus Abbildung 6.36

```
1  !n1->f1->n2->f2->f12->!n10
2  !n1->f1->n2->f2->f8->n8->f9->n9->f10->f5->n6->f6->!n7
3  !n1->f1->n2->f2->f11->f5->n6->f6->!n7
4  !n1->f1->n2->f2->f3->g1->m2->g2->g12->f4->f5->n6->f6->!n7
5  !n1->f1->n2->f2->f3->g1->m2->g2->g8->m8->g9->m9->g10->g5->m6->g6->f4↙
       ↳ ->f5->n6->f6->!n7
6  !n1->f1->n2->f2->f3->g1->m2->g2->g11->g5->m6->g6->f4->f5->n6->f6->!↙
       ↳ n7
7  !n1->f1->n2->f2->f3->g1->m2->g2->g3->h1->o2->h2->h3->o4->h4->h5->o6↙
       ↳ ->h6->g4->g5->m6->g6->f4->f5->n6->f6->!n7
8  !n1->f1->n2->f2->f3->g1->m2->g2->g3->h1->o2->h2->h12->g4->g5->m6->g6↙
       ↳ ->f4->f5->n6->f6->!n7
9  !n1->f1->n2->f2->f3->g1->m2->g2->g3->h1->o2->h2->h8->o8->h9->o9->h10↙
       ↳ ->h5->o6->h6->g4->g5->m6->g6->f4->f5->n6->f6->!n7
10 !n1->f1->n2->f2->f3->g1->m2->g2->g3->h1->o2->h2->h11->h5->o6->h6->g4↙
       ↳ ->g5->m6->g6->f4->f5->n6->f6->!n7
```

Die Menge der Ausführungspfade ist erneut um drei angewachsen, die neu hinzugekommenen Ausführungspfade sind in analoger Weise strukturell identisch zu den Ausführungspfaden aus dem vorherigen Verfeinerungsschritt.

Es wurde direkt nachgewiesen, dass die Transformation für das betrachtete Aktivitätsdiagramm auf der ersten Stufe Abbildung 6.35 ein korrektes Ergebnis erzeugt. Es wurde weiterhin

direkt nachgewiesen, dass für die erste Verfeinerungsstufe in Abbildung 6.36 und für die darauf folgende Verfeinerungsstufe jeweils korrekte Ergebnisse generiert werden. Implizit wurde so nachgewiesen, dass die Transformation für Modelle mit eingebetteten Teilmodellen korrekte Ergebnisse erzeugt, wenn für diese Teilmodelle korrekte Ergebnisse erzeugt werden. Hieraus kann gefolgert werden, dass auch für weitere Verfeinerungsstufen der Modellierungsmuster Alternative und Bypass die Transformation ebenfalls korrekte Testfallmodelle generiert.

6.3.2.3 Zyklus

Zu den zusammengesetzten Modellierungsmustern gehört ebenfalls der Zyklus, welcher bereits von Staines [330] identifiziert wurde. Das Muster strukturiert die atomaren Modellierungsmuster disjunktive Verzweigung (vgl. Abschnitt 6.3.1.2) und disjunktive Vereinigung (vgl. Abschnitt 6.3.1.3) in einer solchen Weise, dass ein Zyklus im Kontrollfluss entsteht.

Mit Ausnahme der Richtung der Kontrollflüsse an den Vereinigungs- bzw. Verzweigungsknoten entspricht der Zyklus strukturell dem Bypass (vgl. Abschnitt 6.3.2.2), stellt für das Testen jedoch eine vergleichsweise hohe Herausforderung dar, weil Ausführungspfade in Teilen mehrfach ausgeführt werden müssen, wodurch insbesondere bei ineinander verschachtelten Zyklen die Komplexität des Testfallmodells rasch auf ein Maß ansteigt, das für einen menschlichen Tester nicht mehr zu beherrschen ist.

Das in Abschnitt 5.3.2 definierte Permutationssequenzüberdeckungskriterium fordert, dass der Transformationsalgorithmus zur Generierung von Tests aus UML-Aktivitätsdiagrammen Ausführungspfade erzeugt, so dass jeder Zyklus im Kontrollfluss mindestens einmal durchlaufen wird. Die Validierung des Transformationsalgorithmus hat deshalb das Ziel, den Nachweis zu erbringen, dass die Transformation gegenüber dieser Forderung korrekte Ergebnisse (d. h. Ausführungspfade) erzeugt.

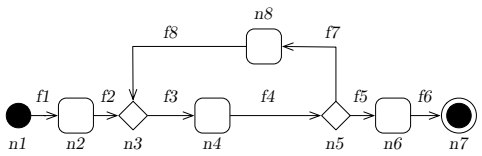


Abbildung 6.38: Modellierungsmuster Zyklus

Analog zur Validierung der Transformation atomarer Modellierungsmuster wird auch zur Validierung des Modellierungsmusters Zyklus ein minimales Aktivitätsdiagramm gebildet. Abbildung 6.38 zeigt ein mögliches minimales Aktivitätsdiagramm, das aus dem Modellierungsmuster gebildet werden kann. Bewusst wurde eine Variante gewählt, in welcher sowohl auf dem Kontrollflussabschnitt vor der Einleitung des Zyklus (Verzweigung an $n5$) eine Aktivität modelliert ist als auch auf der Rückführung des Kontrollflusses (Knoten $n3$). Dieses Aktivitätsdiagramm soll verwendet werden, um die Korrektheit der Transformation für unterschiedliche Verfeinerungsvarianten zu zeigen, in denen jeweils die Knoten $n4$ bzw. $n8$ durch andere Modellierungsmuster, insbesondere durch weitere Zyklen verfeinert werden.

6.3.2.3.1 Basisvariante

In der Basisvariante der Validierung der Transformation des Musters Zyklus wurde der Transformationsalgorithmus auf das in Abbildung 6.38 dargestellte Aktivitätsdiagramm angewendet. In Sinne des für die Transformation definierten Permutationssequenzüberdeckungskriteriums wird von der Transformation erwartet, ein Testfallmodell zu generieren, welches zwei Ausführungspfade aus dem Aktivitätsdiagramm generiert: einen Ausführungspfad, der die Verzweigung an Knoten $n5$ so auswertet, dass der Zyklus nicht durchlaufen wird und ein Pfad, der die Verzweigung so auswertet, dass der Zyklus ausgewertet wird.

Quellcodefragment 6.9: Textuelle Repräsentierung der aus dem Aktivitätsdiagramm in Abbildung 6.38 generierten Ausführungspfade

```
1 !n1->f1->n2->f2->f3->n4->f4->f5->n6->f6->!n7
2 !n1->f1->n2->f2->f3->n4->f4->f7->n8->f8->f3->n4->f4->f5->n6->f6->!n7
```

Eine Inspektion der Zwischenergebnisse (d. h. Petri-Netz, Ausführungspfade, plattformunabhängiges Testfallmodell) der Anwendung der Transformation auf das in Abbildung 6.38 dargestellte Aktivitätsdiagramm hat ergeben, dass der Transformationsalgorithmus i. S. des Abdeckungskriteriums korrekte Ausführungspfade generiert. Anhand der in Quellcodefragment 6.9 dargestellten Ausführungspfade ist erkennbar, dass der Algorithmus tatsächlich zwei Ausführungspfade für das Aktivitätsdiagramm in Abbildung 6.38 generiert hat. Der Ausführungspfad in Zeile 2 in Quellcodefragment 6.9 bildet den an Knoten $n5$ eingeleiteten Zyklus ab, der den Kontrollfluss über Knoten $n8$ zu Knoten $n3$ zurückführt, so dass Knoten $n4$ in auf diesem Ausführungspfad mehrfach ausgeführt wird. Dieses Transformationsergebnis erfüllt die Forderung des Permutationssequenzüberdeckungskriteriums, womit die Korrektheit der Transformation für die Basisvariante Modellierungsmusters Zyklus nachgewiesen ist.

6.3.2.3.2 In einen Zyklus eingebettete Sequenz

Das in Abschnitt 6.3.2.3.1 dargestellte Beispiel der Transformation eines Aktivitätsdiagramms deckt zunächst nur den Basisfall ab, in welchem auf Kontrollflüssen, durch welche der Zyklus gebildet wird, lediglich eine einzige Aktivität modelliert ist. Ziel der Validierung ist nachzuweisen, dass beliebig komplexe Modelle basierend auf dem Muster Zyklus ebenfalls konform zum Permutationssequenzüberdeckungskriterium zu Testfallmodellen transformiert werden.

Zu diesem Zweck wird das Aktivitätsdiagramm in Abbildung 6.38 um zusätzliche Modellierungselemente erweitert und gezeigt, dass die aus der Transformation resultierenden Ausführungspfade weiterhin das Permutationssequenzüberdeckungskriterium erfüllen. Die generierten Ausführungspfade unterscheiden sich lediglich lokal an denjenigen Stellen, an denen auch im erweiterten Modell eine Änderung vorgenommen wurde.

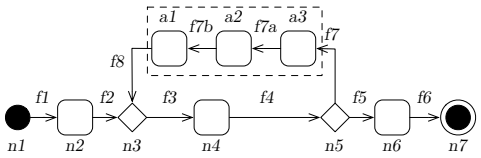


Abbildung 6.39: Muster Zyklus mit eingebetteter Sequenz

In Abbildung 6.39 ist das modifizierte Aktivitätsdiagramm dargestellt. Durch Verfeinerung des Knotens $n8$ wurde das ursprüngliche Aktivitätsdiagramm in Abbildung 6.38 zu einem komplexeren Modell erweitert. Das Modell wurde um eine Sequenz erweitert, dessen korrekte Transformation bereits erwiesen ist.

Um weiterhin dem Permutationssequenzüberdeckungskriterium zu genügen, muss die Transformation aus diesem Modell zwei Ausführungspfade erzeugen, die analog zum Modell in Abbildung 6.38 jeweils einen Ausführungspfad ohne bzw. mit Ausführung des Zyklus repräsentieren. Darüber hinaus dürfen sich die aus dem Modell in Abbildung 6.39 generierten Ausführungspfade nur an der Stelle der Verfeinerung des Modells unterscheiden, d. h. konkret ausschließlich an demjenigen Pfad, auf dem die hinzugefügte Sequenz modelliert ist.

Das Ergebnis der Transformation ist in Quellcodefragment 6.10 abgebildet. Konform zum Permutationssequenzüberdeckungskriterium hat der Algorithmus zwei Ausführungspfade generiert, die beide Fälle (Durchlauf mit und ohne Zyklus) repräsentieren. Zeile 2 in Quellcodefragment 6.10 ist zu entnehmen, dass die Knotensequenz $a1$, $a2$, $a3$ ebenfalls in korrekter Reihenfolge abgebildet wurde.

Quellcodefragment 6.10: Textuelle Repräsentierung der aus dem Aktivitätsdiagramm in Abbildung 6.39 generierten Ausführungspfade

```
1 !n1->f1->n2->f2->f3->n4->f4->f5->n6->f6->!n7
2 !n1->f1->n2->f2->f3->n4->f4->f7->n8->f8->f3->n4->f4->f5->n6->f6->!n7
```

In Abbildung 6.40 sind die Ausführungspfade, durch welche der Durchlauf des Zyklus repräsentiert ist, aus den Aktivitätsdiagrammen in Abbildung 6.38 und Abbildung 6.39 einander gegenübergestellt. Zu erkennen ist, dass sich beide Ausführungspfade ausschließlich lokal auf denjenigen Bereich beschränkt voneinander unterscheiden, an dem das Aktivitätsdiagramm durch Einfügen einer Sequenz verfeinert wurde.

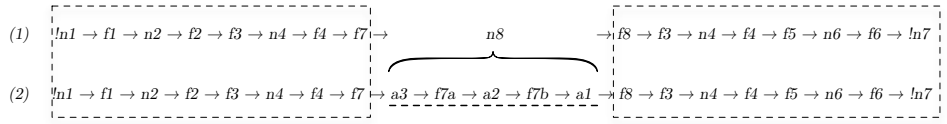


Abbildung 6.40: Gegenüberstellung der Zykluspfade aus Abbildung 6.38 (1) und Abbildung 6.39 (2)

6.3.2.3.3 Verschachtelte Zyklen

Es wurde bereits gezeigt, dass der Transformationsalgorithmus für einfache Zyklen korrekte Ergebnisse erzeugt (vgl. Abschnitt 6.3.2.3.1). Die atomaren Modellierungsmuster disjunktive Verzweigung und disjunktive Vereinigung können jedoch auch so verwendet werden, dass ineinander verschachtelte Zyklen entstehen.

In Abbildung 6.41 ist das bereits in Abbildung 6.38 dargestellte Aktivitätsdiagramm erneut abgebildet, allerdings wurden die Knoten $n4$ und $n8$ jeweils mit einer weiteren Instanz des in Abbildung 6.38 dargestellten Aktivitätsdiagramms verfeinert. Analog zum Vorgehen bei der Validierung der Modellierungsmuster Bypass und Alternative wurde hier ebenfalls der Ansatz gewählt, das zu validierende Modell mit sich selbst zu verfeinern (Namenschema analog zu Abschnitt 6.3.2.2).

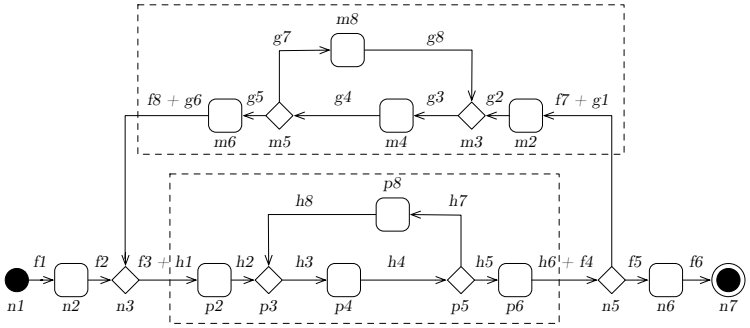


Abbildung 6.41: Modellierungsmuster Zyklus mit eingebettetem Zyklus

Auf dieses Modell wurde der Algorithmus zur Generierung von Testfällen angewendet und die Zwischenergebnisse wurden einer manuellen Inspektion unterzogen. Die generierten Ausführungspfade sind in Quellcodefragment 6.11 abgebildet.

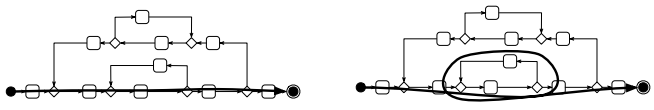
Quellcodefragment 6.11: Textuelle Repräsentierung der aus dem Aktivitätsdiagramm in Abbildung 6.41 generierten Ausführungspfade

```
1 !n1->f1->n2->f2->f3->h1->p2->h2->h3->p4->h4->h5->p6->h6->f4->f5->n6↙
      ↳ ->f6->!n7
2 !n1->f1->n2->f2->f3->h1->p2->h2->h3->p4->h4->h7->p8->h8->h3->p4->h4↙
      ↳ ->h5->p6->h6->f4->f5->n6->f6->!n7
3 !n1->f1->n2->f2->f3->h1->p2->h2->h3->p4->h4->h5->p6->h6->f4->f7->g1↙
      ↳ ->m2->g2->g3->m4->g4->g5->m6->g6->f8->f3->h1->p2->h2->h3->p4->↙
      ↳ h4->h5->p6->h6->f4->f5->n6->f6->!n7
4 !n1->f1->n2->f2->f3->h1->p2->h2->h3->p4->h4->h5->p6->h6->f4->f7->g1↙
      ↳ ->m2->g2->g3->m4->g4->g7->m8->g8->g3->m4->g4->g5->m6->g6->f8->↙
      ↳ f3->h1->p2->h2->h3->p4->h4->h5->p6->h6->f4->f5->n6->f6->!n7
5 !n1->f1->n2->f2->f3->h1->p2->h2->h3->p4->h4->h7->p8->h8->h3->p4->h4↙
      ↳ ->h5->p6->h6->f4->f7->g1->m2->g2->g3->m4->g4->g5->m6->g6->f8->↙
      ↳ f3->h1->p2->h2->h3->p4->h4->h7->p8->h8->h3->p4->h4->h5->p6->h6↙
      ↳ ->f4->f5->n6->f6->!n7
6 !n1->f1->n2->f2->f3->h1->p2->h2->h3->p4->h4->h7->p8->h8->h3->p4->h4↙
      ↳ ->h5->p6->h6->f4->f7->g1->m2->g2->g3->m4->g4->g7->m8->g8->g3->↙
      ↳ m4->g4->g5->m6->g6->f8->f3->h1->p2->h2->h3->p4->h4->h7->p8->h8↙
      ↳ ->h3->p4->h4->h5->p6->h6->f4->f5->n6->f6->!n7
```

Eine graphische Aufbereitung der textuellen Repräsentierung der Ausführungspfade ist in Abbildung 6.42 dargestellt. Zu erkennen ist, dass konform zum Permutationssequenzüberdeckungskriterium alle Zyklen, auch in Kombination miteinander, im Aktivitätsdiagramm durch einen Ausführungspfad abgedeckt sind, womit die Korrektheit der Transformation im Sinne des Abdeckungskriteriums nachgewiesen ist.

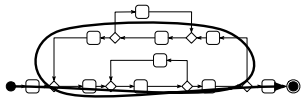
Um sicherzustellen, dass der Transformationsalgorithmus auch für mehrfach ineinander verschachtelte Zyklen korrekte Ergebnisse generiert, wurde in einem weiteren Schritt der Validierung erneut eine Variante des in Abbildung 6.38 dargestellten Aktivitätsdiagramms erstellt. Diesmal wurde Knoten $n4$ durch eine weitere Instanz des Aktivitätsdiagramms ersetzt und auch in diesem Aktivitätsdiagramm der Knoten $n4$ durch eine weitere Instanz desselben Aktivitätsdiagramms ersetzt. Im Ergebnis entsteht das in Abbildung 6.43 dargestellte Aktivitätsdiagramm mit einem dreifach verschachtelten Zyklus.

Das Ergebnis der Transformation des in Abbildung 6.43 dargestellten Aktivitätsdiagramms ist in Quellcodefragment 6.12 abgebildet. Eine manuelle Überprüfung hat auch hier ergeben, dass konform zum Permutationssequenzüberdeckungskriterium alle Zyklen im Aktivitätsdia-

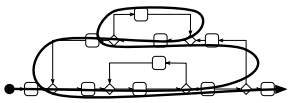


(a) Zeile 1, Quellcodefragment 6.11

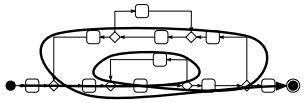
(b) Zeile 2, Quellcodefragment 6.11



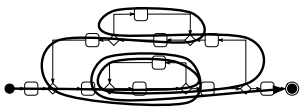
(c) Zeile 3, Quellcodefragment 6.11



(d) Zeile 4, Quellcodefragment 6.11



(e) Zeile 5, Quellcodefragment 6.11



(f) Zeile 6, Quellcodefragment 6.11

Abbildung 6.42: Graphische Aufbereitung der Transformationsergebnisse für das Aktivitätsdiagramm in Abbildung 6.41

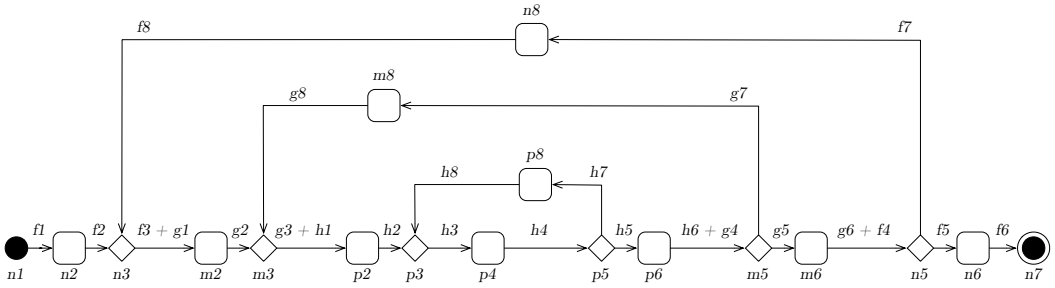


Abbildung 6.43: Modellierungsmuster Zyklus mit dreifach verschachteltem Zyklus

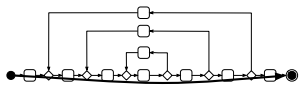
gramm mindestens einmal durchlaufen wurden. Hervorzuheben ist, dass einige Zyklen, insbesondere der durch Kontrollfluss $h7$ in Abbildung 6.43 häufiger als einmal durchlaufen wurde. Konkret wird dieser Zyklus in einem der generierten Testfälle viermal durchlaufen, einmal direkt und zusätzlich als Teil des Durchlaufens der durch die Kontrollflüsse $g7$ und $f7$ eingeleiteten Zyklen.

Quellcodefragment 6.12: Textuelle Repräsentierung der aus dem Aktivitätsdiagramm in Abbildung 6.43 generierten Ausführungspfade

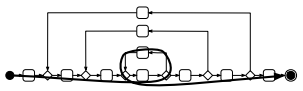
```
1 !n1->f1->n2->f2->f3->g1->m2->g2->g3->h1->p2->h2->h3->p4->h4->h5->p6↙
      ↳ ->h6->g4->g5->m6->g6->f4->f5->n6->f6->!n7
2 !n1->f1->n2->f2->f3->g1->m2->g2->g3->h1->p2->h2->h3->p4->h4->h7->p8↙
      ↳ ->h8->h3->p4->h4->h5->p6->h6->g4->g5->m6->g6->f4->f5->n6->f6↙
      ↳ ->!n7
3 !n1->f1->n2->f2->f3->g1->m2->g2->g3->h1->p2->h2->h3->p4->h4->h5->p6↙
      ↳ ->h6->g4->g7->m8->g8->g3->h1->p2->h2->h3->p4->h4->h5->p6->h6->↙
      ↳ g4->g5->m6->g6->f4->f5->n6->f6->!n7
4 !n1->f1->n2->f2->f3->g1->m2->g2->g3->h1->p2->h2->h3->p4->h4->h7->p8↙
      ↳ ->h8->h3->p4->h4->h5->p6->h6->g4->g7->m8->g8->g3->h1->p2->h2->↙
      ↳ h3->p4->h4->h7->p8->h8->h3->p4->h4->h5->p6->h6->g4->g5->m6->g6↙
      ↳ ->f4->f5->n6->f6->!n7
5 !n1->f1->n2->f2->f3->g1->m2->g2->g3->h1->p2->h2->h3->p4->h4->h5->p6↙
      ↳ ->h6->g4->g5->m6->g6->f4->f7->n8->f8->f3->g1->m2->g2->g3->h1->↙
```

```
      ↳ p2->h2->h3->p4->h4->h5->p6->h6->g4->g5->m6->g6->f4->f5->n6->f6↙
      ↳ ->!n7
6  !n1->f1->n2->f2->f3->g1->m2->g2->g3->h1->p2->h2->h3->p4->h4->h7->p8↙
      ↳ ->h8->h3->p4->h4->h5->p6->h6->g4->g5->m6->g6->f4->f7->n8->f8->↙
      ↳ f3->g1->m2->g2->g3->h1->p2->h2->h3->p4->h4->h7->p8->h8->h3->p4↙
      ↳ ->h4->h5->p6->h6->g4->g5->m6->g6->f4->f5->n6->f6->!n7
7  !n1->f1->n2->f2->f3->g1->m2->g2->g3->h1->p2->h2->h3->p4->h4->h5->p6↙
      ↳ ->h6->g4->g7->m8->g8->g3->h1->p2->h2->h3->p4->h4->h5->p6->h6->↙
      ↳ g4->g5->m6->g6->f4->f7->n8->f8->f3->g1->m2->g2->g3->h1->p2->h2↙
      ↳ ->h3->p4->h4->h5->p6->h6->g4->g7->m8->g8->g3->h1->p2->h2->h3->↙
      ↳ p4->h4->h5->p6->h6->g4->g5->m6->g6->f4->f5->n6->f6->!n7
8  !n1->f1->n2->f2->f3->g1->m2->g2->g3->h1->p2->h2->h3->p4->h4->h7->p8↙
      ↳ ->h8->h3->p4->h4->h5->p6->h6->g4->g7->m8->g8->g3->h1->p2->h2->↙
      ↳ h3->p4->h4->h7->p8->h8->h3->p4->h4->h5->p6->h6->g4->g5->m6->g6↙
      ↳ ->f4->f7->n8->f8->f3->g1->m2->g2->g3->h1->p2->h2->h3->p4->h4->↙
      ↳ h7->p8->h8->h3->p4->h4->h5->p6->h6->g4->g7->m8->g8->g3->h1->p2↙
      ↳ ->h2->h3->p4->h4->h7->p8->h8->h3->p4->h4->h5->p6->h6->g4->g5->↙
      ↳ m6->g6->f4->f5->n6->f6->!n7
```

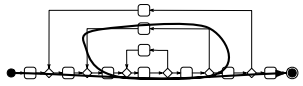
In Abbildung 6.44 wurden die in Quellcodefragment 6.12 dargestellten Ausführungspfade zum besseren Verständnis graphisch aufbereitet dargestellt. Insbesondere der in Abbildung 6.44h dargestellte Fall stellt sowohl bei der manuellen Testfallerstellung als auch bei dessen Durchführung eine besondere kognitive Leistung für einen menschlichen Tester dar. Er beinhaltet den Durchlauf aller im Modell vorkommenden Zyklen, ggf. in Kombination miteinander. Der Tester muss in jedem Schritt die jeweils gültigen Vorbedingungen herstellen, eine Anzahl von Testschritten durchführen (d. h. Eingabe von Testdaten, Manipulation interaktiver Elemente des UI etc.) und anschließend Nachbedingungen prüfen.



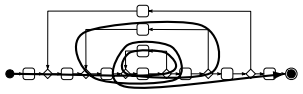
(a) Zeile 1, Quellcodefragment 6.12



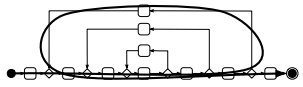
(b) Zeile 2, Quellcodefragment 6.12



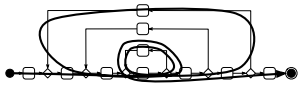
(c) Zeile 3, Quellcodefragment 6.12



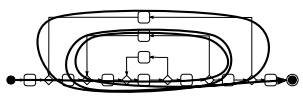
(d) Zeile 4, Quellcodefragment 6.12



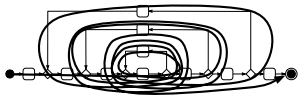
(e) Zeile 5, Quellcodefragment 6.12



(f) Zeile 6, Quellcodefragment 6.12



(g) Zeile 7, Quellcodefragment 6.12



(h) Zeile 8, Quellcodefragment 6.12

Abbildung 6.44: Graphische Aufbereitung der Transformationsergebnisse für das Aktivitätsdiagramm in Abbildung 6.43

Ein Vergleich der Ergebnisse zu den in Abbildung 6.38, Abbildung 6.41 sowie in Abbildung 6.43 dargestellten Aktivitätsdiagrammen hat ergeben, dass der Transformationsalgorithmus stabil gegenüber einer verschachtelten Einbettung von Zyklen ist. Insbesondere bleiben

alle Ausführungspfade, die bei einer 2-fachen verschachtelten Einbettung konform zum Permutationssequenzüberdeckungskriterium generiert wurden auch bei der 3-fachen verschachtelten Einbettung erhalten und werden zusätzlich durch kombinierte Zyklendurchläufe ergänzt.

6.3.2.3.4 Überlappende Zyklen

Die atomaren Modellierungsmuster disjunktive Verzweigung und disjunktive Vereinigung können ebenfalls in einer solchen Weise miteinander verknüpft werden, dass überlappende Zyklen entstehen. Dieser Fall ist in Abbildung 6.45 dargestellt. Hier entspringt einem oberflächlich einfach anmutenden Modell rasch eine komplexe Menge alternativer Ausführungspfade durch das modellierte System, die jeweils in einem individuellen Test adressiert werden müssen.

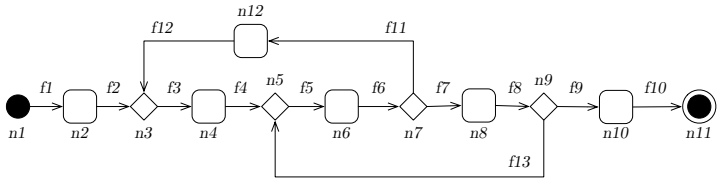
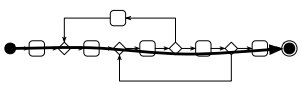


Abbildung 6.45: Modellierungsmuster überlappende Zyklen

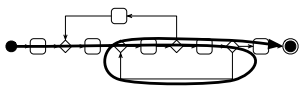
In einem analogen Vorgehen zu den vorherigen Mustern wurde auch auf dieses Modell der Transformationsalgorithmus angewendet und die Zwischenergebnisse sowie das resultierende Testfallmodell einer manuellen Prüfung unterzogen. Für das in Abbildung 6.45 dargestellte Modell konnte nachgewiesen werden, dass der Transformationsalgorithmus im Sinne des Permutationssequenzüberdeckungskriteriums korrekte Ergebnisse generiert. Die Ausführungspfade sind in Quellcodefragment 6.13 dargestellt und in Abbildung 6.46 graphisch aufbereitet.

Quellcodefragment 6.13: Textuelle Repräsentierung der aus dem Aktivitätsdiagramm in Abbildung 6.45 generierten Ausführungspfade

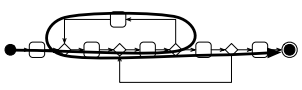
```
1 !n1->f1->n2->f2->f3->n4->f4->f5->n6->f6->f7->n8->f8->f9->n10->f10->!↙
      ↳ n11
2 !n1->f1->n2->f2->f3->n4->f4->f5->n6->f6->f7->n8->f8->f13->f5->n6->f6↙
      ↳ ->f7->n8->f8->f9->n10->f10->!n11
3 !n1->f1->n2->f2->f3->n4->f4->f5->n6->f6->f11->n12->f12->f3->n4->f4->↙
      ↳ f5->n6->f6->f7->n8->f8->f9->n10->f10->!n11
4 !n1->f1->n2->f2->f3->n4->f4->f5->n6->f6->f11->n12->f12->f3->n4->f4->↙
      ↳ f5->n6->f6->f7->n8->f8->f13->f5->n6->f6->f7->n8->f8->f9->n10->↙
      ↳ f10->!n11
```



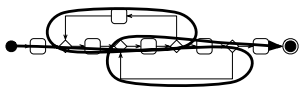
(a) Zeile 1, Quellcodefragment 6.13



(b) Zeile 2, Quellcodefragment 6.13



(c) Zeile 3, Quellcodefragment 6.13



(d) Zeile 4, Quellcodefragment 6.13

Abbildung 6.46: Graphische Aufbereitung der Transformationsergebnisse für das Aktivitätsdiagramm in Abbildung 6.45

Hiermit ist nachgewiesen, dass der Algorithmus für den einfachen Fall überlappender Zyklen korrekt ist. Um nachzuweisen, dass der Algorithmus auch für komplexe Varianten dieses Modellierungsmusters korrekte Ausführungspfade erzeugt, wurde analog zum Vorgehen in den vorherigen Abschnitten das Muster in sich selbst eingebettet. Konkret wurde Knoten $n6$ in Abbildung 6.45 durch eine weitere Instanz des Modellierungsmusters ersetzt. Alternativ zu Knoten $n6$ hätte auch Knoten $n12$ gewählt werden können. Knoten $n6$ zeichnet sich jedoch dadurch aus, dass er von beiden Zyklen im Modell abgedeckt wird.

Ziel ist es, den Nachweis zu erbringen, dass die Verfeinerung des Knotens $n6$ zu komplexeren Ausführungspfaden (d. h. Testfällen) führt, die aus dem Aktivitätsdiagramm in Abbildung 6.45 generierten Ausführungspfade (Quellcodefragment 6.13, Abbildung 6.46) aber weiterhin strukturgleich in der Testfallmenge enthalten sind. Dann wäre auch für dieses Muster der Nachweis erbracht, dass der Transformationsalgorithmus gegenüber strukturellen Verfeinerungen des Musters verschachtelter Zyklen stabil ist und korrekte Ergebnisse erzeugt.

Das verfeinerte Modell ist in Abbildung 6.47 dargestellt. Die Transformation hat insgesamt 16 Testfälle generiert, die in Abschnitt C.1 im Anhang abgedruckt sind. Zur Vergleichbarkeit mit den Transformationsergebnissen aus dem in Abbildung 6.45 abgedruckten Modell ist in Abbildung 6.48 eine graphische Aufbereitung der Transformationsergebnisse des verfeinerten Modells dargestellt.

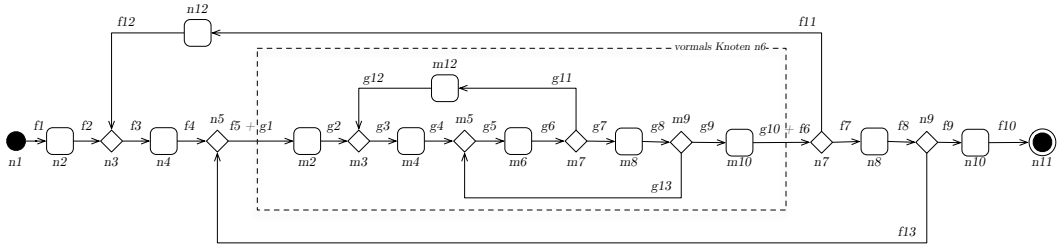
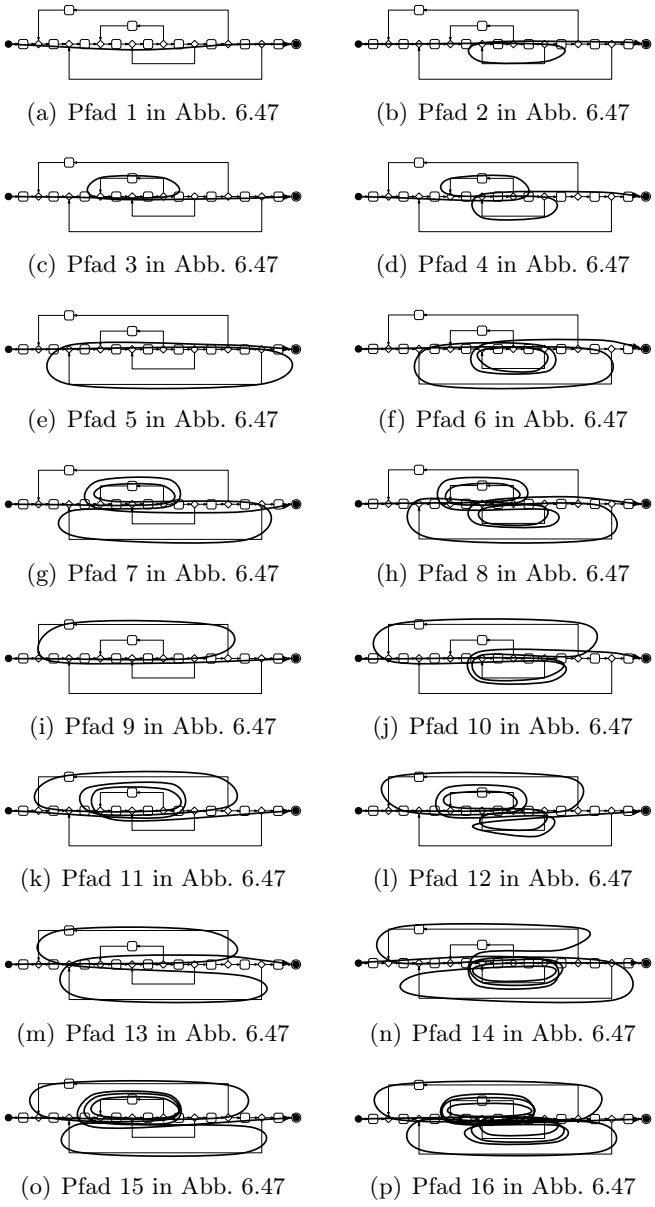


Abbildung 6.47: Modellierungsmuster überlappende, eingebettete Zyklen

Nachvollziehbar ist, dass sich die Anzahl der Ausführungspfade im Vergleich zum Modell in Abbildung 6.45 von vier auf 16 erhöht hat. Aus dem in Abbildung 6.45 dargestellten Modell wurden die vier in Abbildung 6.46 (bzw. in Quellcodefragment 6.13) abgedruckten Ausführungspfade erstellt. Das Modell wurde an Knoten $n6$ in sich selbst eingebettet. Jeder der vier Pfade des Ursprungsmodells wird nach der Verfeinerung also durch die vier Pfade des inneren Modells substituiert. Die Pfade in Abbildung 6.48a bis Abbildung 6.48d beispielsweise repräsentieren den Pfad in Abbildung 6.46a, die Pfade in Abbildung 6.48e bis Abbildung 6.48h repräsentieren den Pfad in Abbildung 6.46b usw.

Die Pfade in Abbildung 6.48e bis Abbildung 6.48h repräsentieren den Pfad in Abbildung 6.46b, weil sie gemeinsam haben, dass in allen vier Pfaden der durch $f13$ eingeleitete Zyklus durchlaufen wird. Innerhalb dieses Durchlaufs werden alle vier Ausführungsvarianten der inneren überlappenden Zyklen ausgeführt.

Es konnte somit auch für dieses Modell gezeigt werden, dass die Transformation auch dann korrekte Ausführungspfade erzeugt, wenn einzelne Knoten innerhalb des Modells durch andere Modellierungsmuster verfeinert werden. Auch nach einer Verfeinerung, wie am Beispiel



(a) Pfad 1 in Abb. 6.47 (b) Pfad 2 in Abb. 6.47

(c) Pfad 3 in Abb. 6.47 (d) Pfad 4 in Abb. 6.47

(e) Pfad 5 in Abb. 6.47 (f) Pfad 6 in Abb. 6.47

(g) Pfad 7 in Abb. 6.47 (h) Pfad 8 in Abb. 6.47

(i) Pfad 9 in Abb. 6.47 (j) Pfad 10 in Abb. 6.47

(k) Pfad 11 in Abb. 6.47 (l) Pfad 12 in Abb. 6.47

(m) Pfad 13 in Abb. 6.47 (n) Pfad 14 in Abb. 6.47

(o) Pfad 15 in Abb. 6.47 (p) Pfad 16 in Abb. 6.47

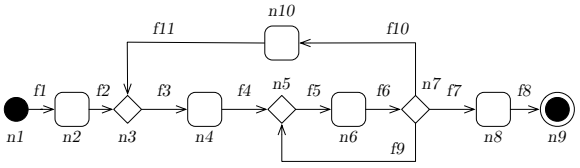
Abbildung 6.48: Graphische Aufbereitung der Transformationsergebnisse für das Aktivitätsdiagramm in Abbildung 6.47

der Selbsteinbettung des in Abbildung 6.45 dargestellten Aktivitätsdiagramms in Knoten $n6$ demonstriert, bleiben die für das Testen relevanten strukturellen Eigenschaften der generierten Ausführungspfade erhalten, d. h. alle mit Aktivitäten assoziierten Testschritte werden in der im Sinne des Permutationssequenzüberdeckungskriteriums korrekten Reihenfolge ausgeführt.

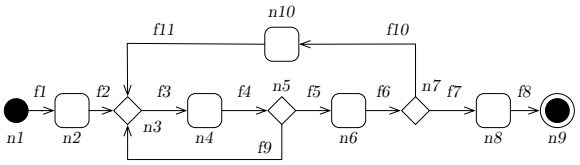
6.3.2.3.5 Komplexe Zyklen

Die atomaren Modellierungsmuster disjunktive Verzweigung und disjunktive Vereinigung sind nicht auf lediglich zwei Verzweigungs- bzw. Vereinigungsalternativen beschränkt. Hieraus ergibt sich unmittelbar, dass Zyklen auch vielfältiger und komplexer als in den bisher diskutierten Mustern gestaltet werden können, wodurch die Komplexität der Testfälle weiter ansteigt. Um die Korrektheit des Transformationsalgorithmus auch für diese Varianten des

Modellierungsmusters zu prüfen, wurde er auf die in Abbildung 6.49a und Abbildung 6.49b dargestellten Sonderfälle der in Abschnitt 6.3.2.3.1 diskutierten Basisvariante des Musters Zyklus angewendet.



(a) Modellierungsmuster Zyklus mit mehreren zykluseinleitenden Kontrollflüssen an einem Verzweigungsknoten



(b) Modellierungsmuster Zyklus mit mehreren zyklusvereinenden Kontrollflüssen an einem Vereinigungsknoten

Abbildung 6.49: Modellierungsmuster Zyklus mit komplexen Verzweigungen bzw. Vereinigungen

Quellcodefragment 6.14: Textuelle Repräsentierung der aus dem Aktivitätsdiagramm in Abbildung 6.49a generierten Ausführungspfade

```
1 !n1->f1->n2->f2->f3->n4->f4->f5->n6->f6->f7->n8->f8->!n9
2 !n1->f1->n2->f2->f3->n4->f4->f5->n6->f6->f9->f5->n6->f6->f7->n8->f8↙
    ↳ ->!n9
3 !n1->f1->n2->f2->f3->n4->f4->f5->n6->f6->f10->n10->f11->f3->n4->f4->↙
    ↳ f5->n6->f6->f7->n8->f8->!n9
4 !n1->f1->n2->f2->f3->n4->f4->f5->n6->f6->f10->n10->f11->f3->n4->f4->↙
    ↳ f5->n6->f6->f9->f5->n6->f6->f7->n8->f8->!n9
```

Quellcodefragment 6.15: Textuelle Repräsentierung der aus dem Aktivitätsdiagramm in Abbildung 6.49b generierten Ausführungspfade

```
1 !n1->f1->n2->f2->f3->n4->f4->f5->n6->f6->f7->n8->f8->!n9
2 !n1->f1->n2->f2->f3->n4->f4->f5->n6->f6->f10->n10->f11->f3->n4->f4->↙
    ↳ f5->n6->f6->f7->n8->f8->!n9
3 !n1->f1->n2->f2->f3->n4->f4->f9->f3->n4->f4->f5->n6->f6->f7->n8->f8↙
    ↳ ->!n9
```

In Quellcodefragment 6.14 und Quellcodefragment 6.15 sind die generierten Ausführungspfade abgebildet. Eine Inspektion der Ergebnisse hat ergeben, dass die Transformation auch für diese Varianten korrekte Testfälle erzeugt. Auf eine weitere Verfeinerung der in Abbildung 6.49 dargestellten Modell wurde aufgrund der in den vorherigen Abschnitten beobachteten Ergebnisse verzichtet. Es hat sich dort bereits gezeigt, dass eine Verfeinerung eines Modells strukturelle Eigenschaften der generierten Ausführungspfade im Sinne des Permutationssequenzüberdeckungskriteriums erhält. Konkret für die in Abbildung 6.49 dargestellten Fälle bedeutet das, dass die in Quellcodefragment 6.14 sowie Quellcodefragment 6.15 dargestellten Ausführungspfade strukturell erhalten bleiben. Alle Vorkommen verfeinerter Knoten werde durch die Ausführungspfade des substituierenden Modells ersetzt, wobei die Anzahl der Testfälle analog zu dem in Abschnitt 6.3.2.3.4 untersuchten Fall anwächst.

6.3.2.3.6 Verkettete Zyklen

Ein weiterer Komplexitätstreiber des Testens, der auf dem Muster Zyklus basiert, sind miteinander verkettete Zyklen. Auch für Modelle nach diesem Muster wurde der Transformationsalgorithmus validiert. In Abbildung 6.50 ist ein Beispiel verketteter Zyklen abgebildet.

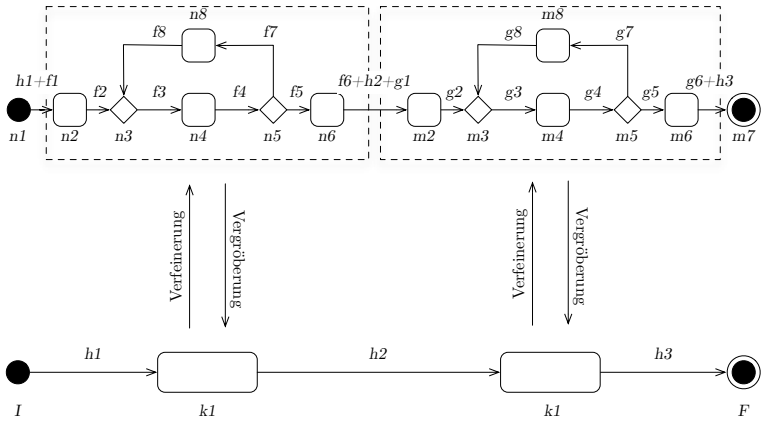


Abbildung 6.50: Konkatenation des Modellierungsmusters Zyklus

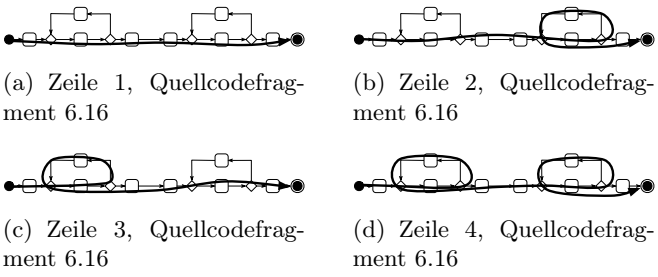
Dargestellt ist in Abbildung 6.50 eine Konkatenation des bereits in Abschnitt 6.3.2.3 eingeführten Aktivitätsdiagramms in Abbildung 6.38. Konkret handelt es sich bei dem in Abbildung 6.50 dargestellten Aktivitätsdiagramm um eine Variante des Modellierungsmusters Sequenz (vgl. Abschnitt 6.3.2.1) der Länge zwei, wobei beide Aktivitäten jeweils durch eine zyklische Struktur verfeinert wurden.

In Abschnitt 6.3.2.1 wurde die Korrektheit der Transformation des Modellierungsmusters Sequenz nachgewiesen. Es verbleibt der Nachweis zu erbringen, dass die Transformation auch dann ein im Sinn des Permutationssequenzüberdeckungskriteriums korrektes Ergebnis erzeugt, wenn Aktivitäten durch Modellierungsmuster höherer Ordnung verfeinert werden.

In Quellcodefragment 6.16 sind die aus dem in Abbildung 6.50 dargestellten Aktivitätsdiagramm generierten Ausführungssequenzen abgebildet. Alle vier Ausführungspfade durchlaufen das Aktivitätsdiagramm vom Anfangs- bis zum Endknoten und alle Aktivitäten werden in der modellierten Reihenfolge durch Testfälle abgebildet. Wie vom Permutationssequenzüberdeckungskriterium gefordert, werden alle im Modell existierenden Zyklen mindestens einmal durchlaufen. In Abbildung 6.51 sind die Ausführungspfade graphisch aufbereitet dargestellt.

Quellcodefragment 6.16: Textuelle Repräsentierung der aus dem Aktivitätsdiagramm in Abbildung 6.50 generierten Ausführungspfade

```
1 !I->h1->f1->n2->f2->f3->n4->f4->f5->n6->f6->h2->g1->m2->g2->g3->m4->↙
      ↳ g4->g5->m6->g6->h3->!F
2 !I->h1->f1->n2->f2->f3->n4->f4->f5->n6->f6->h2->g1->m2->g2->g3->m4->↙
      ↳ g4->g7->m8->g8->g3->m4->g4->g5->m6->g6->h3->!F
3 !I->h1->f1->n2->f2->f3->n4->f4->f7->n8->f8->f3->n4->f4->f5->n6->f6->↙
      ↳ h2->g1->m2->g2->g3->m4->g4->g5->m6->g6->h3->!F
4 !I->h1->f1->n2->f2->f3->n4->f4->f7->n8->f8->f3->n4->f4->f5->n6->f6->↙
      ↳ h2->g1->m2->g2->g3->m4->g4->g7->m8->g8->g3->m4->g4->g5->m6->g6↙
      ↳ ->h3->!F
```



(a) Zeile 1, Quellcodefragment 6.16

(b) Zeile 2, Quellcodefragment 6.16

(c) Zeile 3, Quellcodefragment 6.16

(d) Zeile 4, Quellcodefragment 6.16

Abbildung 6.51: Graphische Aufbereitung der Transformationsergebnisse für das Aktivitätsdiagramm in Abbildung 6.50

6.3.2.4 Zusammenfassung

In den vorangegangenen Abschnitten 6.3.2.1 bis 6.3.2.3 wurden die in Abschnitt 6.3.1 identifizierten atomaren Modellierungsmuster zu komplexeren Modellierungsmustern zusammengesetzt. Mit dem Modellierungselement *UML::CallBehaviorAction* bietet das Metamodell der UML die Möglichkeit, Teilmodelle von UML-Aktivitätsdiagrammen zu vergröbern oder individuelle Aktivitäten durch Teilmodell zu verfeinern, wobei jedes nichtatomare Aktivitätsdiagramm durch Vergröberung eines knotenberandeten Teilmodells zu einer einzelnen Aktivität auf die in Abschnitt 6.3.1 diskutierten atomaren Modellierungsmuster reduziert werden kann.

Basierend auf dieser Eigenschaft des UML-Metamodells wurde in den vorangegangen Abschnitten gezeigt, dass der in dieser Dissertation erarbeitete Transformationsalgorithmus auch für komplexe Modelle korrekte Ergebnisse im Sinne des Permutationssequenzüberdeckungskriteriums generiert. Der Nachweis erfolgt hierbei basierend auf dem bereits in Abschnitt 6.3.1 erbrachten Nachweis der Korrektheit der Transformation atomarer Modellierungsmuster.

Der Transformationsalgorithmus basiert auf der Überführung von Aktivitätsdiagrammen zu Petri-Netzen und der Generierung von Ausführungspfaden durch Traversierung der Erreichbarkeitsgraphen der Petri-Netze. In der Semantik des in dieser Dissertation entworfenen Transformationsalgorithmus ist die Verfeinerung einer Aktivität eines Aktivitätsdiagramms äquivalent zur Verfeinerung der zu dieser Aktivität gehörenden Stelle des aus dem Aktivitätsdiagramm erzeugten Petri-Netzes mit einem stellenberandeten Teilnetz, welches die verfeinernde Aktivität abbildet (vgl. Abbildung 6.52). Bei der Generierung von Testfällen manifestiert sich diese Eigenschaft darin, dass die generierten Ausführungspfade Kompositionen der jeweils aus den atomaren Modellierungsmustern generierten Teilsequenzen sind.

Eine Validierung durch manuelle Inspektion der Transformationsergebnisse kann für komplexe Modelle nicht erfolgen, weil die Komplexität der Testfälle rasch für menschliche Akteure unüberschaubar wird. Deshalb wurde die Transformation von UML-Aktivitätsdiagramm zu Testfallmodell auf Modelle angewendet, die gezielt so konstruiert wurden, dass strukturelle Eigenschaften der Transformation in den Ergebnissen erkennbar sind und Rückschlüsse auf das Verhalten der Transformation bei komplexeren Modellen zulassen.

In allen Fällen wurden aus den Aktivitätsdiagrammen im Sinne des Überdeckungskriteriums korrekte Ausführungspfade generiert. Aufgrund der Funktionsweise der Transformation wird gefolgert, dass die Transformation auch dann korrekte Ergebnisse generiert, wenn Modelle weiter verfeinert werden.

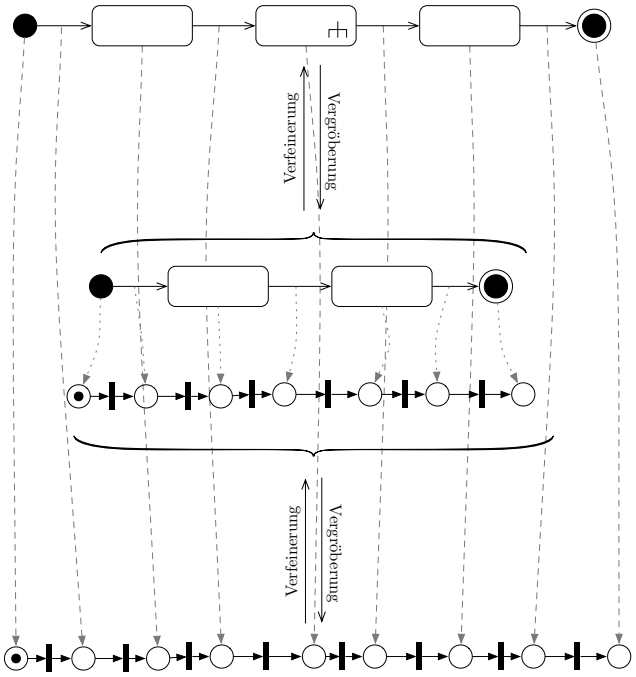


Abbildung 6.52: Äquivalenz von Petri-Netz und Aktivitätsdiagramm in der Semantik der Transformation

6.4 Validierung der Generierung von Calabash-Tests

Schritt 4 der in Abschnitt 5.1 vorgestellten Methode zur Generierung von Tests aus UML-Aktivitätsdiagrammen ist die Erzeugung technologiespezifischer Tests aus Testfallmodellen. Die im Sinne des Permutationssequenzüberdeckungskriteriums korrekte Generierung von Testmodellen aus Aktivitätsdiagrammen wurde bereits in Abschnitt 6.3 bestätigt. Im nächsten Schritt der Validierung ist also zu zeigen, dass auch die M2T-Transformation von Testfallmodell zu Calabash-Tests ebenfalls korrekt ist.

In Abschnitt 5.4 wurde die Methode zur Transformation von Testfallmodell zu Calabash-Tests diskutiert. Es wurde dargelegt, wie individuelle Elemente eines Testfallmodells auf Calabash-Testschritte abgebildet werden. Ziel der Validierung dieses Transformationsschritts ist es, den Nachweis zu erbringen, dass zu jedem Testfallmodellelement der korrekte Calabash-Testschritt erzeugt wird. Die Überführung von Testfallmodell zu Calabash-Tests ist eine M2T-Transformation, d. h. aus einem Artefakt der Modellierung wird Text, konkret eine Menge von Calabash-Instruktionen, generiert. Das in Abschnitt 5.2.3.2 vorgestellte Metamodell zur Testfallmodellierung definiert Modellierungselemente, die bei der Erstellung von Testfallmodellen verwendet werden können. Wenngleich Testfallmodelle vom Grundsatz her auch manuell erstellt werden können, wird diese Aufgabe durch den Transformationsalgorithmus automatisiert, so dass durch die in Abschnitt 6.3 durchgeführte Validierung der Testfallgenerierung sichergestellt ist, dass im Testfallmodell nur gültige (d. h. syntaktisch korrekte) Modellierungselemente vorkommen und nur gültig miteinander in Relation stehen.

In Abschnitt 5.5.3.2 wurden im Rahmen der Diskussion der angepassten Implementierung des Calabash-Frameworks Calabash-Testschritte vorgestellt (vgl. auch Anhang A). Validie-

rungsziel ist es zu überprüfen, ob aus Testfallmodellen eine Zeichenfolge generiert wird, die das zu überprüfende Testfallmodellelement im Calabash-Syntax repräsentiert. Ein probates Vorgehen zur Überprüfung der Korrektheit dieser Transformation sind Unit-Tests.

Zur Validierung der M2T-Transformation zur Erzeugung von Calabash-Tests wurden Unit-Tests erstellt. Diese erzeugen prototypische Testfallmodelle, auf welche dann die M2T-Transformation angewendet wurde, um die Spezifikationskonformität des resultierenden Calabash-Features zu prüfen. Das Calabash-Feature wurde hierzu als Zeichenfolge interpretiert, die mit konstanten Zeichenfolgen in der Funktion eines Testorakels verglichen wurden. Jede Zeichenfolge des Orakels repräsentiert hierbei ein Calabash-Fragment, durch welches Testfallmodellelement repräsentiert wird.

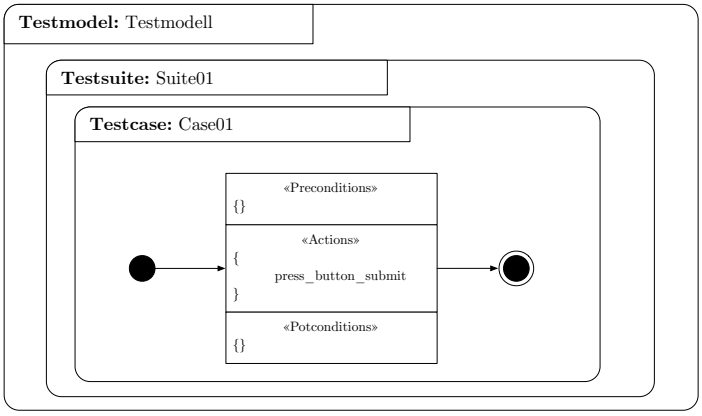


Abbildung 6.53: Beispiel eines minimalen Testfallmodells zur Validierung der M2T-Transformation zu Calabash

In Abbildung 6.53 ist beispielhaft eine graphische Repräsentierung eines Testfallmodells dargestellt. Es modelliert ein *Test::TestModel*-Element, welches ein *Test::TestSuite*-Element enthält, in welches wiederum ein einzelnes *Test::TestCase*-Element eingebettet ist. Strukturell entspricht dieses Modell dem Ergebnis einer Transformation von UML-Aktivitätsdiagramm zu Testfallmodell (vgl. Abschnitt 5.3.3.2).

Das in Abbildung 6.53 dargestellte Beispiel modelliert konkret einen Testfall, dessen einziger Inhalt ein *Test::Action*-Element ist, welches das Anklicken einer interaktiven Schaltfläche in einem UI einer mobilen App ist. Der abgebildete Testfall dient der Validierung der M2T-Transformation und ist deshalb bewusst minimal gehalten und enthält keine Vor- oder Nachbedingungen, wie es bei einem tatsächlichen Testfall der Fall wäre.

Zur Validierung der M2T-Transformation wurden für die Modellierungselemente des Metamodells zur Testfallmodellierung, die Interaktionen mit dem SUT (z. B. Texteingabe, Simulation von Sensordaten) und für Elemente, die Vor- oder Nachbedingungen modellieren (z. B. Textanzeige, Screenshot-Vergleich) jeweils Unit-Tests erstellt, die analog zur Abbildung 6.53 ein minimales Testfallmodell erzeugen. Anschließend wurde die M2T-Transformation darauf angewendet und mittels einer JUnit-Assertion überprüft, ob eine für dieses Modellelement korrekte Zeichenfolge erzeugt wird, die das Modellelement im Calabash-Syntax repräsentiert. Konkret für das in Abbildung 6.53 dargestellte Testfallmodell wurden die in den Codeabschnitten 6.17 und 6.18 abgedruckten Unit-Tests erstellt.

Quellcodefragment 6.17: Unit Test zum in Abbildung 6.53 dargestellten Testfallmodell

```
1  package de.paluno.mobiletetsing.testgenerationplugin.handlers.tests;
2
3  import static org.hamcrest.CoreMatchers.containsString;
4
5  public class CalabashConversionTest {
6
7     // define test fixture
8     public static final String FIXTURE_MODEL_NAME = "Testmodell";
9     public static final String FIXTURE_SUITE_NAME = "Suite01";
10    public static final String FIXTURE_CASE_NAME = "Case01";
11    public static final String FIXTURE_PRESS_SUBMIT = "Then␣I␣press␣↙
          ↳ view␣with␣id␣\"button_submit\"";
12
13    /**
14     * prepare empty test model
15     */
16    @Before
17    public void setup() {
18
19       factory = TestcaseModellingFactory.eINSTANCE;
20       assertNotNull(factory);
21
22       testModel = factory.createTestModel();
23       assertNotNull(testModel);
24
25       testModel.setLabel(FIXTURE_MODEL_NAME);
26       assertEquals(FIXTURE_MODEL_NAME, testModel.getLabel());
27
28       testSuite = factory.createTestSuite();
29       assertNotNull(testSuite);
30
31       testSuite.setLabel(FIXTURE_SUITE_NAME);
32       assertEquals(FIXTURE_SUITE_NAME, testSuite.getLabel());
33
34       // ...
35
36    }
37
38    /**
39     * test for ClickUiElementAction
40     */
41    @Test
42    public void testClickUiElementAction() {
43
44       TestStep step = factory.createTestStep();
45       assertNotNull(step);
46
47       testCase.getStep().add(step);
48
49       Edge edge01 = factory.createEdge();
50       assertNotNull(edge01);
51       edge01.setSource(initialNode);
52       edge01.setTarget(step);
53
54       Edge edge02 = factory.createEdge();
55       assertNotNull(edge02);
56       edge02.setSource(step);
57       edge02.setTarget(finalNode);
58
59       // ..
60
61    }
62
63 }
```

In den Zeilen 8-11 in Quellcodefragment 6.17 werden einige Konstanten definert, die im Test bei der Durchführung von Assertions verwendet werden. In Zeilen 13-36 erfolgt der Aufbau des zur eigentlichen Testdurchführung verwendeten Testfallmodells. Das spezifische zu testende Artefakt wird in den Zeilen 38-61 erzeugt.

In Quellcodefragment 6.18 ist erneut ein Ausschnitt aus dem zu Abbildung 6.53 gehörigen Unit-Test abgedruckt, hier allerdings mit einem Fokus auf die eigentliche Testdurchführung. In

den Zeilen 10-17 erfolgt die Konfiguration des Testobjektes, d. h. des Testfallmodells, konkret das Hinzufügen eines Elements vom Typ *Test::ClickUIElementAction*. Der eigentliche Test der M2T-Transformation erfolgt in Zeile 21 in Quellcodefragment 6.18 als JUnit-Assertion, welche die durch die M2T-Transformation generierte Zeichenfolge mit einem Orakel vergleicht.

Analog zu diesem Beispiel konnte für die im Metamodell zur Testfallmodellierung (vgl. Abschnitt 5.2.3.2) definierten Modellierungselemente die Korrektheit der M2T-Transformation nachgewiesen werden. Bei der Durchführung der Fallstudien in Abschnitt 6.5 wurde die M2T-Transformation zusätzlich im Zusammenwirken mit der Generierung von Testfällen überprüft.

Quellcodefragment 6.18: Unit Test zum in Abbildung 6.53 dargestellten Testfallmodell, Fokus Testartefakt

```
1
2       /**
3        * test for ClickUiElementAction
4        */
5       @Test
6       public void testClickUiElementAction() {
7
8           // ..
9
10          ClickUIElementAction clickAction = ↙
                ↳ TestcaseModellingActionFactory.eINSTANCE.↙
                ↳ createClickUIElementAction();
11
12          ViewIdTarget target = TestcaseModellingCommonFactory.eINSTANCE↙
                ↳ .createViewIdTarget();
13          target.setViewid("button_submit");
14
15          clickAction.setViewid(target);
16
17          step.getAction().add(clickAction);
18
19
20          String calabash = new CalabashAndroidConverter().convert(↙
                ↳ testSuite);
21          assertThat(calabash, containsString(FIXTURE_PRESS_SUBMIT));
22
23      }
24
25  }
```

6.5 Fallstudien

Zur funktionalen Evaluierung der Testfallgenerierung und automatisierten Ausführung von Tests wurden Fallstudien durchgeführt, die den in dieser Dissertation untersuchten Ansatz zur Testautomatisierung ganzheitlich von der Modellierung bis hin Testausführung anwenden.

Die erste Fallstudie evaluiert in Abschnitt 6.5.1 die Testautomatisierung an der in Kapitel 3 eingeführten und in der Dissertation durchgängig als Beispiel verwendeten Anwendung Mobiler Taxiruf. Es handelt sich um ein akademisches Fallbeispiel, das inhaltlich an existierende mobile Anwendungen angelehnt ist. Die Fallstudie dient der Untersuchung des Verfahrens der Testfallgenerierung aus UML-Aktivitätsdiagrammen anhand einer mobilen Anwendung, die den Kontextparameter Standort verwendet und deren dynamische und strukturelle Eigenschaften vollständig bekannt sind.

Eine funktionale Evaluierung der Testfallgenerierung in Verbindung mit dem Kontextsimulator zur Unterstützung automatisierter Tests für solche Anwendungen, die weitere physikalische Kontextparameter verwenden, insbesondere Sensordaten, erfolgt in den Abschnitten 6.5.2 bis 6.5.3. In diesen Abschnitten werden Fallstudien an Apps mit einem besonderen Fokus

auf die Verwendung von Sensordaten diskutiert, wobei in der Fallstudie Smart Thermometer in Abschnitt 6.5.2 die solitäre Verwendung von Sensordaten (d. h. Sensordaten werden ohne Bezug zueinander verwendet) im Fokus steht, während in den Fallstudien Wasserwagen-App und Kompass-App die Daten mehrerer Sensoren aggregiert verwendet werden, um einen Anwendungsfall zu realisieren.

6.5.1 Mobiler Taxiruf

In Kapitel 3 wurde die Beispielanwendung Mobiler Taxiruf eingeführt. Dieses Beispiel wurde durchgängig in dieser Dissertation verwendet, um die besonderen Herausforderungen des Testens mobiler, kontextsensitiver Apps zu diskutieren. Die App repräsentiert einen Geschäftsprozess, der in vielfältigen Implementierungen in den App Stores für unterschiedliche mobile Plattformen erhältlich ist (z. B. mytaxi™ [198]). Zur Untersuchung der Modelltransformation und der Testautomatisierung in der Fallstudie wurde die Anwendung auf wesentliche Merkmale reduziert und von untergeordneter Funktionalität wie beispielsweise dem Hinterlegen von Nutzerdaten oder Zahlungsinformationen abstrahiert.

Aspekte der Kontextsensitivität manifestieren sich in der Verwendung des Standorts des Anwenders als Eingabevektor in die App. Die Ermittlung des Standorts des Anwenders erfolgt durch Verwendung des GPS-Moduls des Geräts. Verfügt das Gerät nicht über ein GPS-Modul oder kann in einem Zeitraum von 30 Sekunden der Standort des Anwenders nicht bestimmt werden oder es erfolgt keine Bestätigung durch den Anwender, wird dem Anwender alternativ zur Kartenansicht ein Dialog zur manuellen Eingabe seines Standorts angezeigt. Die Taxibestellung erfolgt über eine Schaltfläche auf dem UI. Eine Bestellung kann ausgelöst werden, wenn eine Standortinformation vorliegt, die entweder manuell eingegeben wurde oder vom GPS-Modul erzeugt wurde und eine Genauigkeit von höchstens 150 Meter ausweist.

Im Folgenden wird zunächst in Abschnitt 6.5.1.1 das Modell der Anwendung im Detail diskutiert und gezeigt, wie das UML-Profil zur Testfallmodellierung (vgl. Abschnitt 5.2.4) auf das UML-Aktivitätsdiagramm angewendet wurde, das den Anwendungsinhalt modelliert. Als Gegenstand der Fallstudie wird in Abschnitt 6.5.1.2 eine prototypische Implementierung der App für die Plattform Android in der Version 5.1.1 vorgestellt. Auf die prototypische Implementierung werden die in Abschnitt 6.5.1.3 diskutierten Tests angewendet, die automatisch unter Verwendung der in Kapitel 5 vorgestellten Methode generiert wurden. Eine Analyse der Ergebnisse erfolgt in Abschnitt 6.5.1.4.

6.5.1.1 Modell der Anwendung

Ausschnitte aus dem Modell der App Mobiler Taxiruf wurden bereits in den Kapiteln 3 bis 5 unter unterschiedlichen Aspekten vorgestellt. Das vollständige Verhaltensmodell der App in Form eines UML-Aktivitätsdiagramms ist in den Abbildungen 6.54 bis 6.60 dargestellt. Aus Gründen der Übersichtlichkeit bedient sich das Modell des Modellierungselements *UML::CallBehaviorAction*, um die Komplexität des Modells auf die in Abbildung 6.54 dargestellte Abstraktionsebene zu reduzieren. Diese Komplexitätsreduktion ist im Einklang mit dem in Abschnitt 6.3.1 und Abschnitt 6.3.2 diskutierten Vorgehen, ein komplexes Modell durch sukzessive Verfeinerung aus Teilmodellen geringer Komplexität zusammenzusetzen.

6.5.1.1.1 Ortsbasierter Taxiruf

In Abbildung 6.54 ist das Modell der App auf der höchsten Abstraktionsebene dargestellt. Das Modell besteht aus insgesamt sieben Aktivitätsdiagrammen, durch welche die Anwendung in zunehmendem Detailgrad spezifiziert wird. Das Aktivitätsdiagramm Mobiler Taxiruf enthält sechs Elemente des Typs *UML:CallBehaviorAction*, die das Modell verfeinern. Auf einige Kontrollflüsse wurde der Stereotyp *TestFlow* des UML-Profils zur Integration der Testfallmodellierung in die Systemmodellierung angewendet. Konkret wurden jene Kontrollflüsse mit dem Stereotyp *TestFlow* annotiert, die eine Guard-Condition spezifizieren. Die Testautomatisierung verwendet die mit dem Kontrollfluss assoziierten Testdaten zur Steuerung des SUT.

Am Verzweigungskonten $da2$ können sechs disjunkte Ereignisse eintreten: (1) der Anwender entschließt sich, die Taxibestellung abzubrechen, (2) der durch das GPS-Modul ermittelte Standort des Anwenders befindet sich außerhalb Deutschlands, (3) die ermittelte Standortinformation ist von zu geringer Qualität, (4) der Anwender entschließt sich, seinen Standort manuell einzugeben, (5) nach Ablauf von 30 Sekunden liegt keine Standortinformation vor oder der Anwender hat nicht bestätigt oder (6) der Anwender bestätigt die per GPS-Modul ermittelte Standortinformation. Im Aktivitätsdiagramm wurde keine der Aktivitäten mit dem Stereotyp *TestStep* annotiert, da alle Aktivitäten durch ein Aktivitätsdiagramm verfeinert werden, in welchem das jeweilige Verhalten im Detail spezifiziert ist. Alternativ zur Verwendung von *UML:CallBehaviorAction* gäbe es die Möglichkeit, die in den verfeinernden Aktivitätsdiagrammen modellierten Aktivitäten direkt auf der obersten Ebene zu modellieren, wodurch die Lesbarkeit des Modells jedoch eingeschränkt würde.

Um die Abbildung 6.54 übersichtlich zu halten, sind nur einige der Testdatenobjekte dargestellt. Das für die Fallstudie erstellte Modell wurde folgendermaßen mit Testdaten annotiert: Auf den Kontrollfluss $fa5$ wurde der Stereotyp *TestFlow* angewendet. Hierdurch wird der Kontrollfluss durch die Attribute *Test::Precondition*, *Test::Action* und *Test::Postcondition* erweitert. Es handelt sich jeweils um Listen-Typen, die vom Grundsatz her eine beliebige Anzahl Vorbedingungen, Aktionen oder Nachbedingungen aus dem Metamodell zur Kontextmodellierung (vgl. Abschnitt 5.2.2.2) bzw. aus dem Metamodell zur Testfallmodellierung (vgl. Abschnitt 5.2.3.2) aufnehmen können. Tatsächlich wurde der Kontrollfluss $fa5$ mit einem Element des Typs *Test::ClickAction* annotiert, welches das Anklicken des Abbrechen-Buttons auf dem UI der App repräsentiert.

Kontrollfluss $fa7$ repräsentiert den Fall, dass per GPS eine Standortinformation ermittelt wurde, die vom Anwender bestätigt wird. Der Kontrollfluss ist mit einem Element vom Typ *Test::ClickAction* annotiert, welches das Anklicken des Übernehmen-Buttons auf dem UI der App repräsentiert.

Kontrollfluss $fa8$ repräsentiert den Fall, dass per GPS ein außerhalb Deutschlands liegender Standort ermittelt wird. In diesem Fall ist der Taxi-Dienst nicht verfügbar und die App terminiert mit einem Fehlerdialog, der separat in einem Aktivitätsdiagramm modelliert ist.

Kontrollfluss $fa15$ repräsentiert den durch die Verzweigung $da16$ eingeleiteten Zyklus, der die App zur Aktivität Standortbestimmung zurückführt. Im UI der App wird dieser Zyklus durch die Zurück-Taste der Android-UI eingeleitet. Der Kontrollfluss ist deshalb mit einem Element des Typs *Test::BackButtonAction* annotiert.

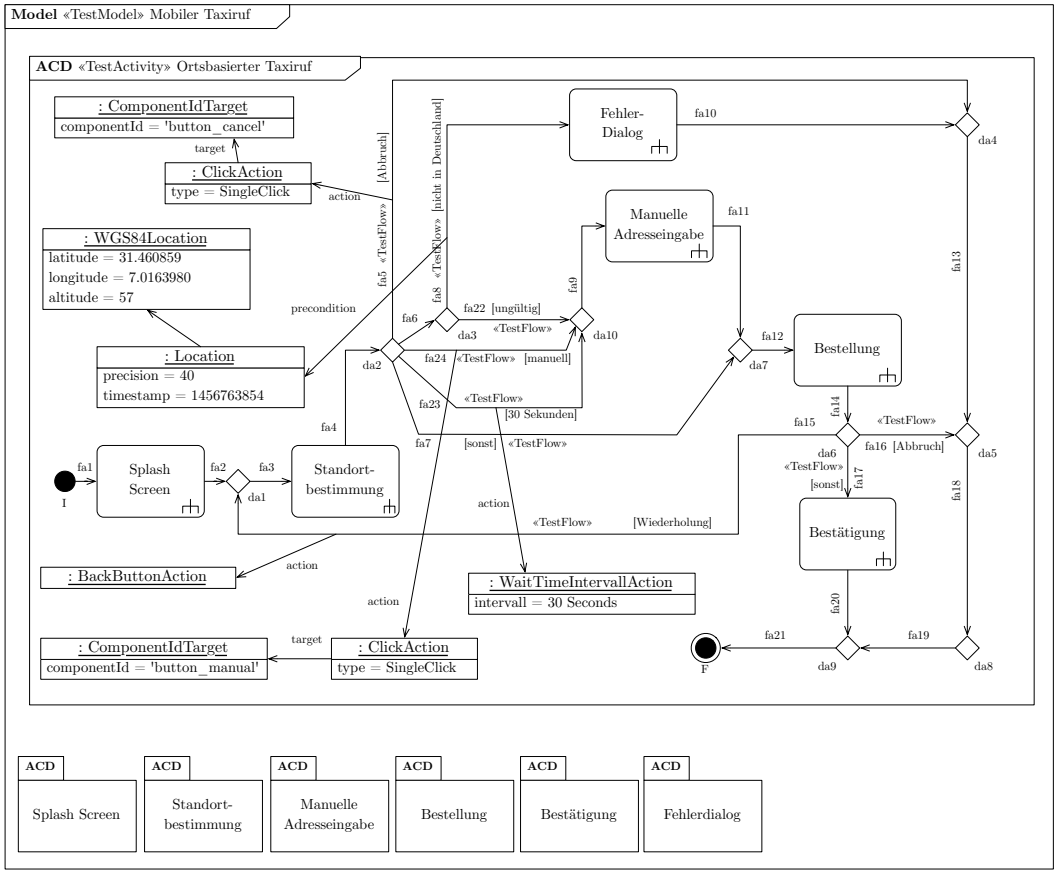


Abbildung 6.54: Aktivitätsdiagramm Mobiler Taxiruf auf höchster Abstraktionsebene

Kontrollfluss $fa16$ repräsentiert analog zu $fa5$ den Abbruch der Taxibestellung durch den Anwender.

Kontrollfluss $fa17$ repräsentiert die Auslösung der Taxibestellung durch den Anwender. Modelliert ist diese durch ein Element des Typs *Test::ClickAction* am Übernehmen-Button der Aktivität Bestellung.

Kontrollfluss $fa22$ ist analog zu $fa8$ mit einem Element des Typs *Context::Location* annotiert. Diesmal mit einem Koordinatenpaar innerhalb Deutschlands, allerdings mit einer Genauigkeit von 500 Metern. Zur Verwendung eines per GPS ermittelten Standorts wird gefordert, dass dessen Genauigkeit kleiner als 150 Meter ist. Deshalb wechselt die App ersatzweise zur manuellen Adresseingabe.

Kontrollfluss $fa23$ repräsentiert den Fall, dass entweder innerhalb von 30 Sekunden kein Standort ermittelt werden kann oder der Anwender den per GPS ermittelten Standort nicht bestätigt. Der Kontrollfluss ist mit einem Element von Typ *Test::WaitTimeIntervalAction* annotiert.

Kontrollfluss $fa24$ repräsentiert den Entschluss des Anwenders, die vom GPS-Modul erzeugte Standortinformation zu ignorieren und stattdessen manuell eine Adresse einzugeben. Der Kontrollfluss ist deshalb mit einem *Test::ClickAction*-Element annotiert, welches das Anklicken des Buttons zum Wechsel zur manuellen Eingabe repräsentiert.

6.5.1.1.2 Splash Screen

Nachdem die Anwendung durch den Anwender gestartet wurde, wird zunächst für die Dauer von zwei Sekunden ein Begrüßungsbildschirm angezeigt. Das Teilmodell in Abbildung 6.55 modelliert das Aktivitätsdiagramm Splash Screen, welches die Anzeige des Begrüßungsbildschirms realisiert und den Kontrollfluss dann unmittelbar an die Aktivität Standortbestimmung übergibt.

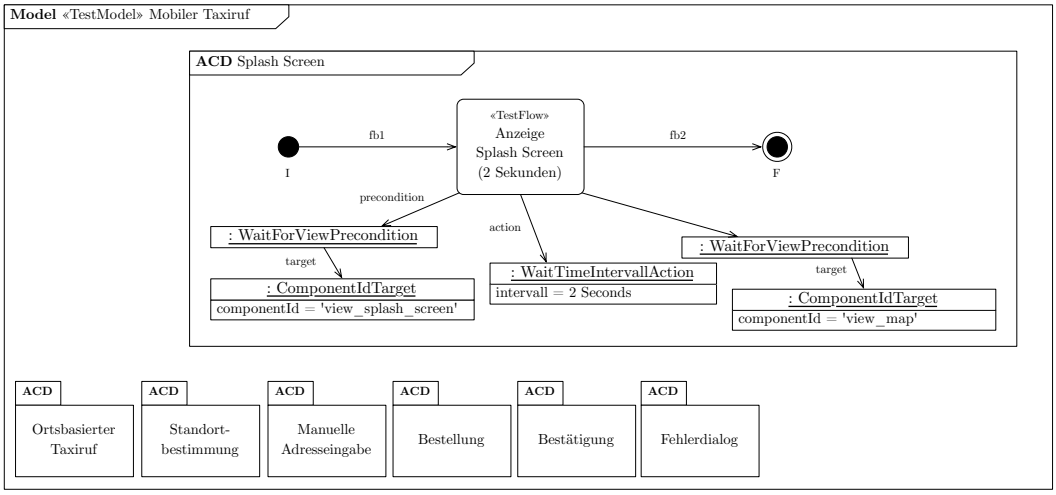


Abbildung 6.55: Aktivitätsdiagramm Mobiler Taxiruf, Aktivität Splash Screen

Relevant für die Testausführung ist, dass (1) der Splash Screen tatsächlich auf dem UI angezeigt wird und (2) anschießend nach einer Zeitdauer von zwei Sekunden die Kontrolle an die folgende Aktivität übergeben wird. Im Modell der Testdaten wird das abgebildet, indem an der Aktivität ein Element des Typs *Test::WaitForViewPrecondition* als Vorbedingung annotiert ist, welche das UI der Aktivität Splash Screen referenziert. Das Abwarten des 2-Sekunden-Intervalls wird durch ein Element des Typs *Test::WaitTimeIntervalAction* abgebildet. Durch die Nachbedingung vom Typ *Test::WaitForViewPostcondition* wird sichergestellt, dass das Zeitintervall von zwei Sekunden nicht überschritten wird.

6.5.1.1.3 Standortbestimmung

Die Aktivität Standortbestimmung in Abbildung 6.56 wird gestartet, sobald der Begrüßungsbildschirm terminiert. Fortgesetzt wird mit der Aktivität Bestimmung des letzten bekannten Standorts. Die APIs mobiler Plattformen bieten hierzu i. d. R. Funktionen an, die den letzten bekannten Standort ohne Zeitverlust zurückliefern, wenn diese Information vorliegt.

Diese Standortinformation kann nicht existent, unpräzise oder veraltet sein. Aus diesem Grund schließt sich als nächster Schritt eine Aktivität zur Bewertung der Gültigkeit der Standortinformation an. Um sicherzustellen, dass zur Anzeige der Kartendarstellung im darauffolgenden Schritt eine Standortinformation verfügbar ist, fließt in die Aktivität Auswertung Gültigkeit der Standortbestimmung ein Standwert aus einem Datencontainer ein. Die Aktivität Auswertung Gültigkeit der Standortbestimmung stellt sicher, dass in die Folgeaktivität Aktualisierung der Kartenansicht eine Standortinformation einfließt.

Solange der Anwender die Standortinformation nicht durch Interaktion mit der UI der Anwendung bestätigt, wird die Kartenansicht durch Verwendung des GPS-Moduls mit Standortinformationen aktualisiert. Nach Ablauf von 30 Sekunden wird der Kontrollfluss an die Aktivität Manuelle Adresseingabe übergeben. Ebenfalls hat der Anwender die Möglichkeit, aktiv zur Aktivität Manuelle Adresseingabe zu wechseln. Bestätigt der Anwender die Standortinformation (manuell oder via GPS bereitgestellt), wird der Kontrollfluss an der Aktivität Bestellung des übergeordneten Modells fortgesetzt. Die Aktivitäten Aktualisierung der Kartenansicht und Standortbestimmung GPS sind Aufrufe von Funktionen des API. Die werden deshalb nicht auf feinerer Detailebene modelliert.

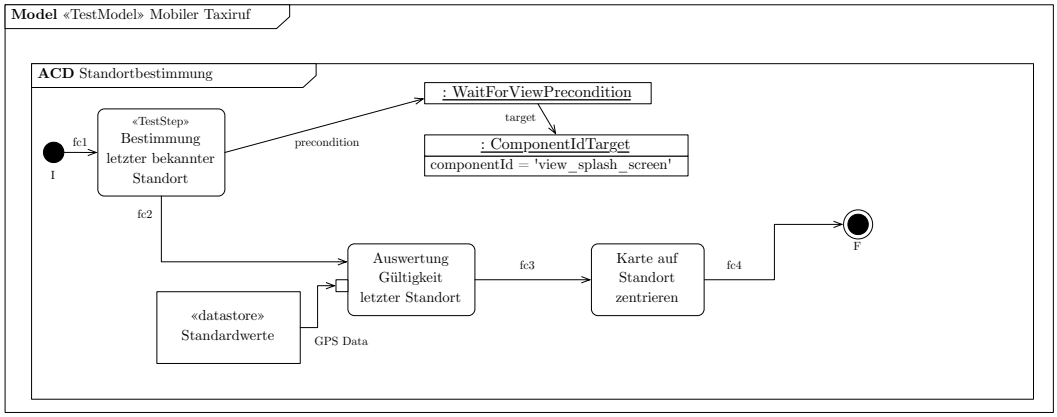


Abbildung 6.56: Aktivitätsdiagramm Mobiler Taxiruf, Aktivität Standortbestimmung

Die Zentrierung der Karte auf spezifische Koordinaten ist eine Funktion, die durch das Android-SDK bereitgestellt wird. Ein Test, ob die Kartenansicht tatsächlich auf die korrekten Koordinaten zentriert dargestellt wird, ist nicht Gegenstand der Fallstudie. Zwar bietet Calabash-Android grundsätzlich auch hierfür eine Step-Definition. Da es sich aber um eine Funktion des Android-SDK handelt, ist ein Defekt dieser Funktion unwahrscheinlich und ließe sich darüber hinaus im Rahmen der Implementierung der App Mobiler Taxiruf auch nicht korrigieren. Die Fallstudie umfasst daher lediglich die Überprüfung, ob die Kartenansicht überhaupt auf dem UI der App angezeigt wird.

6.5.1.1.4 Manuelle Adresseingabe

Entscheidet sich der Anwender, seinen Standort nicht durch das GPS-Modul ermitteln zu lassen, sondern zieht es vor, die Adresse manuell einzugeben oder wenn ungültige Standortinformationen vom GPS-Modul erzeugt werden, wird innerhalb der Aktivität Manuelle Adresseingabe in Abbildung 6.57 eine Eingabemaske angezeigt, in welcher der Anwender Straßenname und Hausnummer sowie Stadt und Postleitzahl in beliebiger Reihenfolge eingeben kann. Nach dem Bestätigen der Adresseingabe wird der Kontrollfluss an der Aktivität Bestellung des übergeordneten Aktivitätsdiagramms fortgesetzt.

Gegenstand des Testens ist zunächst sicherzustellen, dass tatsächlich das UI zur manuellen Adresseingabe angezeigt wird. Aus diesem Grund wurde im Modell auf die Aktivität Anzeige Dialog Adresseingabe der Stereotyp *TestStep* angewendet und mit einem Element Typ *Test::WaitForViewPrecondition* annotiert.

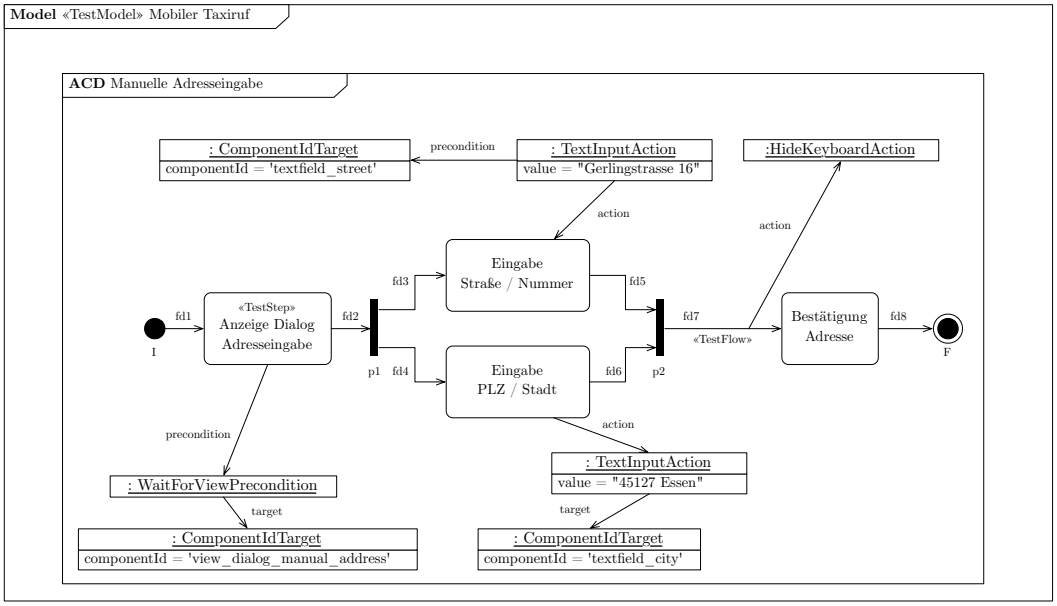


Abbildung 6.57: Aktivitätsdiagramm Mobiler Taxiruf, Aktivität Manuelle Adresseingabe

Die beiden Aktivitäten zur Eingabe von Straße und Hausnummer sowie Stadt und Postleitzahl sind jeweils einem Element des Typs *Test::TextInputAction* annotiert, welche die während der Testausführung zu verwendenden Testdaten spezifizieren.

Auf den Kontrollfluss $fd7$ wurde der Stereotyp *TestFlow* angewendet und ein Element des Typs *Test::HideKeyboardAction* modelliert. Bei der automatisierten Testausführung wird hierdurch das Soft-Keyboard des Android-Systems nach erfolgter Eingabe der Testdaten verborgen.

6.5.1.1.5 Fehlerdialog

Wenn der per GPS ermittelte Standort nicht innerhalb Deutschlands liegt, wird dem Anwender ein Fehlerdialog angezeigt und die App terminiert. Die korrespondierende Aktivität ist in Abbildung 6.58 dargestellt.

Auch in dieser Aktivität sind wieder einige Testdaten modelliert. Hervorzuheben ist die an Kontrollfluss $fg2$ annotierte Nachbedingung vom Typ *Test::TextAssertion*, durch welche bei der Testausführung überprüft wird, ob im Fehlerdialog ein spezifikationskonformer Fehlertext angezeigt wird.

6.5.1.1.6 Bestellung

Nachdem der Anwendung die Adresse des Taxirufs bekannt geworden ist, entweder durch manuelle Eingabe oder durch Verwendung des GPS-Moduls, hat der Anwender die Möglichkeit, den Taxibestellprozess mit der Aktivität Bestellung fortzusetzen. Zusätzlich besteht die Möglichkeit, den Bestellvorgang abzubrechen. In Abbildung 6.59 ist das Diagramm zur Aktivität Bestellen einschießlich der modellierten Testdaten dargestellt.

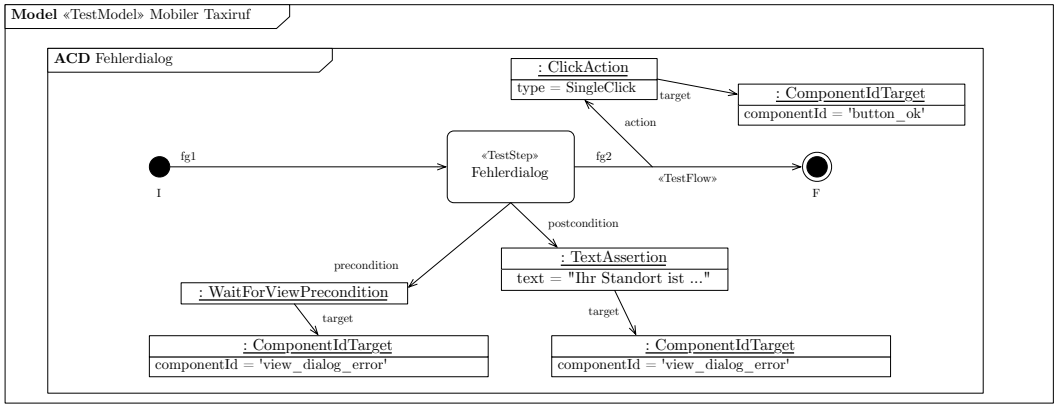


Abbildung 6.58: Mobiler Taxiruf, Aktivität Fehlerdialog

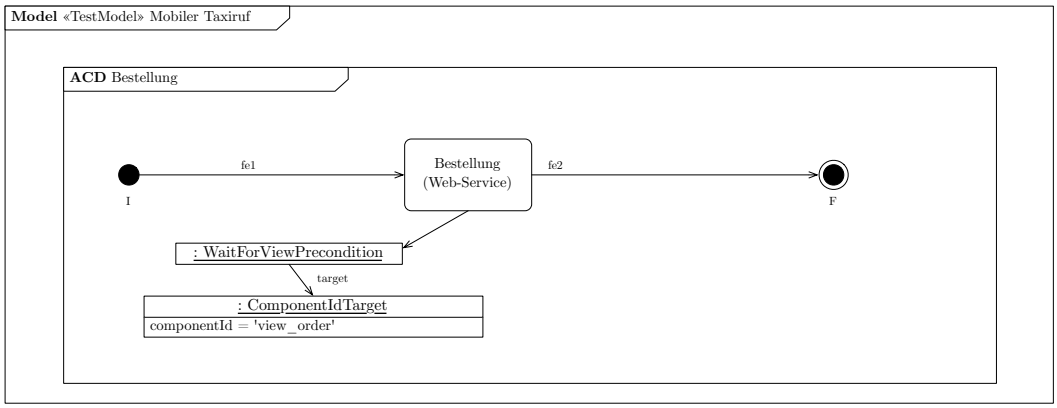


Abbildung 6.59: Aktivitätsdiagramm Mobiler Taxiruf, Aktivität Bestellung

6.5.1.1.7 Bestätigung

Abgeschlossen wird der Prozess der Taxibestellung, nachdem die Bestellung durchgeführt wurde und die verbleibende Wartezeit bis zur Ankunft des Taxis auf dem UI angezeigt wird. Diese Aktivität wird durch das Aktivitätsdiagramm in Abbildung 6.60 modelliert.

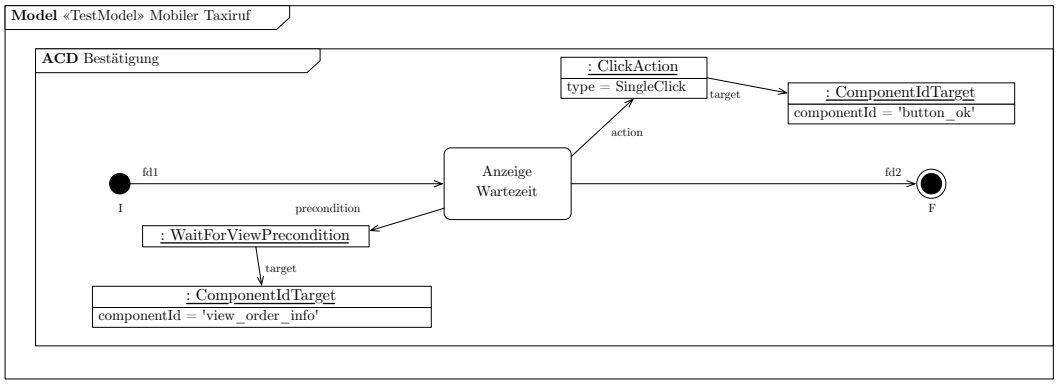


Abbildung 6.60: Aktivitätsdiagramm Mobiler Taxiruf, Aktivität Bestätigung

6.5.1.2 Prototypische Implementierung

Im Rahmen der Fallstudie wurde eine prototypische Implementierung der App Mobiler Taxiruf erstellt. Die App wurde unter Verwendung des Android SDK in der Version 23.0.1 implementiert. Die App selbst unterstützt Android in der Version 5.1.1 Lollipop. Die App implementiert eine Android-*Activity*[5] und sechs Android-*Fragments*[6] zur Abbildung des in Abschnitt 6.5.1.1 vorgestellten Anwendungsverhaltens.

Die Activity implementiert das durch das Aktivitätsdiagramm spezifizierte Anwendungsverhalten. Nach dem Start der Anwendung erzeugt die Activity eine Instanz des Fragments Splash Screen. Dieses implementiert den im Aktivitätsdiagramm Splash Screen in Abbildung 6.55 spezifizierten Begrüßungsbildschirm der Anwendung. Nach Ablauf von 2 Sekunden instruiert das Fragment die Activity, das Fragment Splash Screen durch das Fragment Standortbestimmung zu ersetzten. Die Activity terminiert das Fragment Splash Screen, erzeugt eine Instanz des Fragments Standortbestimmung und zeigt es auf der Benutzungsoberfläche an.

Das Fragment Standortbestimmung verwendet das GPS-Modul zur Ermittlung des Standorts des Geräts und zeigt diesen auf einer in die Anwendungsoberfläche eingebettete *GoogleMaps*-Kartenansicht an. Das GPS-Modul erzeugt die Standortinformation als WGS84-Koordinatenpaar. Das Fragment verwendet Reverse Geocoding[7], um die WGS84-Koordinaten in eine menschenlesbare Darstellung in Form einer Straßenadresse zu transformieren. Die so erzeugte Straßenadresse wird zum einen zur Anzeige im UI verwendet und erfüllt zum anderen die Aufgabe auszuwerten, ob sich der Anwender innerhalb Deutschlands befindet (Guard-Condition an Kontrollfluss $fa8$ in Abbildung 6.54, Abschnitt 6.5.1.1.1). Die WGS84-Darstellung der Standortinformation wird ebenfalls verwendet, um deren Qualität mit der in Guard-Condition an Kontrollfluss $fa22$ in Abbildung 6.54, Abschnitt 6.5.1.1.1 modellierten Anforderung zu vergleichen.

Je nach Qualität der Standortinformation instruiert das Fragment Standortbestimmung die Activity, den Kontrollfluss gemäß des in Abbildung 6.54, Abschnitt 6.5.1.1.1 definierten Anwendungsverhaltens fortzusetzen.

Die übrigen in Abschnitt 6.5.1.1 modellierten Aktivitätsdiagramme wurden ebenfalls in jeweils einem eigenen Fragment abgebildet. Auf die Kommunikation mit einem Backend-System im Rahmen der Fallstudie wurde verzichtet, so dass die übrigen Fragmente nach der Standortbestimmung bzw. der manuellen Adresseingabe trivial sind. Gegenstand des Tests der Anwendung ist zu überprüfen, ob der im Aktivitätsdiagramm in Abschnitt 6.5.1.1.1 modellierte Fluss durch die Anwendung in Abhängigkeit vom Standort des Anwenders korrekt implementiert ist. Bildschirmabdrucke der Anwendung sind nach der Art eines Storyboard in Abbildung 6.61 dargestellt.

[5] Strukturkomponente der Android-App Entwicklung zur Abbildung abgrenzbarer fachlicher Inhalte.

[6] Ebenfalls eine Strukturkomponente der Android-App-Entwicklung; dient der feineren Strukturierung fachlicher Inhalte unterhalb von Activities.

[7] Eine Komponente zum Reverse Geocoding wird vom Android-SDK bereitgestellt.

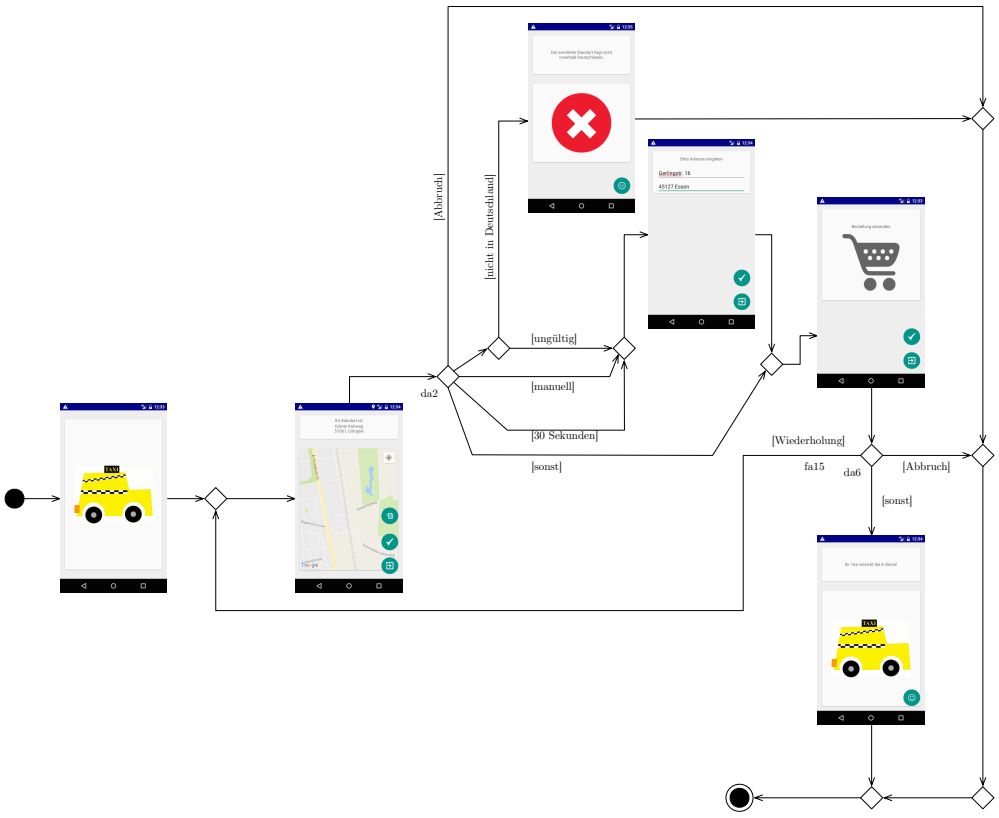


Abbildung 6.61: Storyboard der App Mobiler Taxiruf

6.5.1.3 Generierte Tests

Auf das in Abschnitt 6.5.1.1 diskutierte Modell der App Mobiler Taxiruf wurde der Algorithmus zur Generierung des Testfallmodells und zur Generierung von Calabash-Tests angewendet. Die Modelltransformation hat für dieses Modell insgesamt 30 Testfälle erstellt. Für das wenig komplexe Modell ist diese Anzahl von Testfällen verhältnismäßig groß. Dies ist einerseits in den sechs unterschiedlichen Alternativen an Verzweigungsknoten *da*2 in Abbildung 6.54 in Abschnitt 6.5.1.1.1 und andererseits in dem durch Verzweigungsknoten *da*6 in Abbildung 6.54 in Abschnitt 6.5.1.1.1 mit dem ausgehenden Kontrollfluss $fa15$ begründet.

Ein Ausschnitt aus den generierten Tests ist in Quellcodefragment 6.19 abgedruckt. Beispielhaft ist in Quellcodefragment 6.19 ein Testfall dargestellt, in dem zunächst ein ungültiger Standort vom GPS-Modul erzeugt wurde und der Anwender zum Dialog zur manuellen Adresseingabe geleitet wird.

Quellcodefragment 6.19: Calabash-Test der Anwendung Mobiler Taxiruf

```
1 Feature: ortsbasierter_taxiruf_toplevel
2    Scenario: ortsbasierter_taxiruf_toplevel_testcase_24
3       Then I wait for the view with id "view_splash_screen" to ↙
            ↳ appear
4       Then I wait for 2 seconds
5       Then I wait for the view with id "view_map" to appear
6       Then I wait for the view with id "view_map" to appear
7       Then the location is 51.461001, 7.016339 with accuracy ↙
            ↳ 500.000000 1000 milliseconds ago
8       Then I wait for the view with id "view_dialog_manual_address" ↙
            ↳ to appear
9       When I enter text "Gerlingstrasse 16" into field with id "↙
```

```
        ↳ textfield_street"
10      When I enter text "45127 Essen" into field with id "↙
        ↳ textfield_city"
11      Then I hide the keyboard
12      When I press view with id "button_commit_address"
13      Then I wait for the view with id "view_order" to appear
14      When I press view with id "button_commit_order"
15      Then I wait for the view with id "view_order_info" to appear
16      When I press view with id "button_ok"
```

Die Simulation eines ungültigen Standorts erfolgt in Zeile 7 im Quellcodeabdruck. Diese Zeile instruiert die Testautomatisierung eine Standortinformation mit einem WGS84-Koordinatenpaar, einem Alter von einer Sekunde und einer Genauigkeit von 500 Metern zu simulierten. Gemäß dem Modell der Anwendung in Abbildung 6.54 in Abschnitt 6.5.1.1.1 erfüllt diese Standortinformation die Qualitätsanforderungen nicht (es wird eine Genauigkeit von 150 Metern gefordert) und der Anwender muss mit der manuellen Adresseingabe fortsetzen. In Quellcodefragment 6.19 wird das durch die Zeile 8 abgebildet.

Eine vollständige Übersicht der generierten Testfälle ist in Anhang B abgedruckt.

6.5.1.4 Bewertung

Gegenstand der Fallstudie Mobiler Taxiruf ist die Evaluierung des in dieser Dissertation vorgestellten Ansatzes zur Generierung von Tests aus einem UML-Aktivitätsdiagramm der Anwendung und deren automatisierte Ausführung unter dem Aspekt der Abdeckung des SUT gemäß dem Permutationssequenzüberdeckungskriteriums.

Die Testfallgenerierung hat aus dem in Abschnitt 6.5.1.1 diskutierten Modell der App Mobiler Taxiruf 30 Testfälle mit insgesamt 396 Testschritten erzeugt, die alle durch das Modell abgebildeten Ausführungspfade durch die App abdeckt. Die Anwendung implementiert einen einfachen Anwendungsfall, der zusätzlich auf einen einzelnen Aspekt der Anwendung reduziert ist. Aus diesem Grund ist die Menge der Testfälle aus wiederkehrenden Teil-Testfällen zusammengesetzt. Aufgrund des wenig komplex gestalteten Anwendungsfalls wäre es einem menschlichen Tester hier möglich gewesen, Tests für die einzelnen Aktivitäten des Modells manuell zu erstellen. Durch abschnittweise parallel verlaufende Kontrollflüsse (vgl. Abschnitt 6.5.1.1.4, manuelle Adresseingabe) und den Zyklus im Modell, der es in der Anwendung erlaubt, von der Aktivität Bestellung zur Standortbestimmung zurückzuspringen und den Alternativen bei der Auswertung der Standortinformation entsteht im Ergebnis dennoch eine Komplexität, die für menschliche Akteure nur noch schwer zu überblicken ist.

Durch diese Komplexität, gepaart mit dem Aufwand für ein manuelles Erstellen der Testfälle im Hinblick auf eine vollständige Testabdeckung, entstehen bereits durch die Generierung der Testfälle Vorteile durch die Verwendung der Automatisierungstechnologie. Einerseits ist sichergestellt, dass bei einer maschinellen Generierung von Testfällen gegenüber einer manuellen Erstellung Fehler durch menschlichen Irrtum ausgeschlossen werden können. Sollten Fehler in den generierten Testfällen auftreten, sind sie in einem defekten Modell, d. h. einem Modell, das den intendierten Anwendungsfall falsch abbildet, begründet. Die Erstellung des Modells erfolgt hingegen auf einer höheren Abstraktionsebene als die Erstellung von Testfällen, so dass Fehler zum einen unwahrscheinlicher und zum anderen einfacher zu identifizieren und zu korrigieren sind. Zum anderen ist der Aufwand zur Anpassung der Testfälle geringer, wenn sich das Modell der Anwendung verändert.

Sich verändernde Anforderungen sind ein gewöhnliches Ereignis in der Softwareentwicklung. Tritt eine Änderung der Anforderungen an eine Anwendung ein, so kann diese Änderung in einem Modell der Anwendung mit verhältnismäßig geringem Aufwand abgebildet werden. Durch die Verwendung der in dieser Dissertation vorgestellten Methode und Technologie zur Testfallgenerierung ist der Aufwand zur Anpassung von Testfällen als Konsequenz sich verändernder Anforderungen auf den Aufwand zur Anpassung des Modells und einem erneuten Generieren der Testfälle beschränkt. Dem gegenüber steht der große Aufwand einer manuellen Anpassung aller Testfälle.

Ein weiterer Vorteil ergibt sich aus der Automatisierung der Testausführung durch das angepasste Calabash-Framework (vgl. Abschnitt 5.5.3). Die App verwendet den Kontextparameter Standort. Folglich ist es erforderlich, bei der Testausführung Standortinformationen so zu simulieren, dass alle im Modell der App abgebildeten Fälle durch einen Testfall berücksichtigt werden. Durch die in dieser Dissertation vorgestellte Methode und Technologie zur Testautomatisierung für kontextsensitive Anwendungen ist es möglich, auch diesen Aspekt des Testens zu automatisieren. Die angepasste Implementierung des Calabash-Frameworks bietet über die Funktionalität der Standardimplementierung hinaus die Möglichkeit, Standortinformationen einschließlich Metainformationen wie Alter und Genauigkeit der Standortinformation zu simulieren. In der Fallstudie wird diese Funktion verwendet, um die App mit Standortinformationen unterschiedlicher Qualität zu testen. In dem in Quellcodefragment 6.19 abgebildeten Testfall wird beispielsweise eine ungültige Standortinformation simuliert, die dazu führt, dass der Anwender zur manuellen Adresseingabe weitergeleitet wird. Ebenfalls ist der Fall abgebildet, dass eine gültige Standortinformation simuliert wird (vgl. Anhang B).

In Gegenüberstellung mit einer manuellen Ausführung ergibt sich durch die Verwendung der in dieser Dissertation vorgestellten Technologie zur Testautomatisierung der Vorteil, Standortinformation mit unterschiedlichen Qualitätseigenschaften als Testdaten verwenden zu können. Bei einem manuellen Test auf einem Gerät stünde diese Option ansonsten nicht zur Verfügung. Beispielsweise zur Realisierung des Tests, für den Fall, dass der Anwender versucht ein Taxi an einen Ort außerhalb Deutschlands zu bestellen, hätte ein Tester tatsächlich einen Ort außerhalb Deutschlands aufsuchen müssen oder auf die Verwendung anderer Spezialsoftware zur Simulation von Standortinformationen auf dem Gerät zurückgreifen müssen. Zur Realisierung des Tests, bei dem der Anwender zur manuellen Adresseingabe geführt wird, weil die Standortinformation nicht die geforderte Qualität aufweist, steht im Jahr 2016 keine Technologie zur Verfügung, mit der dieser Test durchgeführt werden kann.

Mit der Fallstudie Mobiler Taxiruf konnte am Beispiel des Kontextparameters Standort gezeigt werden, dass die in dieser Dissertation vorgestellte Methode und Technologie zur Testautomatisierung für kontextsensitive, mobile Anwendungen in der Lage ist, den Testprozess von der Generierung von Testfällen bis zur Ausführung zu automatisieren und hierbei Kontextparameter mit variierenden Qualitätseigenschaften zu simulieren. Die Methode und Technologie repräsentiert daher einen relativen Vorteil gegenüber anderen im Jahr 2016 verfügbaren Technologien, die nicht in der Lage sind, die Generierung und automatisierte Testausführung in einem vergleichbaren Umfang zur realisieren.

6.5.2 Smart Thermometer

Gegenstand dieser Fallstudie ist die mobile Anwendung Smart Thermometer [81]. Sie unterscheidet sich von dem in Abschnitt 6.5.1 untersuchten akademischen Fallbeispiel insofern, als dass es sich um eine öffentlich im Google Play Store der Plattform Android verfügbare App handelt. Sie wurde als Fallstudie ausgewählt, weil sie einerseits die Kontextparameter Umgebungstemperatur, Luftfeuchte und Luftdruck verwendet und andererseits unabhängig von den Inhalten dieser Dissertation durch Dritte erstellt wurde.

Ziel der Durchführung dieser Fallstudie ist es den Nachweis zu erbringen, dass die in dieser Dissertation vorgestellte Methode und Technologie zur Generierung von Testfällen und deren automatische Ausführung geeignet ist, Tests für sensorbasierte Apps durchzuführen. Aus fachlicher Perspektive implementiert die App einen Anwendungsfall mit geringer Komplexität. Die Verwendung von Sensoren stellt allerdings die Anforderung an das Testen, in einer geeigneten Weise Testdaten zu erzeugen und der App während der Testdurchführung zuzuführen. In dieser Dissertation wurde mit dem Sensorsimulator im Rahmen des Werkzeugs zur Testautomatisierung eine technologische Grundlage geschaffen, bei der Testdurchführung Testdaten in die Sensoren des SUT einzuspeisen. Die Funktion des Sensorsimulators wird mit dieser Fallstudie überprüft.

Die App Smart Thermometer implementiert den einfachen Anwendungsfall, die Werte für Umgebungstemperatur, Luftdruck und Luftfeuchtigkeit in einem UI anzuzeigen. Der Anwender kann in einem Einstellungsmenü die Einheit der Darstellung (z. B. Celsius, Fahrenheit) wählen. Verfügt das Gerät, auf dem die Anwendung ausgeführt wird nicht über einen Sensor für die Umgebungstemperatur, kann die App nicht sinnvoll verwendet werden. In Abhängigkeit von der Temperatur färbt die App den Bildschirmhintergrund unterschiedlich ein.

Zur Durchführung der Evaluierung ist diese App deshalb besonders geeignet, weil als Untersuchungsgegenstand der Fallstudie die Simulation von Kontextparametern isoliert betrachtet werden kann. Die Evaluierung kann mit einem konkreten Fokus auf die Simulation von Sensordaten erfolgen.

6.5.2.1 Modell der Anwendung

Das Modell zur App Smart Thermometer reflektiert den einfachen fachlichen Anwendungsfall. Aufgrund der geringen Komplexität des Anwendungsfalls modelliert das in Abbildung 6.62 dargestellte Aktivitätsdiagramm anstelle eines vollständigen Modells der App eine ausgewählte *User Story* (engl. Anwendererzählung), in welcher der Anwender nach dem Öffnen der App zunächst zum Einstellungsdialog navigiert und die App dort so konfiguriert, dass als Einheit zur Anzeige der Temperatur Grad Celsius verwendet wird.

Auf den Kontrollfluss $f1$ wurde der Stereotyp *TestFlow* angewendet und im Objektmodell der Testdatenmodellierung ein Element vom Typ *Test::ClickAction* erstellt, das auf die Schaltfläche mit der Kennung „settings“ verweist. Auf die Aktivität „Setup Einheiten“ wurde der Stereotyp *TestStep* angewendet. Der Transformationsalgorithmus erkennt hieran, dass es sich bei dieser Aktivität um eine testrelevante Nutzerinteraktion handelt. Auch hier ist ein Element des *Test::ClickAction* modelliert, welches auf die Schaltfläche zur Auswahl der Einheit Celsius in den App-Einstellungen verweist. Anklicken des Zurück-Buttons führt zur Haupt-UI

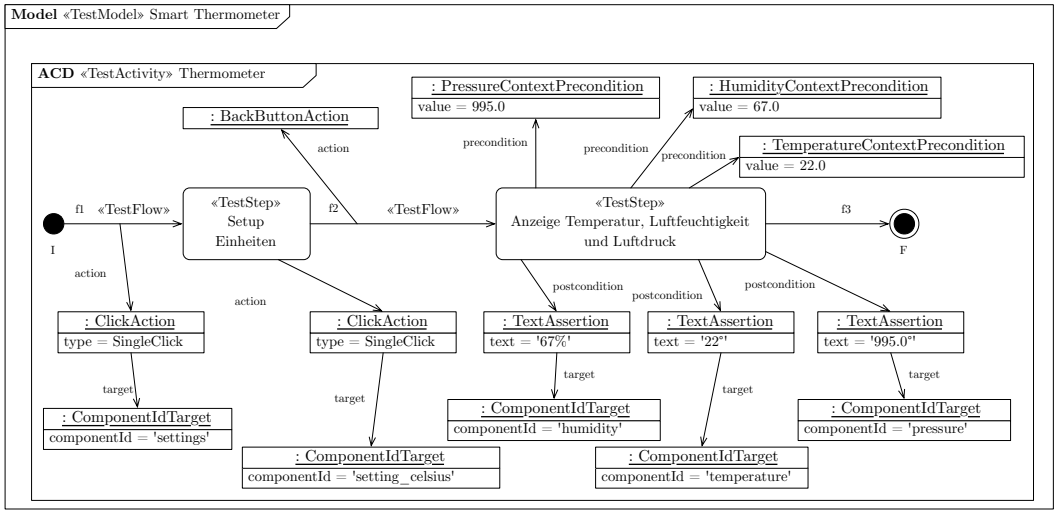


Abbildung 6.62: Aktivitätsdiagramm einer User Story der App Smart Thermometer [81]

der Anwendung, die durch die Aktivität Anzeige Temperatur, Luftfeuchtigkeit und Luftdruck realisiert wird. Diese Aktivität wurde durch Elemente des Typs *Test::ContextPrecondition* so mit Testdaten annotiert, dass im aus der Transformation resultierenden Testfallmodell für diese Aktivität entsprechende Vorbedingungen erzeugt werden. Bei der Testausführung verarbeitet die modifizierte Implementierung des Calabash-Frameworks diese Vorbedingungen in Instruktionen, die den Sensorsimulator (vgl. Abschnitt 5.5.3.1) in der angepassten Implementierung des Android-Betriebsystems so parametrisieren, dass ein Aufruf der Sensor-API genau die im Modell abgebildeten Werte erzeugt.

Der funktionale Anwendungstest wird durch die ebenfalls an der Aktivität Anzeige Temperatur, Luftfeuchtigkeit und Luftdruck modellierten Elemente vom Typ *Test::Postcondition*, konkret *Test::TextAssertion*, realisiert. Im Rahmen der Testdurchführung wird geprüft, ob die Anwendung die modellierten Nachbedingungen erfüllt. Konkret wird geprüft, ob die referenzierten Elemente genau den Text enthalten, der durch die Logik des Anwendungsfalls aus den Sensorwerten der Vorbedingungen bestimmt ist.

6.5.2.2 Generierte Tests

Das in Abschnitt 6.5.2.1 dargestellte Modell des Anwendungsfalls wurde mit der in Abschnitt 6.2 diskutierten prototypischen Implementierung der Testfallgenerierung zu Calabash-Tests transformiert. Durch den Transformationsalgorithmus wurde das in Quellcodefragment 6.20 dargestellte Calabash-Feature erzeugt.

Quellcodefragment 6.20: Calabash-Test der Anwendung Smart Thermometer

```
1 Feature: values_in_celsius_hpa
2
3    Scenario: values_in_celsius_hpa_testcase_0
4
5        # action
6        When I press view with id "settings"
7
8        # action
9        When I press view with id "setting_celsius"
```

```
10        When I press view with id "setting_hpa"
11
12        # action
13        When I go back
14        Then I wait for 1 seconds
15
16        # precondition
17        Given sensor relative humidity value is 67.000000
18        Given sensor ambient temperature value is 21.000000
19        Given sensor pressure value is 995.000000:0
20        # action
21        Then I wait for 1 seconds
22        # postcondition
23        Then I read "67%" in field "humidity"
24        Then I read "21°" in field "temperature"
25        Then I read "995,0" in field "pressure"
```

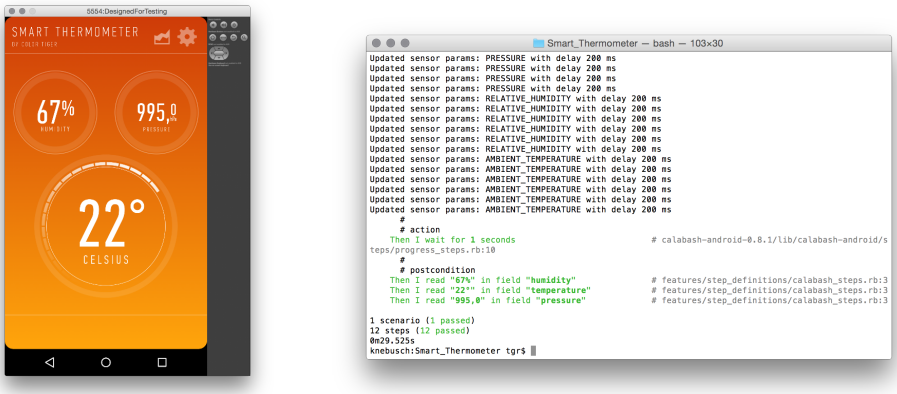
Das Calabash-Feature in Quellcodefragment 6.20 umfasst 12 Instruktionen, die jeweils die einzelnen Modellierungselemente im Aktivitätsdiagramm in Abbildung 6.62 abbilden. Zeile 6 repräsentiert das mit dem Kontrollfluss $f1$ assoziierte *Test::ClickAction*-Element. Die Zeilen 9-10 repräsentieren die mit der Aktivität Setup Einheiten assoziierten Interaktionen des Anwenders mit der App. Die Testschritte in den Zeilen 14 und 21 sind in Abbildung 6.62 aus Gründen der Übersichtlichkeit nicht dargestellt. Tatsächlich sind sie jedoch im Modell enthalten, um die graphische Animation des Übergangs zwischen dem Einstellungsmenü der App (Zeile 4, Rückkehr aus dem Menü) und die graphische Animation der Anzeige bei sich verändernden Messwerten der verwendeten Sensoren (Zeile 21) abzuwarten, um ein konsistentes UI während der Überprüfung der für die Aktivität Anzeige Temperatur, Luftfeuchtigkeit und Luftdruck im Modell hinterlegten Nachbedingungen zu gewährleisten.

Hervorzuheben sind die Zeilen 17-19 in Quellcodefragment 6.20. Diese Instruktionen werden durch die in dieser Dissertation erarbeiteten Anpassungen am Calabash-Framework und am Android-Betriebssystem bereitgestellt. Sie parametrisieren den Sensorsimulator so, dass dieser die im Testfallmodell hinterlegten Daten für die spezifizierten Sensoren generiert. Die Anwendung verwendet die regulären API-Funktionen des Android-SDK, um die Daten für Temperatur, Luftdruck und relative Luftfeuchte zu ermitteln. Durch den Einsatz des Sensorsimulators ist sichergestellt, dass zum Zeitpunkt des Abrufs der Sensordaten sichergestellt ist, dass genau die im Testfallmodell hinterlegten Daten anliegen.

Die Zeilen 23-25 überprüfen, ob die Anwendung als Reaktion auf die durch den Sensorsimulator eingespeisten Daten die im Testfallmodell spezifizierten Nachbedingungen erfüllt. Als Nachbedingungen der Aktivität Anzeige Temperatur, Luftfeuchtigkeit und Luftdruck sind im Modell drei Elemente des Typs *Test::TextAssertion* spezifiziert, die jeweils abbilden, dass die Anwendung sich nur dann anforderungskonform verhält, wenn für Temperatur, Luftdruck und relative Luftfeuchte jeweils die im Modell hinterlegten Werte angezeigt werden. Die Automatisierungstechnologie wertet das UI der Anwendung zur Laufzeit aus und vergleicht die tatsächlich angezeigten Werte mit den im Modell spezifizierten. Tritt eine Abweichung auf, schlägt der Test fehl. In Abbildung 6.63 ist ein Bildschirmabdruck der Testdurchführung dargestellt.

6.5.2.3 Bewertung

Der Testfall umfasst lediglich 12 Instruktionen. Allein durch die Komplexität des Testfalls scheint die Verwendung einer Technologie zur Testfallgenerierung mit anschließender Automa-



(a) Bildschirmabdruck des Android-Emulators während der Testdurchführung

(b) Bildschirmabdruck der Calabash-Konsole während der Testdurchführung

Abbildung 6.63: Bildschirmabdrucke des Android-Emulators und der Calabash-Konsole während der Ausführung des in Quellcodefragment 6.20 dargestellten Tests.

tisierung der Ausführung zunächst nicht gerechtfertigt. Tatsächlich steht in dieser konkreten Fallstudie der Aufwand zur Erstellung eines Modells der Anwendung bzw. eines Anwendungsfalls in einem ungünstigen Verhältnis zum Aufwand einer manuellen Testfallerstellung. Im Fokus der Fallstudie Smart Thermometer steht allerdings nicht die Betrachtung des Vorteils einer Generierung von Testfällen aus einem Modell gegenüber einer manuellen Testfallerstellung. Es soll vielmehr gezeigt werden, dass die hier untersuchte Methode und Technologie zur Testautomatisierung verwendet werden kann, während der Testausführung Testdaten für Sensorwerte reproduzierbar zu erzeugen und in das SUT einzuspeisen.

Auf mobilen Geräten existiert mit den in Abschnitt 2.5 dargestellten Technologien im Jahr 2016 keine Möglichkeit, dem SUT Werte für Temperatur, relative Luftfeuchte und Luftdruck vorzugeben. Ein Tester hätte dann nur die Option, die Anzeigen der App mit separaten Messinstrumenten (d. h. konventionelles analoges oder digitales Thermometer) zu vergleichen. Testen der App mit unterschiedlichen Werten, wäre sonst nur möglich, indem sich der Tester zusammen mit dem Testgerät zu Orten begibt deren Klimaeigenschaften bekannt sind (z. B. Kühlhaus, Sauna).

Wird hingegen ein Android-Emulator mit dem Standardbetriebssystem ohne die in Abschnitt 5.5.2 vorgestellte Modifikation zur Bereitstellung einer Testschnittstelle verwendet, ist Testen sensorbasierter Apps nur möglich, wenn auf der Shell-Konsole für jeden von einer App verwendeten Sensor manuell Werte vorgegeben werden. Einerseits stehen manuell auszuführende Schritte während des Testens einer Automatisierung des Build- und Deployment-Prozesses entgegen und andererseits ist es einem menschlichen Tester unmöglich, Sensordaten mit realistischen Eigenschaften (z. B. Rauschen, Frequenz) in das SUT einzuspeisen.

Mit dieser Fallstudie konnte gezeigt werden, dass der in dieser Dissertation vorgestellte Sensorsimulator seinen Bestimmungszweck erfüllt, Sensordaten in ein SUT unabhängig von den Umgebungsbedingungen einzuspeisen. Der in Abbildung 6.63 dargestellte Calabash-Test wurde sowohl auf dem Android-Emulator als auch auf Android-Geräten (konkret einem Nexus

5 und einem Nexus 10) erfolgreich ausgeführt.

Die in dieser Dissertation vorgestellte Technologie hat in der Fallstudie eine Automatisierung eines Tests ermöglicht, dessen Durchführung auf Android-Geräten mit reproduzierbaren Testdaten ansonsten unmöglich gewesen wäre und auch auf einem Geräteemulator nur mit hohem manuellen Aufwand und ohne Option einer Automatisierung hätte erfolgen können.

6.5.3 Kompass

Im Google Play Store ist eine Vielzahl Apps verfügbar, die einen Kompass implementieren. Grundfunktionalität ist die Anzeige der Orientierung des Geräts relativ zum magnetischen oder zum geographischen Nordpol. Technisch ist diese Funktionalität i. d. R. durch Verwendung des Beschleunigungssensors und des Magnetfeldsensors realisiert. Die in dieser Dissertation vorgestellte Methode und Technologie zur Testautomatisierung für mobile, kontextsensitive Anwendungen ist in der Lage, neben den anderen im Android-SDK definierten Sensoren auch Werte für diese beiden Sensoren zu simulieren.

Dass zur Berechnung von Gier-, Nick-, und Rollwinkel in der App tatsächlich keine anderen Sensoren als Beschleunigungssensor und Magnetfeldsensor verwendet wurden, konnte durch eine Auswertung des Protokolls des Sensorsimulators bei der Testausführung verifiziert werden. Dieser ist zu entnehmen, dass mit dem Start der App genau diese beiden Sensoren aktiviert wurden.

Gegenstand dieser Fallstudie ist die App Kompass Wasserwaage [235] des Herstellers LemonClip. Sie ist ein Vertreter der im Google Play Store verfügbaren Kompassanwendungen, zeichnet sich jedoch dadurch aus, dass sie die Kompassfunktion mit der Funktion einer Wasserwaage vereinigt. Für die Durchführung einer Fallstudie im Rahmen der Evaluierung der Methode und Technologie zur Testautomatisierung ist diese App interessant, weil sie beide Sensoren gemeinsam zur Realisierung der Funktionalität verwendet. Die Messwerte beider Sensoren werden mathematisch miteinander verknüpft, um Gier-, Nick- und Rollwinkel des Geräts zu berechnen. Die App wurde zum Zeitpunkt der Durchführung der Fallstudie ca. 26.000 Mal heruntergeladen und ist mit vier von fünf Sternen in der Bewertungsmetrik des Google Play Store verhältnismäßig positiv positioniert.

Ziel dieser Fallstudie ist es, am Praxisbeispiel zu zeigen, dass das in dieser Dissertation erstellte Automatisierungs-Framework verwendet werden kann, um einen Test für eine Anwendung zu automatisieren, die mehrere Sensoren zur Implementierung eines Anwendungsfalls, hier konkret die Orientierung des Geräts relativ zur Erdoberfläche, verwendet.

6.5.3.1 Modell der Anwendung

Die App Kompass Wasserwaage implementiert einen Anwendungsfall, der sich dem Anwender in einem einzelnen UI präsentiert. Neben dem Haupt-UI, in dem der gesamte Inhalt der App dargestellt wird, bietet die App ebenfalls eine Einstellungsansicht, in welcher der Anwender Einheiten der Anzeige und das Aktualisierungsintervall anpassen kann.

Inhalt dieser Fallstudie ist es zu überprüfen, ob der Sensorsimulator die an ihn gestellte Anforderung das SUT mit Testdaten zu versorgen, die vom SUT als originäre Sensordaten akzeptiert werden, erfüllt. Die Fallstudie ist deshalb auf das Haupt-UI der App beschränkt

Abbildung 6.64: Bildschirmabdruck der App Kompass Wasserwaage [235] im Google Play Store

und berücksichtig die Optionen im Einstellungsmenü der App nicht. Das in Abbildung 6.65 dargestellte Modell ist aus diesem Grund auf ein Aktivitätsdiagramm beschränkt, dass lediglich eine einzelne Aktivität modelliert. Zwar ist anzunehmen, dass durch den Entwickler der Anwendung ein komplexeres Modell erstellt worden wäre. Aus der Perspektive eines fachlichen Akzeptanztests ist das in Abbildung 6.65 abgebildete Modell jedoch ausreichend. Die in dieser Dissertation vorgestellte Methode und Technologie zur Testautomatisierung kann vom Grundsatz her als Werkzeug zur Durchführung von Black-Box-Tests verwendet werden, d. h. Kenntnisse der konkreten Implementierung der App sind nicht erforderlich.

In der Fallstudie wird das Werkzeug zur Testfallgenerierung und Testausführung jedoch im Grey-Box-Test-Betrieb verwendet. Da es sich um eine App aus dem Google Play Store eines kommerziellen Anbieters handelt, sind Details über die Implementierung unbekannt. Im Modell der App werden allerdings Detailkenntnisse des UI der App verwendet. Die von den *Test::TextAssertion*-Elementen referenzierten *Test::ComponentIdTarget*-Element spezifizieren Gestaltungselemente des UI. Deren interne Bezeichnung wurde mit dem Werkzeug Android Device Monitor [129] ermittelt. Das als Grundlage des Testautomatisierungswerkzeugs verwendete Calabash-Framework definiert zwar eine Assertion, die Textinhalte des UI anonym auswertet, d. h. ohne Kenntnis der internen Bezeichnung eines UI-Elements. Für den konkreten Test ist sie ungeeignet, weil auf der UI der Anwendung an mehreren Stellen identisch strukturierter Text angezeigt wird. Eine eindeutige wechselseitige Zuordnung von Testdaten und UI-Elementen ist daher unmöglich.

Modelliert ist eine einzelne Aktivität im Aktivitätsdiagramm, auf welche der Stereotyp *TestStep* angewendet wurde. Das Objektmodell der Testdaten spezifiziert als Vorbedingung ein Element des Typs *Context::DeviceOrientationPrecondition*. Es modelliert für diesen Testfall eine spezifische Orientierung des Gerätes relativ zur Erdoberfläche, konkret einen Gierwinkel von 42°, einen Rollwinkel von 2° und einen Nickwinkel von −20°. Geometrisch repräsentiert das eine Orientierung des Geräts, in welcher die Längsachse des Geräts einen 42°-Winkel Nordost vom magnetischen Nordpol der Erde bildet, das Gerät dem Anwender 20° aus der Waagerechten entgegen geneigt ist und zudem um 2° um die Gerätelängsachse gekippt ist.

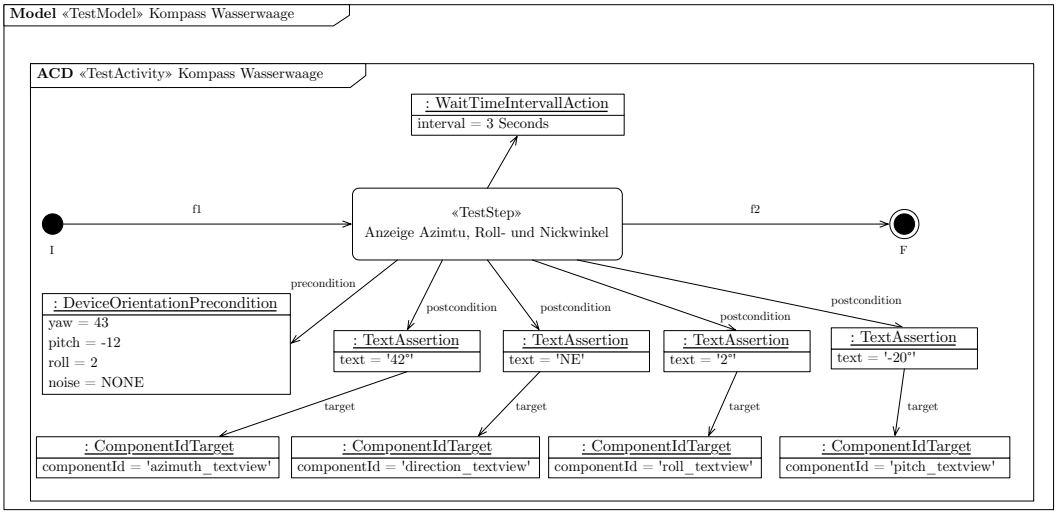


Abbildung 6.65: Aktivitätsdiagramm der App Kompass Wasserwaage [235]

Die Vorbedingung spezifiziert, dass bei der Simulation von Sensordaten kein Sensorrauschen generiert werden soll. Die App berechnet für jeden neuen Sensorwert des Beschleunigungssensors bzw. des Magnetfeldsensors erneut die Orientierung des Geräts und aktualisiert das UI. Da diese Sensoren in rascher Folge neue Messwerte produzieren, werden auf dem UI ständig wechselnde Werte angezeigt. Eine Auswertung ist dann nicht möglich, da der Sensorsimulator mit einem zu erzeugenden Mittelwert parametrisiert wird.

Die durch das Sensorrauschen verursachte häufige Aktualisierung des UI manifestiert sich in der App einerseits in der Animation der graphischen Kompassansicht und andererseits in Flimmern textuell angezeigter Werte. Hier hätte die UX der App durch Anwendung eines Tiefpassfilters auf die Sensorwerte verbessert werden können.

Wenn sich Sensormesswerte ändern, wird die graphische Darstellung des Kompass in der App animiert. Als während des Tests auszuführende Interaktion spezifiziert das Modell deshalb ein Element vom Typ *Test::WaitTimeIntervallAction*. Hierdurch wird die Zeit überbrückt, die von der App zur graphischen Animation des UI benötigt wird.

Als Nachbedingungen spezifiziert das Objektmodell vier Elemente des Typs *Test::TextAssertion*. Diese referenzieren spezifische Elemente des UI der App und prüfen, ob für Gier-, Roll- und Nickwinkel auf dem UI die korrekten (d. h. die in der Vorbedingung spezifizierten) Werte angezeigt werden.

6.5.3.2 Generierte Tests

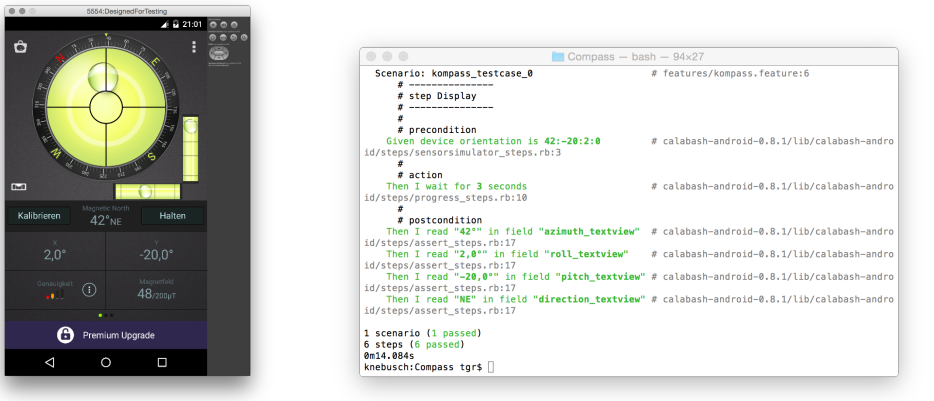
Auf das in Abschnitt 6.5.3.1 dargestellte Modell der App Kompass Wasserwaage wurde die Transformation Aktivitätsdiagramm zu Testfallmodell zu Calabash-Test angewendet. Hierdurch wurde das in Quellcodefragment 6.21 dargestellte Calabash-Feature erzeugt. Das Calabash-Feature in Quellcodefragment 6.21 umfasst sechs Instruktionen. In Zeile 13 wird die Vorbedingung des Tests spezifiziert. Bei der Ausführung wird hierdurch der Sensorsimulator instruiert, zu den Werten für Gier-, Nick- und Rollwinkel passende Werte für den Beschleunigungssensor und den Magnetfeldsensor zu generieren. Das mathematische Modell zur Berech-

nung dieser Werte wurde aus der Biblothek SensorSimulator™ von OpenIntents übernommen und die angepasste Calabash-Implementierung integriert.

Quellcodefragment 6.21: Calabash-Test der Anwendung Kompass Wasserwaage

```
1  Feature: kompass
2
3      # not defined
4
5
6      Scenario: kompass_testcase_0
7
8          # ---------------
9          # step Display
10         # ---------------
11         #
12         # precondition
13         Given device orientation is 42:-20:2:0
14         #
15         # action
16         Then I wait for 3 seconds
17         #
18         # postcondition
19         Then I read "42°" in field "azimuth_textview"
20         Then I read "2,0°" in field "roll_textview"
21         Then I read "-20,0°" in field "pitch_textview"
22         Then I read "NE" in field "direction_textview"
```

In den Zeilen 19-22 in Quellcodefragment 6.21 erfolgt die Auswertung der Nachbedingung durch Inspektion des UI der Anwendung. In Abbildung 6.66 sind Bildschirmabdrucke des Android-Emulators und der Calabash-Konsole während der Testausführung dargestellt.



(a) Bildschirmabdruck des Android-Emulators während der Testdurchführung

(b) Bildschirmabdruck der Calabash-Konsole während der Testdurchführung

Abbildung 6.66: Bildschirmabdrucke des Android-Emulators und der Calabash-Konsole während der Ausführung des in Quellcodefragment 6.21 dargestellten Tests

6.5.3.3 Bewertung

Aufgrund des trivialen Anwendungsfalls der in der Fallstudie untersuchten App besteht der aus dem Modell in Abbildung 6.65 in Abschnitt 6.5.3.1 generierte Test nur aus wenigen Instruktionen. Der Vorteil der automatischen Generierung des Tests aus einem Modell der Anwendung

bzw. eines Anwendungsfalls tritt hier hinter den Vorteilen der Automatisierung der Testausführung zurück. Dieser Test hätte in dieser Form mit geringem Aufwand manuell erstellt werden können. Ziel der Fallstudie war jedoch der Nachweis, dass die in dieser Dissertation vorgestellte Technologie zur Testautomatisierung mit Simulation von Kontextparametern, konkret Sensordaten, das Testen mobiler Apps durch Reduzierung manuellen Aufwands unterstützt. Erst durch die in dieser Dissertation vorgestellte Technologie zur Testautomatisierung ist es überhaupt möglich, die in der Fallstudie untersuchte App adäquat und automatisiert zu testen.

Manuelles Testen der untersuchten App auf einem Gerät wäre einem menschlichen Akteur nur unter Zuhilfenahme von Hilfsmitteln wie einem physikalischen Kompass, einer physikalischen Wasserwaage und einem Winkelmesser möglich. Selbst bei Verwendung dieser Hilfsmittel kann nicht gewährleistet werden, dass der Testkontext zu jeder Testausführung reproduziert werden kann. Die App verwendet mit dem Erdmagnetfeld eine physikalische Eigenschaft ihrer Umgebung, die von den menschlichen Sinnen nicht direkt wahrnehmbar ist. Bei einem manuellen Test müsste folglich geeignetes technisches Messgerät verwendet werden, um die Homogenität und Störungsfreiheit des Erdmagnetfelds im Umfeld des Geräts zum Zeitpunkt des Testens zu ermitteln und zu bewertet. Durch die Bereitstellung eines solchen Messgeräts entsteht ein finanzieller Aufwand, der dem Nutzen mit großer Wahrscheinlichkeit nicht angemessen gegenübersteht.

Manuelles Testen auf einem Gerät mit Simulation von Sensordaten ist hingegen mit keiner der im Rahmen dieser Dissertation untersuchten und im Jahr 2016 verfügbaren Technologie möglich. Ohne die in dieser Dissertation implementierte Modifikation des Android-Betriebssystems zur Bereitstellung einer Schnittstelle zur Einspeisung von Sensordaten ist es unmöglich, Testdaten zu reproduzieren.

Testen in einem Emulator ist ebenfalls nur mit Einschränkungen möglich. Zunächst verfügt der Android-Emulator nicht über physikalische Sensoren, so dass eine haptische Interaktion mit dem Emulator prinzipiell ausscheidet. Es besteht zwar vom Grundsatz her die Option, dem Emulator an einer Telnet-Konsole simulierte Sensordaten zuzuführen. Die in der Fallstudie untersuchte App verwendet jedoch zwei Sensoren gleichzeitig für die Berechnung einer fachlichen Entität. Der Tester müsste folglich für jeden der verwendeten Sensoren Werte bestimmen, die den Testfall (d. h. die Orientierung des Geräts) repräsentieren und an der Telnet-Konsole eingeben. Neben der Fehleranfälligkeit solchen Vorgehens ist es zur Integration in ein automatisiertes Build-System ebenfalls ungeeignet. Die Option, Sensordaten über eine Telnet-Konsole einem realen Android-Gerät zuzuführen, existiert nicht.

Die in dieser Dissertation vorgestellte Methode und Technologie zur Testautomatisierung eröffnet für solche kontextsensitiven Anwendungen wie der in dieser Fallstudie untersuchten somit erstmals die Möglichkeit, Anwendungstests mit Simulation von Sensordaten sowohl auf realen Android-Geräten als auch auf Emulatoren zu automatisieren.

6.6 Bewertung im Vergleich zu alternativen Technologien

In Abschnitt 2.5 wurde das technologische Umfeld im Bereich Testen mobiler Anwendungen diskutiert. Es wurde hierbei herausgestellt, dass keine der im Jahr 2016 etablierten Technolo-

gien zur Generierung von Testfällen oder zu automatisierten Testausführung in der Lage ist, fachliche Testfälle aus einem Modell der Anwendung zu generieren oder Kontextparameter in automatisierten Tests zu berücksichtigen.

In den Abschnitten 6.3, 6.4 und insbesondere in Abschnitt 6.5 konnte gezeigt werden, dass die in dieser Dissertation erarbeitete modellbasierte Methode und Technologie zur Testautomatisierung (1) geeignet ist, fachliche Tests aus UML-Aktivitätsdiagrammen zu generieren und (2) diese Tests automatisiert auszuführen und hierbei (3) Testdaten als Sensordaten in das SUT einzuspeisen. Über den objektiven Nachweis der Machbarkeit hinaus, d. h. der Testfallgenerierung aus Aktivitätsdiagrammen und dem reproduzierbaren Einspeisen von Testdaten als Sensordaten in ein SUT im Rahmen eines automatisierten Tests, konnte in den vorangegangenen Abschnitten gezeigt werden, dass der in dieser Dissertation erarbeitete Ansatz zur Testautomatisierung nützlich, kompatibel, erprobbar und beobbachtbar im Sinne Rogers Diffusionstheorie ist (vgl. Kapitel 6).

Durch die Verwendung der in dieser Dissertation erarbeiteten Methode und Technologie entsteht ein relativer Vorteil gegenüber existierenden Werkzeugen, weil sie die Automatisierung von Tests in einem Umfang ermöglicht, den keines der existierenden Werkzeuge unterstützt. Die Basistechnologie des Testens JUnit (vgl. Abschnitt 2.5.1.1) für Android-Apps gibt keinerlei Unterstützung bei der Erstellung von Tests und ist darüber hinaus nicht in der Lage, Kontextparameter zu simulieren und in das SUT einzuspeisen. Die Werkzeuge Robotium, Espresso, UIAutomator, Monkeyrunner und Appium (vgl. Abschnitte 2.5.2.1 bis 2.5.2.5) erweitern JUnit zwar in einigen Aspekten, sind aber ebenfalls nicht in der Lage, Kontextparameter in Tests abzubilden oder bei der Testfallerstellung zu unterstützen.

Der in dieser Dissertation vorgestellte Ansatz zur Testautomatisierung adressiert diese Problemstellungen hingegen durch die Bereitstellung einer Technologie zur Generierung von Testfällen aus Aktivitätsdiagrammen und deren automatisierte Ausführung unter Berücksichtigung von Kontextparametern. In Ergänzung zu JUnit-Tests stellt der hier vorgestellte Ansatz erstmals eine Technologie zur Verfügung, die es auch nicht-technischen Akteuren ermöglicht, Tests für mobile, kontextsensitive Anwendungen zu erstellen und durchzuführen. Mit existierenden Technologien ist die Durchführung solcher Tests Akteuren vorbehalten, die nicht nur die Werkzeuge des Android-SDK sicher beherrschen, sondern zudem auch sicheren Umgang mit mehreren Programmiersprachen und Test-Frameworks haben.

6.7 Zusammenfassung

In den vorangegangen Abschnitten 6.1 bis 6.6 wurde die in dieser Dissertation vorgestellte Methode und Technologie zur Automatisierung von Tests für mobile, kontextsensitive Anwendungen evaluiert. In Abschnitt 6.1 wurden zunächst die Methode und die Kriterien zur Evaluierung vorgestellt. Der Nachweis der Korrektheit der Testfallgenerierung im Sinne des in Abschnitt 5.3.2 definierten Testabdeckungskriteriums und die korrekte Interpretation der generierten Tests durch das Automatisierungswerkzeug (d. h. die Testdurchführung) wurden als konkrete Ziele der Evaluierung herausgestellt.

Die Grundlage der Evaluierung bildet die in Abschnitt 6.2 vorgestellte prototypische Implementierung der Werkzeuge zur Testfallgenerierung und Testautomatisierung. Kernbe-

standteile dieser Implementierung sind die Realisierungen der in Abschnitt 5.2.2.2 sowie in Abschnitt 5.2.3.2 vorgestellten Metamodelle für Kontext- und Testfallmodellierung mit UML-basierten Technologien. Der Testfallgenerator wurde als Plug-In für die Eclipse-Rich-Client-Platform umgesetzt, da so eine Kompatibilität zu existierenden Werkzeugen des MDA/MDSD hergestellt werden konnte. Die prototypische Implementierung der Testausführung erfolgte durch eine Modifikation des Android-Betriebssystems und des Calabash-Frameworks zur Bereitstellung von Schnittstellen zur Simulation von Sensordaten.

Eine akademische Validierung der Testfallgenerierung wird in Abschnitt 6.3 diskutiert. Basierend auf der korrekten Transformation rudimentärer Modellierungsmuster wurde gezeigt, dass der Transformationsalgorithmus zur Generierung von UML-Aktivitätsdiagrammen zu Testfallmodellen auch für komplexe Aktivitätsdiagramme im Sinne des Abdeckungskriteriums korrekte Ausführungspfade durch ein SUT erzeugt.

Der technologiespezifische Aspekt der Testautomatisierung für mobile, kontextsensitive Anwendungen wird in dieser Dissertation am Beispiel der Automatisierungstechnologie Calabash für die Plattform Android untersucht. Die Korrektheit der Transformation von Testfallmodellen zu Calabash-Tests ist Gegenstand des Abschnitts 6.4.

Die akademische Evaluierung der Testfallgenerierung wird durch die in Abschnitt 6.5 diskutierten Fallstudien ergänzt. Mit einem Fokus auf die Generierung von Testfällen wird in Abschnitt 6.5.1 das konsistent in dieser Dissertation verwendete Fallbeispiel Mobiler Taxiruf zur Evaluierung herangezogen (vgl. Abschnitt 6.5.1). Der Aspekt der Simulation von Kontextparametern wird in den Fallstudien Smart Thermometer (vgl. Abschnitt 6.5.2) sowie Kompass-App (vgl. Abschnitt 6.5.3) im Detail und unter Berücksichtigung verschiedener Gerätesensoren untersucht.

In Abschnitt 6.6 wird der in dieser Dissertation vorgestellte Ansatz zur Testautomatisierung im Vergleich zu existierenden Technologien bewertet. Es konnte gezeigt werden, dass durch die Verwendung der in dieser Dissertation vorgestellten Methode und Technologie zur Testautomatisierung ein relativer Vorteil gegenüber existierenden Methoden und Werkzeugen entsteht. Der hier vorgestellte Ansatz ermöglicht die Automatisierung von Tests, die mit existierenden Werkzeugen ansonsten nicht automatisierbar sind und auch manuell nicht alle Aspekten des jeweiligen kontextsensitiven Anwendungsfalls getestet werden können.

Kapitel 7

Fazit und Ausblick

In diesem Kapitel werden die Ergebnisse dieser Dissertation zusammengefasst und in Beziehung zur in Abschnitt 1.2.4 vorgestellten Forschungshypothese interpretiert. Ebenfalls im Hinblick auf die Forschungshypothese wird der Beitrag dieser Dissertation in Bezug zum Stand der Wissenschaft gesetzt und Voraussetzungen zum erfolgreichen Einsatz der hier vorgestellten Methode zur Testfallgenerierung und des Werkzeugs zur Testautomatisierung in der Praxis diskutiert. Es folgt eine kritische Betrachtung der Ergebnisse und ein Ausblick auf zukünftige Forschungs- und Entwicklungsfragestellungen, die sich während der Bearbeitung des Themas ergeben haben.

7.1 Beitrag der Arbeit

Bei der Erarbeitung der einzelnen Lösungselemente wurde eine Vielzahl von Themen der Softwaretechnik, von der Modellierung über die Entwicklung bis hin zum Testen von mobilen Anwendungen, in der Breite betrachtet. Die konkreten Themen Modellierung von Kontextparametern und Testdaten, die Generierung von Testfällen und deren automatisierte Ausführung mit Simulation von Sensordaten wurden mit dem Fokus auf eine Methode und eine Technologie zur Unterstützung der Testautomatisierung für mobile Anwendungen in der Tiefe betrachtet. In diesem Abschnitt wird der Beitrag dieser Dissertation reflektiert und in Bezug zum Stand von Wissenschaft und Technik gesetzt:

1. In einer umfassenden Untersuchung wissenschaftlicher Literatur wurden sowohl die Herausforderungen der Entwicklung und insb. des Testens kontextsensitiver Anwendungen in unterschiedlichen Einsatzszenarien als auch unterschiedliche methodische und technische Lösungsansätze zur Testfallgenerierung und Automatisierung betrachtet. Hierbei wurden ergänzend zu existierenden wissenschaftlichen Arbeiten konzeptuelle und technischen Problemstellungen des Testens mobiler Anwendungen unterschieden. In die erste Kategorie fallen Faktoren, die das Testen von Anwendungen methodisch erschweren oder unmöglich machen. Hierzu gehören beispielsweise Ressourcen, die durch Dritte kontrolliert werden oder die reproduzierbare Bereitstellung eines Ausführungskontext, der Faktoren umfasst, die mit vertretbarem Aufwand nicht kontrolliert werden können. Zu den technischen Herausforderungen gehört die Abwesenheit von geeigneten Technologien, die eine Analyse eines SUT zur Laufzeit, die simulierte Interaktion eines An-

wenders mit dem SUT oder die Simulation von Kontextparametern ermöglichen. Eine Unterscheidung zwischen konzeptuellen und technischen Problemstellungen ist relevant, weil sie jeweils unterschiedliche Herangehensweisen beim Entwurf von Lösungsansätzen erfordern. Es konnte gezeigt werden, dass im Jahr 2016 verfügbare Methoden und Technologien nicht geeignet sind, das Testen kontextsensitiver, mobiler Anwendungen adäquat zu unterstützen.

2. Es wurde ein UML-basiertes Metamodell zur Modellierung von Kontextparametern entworfen, das die Berücksichtigung dieser Parameter bei der Anwendungsmodellierung ermöglicht. Das Metamodell zur Kontextmodellierung ist Bestandteil des Metamodells zur Testfallmodellierung und ermöglicht die Integration von Kontextparametern in die Testautomatisierung. Zwar betrachten bereits eine Vielzahl wissenschaftlicher Arbeiten die Modellierung von Kontext im Umfeld mobiler Anwendung, ein vergleichbar umfassender Lösungsansatz, der ein Metamodell zur Kontextmodellierung im konkreten Anwendungszusammenhang des Testens betrachtet, war bislang jedoch nicht Forschungsgegenstand in der wissenschaftlichen Literatur. Das Metamodell zur Kontextmodellierung ermöglicht die Erstellung eines Objektmodells im Rahmen der Systemmodellierung, das testrelevante Kontextparameter abbildet und bei der Generierung von Testfällen zu Testdaten transformiert wird.

3. Es wurde ein UML-basiertes Metamodell zur Modellierung von Testfällen entworfen, das die Integration der Testfallmodellierung in die Systemmodellierung mit der UML ermöglicht. Das Metamodell ermöglicht es, einen semiformalen, syntaktischen und semantischen Zusammenhang zwischen dem Modell einer Anwendung und Testdaten herzustellen und bildet damit die Grundlage für die Generierung von Tests aus einem Systemmodell. Auch die Modellierung von Tests bzw. die Generierung von Tests aus Systemmodellen ist bereits vielfältig Gegenstand wissenschaftlicher Untersuchungen gewesen. Jedoch wurde das Thema bislang nicht ganzheitlich mit Bezug zu kontextsensitiven, mobilen Anwendungen betrachtet.

4. Es wurde ein UML-Profil entwickelt, welches das Metamodell der UML um Modellierungselemente erweitert, individuelle Aktivitäten und Kontrollflüsse als testrelevant zu kennzeichnen, so dass nur explizit als testrelevant gekennzeichnete Modellierungselemente von Aktivitätsdiagrammen in die resultierende Testfallmenge einfließen. Tester haben so die Möglichkeit, bei der Modellierung des SUT solche Aktivitäten zu selektieren, die eine Interaktion des Anwenders mit dem SUT oder Kontextsensitivität im SUT repräsentieren. Vergleichbare Ansätze zur modellbasierten Generierung von Testfälle bieten i. d. R. keine Mechanismen, Kontextparameter in Tests zu berücksichtigen oder von irrelevanten Elementen des Anwendungsmodells zu abstrahieren.

5. Es wurde ein Verfahren zur Generierung von Testfällen aus UML-Aktivitätsdiagrammen entwickelt. Es basiert auf der Transformation von UML-Aktivitätsdiagrammen zu Petri-Netzen, konkret Bedingungs-/Ereignisnetzen, deren Erreichbarkeitsgraph zur Berechnung von Ausführungspfaden durch das Aktivitätsdiagramm verwendet wird. Die Transformation gewährleistet, dass die generierten Testfälle das Modell vollständig über-

decken, d. h. jede modellierte Aktivität wird von mindestens einem Testfall ausgeführt. Weiterhin ist gewährleistet, dass jeder im Modell enthaltene Zyklus mindestens einmal durchlaufen wird und Aktionen auf nebenläufigen Kontrollflussabschnitten zu Permutationen sequenzialisiert werden, die zulässige Ausführungssequenzen aus der Perspektive eines menschlichen Anwenders repräsentieren, der sequenziell mit der Anwendung interagiert. In Abgrenzung zu vergleichbaren Ansätzen, die bereits früher in der wissenschaftlichen Literatur betrachtet wurden, stellt der hier vorgestellte Ansatz keine besonderen Anforderungen an die Wohlgeformtheit von Eingabemodellen bezüglich einer konsistenten, paarweisen Verwendung von Parallelisierung- und Synchronisierungsknoten.

6. Es wurde ein existierendes Testautomatisierungs-Framework um Funktionen erweitert, während der Testausführung Kontextparameter, insb. Sensordaten zu simulieren. Im Jahr 2016 existiert keine Testautomatisierungstechnologie, die es in einem vergleichbaren Umfang gestattet, den Betriebskontext einer Anwendung beim Testen zu kontrollieren.

7. Das Testautomatisierungswerkzeug wurde durch eine modifizierte Implementierung des Android-Betriebssystems ergänzt, in welches Schnittstellen zur Einspeisung von Testdaten integriert wurden. Das modifizierte Betriebssystem kann sowohl auf Geräten der Nexus-Familie als auch auf dem Android-Emulator installiert werden und ermöglicht es im Zusammenhang mit dem Testautomatisierungs-Framework erstmals, Kontextparameter zuverlässig bei der Testausführung zu reproduzieren, auf die mit alternativen Technologien bislang kein Einfluss genommen werden konnte.

Die Integration dieser Komponenten ermöglicht es, kontextsensitive, mobile Anwendungen automatisiert zu testen, die bislang nur manuell und ohne Reproduzierbarkeit von Kontextparametern getestet werden konnten.

7.2 Kritische Betrachtung und Konklusion

In dieser Dissertation wurde die Forschungshypthese untersucht, dass die Berücksichtigung von Kontextinformationen im Systementwurf durch Bereitstellung geeigneter Modellierungsmethoden und Werkzeuge dabei helfen kann, die Abhängigkeiten individueller Softwareartefakte von Kontextinformationen herauszustellen und deren Bedeutung für die Funktion einer entstehenden Anwendung zu verdeutlichen. Ein weiterer Teil der Forschungshypothese ist, dass die formale Berücksichtigung von Kontextinformationen in der Systemmodellierung die Grundlage zur automatisierten Generierung kontextsenstiver Testfälle bildet, die den Prozess der Testfallerstellung in einigen Aspekten von menschlicher Arbeitskraft entkoppeln kann. Hierdurch können menschliche Fehler bei der Testfallerstellung vermieden und die zur Testfallerstellung aufgewendete Arbeitszeit reduziert werden. Ein Werkzeug zur automatisierten Durchführung von Tests unter Berücksichtigung von Kontextinformationen (z. B. durch Injektion von Standort- oder Sensordaten in das SUT) hilft dabei, den Aufwand der Testdurchführung wesentlich zu reduzieren. Tester sind nicht länger gezwungen ihren Arbeitsplatz zur Durchführung von Tests zu verlassen.

Vor dem Hintergrund dieser Forschungshypothese werden in den folgenden Abschnitten die Ergebnisse dieser Dissertation im Hinblick auf Limitierungen kritisch betrachtet.

1. Durch die Metamodelle zur Kontext- bzw. Testfallmodellierung und das UML-Profil zur Testfallmodellierung wurde eine technische Grundlage geschaffen, Aspekte der Kontextsensitivität individueller App-Komponenten im Modell der Anwendung herauszustellen. Die in Kapitel 6 diskutierten akademischen Fallbeispiele und Fallstudien an real verfügbaren Apps aus dem Google Play Store konnten die Forschungshypothese in dieser Hinsicht bestätigen. Voraussetzung zur Anwendung dieser Modellierungswerkzeuge ist jedoch, dass zu einer Anwendung Modelle existieren. Agile Vorgehensmodelle erlangen insb. im Umfeld der App-Entwicklung zunehmend Popularität, da sie in kurzen Intervallen potenziell auslieferbare Softwareartefakte erzeugen, die ein Auftraggeber ohne nennenswerten Aufwand auf einem mobilen Gerät ausprobieren kann. Die kurzen Iterationen der Entwicklung und potenziell hohe Änderungshäufigkeit können den Verzicht auf die Modellierung der zu entwickelnden Anwendung begünstigen. In diesem Fall könnten stellvertretend ersatzweise für ein vollständiges Anwendungmodell Aktivitätsdiagramme des jeweiligen Anwendungsfalls erstellt werden. Eine vollständige Abdeckung der App wäre dann u. U. nicht mehr gewährleistet, wenn durch die Modelle einzelner Anwendungsfälle deren Gesamtzusammenhang nicht vollständig abgebildet wird. Es bliebe dann lediglich der Vorteil der Simulation von Kontextparametern erhalten.

2. Die Generierung von technologiespezifischen Testfällen aus dem Testfallmodell wurde in dieser Dissertation beispielhaft anhand des Testautomatisierungs-Frameworks Calabash umgesetzt. Einerseits ergibt sich hieraus der Vorteil, dass die generierten Testfälle durch die Anlehnung ihrer Syntax an die natürliche Sprache auch für Akteure ohne technischen Hintergrund unmittelbar nachvollziehbar sind. Die Attraktivität von Frameworks für individuelle Entwicklungsprojekte ist jedoch von vielen Faktoren abhängig. Inbesondere das Framework Calabash basiert auf einem komplexen Technologie-Stack, der aufgrund vielfältiger Abhängigkeiten zu Laufzeitumgebungen für Kompilate unterschiedliche Programmiersprachen in seiner Komplexität schwer beherrschbar ist. Die Implementierung der M2T-Transformation aus dem Testfallmodell zu technologiespezifischen Testfällen für ein alternatives Testautomatisierungs-Framework (vgl. Abschnitt 2.5) ist jedoch mit einem hohen Aufwand verbunden, der je nach Art und Umfang des zu realisierenden Entwicklungsprojekts nicht gerechtigfertigt sein könnte. Der in der Forschungshypothese postulierte Vorteil der Aufwandsreduktion durch Ablösung menschlicher Arbeitskraft könnte dann durch den Aufwand zur Implementierung der M2T-Transformation für ein alternatives Testautomatisierungs-Framework aufgezehrt werden.

3. Der Ansatz zur Generierung von Testfällen aus Anwendungsmodellen ist vom Grundsatz her plattformunabhängig. Die Implementierung des Testautomatisierungswerkzeugs und des Sensorsimulators ist in der prototypischen Implementierung jedoch auf die Plattform Android festgelegt. Der Markt mobiler Anwendungen wird im Jahr 2016 von wenigen Anbietern beherrscht. Der weitaus größte Marktanteil entfällt zwar auf die Plattform Android, andererseits ist Android im Jahr 2016 ebenfalls die einzige relevante Plattform, die in einem OSS-Modell zur Modifikation durch Dritte geöffnet ist. Andere mobile Plattformen können derzeit aufgrund ihres Closed-Source-Software-Modells nicht um Testschnittstellen erweitert werden. Hierdurch wird der in dieser Dissertation unter-

suchte Ansatz zur Simulation von Kontextparametern effektiv auf die Plattform Android limitiert.

Abseits dieser Limitierungen konnte durch die in Kapitel 6 durchgeführte Evaluierung die Wirksamkeit des vorgestellten Ansatzes zur modellbasierten Automatisierung von Tests für kontextsensitive, mobile Anwendungen im Sinne der zugrundegelegten Forschungshypothese gezeigt werden.

7.3 Ausblick

Im Rahmen der Entwicklung der Methode und des Werkzeugs zur Modelltransformation von UML-Aktivitätsdiagrammen zu Testfallmodellen und schließlich zu Calabash-Tests und deren automatisierter Ausführung, haben sich eine Reihe von Forschungs- und Entwicklungsfragestellungen ergeben, die den hier vorgestellten Ansatz zur Testautomatisierung in den Aspekten Testabdeckung, Testdatengenerierung und UX verbessern oder ergänzen können.

7.3.1 Erweiterungen der Methode

Die in Abschnitt 5.1 vorgestellte Methode zur modellbasierten Testautomatisierung hat die Generierung von Testfällen aus UML-Aktivitätsdiagrammen zum Ziel. Die Erstellung von Testdaten, durch welche kontextsensitive Aspekte eines SUT adäquat abgebildet werden, ist hingegen weiterhin eine manuell durchzuführende Aktivität. Im Rahmen der Diskussion von Anforderungen an ein Metamodell zur Kontextmodellierung in Abschnitt 5.2.2.1 wurde bereits die Option adressiert, Testdaten aus geeigneten Elementen des Metamodells zur Modellierung von Kontext- und Testfallmodellen zu interpolieren oder zu extrapolieren.

Von besonderem Interesse sind hier beispielsweise Randbereichstests, in denen eine kontextsensitive, mobile App gezielt in Wertebereichsgrenzen relevanter Kontextparameter mit Testdaten stimuliert wird. Eine effiziente Strategie, solche Testdaten zu identifizieren bzw. zu generieren ist ein Forschungsgebiet, das die Automatisierung von Testaktivitäten weiter fördern kann und somit zu einer weiteren Optimierung des Ressourceneinsatzes im Umfeld der Qualitätssicherung beitragen kann.

Das in Abschnitt 5.2.2.2 diskutierte Metamodell zur Kontextmodellierung definiert bereits einige Modellierungselemente (z. B. *Context::CircleStripRegion* oder *Context::PolygonalRegion*), die geeignet sind, durch adäquate Analysemethoden zur Interpolation bzw. Extrapolation von Testdaten verwendet zu werden. Das Element *Context::PolygonalRegion* könnte beispielsweise verwendet werden, sowohl gültige als auch ungültige WGS84-Koordinatenpaare für einen Testfall entlang der polygonalen Begrenzung einer geographischen Region zu berechnen, mit denen eine Menge fachlicher Tests parametrisiert werden können.

Ein weiteres Forschungsfeld, das in dieser Dissertation nicht betrachtet wurde, aber im Umfeld kontextsensitiver Anwendungen relevant ist, ist die Generierung realistischer Testdaten für Standort- oder Sensordaten. Die Simulation von Sensordaten, die ein natürliches Rauschen der Sensormesswerte nachbilden, wurde in Abschnitt 5.5.3.1 adressiert, weil es für die Simulation von Sensordaten essentiell ist. Die Abweichung simulierter Werte vom Mittelwert ist gering und aufgrund der Frequenz der Aktualisierung bei der Implementierung

geeigneter Gegenmaßnahmen (z. B. Tiefpassfilter) kaum geeignet, die Implementierung eines Anwendungfalls grundsätzlich zu korrumpieren.

Standortdaten hingegen werden vom GPS-Modul niederfrequent im Vergleich zu Sensordaten an eine App geliefert, unterliegen aber ebenfalls einem Messfehler. Die Verwendung eines Tiefpassfilters kann hier zu unerwünschten Seiteneffekten führen, die gegenüber einer ungefilterten Verwendung keinen Vorteil darstellt. In Abbildung 7.1 ist beispielsweise die Abweichung einer via GPS gemessenen Route (grün) von der tatsächlich zurückgelegten Route (schwarz) dargestellt.

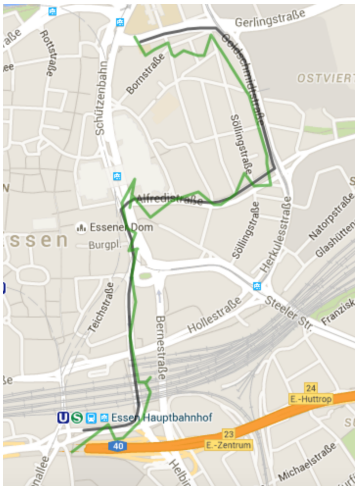


Abbildung 7.1: Abweichung einer via GPS gemessenen Route (grün) von der tatsächlich zurückgelegten Route (schwarz) [266]

Von einer App wird gefordert, solche Abweichungen durch geeignete Algorithmen zu kompensieren (im konkreten Fall in Abbildung 7.1 beispielsweise die *Minimum Distance Threshold Method* [119] zur Berechnung der Länge einer Wegstrecke). Solche Algorithmen zu testen erfordert die Erstellung von Testdaten, die den Messfehler eines GPS-Moduls adäquat approximieren. Das Problem der Testdatenerstellung wurde zwar in einer im Rahmen dieser Dissertation betreuten Bachelorarbeit von Ole Meyer [266] adressiert, ist darüber hinaus aber weitgehend unerforscht und repräsentiert daher einen Anschlusspunkt für zukünftige Forschungen.

Die Methode zur Testfallgenerierung ist in der hier vorgestellten Variante zwar so ausgelegt, dass Zyklen im Kontrollfluss eines SUT bei der Testausführung berücksichtigt werden. Das Metamodell zur Testfallmodellierung (vgl. Abschnitt 5.2.3.2) erlaubt es in der hier vorgestellten Form jedoch nur, pro Kontrollfluss bzw. Aktivität einen einzelnen Satz kontextsensitive Testdaten zu spezifizieren. Dieser darf zwar eine beliebige Anzahl Vor- bzw. Nachbedingungen und Aktionen spezifizieren. Bei mehrfacher Ausführung einer Aktivität werden jedoch auch bei einer mehrfachen Ausführung einer Aktion in der hier erstellten Implementierung stets dieselben Testdaten verwendet.

Während die Generierung von Testfällen zwar garantiert, dass alle relevanten Aktivitäten und Kontrollflüsse in Tests abgebildet werden, ist nicht gewährleistet, dass eine adäquate Testabdeckung auch im Hinblick auf die Testdaten erreicht wird. In zukünftigen Forschungsarbeiten wäre hier zu betrachten, wie Metamodelle zur Kontext- bzw. Testfallmodellierung gestaltet sein müssen, um unterschiedliche Testdaten für jede Testausführung abzubilden.

Beispielsweise wäre es sinnvoll, dass eine Aktivität, die aufgrund eines Zyklus mehrfach ausgeführt wird, bei jedem Durchlauf mit unterschiedlichen Testdaten parametrisiert wird. Dieses Problem ist insb. für verschachtelte Zyklen nicht trivial zu lösen, weil für einen konkreten Test einer Aktivität relevant sein kann, im Kontext welchen Zyklusdurchlaufs sie ausgeführt wird.

7.3.2 Erweiterungen des Werkzeugs

Die in dieser Dissertation entwickelte prototypische Implementierung des Algorithmus zur Testfallgenerierung ist als Eclipse-Plug-In basierend auf dem EMF ausgeführt. Sie akzeptiert UML-Modelle im XMI-Format als Eingabe. Die Erstellung der UML-Aktivitätsdiagramme kann zwar in jedem beliebigen UML-Editor erfolgen, der in der Lage ist, das Modell der Anwendung und insb. das Objektmodell der Testdaten im XMI-Format zu exportieren. Das in Abschnitt 5.2.4 vorgestellte UML-Profil zur Testfallmodellierung stellt allerdings Modellierungselemente bereit, für die in Standardeditoren keine graphischen Syntaxelemente existieren. Die Modellierung des Objektmodells der Testdaten kann also nur in einem rudimentären, generischen Editor wie dem des EMF erfolgen (vgl. Abbildung 6.3, Abschnitt 6.2.2).

Als wesentliche Hürde zur Akzeptanz einer Innovation in einer Zielgruppe identifiziert Rogers [301] die Kompatibilität zu bestehenden Konzepten, Werkzeugen und Denkwelten innerhalb der Zielgruppe. Durch Implementierung eines graphischen Editors, der Syntaxelemente für die Modellierungselemente des Metamodells zur Testfallmodellierung (vgl. Abschnitt 5.2.3.2) und des Metamodells zur Kontextmodellierung (vgl. Abschnitt 5.2.2.2) bereitstellt, beispielsweise ebenfalls als Eclipse-Plug-In unter Verwendung des *Eclipse Graphical Modelling Framework* (GMF), kann die Kompatibilität mit existierenden Werkzeugen erhöht werden. Hierdurch kann die Akzeptanz des vorgestellten Ansatzes im Umfeld des MDSD und insb. des MDT erhöht werden. Die Modellierung von Testdaten könnte dann nahtlos in die Modellierung des Aktivitätsdiagramms der Anwendung integriert werden.

Während das in Abschnitt 5.2.2.2 vorgestellte Metamodell zur Kontextmodellierung bereits Entitäten zur Modellierung von Eigenschaften des Funknetzwerks berücksichtigt (vgl. Abschnitt 5.2.2.2.5), ist deren Simulation nicht Bestandteil der in Abschnitt 5.5.2 vorgestellten und im Rahmen dieser Dissertation erstellten prototypischen Implementierung der Anpassung des Android-OS. Zur Verwendung von Eigenschaften des verfügbaren Funknetzwerks als Kontextparameter des Testens kann analog zu der Schnittstelle zur Einspeisung von Sensordaten in das SUT eine Schnittstelle im Android-Betriebssystem implementiert werden.

Das Unternehmen OpenSignal Inc. [271] vermisst in einem Crowdsourcing-Ansatz weltweit die Abdeckung und die Qualität mobiler Funknetzwerke in Relation zu einem wählbaren ISP. OpenSignal Inc. bietet eine Programmierschnittstelle an, über welche Informationen zur ISP-spezifischen Netzwerkabdeckung zu einem Standort abgerufen werden können. Es bietet sich eine Automatisierung der Simulation von Netzwerkparametern beim Testen ortsbasierter Anwendungen an, bei der neben dem Standort zugleich Qualitätseigenschaften der Netzwerkverbindung simuliert werden könnten. Hierzu müssten Netzwerkeigenschaften aus dem Datenbestand von OpenSignal Inc. abgerufen und in die Kontextsimulation eingespeist werden.

Die im Rahmen dieser Dissertation erstellte prototypische Implementierung der Testautomatisierung ist als Kommandozeilenwerkzeug ausgeführt. Eine Integration in CI-Umgebungen wie beispielsweise Jenkins [215] kann dazu beitragen, den Automatisierungsprozess der Bereit-

stellung von Software von der IDE bis zum App Store um modellbasierte Testautomatisierung mit Simulation von Kontextparametern zu ergänzen. Manuelle Aktivitäten könnten auf diese Weise auf die Erstellung eines Modells des SUT und dessen Anreicherung mit Testdaten reduziert werden. Aufwändige manuelle Testverfahren, die mit Technologien des Jahres 2016 zur Bewertung der Spezifikationskonformität einer kontextsensitiven Anwendung notwendig sind, könnten dann entfallen.

Die in dieser Dissertation erarbeitete Modifikation des Android-Betriebssystems hat gezeigt, dass das Paradigma „Designed for Testability“ angewendet auf das Betriebssystem anstelle einer konkreten Anwendung Optionen eröffnet, Black-Box-Tests unter Verwendung von Testdaten durchzuführen, die anderenfalls nicht in ein SUT eingespeist werden könnten. Diese Technologie eröffnet zugleich die Möglichkeit, auch Testdaten abseits von Kontextparametern in das SUT einzuspeisen. Rudimentär wurde dieser Ansatz bereits für Sprachsteuerung in Android-Apps umgesetzt (vgl. Griebe et al. [157]). Zukünftige Forschungsarbeiten könnten ähnliche Ansätze untersuchen, um weitere Interaktionsmodi mit mobilen Anwendungen, die hohe Anforderungen an das Testen stellen, beispielsweise Multi-Touch-Gesten, für Automatisierungstechnologien zu erschließen.

Anhang A

Calabash Steps für kontextsensitive Tests

Im Rahmen dieser Dissertation wurde ein Testautomatisierungswerkzeug implementiert, dass es erlaubt Tests für mobile, kontextsensitive Anwendungen für die Plattform Android durchzuführen. Das Werkzeug setzt sich aus Komponenten zusammen: einer um Testschnittstellen erweiterten Implementierung des Android-Betriebssystems (vgl. Abschnitt 5.5.2) und einer angepassten Implementierung des Calabash-Frameworks (vgl. Abschnitt 5.5.3). Neben einer Komponente zur Simulation von Sensordaten wurden Calabash-Step-Defintions erstellt, die es ermöglichen, in Calabash-Features Kontextparameter zu spezifizieren.

In den folgenden Abschnitten werden diese Calabash-Features dokumentiert. Dargestellt ist jeweils die Step-Definition im Syntax der DSL Gherkin. Eingabewerte sind jeweils durch reguläre Ausdrücke codiert.

Quellcodefragment A.1: Calabash-Step-Definition zur Spezifikation einer Uhrzeit im Testfall

```
Given /^Time is ([+]?\d+):([+]?\d+):([+]?\d+)$/
```

Quellcodefragment A.1 ermöglicht die Spezifikation der Angabe einer Uhrzeit in einem Testfall. Das gegenwärtig im Zielsystem gesetzte Datum bleibt gültig. Der Step korrespondiert mit dem Kontextparameter Zeit (vgl. Abschnitt 3.1.2.1.2).

Quellcodefragment A.2: Calabash-Step-Definition zur Spezifikation einer Datumsangabe im Testfall

```
Given /^Date is ([+]?\d+):([+]?\d+):([+]?\d+)$/
```

Quellcodefragment A.2 ermöglicht die Spezifikation einer Datumsanbgabe in einem Testfall. Die gegenwärtig im Zielsystem gesetzte Uhrzeit bleibt gültig. Der Step korrespondiert mit dem Kontextparameter Zeit (vgl. Abschnitt 3.1.2.1.2).

Quellcodefragment A.3: Calabash-Step-Definition zur Spezifikation einer Datumsangabe mit Uhrzeit im Testfall

```
Given /^Datetime is ([+]?\d+):([+]?\d+):([+]?\d+):([+]?\d+):([+]?\d
    ↪ +):([+]?\d+)$/
```

Quellcodefragment A.3 kombiniert die Steps in Quellcodefragment A.1 und Quellcodefragment A.2.

Quellcodefragment A.4: Calabash-Step-Definition zur Spezifikation von Roll-, Nick und Gierwinkel im Testfall

```
Given /^device orientation is ([-+]?\d+):([-+]?\d+):([-+]?\d+):(\d+)↙
    ↳ $/
```

Quellcodefragment A.4 ermöglicht die Spezifikation von Roll-, Nick und Gierwinkel des Geräts in einem Testfall. Der vierte Parameter erlaubt die Simulation von Sensorrauschen (Wert 1) oder eines Rauschfreien Signals (Wert 0).

Quellcodefragment A.5: Calabash-Step-Definition zur Spezifikation diskreter Geräteorientierung

```
Given /^device orientation is landscape right$/
Given /^device orientation is landscape left$/
Given /^device orientation is upside
Given /^device orientation is portrait$/
```

Quellcodefragment A.4 ermöglicht die Spezifikation diskreter Geräteorientierungen in einem Testfall.

Quellcodefragment A.6: Calabash-Step-Definition zur Spezifikation von Sensordaten

```
Given /^sensor accelerometer values are ([-+]?\d+\.\d+):([-+]?\d+\.\↙
    ↳ d+):([-+]?\d+\.\d+):(\d+)$/
Given /^sensor geomagnetic field values are ([-+]?\d+\.\d+):([-+]?\d↙
    ↳ +\.\d+):([-+]?\d+\.\d+):(\d+)$/
Given /^sensor orientation values are ([-+]?\d+\.\d+):([-+]?\d+\.\d↙
    ↳ +):([-+]?\d+\.\d+):(\d+)$/
Given /^sensor gyroscope values are ([-+]?\d+\.\d+):([-+]?\d+\.\d+)↙
    ↳ :([-+]?\d+\.\d+):(\d+)$/
Given /^sensor pressure value is ([+]?\d+\.\d+):(\d+)$/
Given /^sensor gravity values are ([-+]?\d+\.\d+):([-+]?\d+\.\d+)↙
    ↳ :([-+]?\d+\.\d+):(\d+)$/
Given /^sensor linear accelerometer values are ([-+]?\d+\.\d+)↙
    ↳ :([-+]?\d+\.\d+):([-+]?\d+\.\d+):(\d+)$/
Given /^sensor rotation vector values are ([-+]?\d+\.\d+):([-+]?\d↙
    ↳ +\.\d+):([-+]?\d+\.\d+):(\d+)$/
Given /^sensor magnetic field uncalibrated values are ([-+]?\d+\.\d↙
    ↳ +):([-+]?\d+\.\d+):([-+]?\d+\.\d+):(\d+)$/
Given /^sensor game rotation vector values are ([-+]?\d+\.\d+)↙
    ↳ :([-+]?\d+\.\d+):([-+]?\d+\.\d+):(\d+)$/
Given /^sensor gyroscope uncalibrated values are ([-+]?\d+\.\d+)↙
    ↳ :([-+]?\d+\.\d+):([-+]?\d+\.\d+):(\d+)$/
Given /^sensor geomagnetic rotation vector values are ([-+]?\d+\.\d↙
    ↳ +):([-+]?\d+\.\d+):([-+]?\d+\.\d+):(\d+)$/
Given /^sensor accelerometer values are ([-+]?\d+\.\d+):([-+]?\d+\.\↙
    ↳ d+):([-+]?\d+\.\d+):(\d+):(\d+)$/
Given /^sensor geomagnetic field values are ([-+]?\d+\.\d+):([-+]?\d↙
    ↳ +\.\d+):([-+]?\d+\.\d+):(\d+):(\d+)$/
Given /^sensor orientation values are ([-+]?\d+\.\d+):([-+]?\d+\.\d↙
    ↳ +):([-+]?\d+\.\d+):(\d+):(\d+)$/
Given /^sensor gyroscope values are ([-+]?\d+\.\d+):([-+]?\d+\.\d+)↙
    ↳ :([-+]?\d+\.\d+):(\d+):(\d+)$/
Given /^sensor pressure value is ([+]?\d+\.\d+):(\d+):(\d+)$/
Given /^sensor gravity values are ([-+]?\d+\.\d+):([-+]?\d+\.\d+)↙
    ↳ :([-+]?\d+\.\d+):(\d+)$/
Given /^sensor linear accelerometer values are ([-+]?\d+\.\d+)↙
    ↳ :([-+]?\d+\.\d+):([-+]?\d+\.\d+):(\d+)$/
Given /^sensor rotation vector values are ([-+]?\d+\.\d+):([-+]?\d↙
    ↳ +\.\d+):([-+]?\d+\.\d+):(\d+)$/
Given /^sensor magnetic field uncalibrated values are ([-+]?\d+\.\d↙
    ↳ +):([-+]?\d+\.\d+):([-+]?\d+\.\d+):(\d+)$/
Given /^sensor game rotation vector values are ([-+]?\d+\.\d+)↙
    ↳ :([-+]?\d+\.\d+):([-+]?\d+\.\d+):(\d+)$/
Given /^sensor gyroscope uncalibrated values are ([-+]?\d+\.\d+)↙
    ↳ :([-+]?\d+\.\d+):([-+]?\d+\.\d+):(\d+)$/
```

```
Given /^sensor geomagnetic rotation vector values are ([-+]?\d+\.\d↙
    ↳ +):([-+]?\d+\.\d+):([-+]?\d+\.\d+):(\d+)$/
Given /^sensor light value is ([+]?\d+\.\d+)$/
Given /^sensor proximity value is ([+]?\d+\.\d+)$/
Given /^sensor relative humidity value is ([+]?\d+\.\d+)$/
Given /^sensor ambient temperature value is ([+]?\d+\.\d+)$/
```

Quellcodefragment A.6 ermöglicht die Spezifikation von Daten in Testfällen für solche Sensoren, die einen kontinuierlichen Datenstrom erzeugen. Für jeden Sensor kann künstliches Rauschen eingestellt werden.

Quellcodefragment A.7: Calabash-Step-Definition zur Spezifikation von Sensordaten für besondere Systemereignisse

```
Given /^sensor step counter value is ([+]?\d+)$/
Given /^sensor significant motion is triggered$/
Given /^sensor step detector is triggered$/
Given /^sensor tilt detector is triggered$/
Given /^sensor wake gesture is triggered$/
Given /^sensor glance gesture is triggered$/
Given /^sensor pick up gesture is triggered$/
```

Quellcodefragment A.7 ermöglicht die Spezifikation von Daten in Testfällen für solche Sensoren, die besondere Systemereignisse repräsentieren, wie etwa Aufheben des Geräts oder Aufwecken aus dem Ruhezustand. Der Schrittzähler des Geräts repräsentiert einen Sonderfall, da zwar kontinuierlich den Beschleunigungssensor auswerte um die zurückgelegten Laufschritte des Anwenders zu zählen, diesen Wert aber nur sporadisch an registrierte Anwendungen ausliefert.

Quellcodefragment A.8: Calabash-Step-Definition zur Spezifikation aufgezeichneter Daten im C'n'R-Verfahren

```
Given /^device receives sensorevent sequence from "([^\"]*)"$/
```

Quellcodefragment A.8 die Spezifikation des Pfades zu einer Datei, die aufgezeichnete Sensordaten enthält, um diese im C'n'R-Verfahren in das SUT einzuspeisen.

Quellcodefragment A.9: Calabash-Step-Definition zum Verbergen des Soft-Keyboards

```
Given /^I hide the keyboard$/
```

Quellcodefragment A.9 spezifiziert eine Step-Definition zum verbergen des Soft-Keyboards.

Quellcodefragment A.10: Calabash-Step-Definition zur Spezifikation einer Standortinformation mit Metainformationen

```
Given /^the location is ([-+]?[0-9]*\.?[0-9]+), ↙
    ↳ ([-+]?[0-9]*\.?[0-9]+) with accuracy ([-+]?[0-9]*\.?[0-9]+) (\↙
    ↳ d+) milliseconds ago$/
```

Quellcodefragment A.10 spezifiziert eine Step-Definition zur Simulation einer Standortangabe mit einer konkreten Präzision und einem Alter.

Anhang B

Testfälle Mobiler Taxiruf

Dargestellt in Quellcodefragment B.1 sind die aus der Fallstudie Mobiler Taxiruf generierten Calabash-Testfälle.

Quellcodefragment B.1: Calabash-Test der Anwendung App Mobiler Taxiruf aus der in Abschnitt 6.5.1 diskutierten Fallstudie

```
Feature: ortsbasierter_taxiruf_toplevel

   Scenario: ortsbasierter_taxiruf_toplevel_testcase_0
      Then I wait for the view with id "view_splash_screen" to ↙
          ↳ appear
      Then I wait for 2 seconds
      Then I wait for the view with id "view_map" to appear
      Then I wait for the view with id "view_map" to appear
      When I press view with id "button_cancel"

   Scenario: ortsbasierter_taxiruf_toplevel_testcase_1
      Then I wait for the view with id "view_splash_screen" to ↙
          ↳ appear
      Then I wait for 2 seconds
      Then I wait for the view with id "view_map" to appear
      Then I wait for the view with id "view_map" to appear
      Then the location is 21.461001, 7.016339 with accuracy ↙
          ↳ 50.000000 1000 milliseconds ago
      Then I wait for the view with id "view_dialog_error" to appear
      Then I read "Der ermittelte Standort liegt nicht innerhalb ↙
          ↳ Deutschlands." in field "textview_error"
      When I press view with id "button_ok"

   Scenario: ortsbasierter_taxiruf_toplevel_testcase_2
      Then I wait for the view with id "view_splash_screen" to ↙
          ↳ appear
      Then I wait for 2 seconds
      Then I wait for the view with id "view_map" to appear
      Then I wait for the view with id "view_map" to appear
      Then the location is 51.461001, 7.016339 with accuracy ↙
          ↳ 50.000000 1000 milliseconds ago
      When I press view with id "button_commit_address"
      Then I wait for the view with id "view_order" to appear
      When I go back
      Then I wait for the view with id "view_map" to appear
      When I press view with id "button_cancel"

   Scenario: ortsbasierter_taxiruf_toplevel_testcase_3
      Then I wait for the view with id "view_splash_screen" to ↙
          ↳ appear
      Then I wait for 2 seconds
      Then I wait for the view with id "view_map" to appear
      Then I wait for the view with id "view_map" to appear
      Then the location is 51.461001, 7.016339 with accuracy ↙
          ↳ 50.000000 1000 milliseconds ago
      When I press view with id "button_commit_address"
      Then I wait for the view with id "view_order" to appear
      When I press view with id "button_cancel"
```

```
Scenario: ortsbasierter_taxiruf_toplevel_testcase_4
   Then I wait for the view with id "view_splash_screen" to ↙
      ↳ appear
   Then I wait for 2 seconds
   Then I wait for the view with id "view_map" to appear
   Then I wait for the view with id "view_map" to appear
   Then the location is 51.461001, 7.016339 with accuracy ↙
      ↳ 50.000000 1000 milliseconds ago
   When I press view with id "button_commit_address"
   Then I wait for the view with id "view_order" to appear
   When I go back
   Then I wait for the view with id "view_map" to appear
   Then the location is 21.461001, 7.016339 with accuracy ↙
      ↳ 50.000000 1000 milliseconds ago
   Then I wait for the view with id "view_dialog_error" to appear
   Then I read "Der ermittelte Standort liegt nicht innerhalb ↙
      ↳ Deutschlands." in field "textview_error"
   When I press view with id "button_ok"

Scenario: ortsbasierter_taxiruf_toplevel_testcase_5
   Then I wait for the view with id "view_splash_screen" to ↙
      ↳ appear
   Then I wait for 2 seconds
   Then I wait for the view with id "view_map" to appear
   Then I wait for the view with id "view_map" to appear
   Then the location is 51.461001, 7.016339 with accuracy ↙
      ↳ 500.000000 1000 milliseconds ago
   Then I wait for the view with id "view_dialog_manual_address" ↙
      ↳ to appear
   When I enter text "Gerlingstrasse 16" into field with id "↙
      ↳ textfield_street"
   When I enter text "45127 Essen" into field with id "↙
      ↳ textfield_city"
   Then I hide the keyboard
   When I press view with id "button_commit_address"
   Then I wait for the view with id "view_order" to appear
   When I go back
   Then I wait for the view with id "view_map" to appear
   When I press view with id "button_cancel"

Scenario: ortsbasierter_taxiruf_toplevel_testcase_6
   Then I wait for the view with id "view_splash_screen" to ↙
      ↳ appear
   Then I wait for 2 seconds
   Then I wait for the view with id "view_map" to appear
   Then I wait for the view with id "view_map" to appear
   Then the location is 51.461001, 7.016339 with accuracy ↙
      ↳ 500.000000 1000 milliseconds ago
   Then I wait for the view with id "view_dialog_manual_address" ↙
      ↳ to appear
   When I enter text "45127 Essen" into field with id "↙
      ↳ textfield_city"
   When I enter text "Gerlingstrasse 16" into field with id "↙
      ↳ textfield_street"
   Then I hide the keyboard
   When I press view with id "button_commit_address"
   Then I wait for the view with id "view_order" to appear
   When I go back
   Then I wait for the view with id "view_map" to appear
   When I press view with id "button_cancel"

Scenario: ortsbasierter_taxiruf_toplevel_testcase_7
   Then I wait for the view with id "view_splash_screen" to ↙
      ↳ appear
   Then I wait for 2 seconds
   Then I wait for the view with id "view_map" to appear
   Then I wait for the view with id "view_map" to appear
   Then the location is 51.461001, 7.016339 with accuracy ↙
      ↳ 500.000000 1000 milliseconds ago
   Then I wait for the view with id "view_dialog_manual_address" ↙
      ↳ to appear
   When I enter text "Gerlingstrasse 16" into field with id "↙
      ↳ textfield_street"
   When I enter text "45127 Essen" into field with id "↙
      ↳ textfield_city"
   Then I hide the keyboard
```

```
   When I press view with id "button_commit_address"
   Then I wait for the view with id "view_order" to appear
   When I press view with id "button_cancel"

Scenario: ortsbasierter_taxiruf_toplevel_testcase_8
   Then I wait for the view with id "view_splash_screen" to ↙
       ↳ appear
   Then I wait for 2 seconds
   Then I wait for the view with id "view_map" to appear
   Then I wait for the view with id "view_map" to appear
   Then the location is 51.461001, 7.016339 with accuracy ↙
       ↳ 500.000000 1000 milliseconds ago
   Then I wait for the view with id "view_dialog_manual_address" ↙
       ↳ to appear
   When I enter text "45127 Essen" into field with id "↙
       ↳ textfield_city"
   When I enter text "Gerlingstrasse 16" into field with id "↙
       ↳ textfield_street"
   Then I hide the keyboard
   When I press view with id "button_commit_address"
   Then I wait for the view with id "view_order" to appear
   When I press view with id "button_cancel"

Scenario: ortsbasierter_taxiruf_toplevel_testcase_9
   Then I wait for the view with id "view_splash_screen" to ↙
       ↳ appear
   Then I wait for 2 seconds
   Then I wait for the view with id "view_map" to appear
   Then I wait for the view with id "view_map" to appear
   Then I wait for 30 seconds
   Then I wait for the view with id "view_dialog_manual_address" ↙
       ↳ to appear
   When I enter text "Gerlingstrasse 16" into field with id "↙
       ↳ textfield_street"
   When I enter text "45127 Essen" into field with id "↙
       ↳ textfield_city"
   Then I hide the keyboard
   When I press view with id "button_commit_address"
   Then I wait for the view with id "view_order" to appear
   When I go back
   Then I wait for the view with id "view_map" to appear
   When I press view with id "button_cancel"

Scenario: ortsbasierter_taxiruf_toplevel_testcase_10
   Then I wait for the view with id "view_splash_screen" to ↙
       ↳ appear
   Then I wait for 2 seconds
   Then I wait for the view with id "view_map" to appear
   Then I wait for the view with id "view_map" to appear
   Then I wait for 30 seconds
   Then I wait for the view with id "view_dialog_manual_address" ↙
       ↳ to appear
   When I enter text "45127 Essen" into field with id "↙
       ↳ textfield_city"
   When I enter text "Gerlingstrasse 16" into field with id "↙
       ↳ textfield_street"
   Then I hide the keyboard
   When I press view with id "button_commit_address"
   Then I wait for the view with id "view_order" to appear
   When I go back
   Then I wait for the view with id "view_map" to appear
   When I press view with id "button_cancel"

Scenario: ortsbasierter_taxiruf_toplevel_testcase_11
   Then I wait for the view with id "view_splash_screen" to ↙
       ↳ appear
   Then I wait for 2 seconds
   Then I wait for the view with id "view_map" to appear
   Then I wait for the view with id "view_map" to appear
   Then I wait for 30 seconds
   Then I wait for the view with id "view_dialog_manual_address" ↙
       ↳ to appear
   When I enter text "Gerlingstrasse 16" into field with id "↙
       ↳ textfield_street"
   When I enter text "45127 Essen" into field with id "↙
       ↳ textfield_city"
```

```
    Then I hide the keyboard
    When I press view with id "button_commit_address"
    Then I wait for the view with id "view_order" to appear
    When I press view with id "button_cancel"

Scenario: ortsbasierter_taxiruf_toplevel_testcase_12
    Then I wait for the view with id "view_splash_screen" to ↙
        ↳ appear
    Then I wait for 2 seconds
    Then I wait for the view with id "view_map" to appear
    Then I wait for the view with id "view_map" to appear
    Then I wait for 30 seconds
    Then I wait for the view with id "view_dialog_manual_address" ↙
        ↳ to appear
    When I enter text "45127 Essen" into field with id "↙
        ↳ textfield_city"
    When I enter text "Gerlingstrasse 16" into field with id "↙
        ↳ textfield_street"
    Then I hide the keyboard
    When I press view with id "button_commit_address"
    Then I wait for the view with id "view_order" to appear
    When I press view with id "button_cancel"

Scenario: ortsbasierter_taxiruf_toplevel_testcase_13
    Then I wait for the view with id "view_splash_screen" to ↙
        ↳ appear
    Then I wait for 2 seconds
    Then I wait for the view with id "view_map" to appear
    Then I wait for the view with id "view_map" to appear
    When I press view with id "button_manual_address"
    Then I wait for the view with id "view_dialog_manual_address" ↙
        ↳ to appear
    When I enter text "Gerlingstrasse 16" into field with id "↙
        ↳ textfield_street"
    When I enter text "45127 Essen" into field with id "↙
        ↳ textfield_city"
    Then I hide the keyboard
    When I press view with id "button_commit_address"
    Then I wait for the view with id "view_order" to appear
    When I go back
    Then I wait for the view with id "view_map" to appear
    When I press view with id "button_cancel"

Scenario: ortsbasierter_taxiruf_toplevel_testcase_14
    Then I wait for the view with id "view_splash_screen" to ↙
        ↳ appear
    Then I wait for 2 seconds
    Then I wait for the view with id "view_map" to appear
    Then I wait for the view with id "view_map" to appear
    When I press view with id "button_manual_address"
    Then I wait for the view with id "view_dialog_manual_address" ↙
        ↳ to appear
    When I enter text "45127 Essen" into field with id "↙
        ↳ textfield_city"
    When I enter text "Gerlingstrasse 16" into field with id "↙
        ↳ textfield_street"
    Then I hide the keyboard
    When I press view with id "button_commit_address"
    Then I wait for the view with id "view_order" to appear
    When I go back
    Then I wait for the view with id "view_map" to appear
    When I press view with id "button_cancel"

Scenario: ortsbasierter_taxiruf_toplevel_testcase_15
    Then I wait for the view with id "view_splash_screen" to ↙
        ↳ appear
    Then I wait for 2 seconds
    Then I wait for the view with id "view_map" to appear
    Then I wait for the view with id "view_map" to appear
    When I press view with id "button_manual_address"
    Then I wait for the view with id "view_dialog_manual_address" ↙
        ↳ to appear
    When I enter text "Gerlingstrasse 16" into field with id "↙
        ↳ textfield_street"
    When I enter text "45127 Essen" into field with id "↙
        ↳ textfield_city"
```

```
   Then I hide the keyboard
   When I press view with id "button_commit_address"
   Then I wait for the view with id "view_order" to appear
   When I press view with id "button_cancel"

Scenario: ortsbasierter_taxiruf_toplevel_testcase_16
   Then I wait for the view with id "view_splash_screen" to ↙
       ↳ appear
   Then I wait for 2 seconds
   Then I wait for the view with id "view_map" to appear
   Then I wait for the view with id "view_map" to appear
   When I press view with id "button_manual_address"
   Then I wait for the view with id "view_dialog_manual_address" ↙
       ↳ to appear
   When I enter text "45127 Essen" into field with id "↙
       ↳ textfield_city"
   When I enter text "Gerlingstrasse 16" into field with id "↙
       ↳ textfield_street"
   Then I hide the keyboard
   When I press view with id "button_commit_address"
   Then I wait for the view with id "view_order" to appear
   When I press view with id "button_cancel"

Scenario: ortsbasierter_taxiruf_toplevel_testcase_17
   Then I wait for the view with id "view_splash_screen" to ↙
       ↳ appear
   Then I wait for 2 seconds
   Then I wait for the view with id "view_map" to appear
   Then I wait for the view with id "view_map" to appear
   Then the location is 51.461001, 7.016339 with accuracy ↙
       ↳ 500.000000 1000 milliseconds ago
   Then I wait for the view with id "view_dialog_manual_address" ↙
       ↳ to appear
   When I enter text "Gerlingstrasse 16" into field with id "↙
       ↳ textfield_street"
   When I enter text "45127 Essen" into field with id "↙
       ↳ textfield_city"
   Then I hide the keyboard
   When I press view with id "button_commit_address"
   Then I wait for the view with id "view_order" to appear
   When I go back
   Then I wait for the view with id "view_map" to appear
   Then the location is 21.461001, 7.016339 with accuracy ↙
       ↳ 50.000000 1000 milliseconds ago
   Then I wait for the view with id "view_dialog_error" to appear
   Then I read "Der ermittelte Standort liegt nicht innerhalb ↙
       ↳ Deutschlands." in field "textview_error"
   When I press view with id "button_ok"

Scenario: ortsbasierter_taxiruf_toplevel_testcase_18
   Then I wait for the view with id "view_splash_screen" to ↙
       ↳ appear
   Then I wait for 2 seconds
   Then I wait for the view with id "view_map" to appear
   Then I wait for the view with id "view_map" to appear
   Then the location is 51.461001, 7.016339 with accuracy ↙
       ↳ 500.000000 1000 milliseconds ago
   Then I wait for the view with id "view_dialog_manual_address" ↙
       ↳ to appear
   When I enter text "45127 Essen" into field with id "↙
       ↳ textfield_city"
   When I enter text "Gerlingstrasse 16" into field with id "↙
       ↳ textfield_street"
   Then I hide the keyboard
   When I press view with id "button_commit_address"
   Then I wait for the view with id "view_order" to appear
   When I go back
   Then I wait for the view with id "view_map" to appear
   Then the location is 21.461001, 7.016339 with accuracy ↙
       ↳ 50.000000 1000 milliseconds ago
   Then I wait for the view with id "view_dialog_error" to appear
   Then I read "Der ermittelte Standort liegt nicht innerhalb ↙
       ↳ Deutschlands." in field "textview_error"
   When I press view with id "button_ok"

Scenario: ortsbasierter_taxiruf_toplevel_testcase_19
```

```
    Then I wait for the view with id "view_splash_screen" to ↙
        ↳ appear
    Then I wait for 2 seconds
    Then I wait for the view with id "view_map" to appear
    Then I wait for the view with id "view_map" to appear
    Then I wait for 30 seconds
    Then I wait for the view with id "view_dialog_manual_address" ↙
        ↳ to appear
    When I enter text "Gerlingstrasse 16" into field with id "↙
        ↳ textfield_street"
    When I enter text "45127 Essen" into field with id "↙
        ↳ textfield_city"
    Then I hide the keyboard
    When I press view with id "button_commit_address"
    Then I wait for the view with id "view_order" to appear
    When I go back
    Then I wait for the view with id "view_map" to appear
    Then the location is 21.461001, 7.016339 with accuracy ↙
        ↳ 50.000000 1000 milliseconds ago
    Then I wait for the view with id "view_dialog_error" to appear
    Then I read "Der ermittelte Standort liegt nicht innerhalb ↙
        ↳ Deutschlands." in field "textview_error"
    When I press view with id "button_ok"

Scenario: ortsbasierter_taxiruf_toplevel_testcase_20
    Then I wait for the view with id "view_splash_screen" to ↙
        ↳ appear
    Then I wait for 2 seconds
    Then I wait for the view with id "view_map" to appear
    Then I wait for the view with id "view_map" to appear
    Then I wait for 30 seconds
    Then I wait for the view with id "view_dialog_manual_address" ↙
        ↳ to appear
    When I enter text "45127 Essen" into field with id "↙
        ↳ textfield_city"
    When I enter text "Gerlingstrasse 16" into field with id "↙
        ↳ textfield_street"
    Then I hide the keyboard
    When I press view with id "button_commit_address"
    Then I wait for the view with id "view_order" to appear
    When I go back
    Then I wait for the view with id "view_map" to appear
    Then the location is 21.461001, 7.016339 with accuracy ↙
        ↳ 50.000000 1000 milliseconds ago
    Then I wait for the view with id "view_dialog_error" to appear
    Then I read "Der ermittelte Standort liegt nicht innerhalb ↙
        ↳ Deutschlands." in field "textview_error"
    When I press view with id "button_ok"

Scenario: ortsbasierter_taxiruf_toplevel_testcase_21
    Then I wait for the view with id "view_splash_screen" to ↙
        ↳ appear
    Then I wait for 2 seconds
    Then I wait for the view with id "view_map" to appear
    Then I wait for the view with id "view_map" to appear
    When I press view with id "button_manual_address"
    Then I wait for the view with id "view_dialog_manual_address" ↙
        ↳ to appear
    When I enter text "Gerlingstrasse 16" into field with id "↙
        ↳ textfield_street"
    When I enter text "45127 Essen" into field with id "↙
        ↳ textfield_city"
    Then I hide the keyboard
    When I press view with id "button_commit_address"
    Then I wait for the view with id "view_order" to appear
    When I go back
    Then I wait for the view with id "view_map" to appear
    Then the location is 21.461001, 7.016339 with accuracy ↙
        ↳ 50.000000 1000 milliseconds ago
    Then I wait for the view with id "view_dialog_error" to appear
    Then I read "Der ermittelte Standort liegt nicht innerhalb ↙
        ↳ Deutschlands." in field "textview_error"
    When I press view with id "button_ok"

Scenario: ortsbasierter_taxiruf_toplevel_testcase_22
```

```
   Then I wait for the view with id "view_splash_screen" to ↙
       ↳ appear
   Then I wait for 2 seconds
   Then I wait for the view with id "view_map" to appear
   Then I wait for the view with id "view_map" to appear
   When I press view with id "button_manual_address"
   Then I wait for the view with id "view_dialog_manual_address" ↙
       ↳ to appear
   When I enter text "45127 Essen" into field with id "↙
       ↳ textfield_city"
   When I enter text "Gerlingstrasse 16" into field with id "↙
       ↳ textfield_street"
   Then I hide the keyboard
   When I press view with id "button_commit_address"
   Then I wait for the view with id "view_order" to appear
   When I go back
   Then I wait for the view with id "view_map" to appear
   Then the location is 21.461001, 7.016339 with accuracy ↙
       ↳ 50.000000 1000 milliseconds ago
   Then I wait for the view with id "view_dialog_error" to appear
   Then I read "Der ermittelte Standort liegt nicht innerhalb ↙
       ↳ Deutschlands." in field "textview_error"
   When I press view with id "button_ok"

Scenario: ortsbasierter_taxiruf_toplevel_testcase_23
   Then I wait for the view with id "view_splash_screen" to ↙
       ↳ appear
   Then I wait for 2 seconds
   Then I wait for the view with id "view_map" to appear
   Then I wait for the view with id "view_map" to appear
   Then the location is 51.461001, 7.016339 with accuracy ↙
       ↳ 50.000000 1000 milliseconds ago
   When I press view with id "button_commit_address"
   Then I wait for the view with id "view_order" to appear
   When I press view with id "button_commit_order"
   Then I wait for the view with id "view_order_info" to appear
   When I press view with id "button_ok"

Scenario: ortsbasierter_taxiruf_toplevel_testcase_24
   Then I wait for the view with id "view_splash_screen" to ↙
       ↳ appear
   Then I wait for 2 seconds
   Then I wait for the view with id "view_map" to appear
   Then I wait for the view with id "view_map" to appear
   Then the location is 51.461001, 7.016339 with accuracy ↙
       ↳ 500.000000 1000 milliseconds ago
   Then I wait for the view with id "view_dialog_manual_address" ↙
       ↳ to appear
   When I enter text "Gerlingstrasse 16" into field with id "↙
       ↳ textfield_street"
   When I enter text "45127 Essen" into field with id "↙
       ↳ textfield_city"
   Then I hide the keyboard
   When I press view with id "button_commit_address"
   Then I wait for the view with id "view_order" to appear
   When I press view with id "button_commit_order"
   Then I wait for the view with id "view_order_info" to appear
   When I press view with id "button_ok"

Scenario: ortsbasierter_taxiruf_toplevel_testcase_25
   Then I wait for the view with id "view_splash_screen" to ↙
       ↳ appear
   Then I wait for 2 seconds
   Then I wait for the view with id "view_map" to appear
   Then I wait for the view with id "view_map" to appear
   Then the location is 51.461001, 7.016339 with accuracy ↙
       ↳ 500.000000 1000 milliseconds ago
   Then I wait for the view with id "view_dialog_manual_address" ↙
       ↳ to appear
   When I enter text "45127 Essen" into field with id "↙
       ↳ textfield_city"
   When I enter text "Gerlingstrasse 16" into field with id "↙
       ↳ textfield_street"
   Then I hide the keyboard
   When I press view with id "button_commit_address"
   Then I wait for the view with id "view_order" to appear
```

```
   When I press view with id "button_commit_order"
   Then I wait for the view with id "view_order_info" to appear
   When I press view with id "button_ok"

Scenario: ortsbasierter_taxiruf_toplevel_testcase_26
   Then I wait for the view with id "view_splash_screen" to ↙
       ↳ appear
   Then I wait for 2 seconds
   Then I wait for the view with id "view_map" to appear
   Then I wait for the view with id "view_map" to appear
   Then I wait for 30 seconds
   Then I wait for the view with id "view_dialog_manual_address" ↙
       ↳ to appear
   When I enter text "Gerlingstrasse 16" into field with id "↙
       ↳ textfield_street"
   When I enter text "45127 Essen" into field with id "↙
       ↳ textfield_city"
   Then I hide the keyboard
   When I press view with id "button_commit_address"
   Then I wait for the view with id "view_order" to appear
   When I press view with id "button_commit_order"
   Then I wait for the view with id "view_order_info" to appear
   When I press view with id "button_ok"

Scenario: ortsbasierter_taxiruf_toplevel_testcase_27
   Then I wait for the view with id "view_splash_screen" to ↙
       ↳ appear
   Then I wait for 2 seconds
   Then I wait for the view with id "view_map" to appear
   Then I wait for the view with id "view_map" to appear
   Then I wait for 30 seconds
   Then I wait for the view with id "view_dialog_manual_address" ↙
       ↳ to appear
   When I enter text "45127 Essen" into field with id "↙
       ↳ textfield_city"
   When I enter text "Gerlingstrasse 16" into field with id "↙
       ↳ textfield_street"
   Then I hide the keyboard
   When I press view with id "button_commit_address"
   Then I wait for the view with id "view_order" to appear
   When I press view with id "button_commit_order"
   Then I wait for the view with id "view_order_info" to appear
   When I press view with id "button_ok"

Scenario: ortsbasierter_taxiruf_toplevel_testcase_28
   Then I wait for the view with id "view_splash_screen" to ↙
       ↳ appear
   Then I wait for 2 seconds
   Then I wait for the view with id "view_map" to appear
   Then I wait for the view with id "view_map" to appear
   When I press view with id "button_manual_address"
   Then I wait for the view with id "view_dialog_manual_address" ↙
       ↳ to appear
   When I enter text "Gerlingstrasse 16" into field with id "↙
       ↳ textfield_street"
   When I enter text "45127 Essen" into field with id "↙
       ↳ textfield_city"
   Then I hide the keyboard
   When I press view with id "button_commit_address"
   Then I wait for the view with id "view_order" to appear
   When I press view with id "button_commit_order"
   Then I wait for the view with id "view_order_info" to appear
   When I press view with id "button_ok"

Scenario: ortsbasierter_taxiruf_toplevel_testcase_29
   Then I wait for the view with id "view_splash_screen" to ↙
       ↳ appear
   Then I wait for 2 seconds
   Then I wait for the view with id "view_map" to appear
   Then I wait for the view with id "view_map" to appear
   When I press view with id "button_manual_address"
   Then I wait for the view with id "view_dialog_manual_address" ↙
       ↳ to appear
   When I enter text "45127 Essen" into field with id "↙
       ↳ textfield_city"
```

```
When I enter text "Gerlingstrasse 16" into field with id "↙
    ↳ textfield_street"
Then I hide the keyboard
When I press view with id "button_commit_address"
Then I wait for the view with id "view_order" to appear
When I press view with id "button_commit_order"
Then I wait for the view with id "view_order_info" to appear
When I press view with id "button_ok"
```

Anhang C

Ausführungspfade

C.1 Überlappende Zyklen

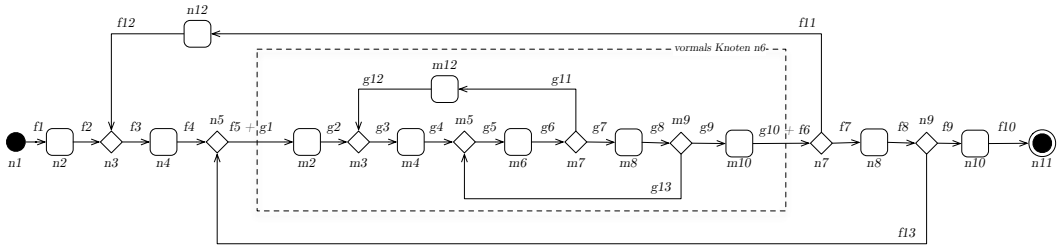


Abbildung C.1: Modellierungsmuster überlappende, eingebettete Zyklen

Quellcodefragment C.1: Textuelle Repräsentierung der aus dem Aktivitätsdiagramm in Abbildung C.1 generierten Ausführungspfade

```
!n1->f1->n2->f2->f3->n4->f4->f5->g1->m2->g2->g3->m4->g4->g5->m6->g6↙
    ↳ ->g7->m8->g8->g9->m10->g10->f6->f7->n8->f8->f9->n10->f10->!n11
!n1->f1->n2->f2->f3->n4->f4->f5->g1->m2->g2->g3->m4->g4->g5->m6->g6↙
    ↳ ->g7->m8->g8->g13->g5->m6->g6->g7->m8->g8->g9->m10->g10->f6->↙
    ↳ f7->n8->f8->f9->n10->f10->!n11
!n1->f1->n2->f2->f3->n4->f4->f5->g1->m2->g2->g3->m4->g4->g5->m6->g6↙
    ↳ ->g11->m12->g12->g3->m4->g4->g5->m6->g6->g7->m8->g8->g9->m10->↙
    ↳ g10->f6->f7->n8->f8->f9->n10->f10->!n11
!n1->f1->n2->f2->f3->n4->f4->f5->g1->m2->g2->g3->m4->g4->g5->m6->g6↙
    ↳ ->g11->m12->g12->g3->m4->g4->g5->m6->g6->g7->m8->g8->g13->g5->↙
    ↳ m6->g6->g7->m8->g8->g9->m10->g10->f6->f7->n8->f8->f9->n10->f10↙
    ↳ ->!n11
!n1->f1->n2->f2->f3->n4->f4->f5->g1->m2->g2->g3->m4->g4->g5->m6->g6↙
    ↳ ->g7->m8->g8->g9->m10->g10->f6->f7->n8->f8->f13->f5->g1->m2->↙
    ↳ g2->g3->m4->g4->g5->m6->g6->g7->m8->g8->g9->m10->g10->f6->f7->↙
    ↳ n8->f8->f9->n10->f10->!n11
!n1->f1->n2->f2->f3->n4->f4->f5->g1->m2->g2->g3->m4->g4->g5->m6->g6↙
    ↳ ->g7->m8->g8->g13->g5->m6->g6->g7->m8->g8->g9->m10->g10->f6->↙
    ↳ f7->n8->f8->f13->f5->g1->m2->g2->g3->m4->g4->g5->m6->g6->g7->↙
    ↳ m8->g8->g13->g5->m6->g6->g7->m8->g8->g9->m10->g10->f6->f7->n8↙
    ↳ ->f8->f9->n10->f10->!n11
!n1->f1->n2->f2->f3->n4->f4->f5->g1->m2->g2->g3->m4->g4->g5->m6->g6↙
    ↳ ->g11->m12->g12->g3->m4->g4->g5->m6->g6->g7->m8->g8->g9->m10->↙
    ↳ g10->f6->f7->n8->f8->f13->f5->g1->m2->g2->g3->m4->g4->g5->m6->↙
    ↳ g6->g11->m12->g12->g3->m4->g4->g5->m6->g6->g7->m8->g8->g9->m10↙
    ↳ ->g10->f6->f7->n8->f8->f9->n10->f10->!n11
!n1->f1->n2->f2->f3->n4->f4->f5->g1->m2->g2->g3->m4->g4->g5->m6->g6↙
    ↳ ->g11->m12->g12->g3->m4->g4->g5->m6->g6->g7->m8->g8->g13->g5->↙
    ↳ m6->g6->g7->m8->g8->g9->m10->g10->f6->f7->n8->f8->f13->f5->g1↙
    ↳ ->m2->g2->g3->m4->g4->g5->m6->g6->g11->m12->g12->g3->m4->g4->↙
```

```
    ↳ g5->m6->g6->g7->m8->g8->g13->g5->m6->g6->g7->m8->g8->g9->m10->↙
    ↳ g10->f6->f7->n8->f8->f9->n10->f10->!n11
!n1->f1->n2->f2->f3->n4->f4->f5->g1->m2->g2->g3->m4->g4->g5->m6->g6↙
    ↳ ->g7->m8->g8->g9->m10->g10->f6->f11->n12->f12->f3->n4->f4->f5↙
    ↳ ->g1->m2->g2->g3->m4->g4->g5->m6->g6->g7->m8->g8->g9->m10->g10↙
    ↳ ->f6->f7->n8->f8->f9->n10->f10->!n11
!n1->f1->n2->f2->f3->n4->f4->f5->g1->m2->g2->g3->m4->g4->g5->m6->g6↙
    ↳ ->g7->m8->g8->g13->g5->m6->g6->g7->m8->g8->g9->m10->g10->f6->↙
    ↳ f11->n12->f12->f3->n4->f4->f5->g1->m2->g2->g3->m4->g4->g5->m6↙
    ↳ ->g6->g7->m8->g8->g13->g5->m6->g6->g7->m8->g8->g9->m10->g10->↙
    ↳ f6->f7->n8->f8->f9->n10->f10->!n11
!n1->f1->n2->f2->f3->n4->f4->f5->g1->m2->g2->g3->m4->g4->g5->m6->g6↙
    ↳ ->g11->m12->g12->g3->m4->g4->g5->m6->g6->g7->m8->g8->g9->m10->↙
    ↳ g10->f6->f11->n12->f12->f3->n4->f4->f5->g1->m2->g2->g3->m4->g4↙
    ↳ ->g5->m6->g6->g11->m12->g12->g3->m4->g4->g5->m6->g6->g7->m8->↙
    ↳ g8->g9->m10->g10->f6->f7->n8->f8->f9->n10->f10->!n11
!n1->f1->n2->f2->f3->n4->f4->f5->g1->m2->g2->g3->m4->g4->g5->m6->g6↙
    ↳ ->g11->m12->g12->g3->m4->g4->g5->m6->g6->g7->m8->g8->g13->g5->↙
    ↳ m6->g6->g7->m8->g8->g9->m10->g10->f6->f11->n12->f12->f3->n4->↙
    ↳ f4->f5->g1->m2->g2->g3->m4->g4->g5->m6->g6->g11->m12->g12->g3↙
    ↳ ->m4->g4->g5->m6->g6->g7->m8->g8->g13->g5->m6->g6->g7->m8->g8↙
    ↳ ->g9->m10->g10->f6->f7->n8->f8->f9->n10->f10->!n11
!n1->f1->n2->f2->f3->n4->f4->f5->g1->m2->g2->g3->m4->g4->g5->m6->g6↙
    ↳ ->g7->m8->g8->g9->m10->g10->f6->f11->n12->f12->f3->n4->f4->f5↙
    ↳ ->g1->m2->g2->g3->m4->g4->g5->m6->g6->g7->m8->g8->g9->m10->g10↙
    ↳ ->f6->f7->n8->f8->f13->f5->g1->m2->g2->g3->m4->g4->g5->m6->g6↙
    ↳ ->g7->m8->g8->g9->m10->g10->f6->f7->n8->f8->f9->n10->f10->!n11
!n1->f1->n2->f2->f3->n4->f4->f5->g1->m2->g2->g3->m4->g4->g5->m6->g6↙
    ↳ ->g7->m8->g8->g13->g5->m6->g6->g7->m8->g8->g9->m10->g10->f6->↙
    ↳ f11->n12->f12->f3->n4->f4->f5->g1->m2->g2->g3->m4->g4->g5->m6↙
    ↳ ->g6->g7->m8->g8->g13->g5->m6->g6->g7->m8->g8->g9->m10->g10->↙
    ↳ f6->f7->n8->f8->f13->f5->g1->m2->g2->g3->m4->g4->g5->m6->g6->↙
    ↳ g7->m8->g8->g13->g5->m6->g6->g7->m8->g8->g9->m10->g10->f6->f7↙
    ↳ ->n8->f8->f9->n10->f10->!n11
!n1->f1->n2->f2->f3->n4->f4->f5->g1->m2->g2->g3->m4->g4->g5->m6->g6↙
    ↳ ->g11->m12->g12->g3->m4->g4->g5->m6->g6->g7->m8->g8->g9->m10->↙
    ↳ g10->f6->f11->n12->f12->f3->n4->f4->f5->g1->m2->g2->g3->m4->g4↙
    ↳ ->g5->m6->g6->g11->m12->g12->g3->m4->g4->g5->m6->g6->g7->m8->↙
    ↳ g8->g9->m10->g10->f6->f7->n8->f8->f13->f5->g1->m2->g2->g3->m4↙
    ↳ ->g4->g5->m6->g6->g11->m12->g12->g3->m4->g4->g5->m6->g6->g7->↙
    ↳ m8->g8->g9->m10->g10->f6->f7->n8->f8->f9->n10->f10->!n11
!n1->f1->n2->f2->f3->n4->f4->f5->g1->m2->g2->g3->m4->g4->g5->m6->g6↙
    ↳ ->g11->m12->g12->g3->m4->g4->g5->m6->g6->g7->m8->g8->g13->g5->↙
    ↳ m6->g6->g7->m8->g8->g9->m10->g10->f6->f11->n12->f12->f3->n4->↙
    ↳ f4->f5->g1->m2->g2->g3->m4->g4->g5->m6->g6->g11->m12->g12->g3↙
    ↳ ->m4->g4->g5->m6->g6->g7->m8->g8->g13->g5->m6->g6->g7->m8->g8↙
    ↳ ->g9->m10->g10->f6->f7->n8->f8->f13->f5->g1->m2->g2->g3->m4->↙
    ↳ g4->g5->m6->g6->g11->m12->g12->g3->m4->g4->g5->m6->g6->g7->m8↙
    ↳ ->g8->g13->g5->m6->g6->g7->m8->g8->g9->m10->g10->f6->f7->n8->↙
    ↳ f8->f9->n10->f10->!n11
```

Tabellenverzeichnis

Abbildungsverzeichnis

Quellcodeverzeichnis

Abkürzungen

ACD Aktivitätsdiagramm

API Application Programming Interface

APNS Apple Push Notification Service

AR Augmented Reality

ART Android Runtime

BDD Behavior Driven Development

BLE Bluetooth Low Energy

BPMN Business Process Model and Notation

CCPP Composite Capabilities/Preference Profile

CI Continuous Integration

CML Context Modeling Language

DSL Domain Specific Language (domänenspezifische Sprache)

EDGE Enhanced Data Rates for GSM Evolution

EMF Eclipse Modeling Framework

EMOF Essential Meta Object Facility

EPK Ereignisgesteuerte Prozesskette

GCM Google Cloud Messaging

GPRS General Packet Radio Service

GPS Golbal Positioning System

GSM Global System for Mobile Communication

HTTP Hypertext Transfer Protocol

IDE Integrated Development Environment

ISP Internet Service Provider

M2M Modell-zu-Modell Transformation

M2T Modell-zu-Text Transformation

MDA Model Driven Architecture

MDSD Model Driven Software Development

MDT Model Driven Testing

MOF Meta-Object-Facility

NFC Near Field Communication

OCL Object Constraint Language

OCL Object Management Group

OSGi Open Services Gateway Initiative

OWL Web Ontologie Language

RCP Rich Client Platform

rOSGi Remote Services for OSGi

RUP Real-Time Streaming Protocol

SCMS Source Code Management System

SDK Software Development Kit

Siri Speech Interpretation and Recognition Interface

SSL Secure Socket Layer

SUT System Under Test

TDD Test Driven Development

UC Ubiquitous Computing

UI User Interface

UML Unified Modeling Language

UMTS Universal Mobile Telecommunications System

UTC Koordinierte Weltzeit

UX User Experience

WAP Wireless Access Protocol

WGS84 World Geodetic System 1984

WLAN Wireless Local Area Network

XMI XML Metadata Interchange

XML Extensible Markup Language

Glossar

Acceptance Testing Durchführung von Akzeptanztests.
Seite(n): 169, 461, siehe Akzeptanztest, Integration Testing, Regression Testing sowie Unit-Testing

Activity Tracking Durch Beobachtung von Kontextparametern durch algorithmische oder heuristische Folgerung auf die Aktivität eines menschlichen Akteurs schließen.
Seite(n): 51–53, 461

Aktivitätsdiagramm, (UML) Eines der sieben Verhaltensdiagramme der UML. Es strukturiert Arbeitsabläufe (engl. Workflows eines Systemmodells schematisch in schrittweise Aktionen und deren Verbindungen mit Kontroll- und Datenflüssen [268].
Seite(n): 41, 60, 61, 87, 167, 201, 209, 210, 220–226, 231, 236, 249, 261, 265, 272–284, 286, 287, 289–321, 343–345, 347, 353–358, 360, 361, 364–404, 406–414, 416, 418–420, 423, 424, 427, 428, 430, 432, 433, 435, 451, 456–458, 461, 463, 475, siehe UML

Akzeptanztest (engl. Acceptance Test) Formales Testen von Benutzeranforderungen; Durchgeführt, um einem Auftraggeber oder einer bevollmächtigten Instanz die Entscheidung auf der Basis der Abnahmekriterien zu ermöglichen, ob ein System anzunehmen ist oder nicht. [192]
Seite(n): 26, 37, 43, 71, 82, 87, 90, 98, 101–103, 105, 115, 155, 159, 161–165, 173, 181, 192, 195, 197–199, 201, 204, 215, 216, 223, 230, 235, 278, 289, 293, 294, 314, 322, 327, 336, 423, 461, 467, 470, siehe Acceptance Testing, Komponententest, Integrationstest, Regressionstest sowie Systemtest

Android (Google) Android ist ein Betriebssystem für Smartphones und Tablet-Computer.
Seite(n): 21, 23, 24, 29, 35, 37, 42, 43, 45, 54, 63, 76–81, 83–85, 90, 92–98, 104–106, 108–110, 125, 136, 140, 141, 143, 161–163, 172, 173, 175, 176, 178, 182, 188–190, 194, 195, 211–214, 216, 217, 222, 230, 269, 319, 321–325, 327, 328, 330–338, 341–343, 345, 347, 350, 352, 353, 355, 360–362, 407, 408, 411, 412, 414, 418–422, 425–428, 431–433, 435–437, 456–458, 461, 467, 468, 470, 472, 475, 478, 481, 487, siehe Apple iOS sowie Microsoft Windows Phone

App Der Begriff *App* ist die Kurzform des Begriffs *Application* (engl. Anwendung, insbesondere Software) [101]. Der Begriff *App* wird insbesondere im deutschen Sprachgebrauch für Anwendungen für mobile Geräte verwendet.
Seite(n): 11, 12, 14, 17, 20–26, 35, 36, 43, 48, 49, 54, 62–65, 67, 69–71, 78–83, 90, 92, 94, 95, 103, 104, 106–111, 115, 116, 125, 126, 128, 129, 134, 135, 138, 139, 141–143,

151, 152, 170–172, 174–185, 188, 190, 191, 193, 194, 197, 205, 212, 213, 215, 216, 231, 247, 256, 257, 260, 264, 267, 275–282, 284, 285, 296, 297, 299, 301, 302, 304, 308, 309, 313, 314, 321, 323–333, 335, 345, 353, 354, 361, 377, 404, 406–409, 411, 412, 414–428, 432–434, 436, 441, 455, 456, 458, 460, 461, 468, 471, 473, 489, 490

Appium Ein OSS-Framework zur Testautomatisierung für native und hybride mobile Anwendungen. [14]
Seite(n): 98, 216, 461, siehe Xamarin sowie Calabash

Apple Weltweit tätiges Soft- und Hardwareunternehmen. Schöpfer des Betriebssystems Apple iOS und Mitbegründer der Smartphone-Ära.
Seite(n): 20, 21, 62, 114, 119, 130, 178, 230, 461, 468, 482

App Store (Apple) App Store des Unternehmens Apple.
Seite(n): 21, 461, siehe App Store, Google Play Store sowie Windows Phone Store

Apple Push Notification Service (APNS) Dienst der Apple iOS-Plattform zum Versand von Nachrichten an mobile Teilnehmer. [17]
Seite(n): 461, 463, siehe GCM

Application Programming Interface (API) Teile von Software, die auf der Ebene von Quellcode anderen Programmen zur Anbindung zur Verfügung gestellt werden.
Seite(n): 15, 20, 21, 36, 39, 49, 54, 63, 77, 96–98, 120, 125, 126, 131, 140, 143, 147, 149, 171–173, 230, 241, 243, 248, 249, 281, 283, 323, 364, 410, 411, 419, 420, 461, 463, 470, 479

App Store Als App Store wird ein digitaler Marktplatz bezeichnet, dessen Aufgabe die Bereitstellung einer technischen Infrastruktur zur Kommerzialisierung von Apps ist.
Seite(n): 12, 20–22, 48, 49, 64, 81, 82, 108, 110, 175, 333, 407, 436, 461, 468, 475, siehe Apple App Store, Google Play Store sowie Windows Phone Store

Assertion Logischer Ausdruck der einen Zustand des Programms beschreibt oder eine Menge von Bedingungen die Programmvariablen an einem definierten Punkt der Programmausführung erfüllen müssen. Beim Testen verwendet, um den Ist-Zustand des SUT mit dem Soll-Zustand zu vergleichen. [192]
Seite(n): 84, 92, 273, 404–406, 423, 461, siehe SUT, Unit-Test sowie Testen

Android Runtime (ART) Die seit Version 5.0 Lollipop eingesetzte Laufzeitumgebung für Android-Anwendungen. ART ist eine virtuelle Maschine, durch welche die zuvor verwendete virtuelle Maschine Dalvik VM auf der Plattform Android abgelöst wird.
Seite(n): 205, 216, 332, 461, 463, 468

Audit, (Software-) Ein unabhängiges Testen von Softwareprodukten und -prozessen, um die Konformität mit Standards, Richtlinien, Spezifikationen, und/oder Prozeduren basierend auf objektiven Kriterien zu bestimmen, einschließlich der Dokumente, welche (1) die Gestaltung oder den Inhalt der zu erstellenden Produkte festlegen, (2) den Prozess der Erstellung der Produkte beschreiben (3) und spezifizieren, wie die Übereinstimmung

mit den Standards und Richtlinien nachgewiesen bzw. gemessen werden kann. [196]
Seite(n): 154, 461

Augmented Reality (AR) Computergestütze Erweiterung der Wahrnehmung der Realität, häufig durch Überlagerung eines Kamera-Live-Bilds. [250]
Seite(n): 49, 50, 63, 97, 108, 110, 111, 135, 136, 141, 144, 151, 192, 255, 461, 463

Backend-System Teil einer Client-Server-IT-Infrastruktur auf der dem Teilnehmer abgewandten Seite. Versorgt Clients mit Informationen und Funktionalitäten, die über Schnittstellen adressiert werden.
Seite(n): 12, 20, 34, 38, 70, 75, 109, 111, 112, 129, 151, 152, 163, 179, 180, 182, 188, 276, 414, 461, 480, siehe Client sowie Server

Base Station Subsystem (BSS) Auch Mobilfunkbasisstation; ein Baustein der Infrastruktur des GSM-Mobilfunknetzes.
Seite(n): 130, 131, 146, 149, 461, 475

Beacon Ortsgebundene technische Einrichtung mit dem Zweck, ein wiedererkennbares Signal zu propagieren. In der IT häufig Quelle eines elektromagnetischen Signals.
Seite(n): 50, 129, 130, 146, 251, 461

Bedarfsträger Ein Bedarfsträger oder Stakeholder eines Systems ist eine Person oder Organisation, die direkt oder indirekt Einfluss auf die Anforderungen des betrachteten Systems hat. [284]
Seite(n): 97, 99, 101, 102, 154, 159, 161, 202, 203, 461, siehe Stakeholder

Behavior Driven Development (BDD) Technik der agilen Softwareentwicklung, die mit einem Fokus auf Automatisierbarkeit versucht, Anforderungen an ein Softwaresystem durch Verwendung einer textuellen DSL zu dokumentieren. [262, 368]
Seite(n): 43, 99, 101, 259, 337, 461, 463, 472

Beta-Tester Ein Tester, dem ein Softwareprodukt zur Durchführung von Tests noch vor der eigentlichen Markteinführung anvertraut wird.
Seite(n): 461, 472, siehe Test sowie Tester

BlackBerry Eine mobile Plattform, eine Klasse mobiler Geräte sowie das Unternehmen (früher Research in Motion), welches die Plattform betreibt und mobile Geräte entwickelt und vertreibt. [46]
Seite(n): 38, 108, 173, 461

Black-Box-Test Ein Test oder eine Menge von Tests, spezifikationsorientiert erstellt und durchgeführt ohne Kenntnis der Struktur oder von Implementierungsdetails des zu testenden Softwaresystems und im Kontrast zu White-Box-Test ohne Kenntnisse des Quellcodes. Auch als *Functional Test* bezeichnet. IEEE [192]
Seite(n): 43, 77, 80, 98, 154, 159–162, 173, 204, 223, 289, 323, 328, 329, 331, 333, 423, 436, 461, 489, siehe White-Box-Test sowie Grey-Box-Test

Bluetooth Industriestandard zur drahtlosen Datenübertragung.
Seite(n): 147, 461, siehe BLE

Bluetooth Low Energy (BLE) Industriestandard zur drahtlosen Datenübertragung, optimiert für einen besonders geringen Energiebedarf.
Seite(n): 50, 130, 251, 461, 463, siehe Bluetooth

Business Process Model and Notation (BPMN) Geschäftsprozessmodell und -notation; eine DSL mit graphischer Syntax zur Spezifikation von Geschäftsprozessen.
Seite(n): 41, 49, 210, 226, 229, 461, 463

Caching Temporäres Speichern von Daten in einem schnellen, flüchtigen Speicher, dem *Cache*. Technologie zur Optimierung schneller Speicherzugriffe.
Seite(n): 19, 95, 461

Calabash Ein Framework zur Automatisierung von Akzeptanztests. Tests werden quasinatürlichsprachlich als Cucumber-Features spezifiziert. [369]
Seite(n): 43, 45, 77, 83, 90, 93, 98–100, 103–106, 163, 165, 166, 204, 210, 216, 217, 221, 259, 260, 273, 288, 318, 320–322, 327–329, 335–338, 341–345, 347, 353–355, 358, 360, 361, 363, 364, 403, 404, 411, 415, 417, 419–421, 423–425, 428, 432, 433, 437–439, 441, 455, 458–461, 470, 481, siehe Calabash-Android, Calabash-iOS, Appium sowie Xamarin

Calabash-Android Android-Inkarnation des Testautomatisierungs-Frameworks Calabash. Es erlaubt die Durchführung automatisierter funktionaler Anwendungstests für die Plattform Android. [121]
Seite(n): 42, 98, 99, 217, 260, 461, 472, 474, siehe Calabash sowie Calabash-iOS

Calabash-Apple iOS Apple iOS-Inkarnation des Testautomatisierungs-Frameworks Calabash. Es erlaubt die Durchführung automatisierter funktionaler Anwendungstests für die Plattform Apple iOS. [122]
Seite(n): 98, 99, 217, 461, 472, 474, siehe Calabash sowie Calabash-Android

Callback (-Methode) Eine Funktion oder Methode, die einer anderen Methode in einem API Aufruf als Parameter übergeben wird, um von dieser aufgerufen zu werden.
Seite(n): 143, 323, 324, 332, 461

Capture and Replay (C'n'R) Technologie der Testautomatisierung, bei der eine manuelle Testausführung zur späteren maschinellen Wiederholung inklusive aller Testdaten aufgezeichnet wird.
Seite(n): 78–81, 83, 207, 210–214, 218, 254, 341, 439, 460, 461

Client Anwendernahe Rolle einer Entität in Client-Server-Architekturmodellen. Software, die lokal auf dem Gerät des Anwenders ausgeführt wird und die Daten und Funktionen von einem Server abruft.
Seite(n): 98, 461, 469, 470, 482, 489, siehe Backend-System

Closed-Source-Software Software, deren Quellcode der Öffentlichkeit i. d. R. nicht zugänglich ist und die nicht durch Dritte modifiziert werden kann oder darf.
Seite(n): 16, 62, 432, 461

Cloud(-Dienst) Eine Architektur eines Computernetzwerkes, die durch Abstraktion und dynamische Allokierung von Ressourcen den Anschein unbegrenzter Kapazität erweckt. Die tatsächliche technische Infrastruktur bleibt dem Anwender i. d. R. verborgen.
Seite(n): 20, 28, 461

Compiler Eine Software, die Quellcode einer Programmiersprache in ausführbaren Bytecode übersetzt, der direkt auf der Zielplattform ausgeführt werden kann. Zielplattform kann eine CPU-Architektur oder eine virtuelle Maschine sein.
Seite(n): 172, 287, 461, 472, 475, 476

Composite Capabilities/Preference Profile Eine Kontextmodellierungssprache. Insbesondere eine spezifische Realisierung nach W3C [356], Reynolds et al. [295].
Seite(n): 232, 461, 463

Constraint Eine Zwangsbedingung, die vom Wert einer Variable erfüllt werden muss. In Software werden *Constraints* beispielsweise als Verzweigungsbedingung verwendet; in der Modellierung z. B. bei UML-Guard-Conditions.
Seite(n): 461, 471

Context Modeling Language (CML) Eine Kontextmodellierungssprache. Insbesondere eine spezifische Realisierung nach Henricksen und Indulska [178, 176, 177].
Seite(n): 233, 461, 463

Continuous Integration (CI) Begriff für eine Technologiefamilie der Softwareentwicklung, die eine nahtlose, kontinuierliche Integration funktionaler Inkremente in ein Softwareprodukt ermöglicht. Mündet im Idealfall unmittelbar in einem für den Anwender verfügbaren Softwareartefakt.
Seite(n): 32, 47, 48, 71, 84, 97, 105, 164, 195, 198, 336, 337, 435, 461, 463, 478

Crash (Software) Ein Crash bezeichnet das spontane, unvorhergesehene Versagen eines Computersystems oder einer seiner Komponenten [192].
Seite(n): 64, 67, 69, 71, 81, 82, 168, 173, 461, 471

Cross-Platform-Technologien Familie von Technologien, die es ermöglichen Anwendungen mit nur einer einzigen Quellcodebasis auf mehreren unterschiedlichen Zielplattformen auszuführen. Wenngleich nicht auf mobile Plattformen beschränkt, wird das Prinzip insbesondere im mobilen Umfeld genutzt, um den Mehraufwand mehrerer plattformspezifischer Entwicklungsprojekte zur Erstellung einer App für mehrere Plattformen zu reduzieren.
Seite(n): 17, 116, 125, 330, 363, 461, 490

Crowdsourcing Outsourcing von Aufgaben und Arbeitsaufträgen an eine große, heterogene Menge von Individuen. [187]
Seite(n): 47, 435, 461, 472, siehe Crowdtesting sowie Outsourcing

Crowdtesting Anwendung von Crowdsourcing auf das Testen von Software. Es kommt eine große Anzahl Individuen als Beta-Tester zum Einsatz, wobei von der Heterogenität der Individuen, deren Geräte und Standorte profitiert wird, um eine Software in einem möglichst breiten Betriebsumfeld zu Testen.
Seite(n): 47, 461, siehe Crowdsourcing

Cucumber Werkzeug zur automatischen Durchführung von Akzeptanztests für Softwaresysteme. Im Rahmen des BDD wird Cucumber zur Beschreibung von Verhaltensaspekten von Softwaresystemen verwendet. Cucumber verwendet die DSL Gherkin und kann aufgrund des formalen Syntax maschinell verarbeitet werden. Die Testautomatisierungstechnologien Calabash-Android/Calabash-iOS basieren auf Cucumber.
Seite(n): 99–105, 455, 459, 461, 470, 472–474, 481, siehe Calabash sowie Gherkin

CUnit Framework zur Implementierung und Ausführung von Unit-Tests in der Programmiersprache C.
Seite(n): 91, 162, 461, siehe Unit-Testing sowie JUnit

Dalivk Virtual Machine Eine virtuelle Maschine, die den durch den Compiler des SDK der Plattform Android erzeugten Byte-Code ausführt. Die Dalvik VM war bis zur Version 5.0 Lollipop die Standardlaufzeitumgebung der Plattform Android. [130]
Seite(n): 205, 461, 468, 472

Deployment Verteilung oder Installation von Software oder -Softwarekomponenten auf Desktop-Computer oder Server.
Seite(n): 107, 125, 461

Desktop (-Computer) Computer, der aufgrund seiner Bauart geeignet ist permanent am Arbeitsplatz oder Schreibtisch (engl. Desk) verwendet zu werden. Insbesondere in Abgrenzung zum Laptop oder zu mobilen Geräten.
Seite(n): 11, 14, 19, 20, 23, 31, 32, 35, 36, 49, 71, 77, 82, 83, 103, 107, 113, 116, 129, 148, 177, 196, 238, 326, 327, 331, 335, 456, 461, 472, 477, siehe Laptop

Domain Specific Language (DSL) Formale Sprache mit deren Hilfe Entitäten eines spezifischen Problemfelds (einer Domäne) exakt beschrieben werden können. Insbesondere mit dem Anspruch, eine Domäne vollständig abzudecken aber zugleich nichts außerhalb der Domäne abzubilden.
Seite(n): 70, 85, 89, 99–101, 208, 210, 259, 263, 437, 461, 463, 469, 470, 472, 473

Domain Test Bereichs- oder Randbereichstest, der ein SUT gezielt mit Eingabewerten an den Grenzen des Zulässigen ausführt. [360, 7, 236, 352]).
Seite(n): 183, 461

Eclipse Modeling Framework (EMF) Ein Eclipse-basiertes Framework zur Modellierung und Code-Generierung.
Seite(n): 70, 355–359, 435, 461, 463, 473

Ecore Das Metamodell des EMF; erlaubt die Definition domänenspezifischer Metamodelle zur Verwendung mit dem EMF.
Seite(n): 356, 461

Emulator Computersystem, häufig eine Software, die ein anderes System in beobachtbaren Eigenschaften nachbildet. Der Begriff leitet sich vom lat. *aemulare*, „nachahmen“ ab. Der konkrete Einsatzzweck bestimmt, welche Eigenschaften des emulierten Systems nachgeahmt werden. Im Gegensatz zum Simulator wird von einem Emulator nicht gefordert, alle internen Zustände des emulierten Systems originalgetreu nachzubilden.
Seite(n): 36–39, 66, 69, 71, 80, 85, 93, 95–97, 103, 152, 158, 171, 174, 179, 181, 183–185, 188, 189, 194, 199, 221, 222, 240, 247, 283, 321, 323, 334, 335, 337, 352, 361, 362, 421, 422, 425, 426, 431, 461, 482, siehe Simulator

Enhanced Data Rates for GSM Evolution (EDGE) Eine Prä-3G Technologie zur drahtlosen Vernetzung mobiler Geräte.
Seite(n): 38, 461, 463, siehe GSM, UMTS, LTE sowie GPRS

Enterprise-App Eine für den internen Unternehmenseinsatz intendierte App.
Seite(n): 23, 461

Entitiy Relationship Model Graphische DSL zur Datenmodellierung.
Seite(n): 58, 461

Ereignisgesteuerte Prozesskette (EPK) Zentraler Bestandteil der SAP-Referenzmodelle und der ARIS-Konzepte und Grundlage modellgetriebener Ansätze für ein durchgängiges und werkzeuggestütztes Geschäftsprozessmanagement. [263]
Seite(n): 41, 49, 229, 461, 463

Essential Meta Object Facility (EMOF) Eine spezielle Metadaten-Architektur und Standard des *Model Driven Engineering*.
Seite(n): 356, 461, 463

Event Ereignis; Kernartefakt der ereignisorientierten Programmierung, beispielsweise eine Interaktion des Anwenders.
Seite(n): 80, 364, 461

Extensible Markup Language (XML) Auszeichnungssprache zur Beschreibung einer Klasse Datenojekte in XML-Dokumenten [57]
Seite(n): 232, 247, 356, 461, 465, 473, 490

Feature (1) Eine einzelne Funktion oder Gruppe von Funktionen, die in einem Softwaresystem eine fachliche Einheit bilden. (2) Ein Artefakt der Testautomatisierungstechnologie Cucumber. Bildet Tests für einzelne Funktionen oder Gruppen von Funktionen ab.
Seite(n): 21, 64, 66, 99, 100, 103, 221, 320, 321, 336, 337, 351, 404, 419, 420, 424, 437, 461, 470, 485

Feldtest Testen einer Software innerhalb ihres zukünftigen Betriebsumfelds, häufig mit vielen Teilnehmern.
Seite(n): 67, 71, 76, 158, 187, 461

First Fix (Lokalisierung) Erstmalige Bestimmung einer diskreten Lokalisierungsinformationen, z. B. mit GPS-Technologie.
Seite(n): 188, 461, siehe Fix

Fix (Lokalisierung) Bestimmung diskreter Lokalisierungsinformationen eines Anwenders oder Geräts, insb. im Umfeld der GPS-Technologie.
Seite(n): 461

General Packet Radio Service (GPRS) Auf dem GSM basierender paketorientierter Datenübertragungsdienst in mobilen Funknetzwerken.
Seite(n): 29, 38, 109, 171, 185, 461, 463, siehe GSM, UMTS, LTE sowie EDGE

geodätisch Abgeleitet von Geodäsie, der Wissenschaft von der Ausmessung und Abbildung der Erdoberfläche.
Seite(n): 30, 55–57, 131, 133, 134, 241, 242, 249, 461

Geofencing Technologie zur geograpischen Beschränkung digitaler Dienste.
Seite(n): 110, 242, 243, 250, 252, 253, 461

Gherkin Die vom Werkzeug Cucumber [83] eingesetzte Sprache zur Beschreibung von Systemverhalten. Sie dient der abstrakten Beschreibung von Verhaltensaspekten eines Softwaresystems. Sie wird zur Beschreibung von Testfällen in der Testautomatisierungstechnologie Calabash-Android [121]/Calabash-iOS [122] verwendet und zeichnet sich durch ihren natürlichsprachlichen Charakter aus.
Seite(n): 99–103, 338, 344, 437, 459, 461, 472, siehe Calabash sowie Cucumber

Glass-Box-Test Alternative Bezeichnung für White-Box-Test.
Seite(n): 159, 461, siehe White-Box-Test

Global Positioning System (GPS) Ein System zur satellitengestützten Zeitmessung und Positionsbestimmung. GPS ist aktuell (Stand 2013) die weltweit wichtigste Technologie zur Ortsbestimmung in militärischen und zivilen Bereichen. GPS verwendet ein System von Satelliten die ständig ein Signal ausstrahlen in welchem die exakte Postion des Satelliten und die Uhrzeit codiert ist. Aus den Laufzeiten der Signale von theoretisch drei, in der Praxis vier, Satelliten können Empfänger ihre eigene Position und Geschwindigkeit berechnen. Je mehr Satelliten sich im Empfangsbereich befinden, desto exakter kann die Positionsbestimmung erfolgen. Effekte wie Urban Canyoning können den Empfang gebietsweise beeinträchtigen. Innerhalb von Gebäuden können GPS-Signale i. d. R. nicht empfangen werden, weshalb eine Positionsbestimmung via GPS innerhalb von Gebäuden nur im Ausnahmefall erfolgreich ist.
Seite(n): 11, 12, 28–30, 38, 50, 53, 56, 58, 64, 69, 70, 76, 79, 85, 107, 109, 111, 119, 120, 125, 126, 130–132, 134, 135, 146, 148–151, 175, 183–186, 188–190, 230, 239, 241, 250,

251, 281, 284, 285, 323, 330, 407–409, 411, 412, 414, 415, 434, 458, 461, 463, 474, 481, 488

Global System for Mobile Communication (GSM) Standard der digitalen Mobilfunktelefonie.

Seite(n): 144, 149, 461, 463, 469, 474, siehe GPRS, UMTS, LTE sowie EDGE

Google Anbieter eines internetbasierten Suchdienstes und Betreiber der mobilen Plattform Android.
Seite(n): 21, 119, 131, 330, 361, 461

Google Cloud Messaging (GCM) Push-Benachrichtigung-dienst der Plattform Android.
Seite(n): 461, 463, siehe APNS

Google Play Store App Store der Plattform Android.
Seite(n): 20, 21, 65, 77, 110, 222, 418, 422, 423, 432, 458, 461, siehe App Store, Apple App Store sowie Windows Phone Store

Grey-Box-Test Ein Test oder eine Menge von Tests, funktionsorientiert erstellt und durchgeführt mit eingeschränkter Kenntnis der Struktur des zu testenden Softwaresystems und eingeschränkten Quellcodekenntnissen.
Seite(n): 159, 161, 162, 423, 461, siehe White-Box-Test sowie Black-Box-Test

Guard (UML) Modellierungselement der UML zur Abbildung von Zwangsbedingungen an Kontrollflüssen. Häufig Regulierung des Markenflusses an Verzweigungsknoten in UML-Aktivitätsdiagrammen.
Seite(n): 231, 236, 280, 281, 386, 408, 414, 461, 471

Handover (Mobilfunknetz) Prozess der Übergabe der Netzwerkverbindung eines mobilen Geräts zwischen zwei BSS, häufig als Resultat von Mobilität des Anwenders.
Seite(n): 149, 461

Hardware Abstraction Layer Abstraktionsschicht zwischen Hardware und Software in Computersystemen, die heterogene Implementierung einer Klasse von Bauteilen gegenüber der Anwendungsschicht homogen adressierbar macht.
Seite(n): 332–334, 345, 361, 461

Hypertext Transfer Protocol (HTTP) Zustandsloses Protokoll zur Übertragung von Daten in Computernetzwerken.
Seite(n): 104, 105, 461, 463

Instrumentierung Die Verwendung spezialisierter Messwerkzeuge zur Beobachtung und Manipulation einer Software während ihrer Ausführung.
Seite(n): 43, 98, 101–105, 204, 205, 215, 335, 363, 461

Integrated Development Environment (IDE) Integrierte Entwicklungsumgebung zur Erstellung von Software. Umfasst mindestens einen Editor und einen Compiler.
Seite(n): 31, 36, 162, 196, 355, 356, 358, 360, 436, 461, 464

Integrationstest (Integration Test) Test, der die Interaktion mehrerer Softwarekomponenten miteinander überprüft.
Seite(n): 71, 92, 159, 162, 163, 198, 461, 476, siehe Akzeptanztest, Integration Testing, Komponententest, Regressionstest sowie Systemtest

Integration Testing Durchführen von Integrationstests.
Seite(n): 461, siehe Acceptance Testing, Integrationstest, Regression Testing sowie Unit-Testing

Internet Service Provider (ISP) Anbieter eines (mobilen) Internetzugangs.
Seite(n): 127, 183, 185, 435, 461, 464

iOS (Apple) Betriebssystem für mobile Kommunikationsgeräte der Firma Apple. Es wurde 2007 zusammen mit dem iPhone als iPhone OS vorgestellt und 2010 in iOS umbenannt. Inzwischen werden die Produktfamilien iPhone, iPod Touch und iPad mit iOS betrieben.
Seite(n): 20, 21, 23, 24, 29, 30, 76, 92, 98, 108, 140, 163, 172, 176, 194, 216, 217, 461, 468, 470, siehe Android sowie Microsoft Windows Phone

JUnit Framework zur Erstellung von Unit-Tests für Software in der Programmiersprache Java.
Seite(n): 78, 80, 84, 91–97, 162, 163, 166, 182, 184, 210, 215, 221, 227, 258, 273, 322, 327, 344, 349, 404, 406, 427, 459, 461, 487, siehe Unit-Testing sowie CUnit

Kompilat Produkt eines Compilers.
Seite(n): 82, 125, 154, 157, 160, 190, 222, 289, 330, 333, 432, 461, 478

Komponententest Siehe Unit-Test.
Seite(n): 91–94, 96, 98, 102, 104, 162, 163, 198, 199, 203, 204, 461, siehe Akzeptanztest, Integrationstest, Regressionstest, Systemtest, Unit-Test sowie Unit-Testing

Kontext Der Kontext einer mobilen Anwendung ist die Gesamtheit aller Parameter ihrer Betriebsumgebung, die sich auf das Laufzeitverhalten der Anwendung auswirken. Hierzu können sowohl physikalische Parameter der Betriebsumgebung, der Standort des Anwenders als auch situative Faktoren zählen (vgl. Schilit et al. [311], Brown et al. [61], Abowd et al. [3], Schmidt et al. [315], Dey [92]). Eine ausführliche Diskussion erfolgt in Abschnitt 3.1.1.4.
Seite(n): 1, 12–17, 23, 24, 26, 29–34, 38–45, 48–63, 65–81, 83–85, 87, 91–98, 102, 105–108, 111–113, 115–129, 133–135, 137–155, 157, 158, 161, 164–175, 177–185, 191–193, 195–197, 199, 201, 204–210, 213–215, 217–222, 224–259, 261, 263–265, 267, 268, 270, 271, 273, 275–279, 281–283, 285–287, 289–291, 303, 308, 310, 312, 314–316, 319, 321–323, 325, 327, 328, 330, 332, 335–338, 341–345, 347–361, 363, 364, 406–408, 417, 418, 422, 426–437, 455, 461, 467, 471, 476, 489

Koordinierte Weltzeit (UTC) Einheitliche Skala zur Angabe von Zeitinformationen (Weltzeit).
Seite(n): 128, 461, 464

Korrektheit (Software) Konsistenz gegenüber einer Spezifikation.
Seite(n): 164–166, 461, siehe Testen

Laptop (-Computer) Computer, der aufgrund seiner Bauart geeignet ist, gelegentlich mobil verwendet zu werden. Insbesondere in Abgrenzung zum Desktop oder zu mobilen Geräten.
Seite(n): 115, 461, 472, siehe Desktop

Load Test Siehe Stresstest.
Seite(n): 196, 205, 461, 483, siehe Smoke-Test

Long Term Evolution (LTE) Standard zu Datenübertragung mit hoher Geschwindigkeit in mobilen Computernetzwerken.
Seite(n): 18, 50, 144, 149, 171, 185, 461, siehe GPRS, GSM, UMTS sowie EDGE

Metaklasse Eine Klasse, deren Instanzen Klassen sind.
Seite(n): 55, 235, 276–278, 281, 461

Metametamodell Ein Metamodell zu einem Metamodell.
Seite(n): 287, 461

Metamodell Ein Modell, das beschreibt wie Modelle einer spezifischen Modellierungssprache aufgebaut sind.
Seite(n): 40, 41, 54–61, 69, 70, 76, 86, 106, 124, 165, 166, 209, 220, 221, 224–226, 228, 230–233, 235–280, 282, 285–288, 290, 291, 294, 296, 297, 299, 315–317, 320, 321, 343, 344, 353–359, 366–368, 370, 372, 373, 378, 382, 389, 402–404, 408, 428, 430, 432–435, 456, 461, 473, 477, 478, 480, 487

Meta Object Facility (MOF) Standard der OMG für das modellgetrieben Softwareengineering.
Seite(n): 356, 461, 464

Middleware Ebene einer Softwarearchitektur, die darüberliegende Ebenen (d. h. Dienste und Schnittstelle) und darunterliegenden Schichten abkoppelt, mit dem Ziel, dass individuelle Komponenten austauschbar werden.
Seite(n): 19, 55, 59, 70, 112, 115, 119, 123, 461

Model-driven Architecture (MDA) Modellbasierter Ansatz zur Spezifikation von Softwaresystemen.
Seite(n): 40, 108, 209, 229, 231, 233, 286, 287, 428, 461, 464, 479, siehe MDSD

Model Driven Software Development (MDSD) Methode der Softwareentwicklung, bei der Code für Software nicht manuell geschrieben, sondern aus Modellen generiert wird.
Seite(n): 40–42, 44, 59, 60, 89, 108, 208, 219, 220, 228, 229, 231, 233, 235–238, 240, 250, 286, 428, 435, 461, 464, 479, siehe MDA

Model Driven Testing (MDT) Die Anwendung von Prinzipien des modellgetriebenen Softwareentwurfs auf die Erstellung von Tests.
Seite(n): 17, 41, 85, 87, 89, 108, 207, 208, 222, 227, 229, 231, 233, 236–238, 240, 250, 255, 435, 461, 464

Modelltransformation Werkzeug zur Überführung von Modellen zwischen unterschiedlichen Abstraktionsebenen oder Metamodelle, das unter Anwendung von Transformationsregeln ein Quellmodell auf ein Zielmodell abbildet.
Seite(n): 42, 101, 201, 261, 461, 478

Modell-zu-Modell Transformation (M2M) Variante der Modelltransformation, bei der aus einem oder mehreren Eingabemodellen ein oder mehrere Ausgabemodelle generiert werden.
Seite(n): 42, 461, 464

Modell-zu-Text Transformation (M2M) Sonderform der Modelltransformation, bei der das Ausgabemodell durch Text repräsentiert wird.
Seite(n): 42, 101, 320, 321, 344, 354, 355, 358, 403, 404, 406, 432, 458, 461, 464

Monkeyrunner (Android) Python-basierte Technologie, die eine API zur Erstellung von Tests und zur Steuerung eines Android-Geräts und dem darauf laufenden SUT implementiert.
Seite(n): 96, 97, 459, 461

Nachbedingung, (Test-) Beschreibt der Zustand testrelevanter Parameter nach der Testausführung. Die Übereinstimmung der Nachbedingungen mit der Spezifikation des SUT gibt Auskunft über dessen Korrektheit. [236]
Seite(n): 59, 84, 92, 99, 100, 102, 103, 167, 174, 195, 198, 231, 259–265, 267, 270, 271, 273, 277, 279, 281, 284, 285, 298, 319, 376, 381, 389, 396, 404, 408, 410, 412, 419, 420, 424, 434, 461, 478, siehe Test sowie Vorbedingung

Nighlty Build Kompilat einer Software, das i. d. R. durch ein CI-System zu einem definierten Zeitpunkt, häufig in der Nacht außerhalb regulärer Büroarbeitszeit, erstellt wird, ohne zwingend Gegenstand eines Rollout zu sein.
Seite(n): 195, 198, 461

Nullmeridian Der Nullmeridian (auch Greenwich Meridian) bezeichnet in der geodätischen Kartographie denjenigen Meridian, der die Zählung der geographischen Länge in östliche und westliche Zählrichtung einteilt.
Seite(n): 131, 182, 461

Object Constraint Language (OCL) Deklarative formale Sprache zur Ergänzung von Randbedingungen in UML-Modellen; Teil des UML-Standards. Die UML stellt Elemente für die Modellierung statischer und dynamischer Aspekte von Softwaresystem bereit. Mit Hilfe der OCL können allgemeine Randbedingungen oder spezifische Bedingungen (z. B. Vor- und Nachbedingungen an Zustandsübergängen, Wertebereiche für Attribute usw.) ergänzt werden.
Seite(n): 280, 461, 464, 478

Object Management Group (OMG) Internationales Konsortium, dass sich mit der Spezifikation von Standards im Umfeld objektorientierter Technologien, UML, MDA und MDSD befasst.
Seite(n): 208, 356, 461, 464, 477, 490

Offshoring Outsourcing von Arbeit, nicht jedoch von Arbeitskraft, in Länder mit niedrigem Lohnniveau. [47]
Seite(n): 47, 461, siehe Outsourcing

Open Services Gateway Initiative (OSGi) Spezifikation einer modularen Service-Plattform für die Programmiersprache Java.
Seite(n): 115, 358, 461, 464, 481

Open-Source-Software (OSS) Software, deren Quellcode offengelegt und unter Berücksichtigung unterschiedlicher Lizenzmodelle zur Verwendung und Modifikation freigegeben ist.
Seite(n): 16, 43, 62, 63, 77, 98, 222, 328, 331, 432, 461, 468

Orakel, (Test-) Entität zur Vorhersage von Erwartungswerten bei der Durchführung von Softwaretests.
Seite(n): 87, 88, 127, 152, 207, 222, 223, 265, 271, 404, 406, 461

ortsbasiert Eine technische Einrichtung oder eine Software, die einen Anwendungsfall unter Verwendung des Standorts des Anwenders implementiert.
Seite(n): 15, 29, 30, 55, 57, 75, 94, 105, 108, 120, 121, 129, 130, 134, 148, 168, 182, 184, 188, 190, 235, 243, 461

Outsourcing Langfristige Auslagerung von Unternehmensteilen im Zuge strategischer Wandlungsprozesse zur vorteilhaften Auslagerung von bisher intern erbrachten Leistungen an externe Dienstleister. [62]
Seite(n): 27, 47, 329, 461, 471, 479, siehe Offshoring

Personal Digital Assistant (PDA) Digitales Gerät, das Funktionen wie Telefonie, Management persönlicher Kontakte und Textnachrichten vereint. Vorläufer des Smartphone.
Seite(n): 75, 76, 107, 461

Pinch Zoom (Geste) Eine *Multi-Touch*-Geste auf berührungsempfindlichen Displays, bei der unter Verwendung von zwei Fingern der dargestellte Ausschnitt eines Inhalts durch Annäherung oder Spreizung der Finger angepasst werden kann.
Seite(n): 461

Plattform Technische Grundlage der Anwendungsprogrammierung, die einerseits Implementierungsdetails Hardware-naher Schichten verbirgt und andererseits Funktionen, Schnittstellen oder ein API für die Anwendungsprogrammierung zur Verfügung stellt.
Seite(n): 11–15, 17, 18, 20–26, 28–30, 32–35, 37–39, 42–45, 49, 54, 59–67, 69, 71–73, 76–82, 87–89, 91, 92, 94, 96, 98, 99, 101–104, 106–109, 113–116, 118, 125, 126, 129, 131, 134, 138, 140, 143, 144, 147, 149, 151, 152, 158, 160–163, 167, 171–179, 182, 185, 188,

189, 194, 195, 197, 200, 204, 205, 208, 211–213, 215–222, 225, 227, 230, 241–243, 247–249, 251, 259, 261, 269, 273, 274, 282, 285, 287–289, 292–294, 315, 319–321, 323–325, 327–330, 333, 335, 337, 338, 342–345, 347, 349, 351–356, 358–360, 392, 407, 410, 418, 428, 432, 433, 437, 456, 461, 468–472, 475, 479, 481, 487, 489, 490

Point and Click (P'n'C) Paradigma der Mensch-Computer Interaktion unter Zuhilfenahme eines Zeigegeräts, mit dem Elemente des UI ausgewählt (*Point*) und Aktiviert (*Click*) werden.
Seite(n): 80, 461

Point of Interest (POI) Geographischer Interessenpunkt; Landmarke.
Seite(n): 110, 126, 186, 187, 193, 206, 231, 461

Profil (UML) Im Kontext der UML ist ein Profil eine Erweiterung des UML-Metamodells mit dem Ziel, Modellierungselemente mit domänenspezifischen Erweiterungen verfügbar zu machen.
Seite(n): 41, 42, 55, 56, 59–62, 86, 87, 209, 219, 220, 224–226, 230, 235, 236, 240, 247, 258, 265, 273–276, 281–284, 286, 290, 291, 294, 297, 301, 303, 310, 314, 315, 319, 344, 350, 351, 353–355, 357, 358, 360, 376, 408, 430, 456, 461, 480, 487

Push-Benachrichtigung Als *Push*-Benachrichtigung wird der Versand von Nachrichten bezeichnet, wenn der Informationsfluss vom Backend-System gesteuert wird. Kommunikationsteilnehmer werden vom Versender der Nachricht über das Vorhandensein einer Nachricht aktiv informiert, ein periodisches Abfragen (*Pull*-Technologie) seitens des Empfängers entfällt.
Seite(n): 11, 42, 461, 475

Framework Bündelung einer Menge thematisch miteinander verwandter Software-Funktionen, häufig mit Programmierschnittstelle.
Seite(n): 78, 83, 84, 91, 92, 94, 95, 98, 104, 106, 119, 125, 163, 215, 216, 221, 321, 322, 327–329, 332, 333, 335–337, 341, 342, 344, 345, 347, 353–355, 357, 358, 361, 403, 417, 419, 420, 422, 423, 427, 428, 431, 432, 437, 461, 468, 470, 472, 476, 481, 487

Randbereichstest Softwaretest, der Funktionen oder Methoden gezielt in den Randbereichen ihres jeweiligen Definitionsbereichs ausführt, um Robustheit gegenüber unzulässigen Eingabeparametern zu prüfen.
Seite(n): 202, 461

RTSP Protokoll zur Übertragung von Audio- und Videodaten.
Seite(n): 38, 461, 464

Regression Im Kontext der Softwareentwicklung ein Rückschritt in der Spezifikationskonformität einer Software gegenüber einer vorhergehenden Version, i. d. R. Defekte, die durch die Implementierung neuer Funktionalität in bereits existierendem Code verursacht oder aufgedeckt werden.
Seite(n): 32, 164, 461, siehe Regressionstest

Regressionstest Selektive Wiederholung von Tests um auszuschließen, dass Modifikation eines Systems oder einer Komponente unbeabsichtigte Effekte, z. B. Defekte, verursacht haben und um sicherzustellen, dass das System oder die Komponente weiterhin ihrer Spezifikation entspricht. IEEE [192]
Seite(n): 73, 162, 164, 176, 198, 350, 461, 481, siehe Akzeptanztest, Integrationstest, Komponententest, Regression Testing sowie Systemtest

Regression Testing englische Bezeichnung für Regressionstest
Seite(n): 461, siehe Acceptance Testing, Integration Testing, Regressionstest sowie Unit-Testing

Remote Services for OSGi OSGi-Spezifikation unter Berücksichtigung distanzierter Akteure.
Seite(n): 115, 461, 464

Resource Description Framework (RDF) Familie von W3C-Spezifikationen zur konzeptuellen Beschreibung und Modellierung von Informationen über Entitäten.
Seite(n): 232, 461

Return on Investment Kennzahl zur Bewertung der Rentabilität einer Investition.
Seite(n): 20, 22, 73, 197, 461

Reverse Geocoding Prozess GPS-Koordinatenpaare zu zugehörige Adressinformationen zu ermitteln. Hierzu gehören beispielsweise Straßenname, Hausnummer, Name der Stadt, Postleitzahl.
Seite(n): 130, 147, 251, 252, 414, 461

Rich-Client-Platform Werkzeug oder Framework zu Integration von Funktionalität in existierende Software.
Seite(n): 355, 358, 360, 428, 461, 464

Robotium Testautomatisierungs-Framework für die Plattform Android.
Seite(n): 80, 83, 84, 94, 95, 103, 104, 162, 163, 204, 213, 349, 427, 459, 461

Rollout Prozess der Einführung eines (Software-)Produkts beim Auftraggeber oder Käufer.
Seite(n): 11, 32, 461, 478

Sandbox Technologie zur isolierten Abgrenzung von Software von anderer Software innerhalb einer Systemumgebung, um zu gewährleisten, dass eine Software ihre Betriebsumgebung nicht beeinflusst.
Seite(n): 85, 103, 215, 216, 461

Scenario Artefakt der Strukturierung von Testfällen in der Calabash bzw. Cucumber.
Seite(n): 260, 321, 461

Scrum Vorgehensmodell der Softwareentwicklung mit einem Fokus auf selbstorganisierende Teams in agilen Projektumgebungen.
Seite(n): 201, 273, 461, siehe Vorgehensmodell, V-Modell, TDD sowie Wasserfallmodell

Secure Socket Layer (SSL) Verschlüsselungsprotokoll zur Datenübertragung in Computernetzwerken.
Seite(n): 34, 39, 461, 464

Sensor Technisches Gerät zur Messung physikalischer oder chemischer Eigenschaften seiner Umgebung.
Seite(n): 12, 14, 15, 24, 31, 35, 36, 39, 43, 49, 51–55, 57–61, 63, 69, 73, 75, 76, 79–85, 92–95, 97, 98, 105, 106, 111, 112, 117–119, 121–126, 135–148, 150, 152, 167, 170–172, 175, 181, 190–195, 205, 209, 214, 218, 221, 230, 233, 235, 244–246, 248, 251, 253–256, 264, 268, 270, 275, 277, 283, 322–343, 345, 348–350, 352–355, 361, 362, 364, 404, 406, 407, 418–422, 424, 426–429, 431–435, 437–439, 457, 460, 461, 482

sensorbasiert Eine technische Einrichtung oder eine Software, die einen Anwendungsfall unter Verwendung von Sensoren implementiert.
Seite(n): 15, 461

Server Computersystem mit dem Bestimmungszweck, Dienste für Teilnehmer und Client-Software bereitzustellen.
Seite(n): 14, 20, 23, 31, 32, 36, 71, 98, 114, 148, 177, 196, 238, 461, 469, 470, 472, 489, siehe Backend-System

SMS ***S****hort* ***M****essage* ***S****ervice* ist ein Dienst zu Übertragung von Textnachrichten zwischen Teilnehmern in mobilen Kommunikationsnetzwerken.
Seite(n): 109, 111, 461

Simulator Computersystem; häufig eine Software, die ein anderes System in beobachtbaren Eigenschaften nachbildet. In Abgrenzung zum Emulator wird gefordert, alle relevanten internen Zustände des simulierten Systems originalgetreu nachzubilden.
Seite(n): 36, 461, 473, siehe Emulator

Speech Interpretation and Recognition Interface (Siri) Softwareprodukt des Unternehmens Apple zur Erkennung natürlicher Sprache in einem persönlichen Assistenzsystem.
Seite(n): 114, 461, 464

Smartphone Mobiltelefon, welches in Abgrenzung zu herkömmlichen Mobiltelefonen über Telefonie hinaus Computerfunktionalität zur Verfügung stellt. Smartphones zeichnen sich durch eine vergleichsweise große Display-Fläche sowie hohe Speicherkapazität und Rechenleistung aus. Obwohl der Smartphone-Begriff nicht an einen einheitlichen Funktionsumfang gebunden ist, gehören Fotographie, Email-Versand, Medienwiedergabe und Verkehrsnavigation zum Standardfunktionsumfang von Smartphones.
Seite(n): 1, 11, 13, 15–22, 24, 28, 35, 36, 48–54, 61, 66, 68–70, 74–77, 80, 103, 107, 109, 110, 112, 113, 115, 117, 119, 124, 127, 129–131, 135, 138, 139, 142, 147, 170, 190–192, 212, 230, 232, 234, 235, 241, 245, 264, 268, 269, 348, 349, 361, 461, 467, 468, 479, 482, 484, 489, siehe Smartwatch

Smartwatch Ein technisches Gerät in Bauform einer Armbanduhr mit Smartphone-ähnlichen Funktionen und Eigenschaften.

Seite(n): 107, 115, 136, 142, 143, 177, 461, siehe Smartphone

Smoke Test Softwaretests, mit dem Ziel, solche Defekte offenzulegen, die einen Absturz der Software verursachen.
Seite(n): 78, 79, 82, 205, 461, 483, siehe Acceptance Testing, Integration Testing, Regression Testing, Systemtest, Unit-Testing sowie Load Test

Socket Kommunikationsendpunkt einer bidirektionalen Kommunikation in Computernetzwerken.
Seite(n): 104, 361, 362, 461

Software Development Kit (SDK) Sammlung von Werkzeugen zur Erstellung von Software.
Seite(n): 13, 14, 20, 21, 30, 63, 77–80, 85, 131, 134, 138, 181, 182, 184, 185, 188, 189, 193, 211, 214, 230, 241, 251, 289, 323–325, 327, 330, 332–334, 337, 343, 352, 411, 414, 420, 422, 427, 461, 464, 472

Sollergebnis Definierter Zustand eines Systems nach dem Ausführen einer definierten Funktion.
Seite(n): 165, 167, 205, 206, 461, siehe Orakel

Source Code Management System (SCM) Softwaresystem zur Verwaltung und Versionierung von Quellcode.
Seite(n): 37, 70, 77, 105, 164, 195, 336, 461, 464

Stakeholder Ein Individuum oder einer Gruppe von Individuen mit einem einem spezifischen Interesse am Verlauf und am Erfolg eines Vorhabens. [117]
Seite(n): 41, 43, 50, 53, 73, 74, 90, 113, 227, 228, 231, 259, 327, 341, 461, 469, siehe Bedarfsträger

Stereotyp (UML) Werkzeug der UML zur Erweiterung einer existierende Metaklasse um zusätzliche Eigenschaften. [268]
Seite(n): 55, 226, 235, 236, 273–284, 290, 291, 294, 296–298, 301, 303, 310, 312, 314–317, 357, 360, 367, 370–373, 375, 376, 381, 387–389, 408, 411, 412, 418, 423, 456, 461

Stresstest Variante des Smoke-Test oder Load Test, der ein Softwaresystem gezielt im Randbereich definierter Lastgrenzen und darüber hinaus betreibt, um den Nachweis der Einhaltung nicht-funktionaler Anforderungen zu erbringen. Myers et al. [261]
Seite(n): 91, 93, 461, 477

Stylus (Eingabegerät) Eingabestift oder Griffel zur Interaktion mit Computersystemen, insbesondere mit berühungsempfindlichen Displays.
Seite(n): 461

Swimlane (UML) Der Begriff *Swimlane* bezeichnet in der Systemmodellierung mit der UML die Partitionierung eines UML-Aktivitätsdiagrammes durch Verwendung des Modellelements *UML::ActivityPartition*. *Swimlanes* werden verwendet, um Knotenelemente in

UML-Aktivitätsdiagrammen zu gruppieren. Kriterien für Gruppierungen können technischer oder fachlicher Natur sein. Beispielsweise kann durch die Verwendung von *Swimlanes* modelliert werden, dass Aktionen von bestimmten Akteuren durchzuführen sind.
Seite(n): 461

Systemtest Softwaretests, bei denen das in seine Betriebsumgebung integrierte System oder Produkt in seiner Gesamtheit überprüft wird (IEEE [193, 192]).
Seite(n): 71, 90, 162, 163, 165, 198, 204, 461, siehe Akzeptanztest, Integrationstest, Komponententest sowie Regressionstest

System-Under-Test Ein ***S**ystem-**U**nder-**T**est* ist ein Softwaresystem, dass Gegenstand einer Testdurchführung ist.
Seite(n): 15, 17, 18, 29, 31–33, 35, 37, 39, 40, 42, 43, 48, 69, 71–76, 78–93, 95–105, 120, 143, 152, 155, 161, 162, 165, 166, 173, 183, 184, 187, 190, 194–199, 201, 202, 204–207, 210, 211, 213–218, 221–224, 226, 227, 240, 245, 249–252, 254, 255, 257, 259, 260, 266, 268, 271, 282, 283, 285, 288–291, 294, 298, 310, 315, 319, 322, 323, 325–330, 335–338, 341–344, 350–353, 355, 361–364, 376, 404, 408, 416, 418, 421, 422, 427–431, 433–436, 439, 461, 464, 468, 472, 478, 485, 487

Tablet-Computer Tragbare Computer in leichter, flacher Bauform werden als Tablet-Computer bezeichnet. Von Notebook-Computern werden Sie durch die Abwesenheit einer integrierten Tastatur und einer kleineren Bildschirmdiagonale abgegrenzt, Smartphones durch eine i. d. R. Bildschirmdiagonale. In weiterer Abgrenzung zum Smartphone verfügen Tablets nicht zwingend über eine mobile Datenverbindung oder eine Telefoniefunktion. Es existieren Mischformen zwischen Smartphone und Tablet-Computer die als *Phablet* oder *Smartlet* (jeweils Amalgamierung der Wörter Smartphone bzw. Phone, Tablet-Computer) bezeichnet werden.
Seite(n): 11, 17–20, 24, 28, 48, 103, 109, 110, 113, 115, 191, 211, 212, 241, 461, 467, 484

Taxonomie Eine Taxonomie (gr. τάξις (*táxis*) „Ordnung“ und νόμος (*nómos*) „Gesetz“) ist ein einheitliches Schema, nach welchem Objekte und (linguistisch) Begriffe klassifiziert und (hierarchisch) geordnet und zueinander in Zusammenhang gesetzt werden [100, 227].
Seite(n): 99, 112, 155, 159, 461

Telnet Das ***Tel**etype **Net**work* ist ein Netzwerkprotokoll zur interaktiven, bidirektionalen, textorientierten Verbindung von Computersystemen [70, 286].
Seite(n): 188, 189, 194, 323, 426, 461

Test Strukturierte Ausführung eines Systems oder einer Komponente unter spezifizierten Bedingungen unter Beobachtung und Dokumentation der Ergebnisse mit der Ziel Evaluierung des Systems oder der Komponente [192].
Seite(n): 11–18, 22, 25–27, 29–45, 47, 48, 51–53, 55, 56, 58, 59, 61, 63–99, 101–106, 108, 110, 111, 115, 116, 118, 126–129, 135, 137–139, 143, 144, 146, 147, 152–188, 190–200, 202–227, 229–231, 234–238, 240, 243, 245–247, 249–262, 264, 265, 267, 268, 270–279, 281–286, 288–292, 294, 296, 298, 300–303, 308–310, 312, 314–316, 318–323, 325–338,

340–342, 344, 345, 347–355, 358, 360–364, 371–373, 375–377, 379, 380, 388, 391, 397, 403–408, 410–412, 414–437, 441, 459–461, 468–470, 472, 473, 475, 476, 478–481, 483–490

Testabdeckung Grad, zu dem Tests die spezifizierten Anforderungen an ein System oder eine Komponente adressieren [192]. Maß für die Testvollständigkeit anhand der Abdeckung von Kontrollstrukturen oder des Kontrollflusses [275, 236].
Seite(n): 159, 200, 461

Testart Eine *Testart* beschreibt, in welcher Art und Weise der Testprozess mit welchen konkreten Zielen durchgeführt wird.
Seite(n): 27, 158–160, 163, 164, 205, 461, 485

Testdaten Daten, die bei der Durchführung von Tests als Eingabe in das *System Under Test* (SUT) verwendet werden. Für gewählte Eingabedaten müssen Ausgabedaten bzw. das Verhalten des SUT bekannt sein, damit der Istzustand des SUT mit dem Sollzustand verglichen werden kann.
Seite(n): 31, 34, 43, 48, 52, 66, 83, 86, 87, 90, 106, 120, 146, 152, 156–158, 160, 163, 175, 182–185, 187, 189–194, 198, 199, 201–203, 209, 210, 215, 218–227, 235, 237, 243, 245–251, 253, 254, 256–258, 261, 273, 275, 276, 278, 280, 282–287, 291, 292, 294, 296, 297, 301, 303, 310, 312, 315, 316, 319, 321, 322, 325–328, 330, 333–338, 341, 342, 344, 345, 349–355, 358, 360, 361, 363, 396, 408, 410, 412, 417–419, 421–423, 426, 427, 429–431, 433–436, 461

Test Driven Development (TDD) Vorgehensmodell der Softwareentwicklung, bei dem Tests das primäre Entwicklungsartefakt sind. Entwickler erstellen Tests für ein Feature bevor dessen Implementierung. Es wird kein Quellcode geschrieben, zu dem nicht bereits ein Test existiert. [32]
Seite(n): 159, 162, 195, 199, 200, 215, 461, 464, siehe Vorgehensmodell, V-Modell, Scrum sowie Wasserfallmodell

Testen Prozess des Ausführens eines Systems oder einer Komponente unter spezifizierten Bedingungen unter Beobachtung und Dokumentation der Ergebnisse und Evaluierung des Systems oder der Komponente. Dient der Feststellung von Abweichungen der Implementierung einer Software von ihrer Spezifikation [192].
Seite(n): 11–13, 15–17, 23, 25–27, 29, 31–41, 43–45, 47, 50, 51, 53–55, 58–60, 62, 64–78, 80–102, 104–109, 111, 118, 120, 124, 126–129, 135, 137–139, 143, 145–148, 152–163, 165–168, 171–175, 177–185, 187–192, 194–202, 204–209, 212, 214–222, 225–227, 229, 230, 234, 238, 240, 242, 243, 246, 247, 249, 250, 256–268, 273, 276–278, 283, 285, 287–289, 292, 294, 308, 310, 314, 316, 319, 321–323, 325–329, 333–335, 338, 347–352, 354, 361–363, 376, 379, 391, 399, 401, 405, 407, 411, 417, 418, 421, 426–431, 434–436, 461, 467, 468, 472, 474, 475, 486, 487

Tester, (Software-) Rolle in Softwareprozessen betraut mit der Durchführung von Tests.
Seite(n): 11, 12, 32–34, 36, 37, 39, 69, 72–75, 80, 86, 88, 89, 93, 97, 98, 102, 105, 143, 152, 156, 178, 181, 184, 187, 188, 190–193, 195–199, 206, 210, 212, 214, 216, 218, 250,

268, 276, 314, 326, 335, 343, 351, 362, 391, 396, 416, 417, 421, 426, 430, 461, 469, siehe Test sowie Testen

Testfall Menge von Testeingabedaten, Ausführungsbedingungen und erwarteten Ergebnissen, die mit dem konkreten Ziel erstellt werden, eine Software oder einen Anwendungspfad durch eine Software auf Konformität zur Anforderungsspezifikation zu überprüfen [193, 192].
Seite(n): 12, 16–18, 22, 25, 27, 29–36, 39–45, 50, 53, 55, 57, 59, 60, 62, 67–71, 74, 76, 78, 79, 83–92, 96–106, 111, 120, 126, 127, 138, 143, 154, 156–162, 164–167, 170, 173–175, 178–180, 182–184, 186, 187, 192, 193, 195–205, 207, 209–211, 213–215, 217, 220–227, 229–231, 234–241, 245–247, 252, 253, 257, 258, 260–267, 270–279, 281–283, 285–288, 290–297, 300, 301, 303, 304, 308–312, 314–322, 328, 335–338, 341–344, 347, 350, 351, 353–366, 368–374, 376, 381, 382, 384, 386–388, 391, 392, 394–406, 408, 415–421, 423, 424, 426–435, 437–439, 441, 456, 459–461, 481

Testschritt Individuellen Aktionen in Testplänen bei der Durchführung von Tests.
Seite(n): 99–101, 104, 105, 165–167, 198, 212, 217, 263, 264, 282, 292, 319, 321, 335, 337, 343, 365, 368, 371, 375, 396, 399, 403, 416, 420, 461

Teststrategie Planmäßiges Vorgehen, welche Teile einer Software mit welchem Aufwand und unter Zuhilfenahme welcher Technologien und Werkzeuge zu testen sind [285].
Seite(n): 15, 22, 25, 27, 91, 155–158, 177, 178, 183, 198, 199, 461

Teststufe Klassifikationsmerkmal für Softwaretests. Eine Teststufe bezeichnet eine Menge gemeinsam durchgeführter Testaktivitäten, die bestimmten Abstraktionsebenen eines Softwaresystems zugeordnet werden können.
Seite(n): 73, 91, 93, 98, 158, 159, 162, 164, 167, 195, 198, 200, 202–204, 210, 461, 486

Traffic-Shaping Technologie zur Bevorzugung bestimmter Kommunikationsprotokolle oder Benutzer in Datenübertragungsnetzwerken.
Seite(n): 38, 461

Testsuite Zusammenstellung inhaltlich miteinander verwandter Tests.
Seite(n): 75, 99, 105, 162, 166, 167, 199, 203, 260, 261, 461

Testzyklus Zeitintervall oder Phase in welcher eine definierte Menge von Aktivitäten der Qualitätssicherung ausgeführt werden.
Seite(n): 158, 461

Testplan Spezifikation und Dokumentation von Testaktivitäten.
Seite(n): 29, 31, 42, 155, 156, 158, 160, 461, 486, siehe Test sowie Testen

Thread Ein Thread bezeichnet einen individuellen sequenziellen Kontrollfluss innerhalb eines Prozesses. Ein Prozess bezeichnet einen Adressraum mit mindestens einem Thread und den zu dessen Ausführung benötigten Ressourcen [195].
Seite(n): 112, 205, 461, 486, 487

Timeout Zeitintervall, den ein Thread, ein Prozess, eine Aktivität oder ein Test in Anspruch nehmen darf, bevor mit einem Fehler abgebrochen wird.
Seite(n): 461

Triangulation Messtechnik zur Bestimmung der Position eines Objekts anhand von Signalen, deren Quelle bekannt ist.
Seite(n): 130, 461

Ubiquitous Computing (UC) Bezeichnet die Allgegenwärtigkeit von Computern und deren nahtlose Integration in das alltägliche Lebensumfeld.
Seite(n): 57, 461, 464

UIAutomator Framework zur TESTAUTOMATISIERUNG aus dem Umfeld der JUnit-Familie für die Plattform Android.
Seite(n): 95, 96, 162, 204, 221, 322, 327, 349, 427, 461, siehe JUnit sowie Akzeptanztest

UI/Application Exerciser Monkey (Android) Werkzeug für die Plattform Android, das in der Lage ist, einen pseudo-zufälligen Strom von Anwenderinteraktionen mit dem SUT (z. B. Anklicken von Schaltflächen, Touch-Gesten) oder Interaktionen mit dem Gerät (z.B. Navigation auf den homescreen) zu erzeugen.
Seite(n): 78, 93, 96, 461

UML 2.0 Testing Profile (U2TP) Das UML 2.0 Testing Profile ist ein UML-PROFIL zur Spezifikation von Softwaretests.
Seite(n): 209, 224, 225, 461

Unified Modeling Language (UML) Graphische Modellierungssprache in der Domäne des Software Engineering. Sie definiert Metamodelle für insgesamt 14 Diagrammarten, die strukturelle und dynamische Aspekte eines Softwaresystems auf variierenden Abstraktionsebenen abbilden. [267, 268]
Seite(n): 18, 32, 40–42, 49, 55, 56, 58–62, 85–87, 89, 106, 167, 201, 209, 210, 219–226, 229–231, 233–236, 238–240, 247, 249, 250, 255, 258, 259, 261, 265, 272–277, 279–284, 286, 287, 289, 290, 292–299, 301, 303, 306–308, 310, 312–319, 321, 322, 332, 343–345, 347, 350–361, 364–372, 374–386, 388, 389, 391, 402–404, 406–408, 416, 427, 428, 430, 433, 435, 456, 457, 461, 464, 467, 471, 475, 478–480, 483, 484, 487

Unit-Test Auch als Komponententest bezeichnet der Unit-Test einen Test für Software auf Quellcode[-Ebene], der überprüft, ob eine isolierte Software-Komponente spezifikationskonform implementiert ist. [193, 192]
Seite(n): 71, 78, 83, 90, 152, 159, 162, 192, 195, 196, 198, 207, 215, 216, 322, 404, 405, 461, 472, 476, 488, siehe Komponententest

Unit-Testing Testen von Softwareartefakten auf Code-Ebene, d. h. das Testen individueller Funktionen und Methoden als kleinste testbare Einheiten von Softwaresystem, herausgelöst aus dem Zusammenhang mit den restlichen Komponenten des Softwaresystems. [194, 192]).

Seite(n): 461, siehe Acceptance Testing, Integration Testing, Regression Testing sowie Unit-Test

Universal Mobile Telecommunications System (UMTS) Das *Universal Mobile Telecommunications System* (UMTS) ist eine Technologie zur drahtlosen Vernetzung mobiler Geräte.
Seite(n): 29, 38, 50, 109, 144, 149, 171, 185, 461, 464, siehe GPRS, GSM, LTE sowie EDGE

Urban Canyoning Effekt der Bildung einer Schlucht durch hohe Gebäude in urbanen Gebieten. Durch Abschirmung durch Gebäude kann die Empfangsqualität elektromagnetischer Signale (z. B. des GPS-Signals) beeinträchtigt werden, so dass ein ausreichend gutes GPS-Signal nur von solchen Satelliten empfangen werden kann, die sich nahe der durch die Gebäude gebildete Schlucht verlaufenden Achse befinden. Die Folge ist, dass sich alle Satelliten, deren Signal empfangen werden kann, in ähnlicher Entfernung zum GPS-Empfänger befinden und deren Signallaufzeiten deshalb nur geringfügig voneinander abweichen. Eine Positionsbestimmung kann dann im Vergleich zu weiträumig verteilen Satelliten nur mit geringer Genauigkeit erfolgen. [60, 202]
Seite(n): 38, 132, 214, 461, 474

User Experience (UX) Nutzungserlebnis des Anwenders bei der Interaktion mit Software.
Seite(n): 20, 24, 51, 54, 62, 79, 126, 424, 433, 461, 465

User-Interface (UI) Einheitliche Skala zur Angabe von Zeitinformationen (Weltzeit).
Seite(n): 23, 37, 39, 42, 67, 68, 72, 74, 76, 78–80, 83, 84, 94–96, 98–100, 102–104, 109, 110, 114, 119, 126, 128, 136, 137, 140, 141, 149, 151, 161–163, 165, 168–170, 172–174, 176, 180, 181, 183, 186, 191, 205, 206, 210–216, 218, 231, 244, 250, 254, 260, 261, 263, 269, 271, 278, 279, 282, 284, 294, 308, 309, 331, 349, 351, 356, 364, 396, 404, 408, 410, 411, 413, 414, 418, 420, 422–425, 455, 461, 464, 480, 488

User-Interface-Test Test des UI einer Software.
Seite(n): 164, 165, 461, siehe Akzeptanztest, Komponententest, Systemtest, Integrationstest sowie Regressionstest

V-Modell Das V-Modell ist ein plangetriebenes Vorgehensmodell der Softwareentwicklung, das den Entwicklungsprozess in zunächst absteigenden Abstraktionsstufen von der Anforderungserhebung bis zum Software-Entwurf und anschließend in aufsteigenden Abstraktionsstufen vom Unit-Test bis zur Abnahme abbildet.
Seite(n): 24, 71, 461, siehe Vorgehensmodell, Scrum, TDD sowie Wasserfallmodell

Vorbedingung, (Test-) Die Menge der Vorbedingungen eines Tests beschreibt den Zustand testrelevanter Parameter vor der Testaufführung. [236]
Seite(n): 42, 59, 92, 95, 96, 99, 100, 102, 103, 166, 167, 198, 231, 249, 250, 258–265, 267, 270–273, 275–277, 279, 281–283, 298, 319, 342, 371, 373, 376, 381, 389, 396, 404, 408, 410, 419, 423, 424, 434, 461, siehe Test sowie Nachbedingung

Vorgehensmodell Organisatorisches Hilfsmittel zur Strukturierung einer Aufgabenstellung in Phasen und Artefakte zum Erreichen eines definiertes Projektziels.
Seite(n): 25, 29, 199, 201, 461, 481, 485, 488, siehe V-Modell, Scrum sowie TDD

Wasserfallmodell Plangetriebenes, lineares, nichtiteratives Vorgehensmodell der Softwareentwicklung.
Seite(n): 24, 461, siehe Vorgehensmodell, Scrum, TDD sowie V-Modell

Web-Anwendung Eine Anwendung, die ihre Benutzerschnittstelle in einem Client-Server-Modell über das Internet anbietet.
Seite(n): 58, 107, 114–116, 205, 209, 461

OWL Die *Web Ontology Language* ist eine Spezifikation des W3C zur formalen Beschreibung von Ontologien.
Seite(n): 234, 461, 464

White-Box-Test Ein Test erstellt und durchgeführt unter Kenntnis von Struktur und Implementierungsdetails (z. B. Bezeichnung von Datentyp von Variablen, Namen und Parameter von Methoden) des zu testenden Softwaresystems auf Quellcodeebene. Steht im Kontrast zum Black-Box-Test. Auch *Glass-Box Test* oder *Structural Test*. IEEE [192]
Seite(n): 68, 72, 154, 159–161, 192, 327, 461, 469, 474, siehe Glass-Box-Test, Grey-Box-Test sowie Black-Box-Test

Wi-Fi Bezeichnung sowohl für einen Markenbegriff als auch für ein Firmenkonsortium, das sich mit der Zertifizierung drahtloser Funkschnittstellen befasst. Umgangssprachlich wird der Begriff *WiFi* synonym für *Wireless Local Area Network* (WLAN) benutzt.
Seite(n): 30, 50, 109, 111, 131, 144, 145, 147, 171, 185, 242, 461, siehe WLAN

Windows Phone (Microsoft) Von Microsoft entwickeltes Betriebssystem für Smartphones.
Seite(n): 24, 76, 81, 82, 108, 163, 172, 461, 489, siehe Android sowie Apple iOS

Windows Phone Store Digitaler Marktplatz für Apps für die Plattform Microsoft Windows Phone.
Seite(n): 20, 21, 461, siehe App Store, Apple App Store sowie Google Play Store

Wireless Access Protocol (WAP) *Wireless Application Protocol*, ein Übertragungsstandard, um Inhalte für Mobiltelefone verfügbar zu machen.
Seite(n): 19, 461, 465

Wireless Local Area Network (WLAN) Drahtloses lokales Computernetzwerk.
Seite(n): 461, 465, 489, siehe Wi-Fi

Workaround Gezieltes umgehen eines bekanntes Fehlerverhaltens eines Systems oder einer Systemkomponente bei der Erstellung einer technischen Lösung.
Seite(n): 35, 461

Workflow Ein formalisierter Arbeitsablauf, häufig im Kontext von Computersystemen.
Seite(n): 60, 100, 201–203, 294, 295, 309, 310, 312, 315, 335, 461, 467

World Geodetic System 1984 (WGS84) Das ***W**orld **G**eodetic **S**ystem 1984* (WGS 84) ist ein Standard der Geodäsie und Kartographie. Es definiert ein Referenzsystem für geographische Koordinaten unter Verwendung eines Referenzellipsoids zur Annäherung der Erdoberfläche. Der Koordinatenursprung des *WGS84*-Referenzsystems liegt im Schwerpunkt der Erde. Koordinaten im *WGS84*-System werden als Tripel aus der geograpischen Länge, der geographischen Breite und der Höhe über dem Punkt auf dem Referenzellipsoid angegeben.
Seite(n): 55, 56, 58, 105, 120, 130–134, 147, 149, 182, 186, 203, 231, 241–243, 249–253, 285, 414, 416, 433, 461, 465

Xamarin Eine *Cross-Platform*-Technologie zur Programmierung mobiler Apps für mehrere Plattformen unter einheitlicher Verwendung der Programmiersprache C# mit integrierter Testautomatisierungslösung.
Seite(n): 106, 216, 363, 461, siehe Appium sowie Calabash

XML Metadata Interchange (XMI) XML *Metadata Interchange* ist ein Standard der OMG zur Repräsentierung objektorientierter Informationen in XML. [159]
Seite(n): 247, 356, 435, 461, 465

Literaturverzeichnis

[1] 3rd Generation Partnership Project (3GPP): *Services and Service Capabilities.* – URL `http://www.3gpp.org/ftp/Specs/html-info/22105.htm`. – Zugriffsdatum: 12.15.2015

[2] Abowd, Gregory D. ; Atkeson, Christopher G. ; Hong, Jason ; Long, Sue ; Kooper, Rob ; Pinkerton, Mike: Cyberguide: A Mobile Context-aware Tour Guide. In: *Wireless Networks - Special Issue: Mobile Computing and Networking: Selected Papers from MobiCom '96* 3 (1997), Oktober, Nr. 5, S. 421–433

[3] Abowd, Gregory D. ; Dey, Anind K. ; Brown, Peter J. ; Davies, Nigel ; Smith, Mark ; Steggles, Pete: Towards a Better Understanding of Context and Context-Awareness. In: *Proceedings of the 1st International Symposium on Handheld and Ubiquitous Computing.* London, UK, UK : Springer-Verlag, 1999 (HUC '99), S. 304–307

[4] Adobe Systems Corp.: *Adobe AIR.* – URL `https://play.google.com/store/apps/details?id=com.adobe.air`. – Zugriffsdatum: 15.10.2015

[5] Agarwal, Sharad ; Mahajan, Ratul ; Zheng, Alice ; Bahl, Victor: Diagnosing Mobile Applications in the Wild. In: *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks.* New York, NY, USA : ACM, 2010 (Hotnets-IX), S. 22–28

[6] Amalfitano, Domenico ; Fasolino, Anna R. ; Tramontana, Porfirio: A GUI Crawling-based Technique for Android Mobile Application Testing. In: *IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops (ICSTW)* IEEE (Veranst.), 2011, S. 252–261

[7] Ammann, Paul ; Offutt, Jeff: *Introduction to Software Testing.* 1. New York, NY, USA : Cambridge University Press, 2008

[8] Anand, Saswat ; Naik, Mayur ; Harrold, Mary J. ; Yang, Hongseok: Automated Concolic Testing of Smartphone Apps. In: *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering.* New York, NY, USA : ACM, 2012 (FSE '12), S. 59:1–59:11

[9] Anderson, Theodore W. ; Darling, Donald A.: Asymptotic Theory of Certain Goodness of Fit Criteria Based on Stochastic Processes. In: *The Annals of Mathematical Statistics* (1952), S. 193–212

[10] Andreas Kaiser: *Integration kontinuierlicher Ortsbestimmung und Geofencing in Android Apps*, Universität Duisburg-Essen, Bachelorarbeit, Oktober 2015

[11] Apfelbaum, Larry ; Doyle, John: Model Based Testing. In: *Software Quality Week Conference*, 1997, S. 296–300

[12] AppBrain.com: *Free vs. Paid Android Apps.* – URL http://www.appbrain.com/stats/free-and-paid-android-applications. – Zugriffsdatum: 26.08.2015

[13] AppBrain.com: *Number of Android Applications.* – URL http://www.appbrain.com/stats/number-of-android-apps. – Zugriffsdatum: 26.08.2015

[14] Appium Community: *Appium.* – URL http://appium.io/index.html. – Zugriffsdatum: 15.01.2016

[15] Apple Inc.: *Apple App Store.* – URL http://itunes.apple.com/de/genre/ios/id36?mt=8. – Zugriffsdatum: 26.08.2015

[16] Apple Inc.: *Apple Health.* – URL https://www.apple.com/ios/whats-new/health. – Zugriffsdatum: 26.08.2015

[17] Apple Inc.: *iOS Developer Documentation: Apple Push Notification Service.* – URL https://developer.apple.com/library/ios/documentation/NetworkingInternet/Conceptual/RemoteNotificationsPG/Chapters/ApplePushService.html. – Zugriffsdatum: 23.01.2016

[18] Apple Inc.: *iOS Developer Documentation: UIViewController.* – URL https://developer.apple.com/library/ios/documentation/UIKit/Reference/UIViewController_Class/. – Zugriffsdatum: 04.12.2015

[19] AppThwack.com: *Xamarin Test Cloud.* – URL https://appthwack.com/landing. – Zugriffsdatum: 11.11.2015

[20] Aspray, William ; Mayadas, Frank ; Vardi, Moshe Y.: Globalization and Offshoring of Software. In: *Report of the ACM Job Migration Task Force, Association for Computing Machinery* (2006)

[21] Ayed, Dhouha ; Delanote, Didier ; Berbers, Yolande: MDD Approach for the Development of Context-aware Applications. In: Kokinov, Boicho (Hrsg.) ; Richardson, DanielC. (Hrsg.) ; Roth-Berghofer, ThomasR. (Hrsg.) ; Vieu, Laure (Hrsg.): *Modeling and Using Context* Bd. 4635. Springer Berlin Heidelberg, 2007, S. 15–28

[22] Bach, James: Test Automation Snake Oil. In: *Proceedings of the 14th International Conference and Exposition on Testing Computer Software (TCS'99)*, 1999

[23] Baker, Paul ; Dai, Zhen R. ; Grabowski, Jens ; Haugen, Oystein ; Schieferdecker, Ina ; Williams, Clay: *Model-Driven Testing: Using the UML Testing Profile.* Secaucus, NJ, USA : Springer-Verlag New York, Inc., 2007

[24] Balasubramanian, Niranjan ; Balasubramanian, Aruna ; Venkataramani, Arun: Energy Consumption in Mobile Phones: a Measurement Study and Implications for Network Applications. In: *Proceedings of the 9th ACM SIGCOMM Conference on Internet Measurement* ACM (Veranst.), 2009, S. 280–293

[25] Baldauf, Matthias ; Dustdar, Schahram ; Rosenberg, Florian: A Survey on Context-aware Systems. In: *International Journal of Ad Hoc and Ubiquitous Computing* 2 (2007), Nr. 4, S. 263–277

[26] Ballendat, Till ; Marquardt, Nicolai ; Greenberg, Saul: Proxemic Interaction: Designing for a Proximity and Orientation-aware Environment. In: *ACM International Conference on Interactive Tabletops and Surfaces*. New York, NY, USA : ACM, 2010 (ITS '10), S. 121–130

[27] Balz, Moritz: *Embedding Model Specifications in Object-Oriented Program Code: A Bottom-up Approach for Model-based Software Development*, Universität Duisburg-Essen, Dissertation, 2011

[28] Basili, Victor R. ; Perricone, Barry T.: Software Errors and Complexity: an Empirical Investigation. In: *Communications of the ACM* 27 (1984), Nr. 1, S. 42–52

[29] Baumeister, Hubert ; Koch, Nora ; Kosiuczenko, Piotr ; Wirsing, Martin: Extending Activity Diagrams to Model Mobile Systems. In: Aksit, Mehmet (Hrsg.) ; Mezini, Mira (Hrsg.) ; Unland, Rainer (Hrsg.): *Objects, Components, Architectures, Services, and Applications for a Networked World* Bd. 2591. Springer Berlin Heidelberg, 2003, S. 278–293

[30] Baur, Xaver ; Groth, K: Psychische Belastung und Beanspruchung. In: *Arbeitsmedizin* (2013), S. 71–76

[31] Beadle, HW P. ; Harper, B ; Maguire Jr, Gerald Q. ; Judge, J: Location Aware Mobile Computing. In: *IEEE/IEE International Conference on Telecommunications (ICT'97). Australia, Melbourne. 1997* IEEE (Veranst.), 1997, S. 1319–1324

[32] Beck, Kent: *Test-Driven Development: by Example*. Addison-Wesley Professional, 2003

[33] Beck, Kent ; Beedle, Mike ; Bennekum, Arie van ; Cockburn, Alistair ; Cunningham, Ward ; Fowler, Martin ; Grenning, James ; Highsmith, Jim ; Hunt, Andrew ; Jeffries, Ron ; Kern, Jon ; Marick, Brian ; Martin, Robert C. ; Mellor, Steve ; Schwaber, Ken ; Sutherland, Jeff ; Thomas, Dave: *Manifesto for Agile Software Development*. 2001

[34] Beck, Kent ; Gamma, Erich ; Saff, David ; Clark, Mike: *JUnit*. – URL `http://www.junit.org`. – Zugriffsdatum: 26.08.2015

[35] Becker, Christian ; Dürr, Frank: On Location Models for Ubiquitous Computing. In: *Personal Ubiquitous Comput.* 9 (2005), Januar, Nr. 1, S. 20–31

[36] Beckschebe, Jessica: Möglichkeiten und Grenzen effektiver Internationaler Marketingstrategien dargestellt am Beispiel der Tourismusbranche. (2003)

[37] Beigl, Michael ; Zimmer, Tobias ; Decker, Christian: A Location Model for Communicating and Processing of Context. In: *Personal and Ubiquitous Computing* 6 (2002), S. 341–357. – ISSN 1617-4909

[38] Bellard, Fabrice: *QEMU.* – URL `http://wiki.qemu.org/Main_Page`. – Zugriffsdatum: 25.10.2015

[39] Berger, Matthias: Marketing in der App-Economy. In: *ONLINE: http://shop. marketing-boerse. de* (2012), S. 847

[40] Berner, Stefan ; Weber, Roland ; Keller, Rudolf K.: Observations and Lessons Learned from Automated Testing. In: *Proceedings of the 27th International Conference on Software Engineering* ACM (Veranst.), 2005, S. 571–579

[41] Bertolino, Antonia: Software Testing Research: Achievements, Challenges, Dreams. In: *Future of Software Engineering* IEEE Computer Society (Veranst.), 2007, S. 85–103

[42] Bettini, Claudio ; Brdiczka, Oliver ; Henricksen, Karen ; Indulska, Jadwiga ; Nicklas, Daniela ; Ranganathan, Anand ; Riboni, Daniele: A Survey of Context Modelling and Reasoning Techniques. In: *Pervasive and Mobile Computing* 6 (2010), April, Nr. 2, S. 161–180

[43] Beydeda, S. ; Book, M. ; Gruhn, V.: *Model-Driven Software Development.* Springer London, Limited, 2005

[44] Biffl, Stefan ; Aurum, Aybuke ; Boehm, Barry ; Erdogmus, Hakan ; Grünbacher, Paul: *Value-based Software Engineering.* Springer Science & Business Media, 2006

[45] Bitbar Inc.: *Testdroid.* – URL `http://testdroid.com`. – Zugriffsdatum: 12.01.2016

[46] BlackBerry Ltd.: *BlackBerry.* – URL `http://www.blackberry.com`. – Zugriffsdatum: 01.02.2016

[47] Blinder, Alan S.: Offshoring: the next industrial revolution? In: *Foreign affairs* (2006), S. 113–128

[48] Bo, Jiang ; Xiang, Long ; Xiaopeng, Gao: Mobiletest: A Tool Supporting Automatic Black Box Test for Software on Smart Mobile Devices. In: *Proceedings of the Second International Workshop on Automation of Software Test* IEEE Computer Society (Veranst.), 2007, S. 8

[49] Boehm, Barry: A View of 20th and 21st Century Software Engineering. In: *Proceedings of the 28th International Conference on Software Engineering.* New York, NY, USA : ACM, 2006 (ICSE '06), S. 12–29

[50] Boehm, Barry W.: *Characteristics of Software Quality.* Bd. 1. North-Holland, 1978

[51] Boehm, Barry W. ; Brown, John R. ; Lipow, Mlity: Quantitative Evaluation of Software Quality. In: *Proceedings of the 2nd International Conference on Software Engineering* IEEE Computer Society Press (Veranst.), 1976, S. 592–605

[52] Böhmer, Matthias ; Hecht, Brent ; Schöning, Johannes ; Krüger, Antonio ; Bauer, Gernot: Falling Asleep with Angry Birds, Facebook and Kindle: a Large Scale Study on Mobile Application Usage. In: *Proceedings of the 13th International Conference on Human-congressomputer Interaction with Mobile Devices and Services* ACM (Veranst.), 2011, S. 47–56

[53] Book, Matthias ; Gruhn, Volker ; Hülder, Malte ; Schäfer, Clemens: Der Einfluss verschiedener Mobilitätsgrade auf die Architektur von Informationssystemen. In: *MCTA*, 2005, S. 117–130

[54] Book, Matthias ; Gruhn, Volker ; Hülder, Malte ; Schäfer, Clemens: A Methodology for Deriving the Architectural Implications of Different Degrees of Mobility in Information Systems. In: Fujita, Hamodio (Hrsg.) ; Mejri, Mohamed (Hrsg.): *Proceedings of the Fourth on New Trends in Software Methodologies, Tools and Techniques (SoMeT)* Bd. 129 IOS Press (Veranst.), 2005, S. 281

[55] Book, Matthias ; Gruhn, Volker ; Hülder, Malte ; Schäfer, Clemens: Mobility in e-commerce. In: *Digital Excellence*. Springer, 2008, S. 41–54

[56] Bowen, John B.: Standard Error Classification to Support Software Reliability Assessment. In: *Proceedings of the National Computer Conference*. New York, NY, USA : ACM, Mai 1980 (AFIPS '80), S. 697–705

[57] Bray, Tim ; Paoli, Jean ; Sperberg-McQueen, C M. ; Maler, Eve ; Yergeau, François: Extensible Markup Language (XML). In: *World Wide Web Consortium Recommendation REC* 16 (1998), S. 16

[58] Briand, Lionel ; Labiche, Yvan: A UML-Based Approach to System Testing. In: *Software and Systems Modeling* 1 (2002), S. 10–42

[59] Broens, Tom ; Halteren, Aart van: SimuContext: Simply Simulate Context. In: *Proceedings of the International Conference on Autonomic and Autonomous Systems*. Washington, DC, USA : IEEE Computer Society, 2006 (ICAS '06), S. 45

[60] Broll, Wolfgang ; Ohlenburg, Jan ; Lindt, Irma ; Herbst, Iris ; Braun, Anne-Kathrin: Meeting Technology Challenges of Pervasive Augmented Reality Games. In: *Proceedings of 5th ACM SIGCOMM Workshop on Network and System Support for Games*. New York, NY, USA : ACM, 2006 (NetGames '06)

[61] Brown, P.J. ; Bovey, J.D. ; Chen, Xian: Context-aware Applications: From the Laboratory to the Marketplace. In: *IEEE Personal Communications* 4 (1997), Nr. 5, S. 58–64. – ISSN 1070-9916

[62] Bruch, Heike: *Outsourcing: Konzepte und Strategien, Chancen und Risiken*. Springer-Verlag, 2013

[63] Bukhari, Sundus ; Waheed, Tabinda: Model-driven Transformation Between Design Models to System Test Models using UML: a Survey. In: *Proceedings of the 2010 National Software Engineering Conference*. New York, NY, USA : ACM, 2010 (NSEC '10)

[64] Bundesverband Digitale Wirtschaft (BVDW) e.V.: *Faszination Mobile: Verbreitung, Nutzungsmuster und Trends*. Bundesverband Digitale Wirtschaft in Kooperation mit Google und TNS Infratest (Veranst.), 2014

[65] Business Wire Inc.: *Business Wire*. – URL `http://www.businesswire.com/news/home/20150330005182/en/Worldwide-Wearables-Market-Forecast-Reach-45.7-Million#.VeRc7GCKG-g`. – Zugriffsdatum: 26.08.2015

[66] Bussmann, Hadumod: *Lexikon der Sprachwissenschaft*. Kröner, 2002

[67] Bylund, Markus ; Espinoza, Fredrik: Using Quake III Arena to Simulate Sensors and Actuators When Evaluating and Testing Mobile Services. In: *CHI '01 Extended Abstracts on Human Factors in Comp. Systems*. New York, NY, USA : ACM, 2001 (CHI '01), S. 241–242. – ISBN 1-58113-340-5

[68] Bylund, Markus ; Espinoza, Fredrik: Testing and Demonstrating Context-aware Services with Quake III Arena. In: *Communications of the ACM* 45 (2002), Nr. 1, S. 46–48

[69] Cantor, Georg: Über eine elementare Frage der Mannigfaltigkeitslehre. In: *Jahresbericht der Deutschen Mathematiker-Vereinigung* 1 (1892), S. 75–78

[70] Carr, C. S.: *RFC 15: Network subsystem for time sharing hosts*. September 1969

[71] Carroll, Aaron ; Heiser, Gernot: An Analysis of Power Consumption in a Smartphone. In: *USENIX Annual Technical Conference* Bd. 14, 2010

[72] Carzaniga, Antonio ; Picco, Gian P. ; Vigna, Giovanni: Designing distributed applications with mobile code paradigms. In: *Proceedings of the 19th International Conference on Software engineering* ACM (Veranst.), 1997, S. 22–32

[73] Castelli, Gabriella ; Rosi, Alberto ; Mamei, Marco ; Zambonelli, Franco: A Simple Model and Infrastructure for Context-aware Browsing of the World. In: *Fifth Annual IEEE International Conference on Pervasive Computing and Communications* IEEE (Veranst.), März 2007, S. 229–238

[74] Cavarra, Alessandra ; Davies, Jim ; Jeron, Thierry ; Mournier, L ; Hartman, Alan ; Olvovsky, Sergey: Using UML for Automatic Test Generation. In: *Proceedings of ISSTA* Bd. 15, 2002

[75] Chen, Harry ; Finin, Tim ; Joshi, Anupam: An Ontology for Context-aware Pervasive Computing Environments. In: *The Knowledge Engineering Review* 18 (2003), Nr. 03, S. 197–207

[76] Chen, Harry ; Tolia, Sovrin ; Sayers, Craig ; Finin, Tim ; Joshi, Anupam: Creating Context-aware Software Agents. In: *Innovative Concepts for Agent-Based Systems*. Springer, 2003, S. 186–197

[77] Cheverst, Keith ; Davies, Nigel ; Mitchell, Keith ; Friday, Adrian: Experiences of Developing and Deploying a Context-aware Aourist Guide: the GUIDE Project. In: *Proceedings of the 6th Annual International Conference on Mobile Computing and Networking*. New York, NY, USA : ACM, 2000 (MobiCom '00), S. 20–31

[78] Cheverst, Keith ; Davies, Nigel ; Mitchell, Keith ; Friday, Adrian ; Efstratiou, Christos: Developing a Context-aware Electronic Tourist Guide: Some Issues and Experiences. In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* ACM (Veranst.), 2000, S. 17–24

[79] Chinh Truc Dao: *Anpassung des Android OS zur Ermöglichung kontextsensitiver Anwendungstests außerhalb emulierter Umgebung*, Universität Duisburg-Essen, Bachelorarbeit, Januar 2015

[80] Collofello, James S. ; Balcom, L B.: *A Proposed Causative Software Error Classification Scheme*. Department of Computer Science, College of Engineering & Applied Sciences, Arizona State University, 1984

[81] Color Tiger: *Smart Thermometer*. – URL `https://play.google.com/store/apps/details?id=com.colortiger.thermo`. – Zugriffsdatum: 12.09.2016

[82] Community, JUnit G.: *Getting Started with JUnit*. – URL `https://github.com/junit-team/junit/wiki/Getting-started`. – Zugriffsdatum: 05.02.2016

[83] Cucumber Ltd.: *cucumber*. – URL `https://cucumber.io`. – Zugriffsdatum: 31.08.2015

[84] Cui, Youjing ; Ge, Shuzhi S.: Autonomous Vehicle Positioning with GPS in Urban Canyon Environments. In: *Robotics and Automation, IEEE Transactions on* 19 (2003), Nr. 1, S. 15–25

[85] Dai, Zhen R.: Model-Driven Testing with UML 2.0 / Computing Laboratory, University of Kent. 2004. – Forschungsbericht

[86] Dai, Zhen R. ; Grabowski, Jens ; Neukirchen, Helmut ; Pals, Holger: From Design to Test with UML. In: *Testing of Communicating Systems*. Springer, 2004, S. 33–49

[87] Dantas, Valéria Lelli L. ; Marinho, Fabiana G. ; Costa, Aline Luiza d. da ; Andrade, Rossana: Testing Requirements for Mobile Applications. In: *24th International Symposium on Computer and Information Sciences (ISCIS)* IEEE (Veranst.), 2009, S. 555–560

[88] DB Vertrieb GmbH: *DB Navigator*. – URL `http://www.bahn.de/p/view/buchung/mobil/db-navigator.shtml`. – Zugriffsdatum: 26.08.2015

[89] Decker, M.: Modelling Location-Aware Access Control Constraints for Mobile Workflows with UML Activity Diagrams. In: *Mobile Ubiquitous Computing, Systems, Services and Technologies, 2009. UBICOMM '09. Third International Conference on*, Oktober 2009, S. 263 –268

[90] Dehlinger, Josh ; Dixon, Jeremy: Mobile Application Software Engineering: Challenges and Research Directions. In: *Workshop on Mobile Software Engineering*, 2011

[91] Der Fachdienst Wissenschaft, Museen und Archive im Fachbereich Kultur der Stadt Hagen: *Hagen - Geschichte, Archäologie und Geologie.* – URL `https://www.facebook.com/geschichtehagen`. – Zugriffsdatum: 19.10.2015

[92] Dey, Anind K.: Understanding and Using Context. In: *Personal and Ubiquitous Computing* 5 (2001), Nr. 1, S. 4–7

[93] Dey, Anind K. ; Abowd, Gregory D. ; Salber, Daniel: A Conceptual Framework and a Toolkit for Supporting the Rapid Prototyping of Context-aware A. In: *Hum.-Comput. Interact.* 16 (2001), Dezember, Nr. 2, S. 97–166. – ISSN 0737-0024

[94] Dey, Anind K. ; Sohn, Timothy ; Streng, Sara ; Kodama, Justin: iCAP: Interactive prototyping of context-aware applications. In: *in Proceedings of Pervasive 2006*, 2006, S. 254–271

[95] Dey, Anind K. ; Wac, Katarzyna ; Ferreira, Denzil ; Tassini, Kevin ; Hong, Jin-Hyuk ; Ramos, Julian: Getting Closer: An Empirical Investigation of the Proximity of User to their Smart Phones. In: *Proceedings of the 13th International Conference on Ubiquitous Computing* ACM (Veranst.), 2011, S. 163–172

[96] Dietz, Paul ; Leigh, Darren ; Yerazunis, William: *Cellular Telephone with Ear Proximity Display and Lighting Control.* Januar 24 2005. – US Patent App. 11/041,500

[97] Diewald, Stefan ; Roalter, Luis ; Möller, Andreas ; Kranz, Matthias: Towards a Holistic Approach for Mobile Application Development in Intelligent Environments. In: *Proceedings of the 10th International Conference on Mobile and Ubiquitous Multimedia* ACM (Veranst.), 2011, S. 73–80

[98] Dixon, WJ: Processing Data for Outliers. In: *Biometrics* 9 (1953), Nr. 1, S. 74–89

[99] Djuknic, Goran M. ; Richton, Robert E.: Geolocation and Assisted GPS. In: *Computer* 34 (2001), Nr. 2, S. 123–125

[100] Duden: *Deutsches Universalwörterbuch (German Edition).* Bibliographisches Institut & FA Brockhaus AG, 2001. – ISBN 3411055049

[101] Duden Online: *Duden Online.* – URL `http://www.duden.de/rechtschreibung/App`. – Zugriffsdatum: 2015-10-06

[102] Dürr, Frank ; Rothermel, Kurt: On a Location Model for Fine-Grained Geocast. In: Dey, AnindK. (Hrsg.) ; Schmidt, Albrecht (Hrsg.) ; McCarthy, JosephF. (Hrsg.):

Ubiquitous Computing Bd. 2864. Springer Berlin Heidelberg, 2003, S. 18–35. – ISBN 978-3-540-20301-8

[103] Dustin, Elfriede ; Rashka, Jeff ; Paul, John: *Automated Software Testing: Introduction, Management, and Performance.* Addison-Wesley Professional, 1999

[104] Eidgenössiche Technische Hochschule Zürich: *Remote Services for OSGi.* – URL http://r-osgi.sourceforge.net. – Zugriffsdatum: 26.08.2015

[105] Eissfeller, Bernd ; Teuber, Andreas ; Zucker, Peter: Indoor-GPS: Ist der Satellitenempfang in Gebäuden möglich. In: *ZfV, Zeitschrift für Vermessung* (2005), Nr. 4, S. 130

[106] Electronic Arts: *Real Racing 3.* – URL https://play.google.com/store/apps/details?id=com.ea.games.r3_row. – Zugriffsdatum: 2015-09-01

[107] Ellison, Scott: *Worldwide and US Mobile Applications, Storefronts, and Developer 2010–2014 Forecast and Year-end 2010 Vendor Shares: The "Appification" of Everything.* 2010

[108] eMarketer: Worldwide Mobile Phone Users: H1 2014 Forecast and Comparative Estimates. Januar 2014. – Forschungsbericht

[109] ERGO Versicherungsgruppe AG: *ERGO App.* – URL https://play.google.com/store/apps/details?id=de.ergo.app. – Zugriffsdatum: 26.08.2015

[110] Essl, Georg ; Rohs, Michael: ShaMus: A Sensor-based Integrated Mobile Phone Instrument. In: *Proceedings of the International Computer Music Conference ICMC (2007)* Bd. 40, 2007, S. 50

[111] eTips.com: *eTips London Travel Guide.* – URL https://play.google.com/store/apps/developer?id=eTips.com. – Zugriffsdatum: 19.10.2015

[112] Fenton, Norman ; Bieman, James: *Software Metrics: a Rigorous and Practical Approach.* CRC Press, 2014

[113] FEO Media AB: *Quizduell.* – URL https://play.google.com/store/apps/details?id=se.feomedia.quizkampen.de.lite. – Zugriffsdatum: 22.10.2015

[114] Fewster, Mark ; Graham, Dorothy: *Software Test Automation: Effective Use of Test Execution Tools.* ACM Press/Addison-Wesley Publishing Co., 1999

[115] Fitbit Inc.: *fitbit.* – URL https://www.fitbit.com. – Zugriffsdatum: 26.08.2015

[116] Forbes Magazine, Forbes Inc..: *How Much Do Average Apps Make?.* – URL http://www.forbes.com/sites/tristanlouis/2013/08/10/how-much-do-average-apps-make/. – Zugriffsdatum: 31.08.2015

[117] Freeman, R E.: *Strategic Management: A Stakeholder Approach.* Cambridge University Press, 2010

[118] Fuggetta, Alfonso ; Picco, Gian P. ; Vigna, Giovanni: Understanding code mobility. In: *Software Engineering, IEEE Transactions on* 24 (1998), Nr. 5, S. 342–361

[119] Ganskopp, David C. ; Johnson, Dustin D.: GPS Error in Studies Addressing Animal Movements and Activities. In: *Rangeland Ecology & Management* 60 (2007), Nr. 4, S. 350–358

[120] Geer, David: Eclipse Becomes the Dominant Java IDE. In: *Computer* 38 (2005), Nr. 7, S. 16–18

[121] GitHub Community Project: *Calabash-Android*. – URL `https://github.com/calabash/calabash-android`. – Zugriffsdatum: 31.08.2015

[122] GitHub Community Project: *Calabash-iOS*. – URL `https://github.com/calabash/calabash-ios`. – Zugriffsdatum: 31.08.2015

[123] GitHub Community Project: *Gherkin*. – URL `https://github.com/cucumber/cucumber/wiki/Gherkin`. – Zugriffsdatum: 31.08.2015

[124] Gladwell, Malcolm: *The Tipping Point: How Little Things Can Make a Big Difference*. Little, Brown and Company, 2000

[125] Godwin-Jones, Robert: Emerging Technologies Mobile-computing Trends: Lighter, Faster, Smarter. In: *About Language Learning & Technology* 3 (2008)

[126] Gomez, Lorenzo ; Neamtiu, Iulian ; Azim, Tayyaba ; Millstein, Todd: Reran: Timing-and Touch-sensitive Record and Replay for Android. In: *35th International Conference on Software Engineering (ICSE)* IEEE (Veranst.), 2013, S. 72–81

[127] Google Inc.: *Activity Recognition API*. – URL `https://developers.google.com/android/reference/com/google/android/gms/location/ActivityRecognitionApi`. – Zugriffsdatum: 09.10.2015

[128] Google Inc.: *Andriod Developer Documentation: Activities*. – URL `http://developer.android.com/guide/components/activities.html`. – Zugriffsdatum: 04.12.2015

[129] Google Inc.: *Andriod Developer Documentation: Android Device Monitor*. – URL `https://developer.android.com/studio/profile/monitor.html`. – Zugriffsdatum: 25.09.2016

[130] Google Inc.: *Andriod Developer Documentation: ART and Dalvik*. – URL `https://source.android.com/devices/tech/dalvik/`. – Zugriffsdatum: 06.02.2016

[131] Google Inc.: *Andriod Developer Documentation: Bound Services*. – URL `http://developer.android.com/guide/components/bound-services.html`. – Zugriffsdatum: 04.12.2015

[132] Google Inc.: *Andriod Developer Documentation: Content Providers.* – URL http://developer.android.com/guide/topics/providers/content-providers.html. – Zugriffsdatum: 04.12.2015

[133] Google Inc.: *Andriod Developer Documentation: Fragments.* – URL http://developer.android.com/guide/components/fragments.html. – Zugriffsdatum: 15.01.2016

[134] Google Inc.: *Andriod Developer Documentation: Getevent.* – URL https://source.android.com/devices/input/getevent.html. – Zugriffsdatum: 15.01.2016

[135] Google Inc.: *Andriod Developer Documentation: Supporting Multiple Screens.* – URL http://developer.android.com/guide/practices/screens_support.html. – Zugriffsdatum: 25.01.2016

[136] Google Inc.: *Andriod Developer Documentation: Testing with the Espresso Framework.* – URL https://google.github.io/android-testing-support-library/docs/espresso/index.html. – Zugriffsdatum: 02.02.2016

[137] Google Inc.: *Andriod Developer Documentation: UI/Application Exerciser Monkey.* – URL http://developer.android.com/tools/help/monkey.html. – Zugriffsdatum: 05.02.2016

[138] Google Inc.: *Andriod Developer Documentation: UIAutomator.* – URL http://developer.android.com/training/testing/ui-testing/uiautomator-testing.html. – Zugriffsdatum: 05.02.2016

[139] Google Inc.: *Andriod Source: Codenames, Tags, and Build Numbers.* – URL https://source.android.com/source/build-numbers.html. – Zugriffsdatum: 15.07.2016

[140] Google Inc.: *Andriod Version History.* – URL https://www.android.com/history/. – Zugriffsdatum: 12.04.2016

[141] Google Inc.: *Android Developer Documentation: Monkeyrunner.* – URL http://developer.android.com/tools/help/monkeyrunner_concepts.html

[142] Google Inc.: *Android Developer Documentation: Sensor Stack.* – URL https://source.android.com/devices/sensors/sensor-stack.html. – Zugriffsdatum: 15.04.2016

[143] Google Inc.: *Android Developer Documentation: Sensors Overview.* – URL http://developer.android.com/guide/topics/sensors/sensors_overview.html. – Zugriffsdatum: 02.02.2016

[144] Google Inc.: *Android Developer Documentation: Services.* – URL http://developer.android.com/guide/components/services.html. – Zugriffsdatum: 04.12.2015

[145] Google Inc.: *Android Developer Documentation: Testing Fundamentals.* – URL http://developer.android.com/tools/testing/testing_android.html. – Zugriffsdatum: 11.01.2016

[146] Google Inc.: *Android Developer Documentation: UI/Application Exerciser Monkey.* – URL http://developer.android.com/tools/help/monkey.html

[147] Google Inc.: *Android Open Source Project.* – URL https://source.android.com. – Zugriffsdatum: 11.04.2016

[148] Google Inc.: *FusedLocationProvider API.* – URL https://developers.google.com/android/reference/com/google/android/gms/location/FusedLocationProviderApi. – Zugriffsdatum: 22.10.2015

[149] Google Inc.: *Google Fit.* – URL https://fit.google.com. – Zugriffsdatum: 26.08.2015

[150] Google Inc.: *Google Fit.* – URL https://play.google.com/store/apps/details?id=com.google.android.apps.fitness. – Zugriffsdatum: 2015-09-06

[151] Google Inc.: *Google Kamera.* – URL https://play.google.com/store/apps/details?id=com.google.android.GoogleCamera. – Zugriffsdatum: 01.10.2015

[152] Google Inc.: *Google Maps Android API.* – URL https://developers.google.com/maps/documentation/android-api/. – Zugriffsdatum: 19.10.2015

[153] Google Inc.: *Google Play-Dienste.* – URL https://play.google.com/store/apps/details?id=com.google.android.gms. – Zugriffsdatum: 15.10.2015

[154] Google Inc.: *Goolge Play Store.* – URL https://play.google.com/store. – Zugriffsdatum: 26.08.2015

[155] Graham, B.A.: *Improvement in telegraphy.* März 7 1876. – URL http://www.google.com/patents/US174465. – US Patent 174,465

[156] Grassi, Vincenzo ; Mirandola, Raffaela ; Sabetta, Antonino: A UML Profile to Model Mobile Systems. In: *UML 2004 –The Unified Modeling Language.* Springer, 2004, S. 128–142

[157] Griebe, Tobias ; Hesenius, Marc ; Gesthüsen, Marc ; Gruhn, Volker: Test Automation for Speech-Based Applications. In: *New Trends in Software Methodologies, Tools and Techniques: Proceedings of the Fifteenth SoMeT_16* 286 (2016), S. 85

[158] Grønli, T.-M. ; Hansen, J. ; Ghinea, G. ; Younas, M.: Mobile Application Platform Heterogeneity: Android vs Windows Phone vs iOS vs Firefox OS. In: *Advanced Information Networking and Applications (AINA), 2014 IEEE 28th International Conference on*, May 2014, S. 635–641

[159] Grose, Timothy J. ; Doney, Gary C. ; Brodsky, Stephen A.: *Mastering XMI: Java Programming with Xmi, XML and UML.* Bd. 21. John Wiley & Sons, 2002

[160] Grubbs, Frank E.: Sample criteria for testing outlying observations. In: *The Annals of Mathematical Statistics* (1950), S. 27–58

[161] Gruhn, Volker ; Pieper, Daniel ; Röttgers, Carsten: *MDA*. 2006

[162] Guerra, Esther: Specification-driven Test Generation for Model Transformations. In: *Theory and Practice of Model Transformations*. Springer, 2012, S. 40–55

[163] Gupta, Priya ; Surve, Prafullakumar: Model-based Approach to Assist Test Case Creation, Execution, and Maintenance for Test Automation. In: *Proceedings of the First International Workshop on End-to-End Test Script Engineering*. New York, NY, USA : ACM, 2011 (ETSE '11), S. 1–7

[164] Haller, Klaus: Mobile Testing. In: *SIGSOFT Software Engineering Notes* 38 (2013), November, Nr. 6, S. 1–8

[165] Ham, Hyung K. ; Park, Young B.: Mobile Application Compatibility Test System Design for Android Fragmentation. In: Kim, Tai-hoon (Hrsg.) ; Adeli, Hojjat (Hrsg.) ; Kim, Haeng-kon (Hrsg.) ; Kang, Heau-jo (Hrsg.) ; Kim, KyungJung (Hrsg.) ; Kiumi, Akingbehin (Hrsg.) ; Kang, Byeong-Ho (Hrsg.): *Software Engineering, Business Continuity, and Education* Bd. 257. Springer Berlin Heidelberg, 2011, S. 314–320

[166] Han, Dan ; Zhang, Chenlei ; Fan, Xiaochao ; Hindle, Abram ; Wong, Kenny ; Stroulia, Eleni: Understanding Android Fragmentation with Topic Analysis of Vendor-specific Bugs. In: *19th Working Conference on Reverse Engineering (WCRE)* IEEE (Veranst.), 2012, S. 83–92

[167] Harman, Mark ; Jia, Yue ; Zhang, Yuanyuan: App Store Mining and Analysis: MSR for App Stores. In: *Proceedings of the 9th IEEE Working Conference on Mining Software Repositories*. Piscataway, NJ, USA : IEEE Press, 2012 (MSR '12), S. 108–111

[168] Harrison, Chris ; Amento, Brian ; Kuznetsov, Stacey ; Bell, Robert: Rethinking the Progress Bar. In: *Proceedings of the 20th Annual ACM Symposium on User Interface Software and Technology*. New York, NY, USA : ACM, 2007 (UIST '07), S. 115–118

[169] Harrison, Chris ; Hudson, Scott E.: Abracadabra: Wireless, High-precision, and Unpowered Finger Input for Very Small Mobile Devices. In: *Proceedings of the 22Nd Annual ACM Symposium on User Interface Software and Technology*. New York, NY, USA : ACM, 2009 (UIST '09), S. 121–124

[170] Harrold, Mary J.: Testing: A Roadmap. In: *Proceedings of the Conference on The Future of Software Engineering*. New York, NY, USA : ACM, 2000 (ICSE '00), S. 61–72

[171] Hartman, Alan ; Katara, Mika ; Olvovsky, Sergey: Choosing a Test Modeling Language: A Survey. In: *Hardware and Software, Verification and Testing*. Springer, 2007, S. 204–218

[172] Hartmann, Jean ; Vieira, Marlon ; Foster, Herbert ; Ruder, Axel: A UML-based Approach to System Testing. In: *Innovations in Systems and Software Engineering* 1 (2005), S. 12–24

[173] Heckel, Reiko ; Lohmann, Marc: Towards Model-Driven Testing. In: *Electronic Notes in Theoretical Computer Science* 82 (2003), Nr. 6, S. 33 – 43

[174] Heinecke, Andreas ; Bruckmann, Tobias ; Griebe, Tobias ; Gruhn, Volker: Generating Test Plans for Acceptance Tests From UML Activity Diagrams. In: *17th IEEE International Conference and Workshops on Engineering of Computer Based Systems (ECBS)* IEEE (Veranst.), 2010, S. 57–66

[175] Heiskanen, Weikko A.: The Earth and its Gravity Field. (1958)

[176] Henricksen, Karen ; Indulska, Jadwiga: Modelling and Using Imperfect Context Information. In: *Proceedings of the Second IEEE Annual Conference on Pervasive Computing and Communications Workshops* IEEE (Veranst.), 2004, S. 33–37

[177] Henricksen, Karen ; Indulska, Jadwiga: Developing Context-aware Pervasive Computing Applications: Models and Approach. In: *Pervasive Mobile Computing* 2 (2006), Februar, Nr. 1, S. 37–64. – ISSN 1574-1192

[178] Henricksen, Karen ; Indulska, Jadwiga ; Rakotonirainy, Andry: Modeling Context Information in Pervasive Computing Systems. In: *Pervasive Computing*. Springer, 2002, S. 167–180

[179] Henze, N.: *Stochastik für Einsteiger: eine Einführung in die faszinierende Welt des Zufalls*. Vieweg + Teubner, 2010. – ISBN 9783834808158

[180] Hering, Ekbert ; Schönfelder, Gert: Sensoren in Wissenschaft und Technik. In: *Funktionsweise und Einsatzgebiete* (2012), S. 323

[181] Herrington, Jack: *Code Generation in Action*. Greenwich, CT, USA : Manning Publications Co., 2003. – ISBN 1930110979

[182] Hofer, Thomas ; Schwinger, Wieland ; Pichler, Mario ; Leonhartsberger, Gerhard ; Altmann, Josef ; Retschitzegger, Werner: Context-awareness on Mobile Devices – The Hydrogen Approach. In: *Proceedings of the 36th Annual Hawaii International Conference on System Sciences* IEEE (Veranst.), 2003, S. 10–pp

[183] Hoffman, Douglas: Cost Benefits Analysis of Test Automation. In: *STAR West* (1999)

[184] Hoffman, Douglas: Test Automation Architectures: Planning for Test Automation. In: *Quality Week*, 1999, S. 37–45

[185] Holl, Konstantin ; Elberzhager, Frank: A Mobile-specific Failure Classification and its Usage to Focus Quality Assurance. In: *40th EUROMICRO Conference on Software Engineering and Advanced Applications (SEAA)* IEEE (Veranst.), 2014, S. 385–388

[186] Holzer, Adrian ; Ondrus, Jan: Trends in mobile application development. In: *Mobile Wireless Middleware, Operating Systems, and Applications-Workshops* Springer (Veranst.), 2009, S. 55–64

[187] Howe, Jeff: The Rise of Crowdsourcing. In: *Wired magazine* 14 (2006), Nr. 6, S. 1–4

[188] Hu, Cuixiong ; Neamtiu, Iulian: Automating GUI Testing for Android Applications. In: *Proceedings of the 6th International Workshop on Automation of Software Test* ACM (Veranst.), 2011, S. 77–83

[189] Hu, Haibo ; Lee, Dik-Lun: Semantic Location Modeling for Location Navigation in Mobile Environments. In: *Proceeding of the International Conference on Mobile Data Management*, 2004, S. 52 – 61

[190] Hülder, Malte: *Context aware support for mobile information systems.*, University of Leipzig, Dissertation, 2010

[191] Ichikawa, Fumiko ; Chipchase, Jan ; Grignani, Raphael: Where's the phone? A Study of Mobile Phone Location in Public Spaces. In: *2nd Asia Pacific Conference on Mobile Technology, Applications and Systems* IEEE Computer Society Press (Veranst.), 2005, S. 1–8

[192] IEEE: IEEE Standard Glossary of Software Engineering Terminology. In: *IEEE Std 610.12-1990* (1990), Dezember, S. 1–84

[193] IEEE: IEEE Standard for Software and System Test Documentation. In: *IEEE Std 829-2008* (2008)

[194] IEEE Standards Board: IEEE Standard for Software Unit Testing. In: *ANSI/IEEE Std 1008-1987* (1986)

[195] IEEE Standards Board: IEEE Standard for Information Technology - Portable Operating System Interface (POSIX(R)) - Base Definitions. In: *IEEE Std 1003.1, 2004 Edition* (2004), April, S. 1–3760

[196] IEEE Standards Board: IEEE Standard for Software Reviews and Audits. In: *IEEE STD 1028-2008* (2008), Aug, S. 1–52

[197] Im, Kyungsoo ; Im, Tacksoo ; McGregor, John D.: Automating Test Case Definition Using a Domain Specific Language. In: *Proceedings of the 46th Annual Southeast Regional Conference on XX*. New York, NY, USA : ACM, 2008 (ACM-SE 46), S. 180–185

[198] Intelligent Apps GmbH: *mytaxi.* – URL `https://de.mytaxi.com/index.html`. – Zugriffsdatum: 15.10.2015

[199] ISO/IEC: *ISO/IEC 9126. Software Engineering – Product quality.* ISO/IEC, 2001

[200] ISO/IEC: *ISO/IEC 19773:2011. Information Technology – Metadata Registries (MDR) Modules.* ISO/IEC, 2011

[201] ITV Studios Germany GmbH: *Quizduell im Ersten.* – URL `https://play.google.com/store/apps/details?id=de.itv.quizduellimersten`. – Zugriffsdatum: 22.10.2015

[202] Jabbour, M. ; Bonnifait, P. ; Cherfaoui, V.: Enhanced Local Maps in a GIS for a Precise Localisation in Urban Areas. In: *Intelligent Transportation Systems Conference, 2006. ITSC '06. IEEE*, September 2006, S. 468 –473

[203] Jeub, Marco ; Herglotz, Christian ; Nelke, Christoph ; Beaugeant, Christophe ; Vary, Peter: Noise Reduction for Dual-Microphone Mobile Phones Exploiting Power Level Differences. In: *IEEE International Conference On Acoustics, Speech and Signal Processing (ICASSP)* IEEE (Veranst.), 2012, S. 1693–1696

[204] Joorabchi, Mona E. ; Mesbah, Ali ; Kruchten, Philippe: Real Challenges in Mobile App Development. In: *International Symposium on Empirical Software Engineering and Measurement* ACM/IEEE (Veranst.), 2013, S. 15–24

[205] Judd, Glenn ; Steenkiste, Peter: Providing Contextual Information to Pervasive Computing Applications. In: *Proceedings of the First IEEE International Conference on Pervasive Computing and Communications*. Washington, DC, USA : IEEE Computer Society, 2003 (PERCOM '03)

[206] Jung, Christian ; Feth, Denis ; Elrakaiby, Yehia: Automatic Derivation of Context Descriptions. In: *IEEE International Intedisciplinary Conference on Cognitive Methods in Situation Awareness and Decision Support (CogSIMA)* IEEE (Veranst.), 2015, S. 70–76

[207] Jung, Hyo T. ; Joo, Sang H.: Transformation of an Activity Model into a Colored Petri Net Model. In: *Trendz in Information Sciences Computing (TISC), 2010*, Dezember 2010, S. 32 –37

[208] Kaaranen, Heikki: *UMTS Networks: Architecture, Mobility and Services*. John Wiley & Sons, 2005

[209] Kaasila, Jouko ; Ferreira, Denzil ; Kostakos, Vassilis ; Ojala, Timo: Testdroid: Automated Remote UI Testing on Android. In: *Proceedings of the 11th International Conference on Mobile and Ubiquitous Multimedia* ACM (Veranst.), 2012, S. 28

[210] Kähäri, Markus ; Murphy, David J.: Mara: Sensor-based Augmented Reality System for Mobile Imaging Device. In: *5th IEEE and ACM International Symposium on Mixed and Augmented Reality*, 2006

[211] Kano, Noriaki ; Seraku, Nobuhiko ; Takahashi, Fumio ; Tsuji, Shinichi: Attractive Quality and Must-be Quality. (1984)

[212] Karhu, K. ; Repo, T. ; Taipale, O. ; Smolander, K.: Empirical Observations on Software Testing Automation. In: *International Conference on Software Testing Verification and Validation (ICST)*, April 2009, S. 201–209

[213] Kasurinen, Jussi ; Taipale, Ossi ; Smolander, Kari: Software Test Automation in Practice: Empirical Observations. In: *Advances in Software Engineering* (2010)

[214] Katzakis, Nicholas ; Hori, Masahiro: Mobile Devices as Multi-DOF Controllers. In: *IEEE Symposium on 3D User Interfaces (3DUI)* IEEE (Veranst.), 2010, S. 139–140

[215] Kawaguchi, Kohsuke: *Jenkins.* – URL `https://jenkins.io`. – Zugriffsdatum: 27.9.2016

[216] Kelly, Steven ; Tolvanen, Juha-Pekka: *Domain-specific Modeling: Enabling Full Code Generation.* John Wiley & Sons, 2008

[217] Ketabdar, Hamed ; Jahanbekam, Amirhossein ; Yuksel, Kamer A. ; Hirsch, Tobias ; Haji Abolhassani, Amin: MagiMusic: Using Embedded Compass (Magnetic) Sensor for Touch-less Gesture Based Interaction with Digital Music Instruments in Mobile Devices. In: *Proceedings of the Fifth International Conference on Tangible, Embedded, and Embodied Interaction* ACM (Veranst.), 2011, S. 241–244

[218] Ketabdar, Hamed ; Moghadam, Peyman ; Naderi, Babak ; Roshandel, Mehran: Magnetic Signatures in Air for Mobile Devices. In: *Proceedings of the 14th International Conference on Human-computer Interaction with Mobile Devices and Services Companion* ACM (Veranst.), 2012, S. 185–188

[219] Ketabdar, Hamed ; Yüksel, Kamer A. ; Roshandel, Mehran: MagiTact: Interaction with Mobile Devices Based on Compass (Magnetic) Sensor. In: *Proceedings of the 15th International Conference on Intelligent user Interfaces* ACM (Veranst.), 2010, S. 413–414

[220] Khalid, Haliyana: On Identifying User Complaints of iOS Apps. In: *35th International Conference on Software Engineering (ICSE)* IEEE (Veranst.), 2013, S. 1474–1476

[221] Khalid, Haliyana ; Shihab, Emad ; Nagappan, Meiyappan ; Hassan, Asif: What Do Mobile App Users Complain About? A Study on Free iOS Apps. (2014)

[222] Khalid, Hammad ; Nagappan, Meiyappan ; Shihab, Emad ; Hassan, Ahmed E.: Prioritizing the Devices to Test Your App on: A Case Study of Android Game Apps. In: *Proceedings of the 22nd International Symposium on Foundations of Software Engineering (ACM SIGSOFT)* ACM (Veranst.), 2014, S. 610–620

[223] Kim, Hee-Woong ; Lee, Hyun L. ; Son, Jung E.: An Exploratory Study on the Determinants of Smartphone App Purchase. In: *The 11th International DSI and the 16th APDSI Joint Meeting, Taipei, Taiwan*, 2011

[224] Kinvey, Inc.: State of Enterprise Mobility / Kinvey, Inc. 2014. – Forschungsbericht

[225] Kleinrock, Leonard: Nomadic Computing — an Apportunity. In: *ACM SIGCOMM Computer Communication Review* 25 (1995), Nr. 1, S. 36–40

[226] Kleinrock, Leonard: Nomadicity: Anytime, Anywhere in a Disconnected World. In: *Mobile Networks and Applications* 1 (1996), Nr. 4, S. 351–357

[227] Kluge, Friedrich: *Etymologisches Warterbuch Der Deutschen Sprache (German Edition).* Walter de Gruyter, 2002. – ISBN 3110174723

[228] Korpipää, Panu ; Jani, M ; Kela, Juha ; Malm, Esko-Juhani et al.: Managing Context Information in Mobile Devices. In: *IEEE Pervasive Computing* (2003), Nr. 3, S. 42–51

[229] Korpipää, Panu ; Mäntyjärvi, Jani: An Ontology for Mobile Device Sensor-based Context Awareness. In: *Modeling and Using Context*. Springer, 2003, S. 451–458

[230] Kotler, Philip ; Burton, Suzan ; Deans, Kenneth ; Brown, Linen ; Armstrong, Gary: *Marketing*. Pearson Higher Education AU, 2012

[231] Kouhi, Ross G.: *Proximity and Ambient Light Monitor*. November 4 1997. – US Patent 5,684,294

[232] Landesrundfunkanstalten der ARD: *Quizduell*. – URL `http://www.daserste.de/unterhaltung/quiz-show/quizduell/index.html`. – Zugriffsdatum: 22.10.2015

[233] Lange, Christian F. ; Chaudron, Michel R. ; Muskens, Johan: In Practice: UML Software Architecture and Design Description. In: *Software, IEEE* 23 (2006), Nr. 2, S. 40–46

[234] Lehman, Meir M.: Programs, Life Cycles, and Laws of Software Evolution. In: *Proceedings of the IEEE* 68 (1980), Nr. 9, S. 1060–1076

[235] LemonClip: *Kompass Wasserwaage*. – URL `https://play.google.com/store/apps/details?id=com.jee.level`. – Zugriffsdatum: 24.09.2016

[236] Liggesmeyer, Peter: *Software-Qualität: Testen, Analysieren und Verifizieren von Software*. Springer Science & Business Media, 2009

[237] Lima Sky LLC: *Doodle Jump*. – URL `https://play.google.com/store/apps/details?id=com.lima.doodlejump`. – Zugriffsdatum: 01.09.2015

[238] Link, Johannes: *Softwaretests mit JUnit: Techniken der testgetriebenen Entwicklung*. 2. Heidelberg : dpunkt, 2005. – ISBN 978-3-89864-325-2

[239] Linzhang, Wang ; Jiesong, Yuan ; Xiaofeng, Yu ; Jun, Hu ; Xuandong, Li ; Guo, Zheng: Generating Test Cases from UML Activity Diagram Based on Gray-Box Method. In: *11th Asia-Pacific Software Engineering Conference* IEEE (Veranst.), 2004, S. 284–291

[240] Liu, Chien-Hung ; Lu, Chien-Yu ; Cheng, Shan-Jen ; Chang, Koan-Yuh ; Hsiao, Yung-Chia ; Chu, Weng-Ming: Capture-Replay Testing for Android Applications. In: *International Symposium on Computer, Consumer and Control (IS3C)*, Juni 2014, S. 1129–1132

[241] Lukas, Bryan A. ; Ferrell, OC: The effect of market orientation on product innovation. In: *Journal of the academy of marketing science* 28 (2000), Nr. 2, S. 239–247

[242] Marc Gesthüsen: *Testautomatisierung sensorbasierter mobiler Anwendungen mit Calabash-Android*, Universität Duisburg-Essen, Bachelorarbeit, Juli 2014

[243] Marc Gesthüsen: *Simulation von Sensordaten zum automatisierten Testen von Android Anwendungen*, Universität Duisburg-Essen, Masterarbeit, November 2016

[244] Mayor, R. ; Piemonte, P. ; Huang, R.K. ; Patel, P.: *Magnetometer Accuracy and Use*. November 22 2011. – URL `https://www.google.com/patents/US8061049`. – US Patent 8,061,049

[245] Meschkowski, Herbert: *Georg Cantor: Leben, Werk und Wirkung*. BI-Wiss.-Verl., 1983

[246] Meszaros, Gerard ; Smith, ShaunM. ; Andrea, Jennitta: The Test Automation Manifesto. In: Maurer, Frank (Hrsg.) ; Wells, Don (Hrsg.): *Extreme Programming and Agile Methods - XP/Agile Universe 2003* Bd. 2753. Springer Berlin Heidelberg, 2003, S. 73–81. – ISBN 978-3-540-40662-4

[247] Microsoft Corporation: *Windows Phone Developer Updates & Offers from Mobile World Congress*. – URL `http://blogs.windows.com/buildingapps/2013/02/25/`. – Zugriffsdatum: 26.08.2015

[248] Microsoft Corporation: *Windows Phone Store*. – URL `http://www.windowsphone.com/de-de/store`. – Zugriffsdatum: 26.08.2015

[249] Milette, Greg ; Stroud, Adam: *Professional Android Sensor Programming*. John Wiley & Sons, 2012

[250] Milgram, Paul ; Takemura, Haruo ; Utsumi, Akira ; Kishino, Fumio: Augmented Reality: A Class of Displays on the Reality-Virtuality Continuum. In: *Photonics for Industrial Applications* International Society for Optics and Photonics (Veranst.), 1995, S. 282–292

[251] Mingsong, Chen ; Xiaokang, Qiu ; Xuandong, Li: Automatic Test Case Generation for UML Activity Diagrams. In: *Proceedings of the 2006 International Workshop on Automation of Software Test*. New York, NY, USA : ACM, 2006 (AST '06), S. 2–8

[252] Mirzaei, Nariman ; Malek, Sam ; Păsăreanu, Corina S. ; Esfahani, Naeem ; Mahmood, Riyadh: Testing Android Apps Through Symbolic Execution. In: *SIGSOFT Software Engineering Notes* 37 (2012), November, Nr. 6, S. 1–5

[253] Mitteldeutscher Rundfunk AG: *MDR Nachrichten*. – URL `https://play.google.com/store/apps/developer?id=Mitteldeutscher+Rundfunk`. – Zugriffsdatum: 22.10.2015

[254] Möller, Andreas ; Michahelles, Florian ; Diewald, Stefan ; Roalter, Luis ; Kranz, Matthias: Update behavior in app markets and security implications: A case study in google play. In: *Proceedings of the 3rd International Workshop on Research in the Large. Held in Conjunction with Mobile HCI.*, 2012, S. 3–6

[255] Moore, Gordon: Moore's law. In: *Electronics Magazine* (1965)

[256] Morla, Ricardo ; Davies, Nigel: Evaluating a Location-based Application: A Hybrid Test and Simulation Environment. In: *IEEE Pervasive computing* (2004), Nr. 3, S. 48–56

[257] Morla, Ricardo ; Davies, Nigel: Modeling and Simulation of Context-Aware Mobile Systems. In: *IEEE Pervasive Computing* (2004), S. 48–56

[258] Muccini, Henry ; Di Francesco, Antonio ; Esposito, Patrizio: Software Testing of Mobile Applications: Challenges and Future Research Directions. In: *Proceedings of the 7th International Workshop on Automation of Software Test*. Piscataway, NJ, USA : IEEE Press, 2012 (AST '12), S. 29–35

[259] Murphy, Amy L. ; Picco, Gian P. ; Roman, Gruia-Catalin: Lime: A middleware for physical and logical mobility. In: *Distributed Computing Systems, 2001. 21st International Conference on*. IEEE (Veranst.), 2001, S. 524–533

[260] Myers, Brad A.: The Importance of Percent-done Progress Indicators for Computer-human Interfaces. In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. New York, NY, USA : ACM, 1985 (CHI '85), S. 11–17

[261] Myers, Glenford J. ; Sandler, Corey ; Badgett, Tom: *The Art of Software Testing*. John Wiley & Sons, 2011

[262] North, Dan et al.: Introducing bdd. In: *Better Software, March* (2006)

[263] Nüttgens, Markus ; Rump, Frank J.: Syntax und Semantik Ereignisgesteuerter Prozessketten (EPK). In: *Promise* Bd. 2, 2002, S. 64–77

[264] Offutt, Jeff ; Abdurazik, Aynur: Generating Tests from UML Specifications. In: *UML'99—The Unified Modeling Language*. Springer, 1999, S. 416–429

[265] Oh, Gyesik ; Hong, Yoo S.: The Impact of Platform Update Interval on Platform Diffusion in a Cooperative Mobile Ecosystem. In: *Journal of Intelligent Manufacturing* (2015), S. 1–10

[266] Ole Meyer: *Entwicklung eines Frameworks zum Testen standortbasierter Anwendungen*, Universität Duisburg-Essen, Bachelorarbeit, Juni 2016

[267] OMG: OMG Unified Modeling Language (OMG UML), Infrastructure Specification (Version 2.4.1) / Object Management Group. http://www.omg.org/spec/UML/2.4.1/, August 2011 (OMG Document Number: formal/2011-08-05). – Forschungsbericht

[268] OMG: OMG Unified Modeling Language (OMG UML), Superstructure Specification (Version 2.5) / Object Management Group. März 2015 (OMG Document Number:formal/15-03-01). – Forschungsbericht

[269] OpenIntents: *SensorSimulator*. – URL `https://github.com/openintents/sensorsimulator`. – Zugriffsdatum: 10.11.2015

[270] OpenIntents: *SensorSimulator*. – URL `http://openintents.googlecode.com/svn/images/screenshots/Release-2.0/sensorsimulator/`. – Zugriffsdatum: 15.04.2016

[271] OpenSignal Inc.: *OpenSignal*. – URL `http://opensignal.com/reports/2015/08/android-fragmentation/`. – Zugriffsdatum: 15.10.2015

[272] Păsăreanu, Corina S. ; Rungta, Neha: Symbolic PathFinder: symbolic execution of Java bytecode. In: *Proceedings of the IEEE/ACM international conference on Automated software engineering* ACM (Veranst.), 2010, S. 179–180

[273] Patel, Shwetak N. ; Pierce, Jeffrey S. ; Abowd, Gregory D.: A Gesture-based Authentication Scheme for Untrusted Public Terminals. In: *Proceedings of the 17th Annual ACM Symposium on User Interface Software and Technology* ACM (Veranst.), 2004, S. 157–160

[274] Pathak, Abhinav ; Hu, Y C. ; Zhang, Ming: Bootstrapping Energy Debugging on Smartphones: A First Look at Energy Bugs in Mobile Devices. In: *Proceedings of the 10th ACM Workshop on Hot Topics in Networks*. New York, NY, USA : ACM, 2011 (HotNets-X)

[275] Patton, Ron: *Software Testing*. Sams Pub., 2006

[276] Perrucci, Gian P. ; Fitzek, Frank H. ; Widmer, Jörg: Survey on Energy Consumption Entities on the Smartphone Platform. In: *Vehicular Technology Conference (VTC Spring)* IEEE (Veranst.), 2011, S. 1–6

[277] Peterson, Benjamin ; Bruckner, Dean ; Heye, Shannon: Measuring GPS Signals Indoors. In: *World congress of the International Association of Institutes of Navigation.*, 1997

[278] Peterson, Benjamin ; Hartnett, Richard ; Ottman, Geffrey: GPS Receiver Structures for the Urban Canyon. In: *Proceedings of the 8th International Technical Meeting of the Satellite Division of The Institute of Navigation (ION GPS 1995)*, 1995, S. 1323–1332

[279] Peterson, James L.: Petri Net Theory and the Modeling of Systems. (1981)

[280] Pettichord, Bret: Seven Steps to Test Automation Success. In: *Proceedings of the International Conference on Software Testing, Analysis, and Review*, 1999

[281] Pettichord, Bret: Design for Testability. In: *Pacific Northwest Software Quality Conference*, 2002

[282] Picco, Gian P. ; Julien, Christine ; Murphy, Amy L. ; Musolesi, Mirco ; Roman, Gruia-Catalin: Software Engineering for Mobility: Reflecting on the Past, Peering into the Future. In: *Proceedings of the on Future of Software Engineering*. New York, NY, USA : ACM, 2014 (FOSE 2014), S. 13–28

[283] Picco, Gian P. ; Murphy, Amy L. ; Roman, Gruia-Catalin: LIME: Linda meets Mobility. In: *Proceedings of the 21st International Conference on Software engineering* ACM (Veranst.), 1999, S. 368–377

[284] Pohl, K ; Rupp, Ch: *Basiswissen Requirements Engineering: Aus- und Weiterbildung zum "Certified Professional for Requirements Engineering." dpunkt. verlag.* 2009

[285] Pol, Martin ; Koomen, Tim ; 0002, Andreas S.: *Management und Optimierung des Testprozesses – ein praktischer Leitfaden für erfolgreiches Software-Testen mit TPI und TMap (2. Aufl.).* dpunkt.verlag, 2002

[286] Postel, J. ; Reynolds, J. K.: *RFC 854: Telnet Protocol Specification.* Mai 1983

[287] Prekop, Paul ; Burnett, Mark: Activities, Context and Ubiquitous Computing. In: *Computer Communications* 26 (2003), Nr. 11, S. 1168–1176

[288] Qui Don Ho: *Entwicklung eines Toolkits zur Unterstützung von automatisiertem Testen von Android Apps mit sensorenbasierter Gestenerkennung*, Universität Duisburg-Essen, Masterarbeit, März 2013

[289] Raento, M. ; Oulasvirta, A. ; Petit, R. ; Toivonen, H.: ContextPhone: A Prototyping Platform for Context-aware Mobile Applications. In: *Pervasive Computing, IEEE* 4 (2005), Jan, Nr. 2, S. 51–59

[290] Raiffeisen e-force GmbH: *Raiffeisen ELBA.* – URL `https://play.google.com/store/apps/details?id=com.isis_papyrus.raiffeisen_pay_eyewdg`. – Zugriffsdatum: 2015-09-05

[291] Ramler, Rudolf ; Wolfmaier, Klaus: Economic Perspectives in Test Automation: Balancing Automated and Manual Testing with Opportunity Cost. In: *Proceedings of the 2006 International Workshop on Automation of Software Test.* New York, NY, USA : ACM, 2006 (AST '06), S. 85–91. – ISBN 1-59593-408-1

[292] Ravindranath, Lenin ; Nath, Suman ; Padhye, Jitendra ; Balakrishnan, Hari: Automatic and Scalable Fault Detection for Mobile Applications. In: *Proceedings of the 12th Annual International Conference on Mobile Systems, Applications, and Services.* New York, NY, USA : ACM, 2014 (MobiSys '14), S. 190–203

[293] Raymond, Eric: The Cathedral and the Bazaar. In: *Knowledge, Technology & Policy* 12 (1999), Nr. 3, S. 23–49

[294] Rekimoto, Jun: Tilting Operations for Small Screen Interfaces. In: *Proceedings of the 9th Annual ACM Symposium on User Interface Software and Technology* ACM (Veranst.), 1996, S. 167–168

[295] Reynolds, Franklin ; Hjelm, Johan ; Dawkins, Spencer ; Singhal, Sandeep: Composite Capability/Preference Profiles (CC/PP): A User Side Framework for Content Negotiation. In: *W3C Note* (1999)

[296] Ridene, Youssef ; Barbier, Franck: A Model-driven Approach for Automating Mobile Applications Testing. In: *Proceedings of the 5th European Conference on Software Architecture: Companion Volume*. New York, NY, USA : ACM, 2011 (ECSA '11)

[297] Ridene, Youssef ; Belloir, Nicolas ; Barbier, Franck ; Couture, Nadine: A DSML for Mobile Phone Applications Testing. In: *Proceedings of the 10th Workshop on Domain-Specific Modeling*. New York, NY, USA : ACM, 2010 (DSM '10)

[298] Robin von Hardenberg: *timetraveler berlin wall*. – URL `https://play.google.com/store/apps/details?id=berlin.timetraveler.timetraveler.freeonline`. – Zugriffsdatum: 24.11.2015

[299] Robinson, Harry: Obstacles and Opportunities for Model-based Testing in an Industrial Software Environment. In: *Proceedings of the 1st European Conference on Model-Driven Software Engineering, Nuremberg, Germany*, 2003, S. 118–127

[300] RobotiumTech: *Robotium*. – URL `https://github.com/robotiumtech/robotium`. – Zugriffsdatum: 02.02.2016

[301] Rogers, Everett M.: *Diffusion of Innovations*. Simon and Schuster, 2010

[302] Rohs, Michael ; Essl, Georg ; Roth, Martin: CaMus: Live Music Performance Using Camera Phones and Visual Grid Tracking. In: *Proceedings of the 2006 Conference on New Interfaces for Musical Expression* IRCAM—Centre Pompidou (Veranst.), 2006, S. 31–36

[303] Roman, Gruia-Catalin ; Picco, Gian P. ; Murphy, Amy L.: Software Engineering for Mobility: A Roadmap. In: *Proceedings of the Conference on the Future of Software Engineering* ACM (Veranst.), 2000, S. 241–258

[304] Rooksby, John ; Rouncefield, Mark ; Sommerville, Ian: Testing in the Wild: The Social and Organisational Dimensions of Real World Practice. In: *Computer Supported Cooperative Work (CSCW)* 18 (2009), Nr. 5-6, S. 559–580

[305] Ruiz, Jaime ; Li, Yang ; Lank, Edward: User-defined Motion Gestures for Mobile Interaction. In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. New York, NY, USA : ACM, 2011 (CHI '11), S. 197–206

[306] Rumbaugh, James ; Jacobson, Ivar ; Booch, Grady: *Unified Modeling Language Reference Manual, The*. Pearson Higher Education, 2004

[307] Runtastic GmbH: *Runtastic Laufen & Fitness*. – URL `https://play.google.com/store/apps/details?id=com.runtastic.android`. – Zugriffsdatum: 2015-09-05

[308] Runtastic GmbH: *Runtastic Squats Kniebeugen*. – URL `https://play.google.com/store/apps/details?id=com.runtastic.android.squats.lite`. – Zugriffsdatum: 2015-09-01

[309] Rutherford, Matthew J. ; Wolf, Alexander L.: A Case for Test-code Generation in Model-Driven Systems. In: *Proceedings of the 2nd International Conference on Generative Programming and Component Engineering*. New York, NY, USA : Springer-Verlag New York, Inc., 2003 (GPCE '03), S. 377–396

[310] Sanaei, Z. ; Abolfazli, S. ; Gani, A. ; Buyya, R.: Heterogeneity in Mobile Cloud Computing: Taxonomy and Open Challenges. In: *Communications Surveys Tutorials, IEEE* 16 (2014), First, Nr. 1, S. 369–392

[311] Schilit, B. ; Adams, N. ; Want, R.: Context-Aware Computing Applications. In: *Proceedings of the 1994 First Workshop on Mobile Computing Systems and Applications*. Washington, DC, USA : IEEE Computer Society, 1994 (WMCSA '94), S. 85–90

[312] Schilit, Bill N. ; Theimer, Marvin M.: Disseminating Active Map Information to Mobile Hosts. In: *IEEE Network* 8 (1994), Nr. 5, S. 22–32

[313] Schilit, William N.: *A System Architecture for Context-Aware Mobile Computing*. 1995

[314] Schmidt, Albrecht ; Aidoo, Kofi A. ; Takaluoma, Antti ; Tuomela, Urpo ; Van Laerhoven, Kristof ; Velde, Walter Van de: Advanced Interaction in Context. In: *Handheld and Ubiquitous Computing* Springer (Veranst.), 1999, S. 89–101

[315] Schmidt, Albrecht ; Beigl, Michael ; Gellersen, Hans-W.: There is More to Context than Location. In: *Computers & Graphics* 23 (1999), Nr. 6, S. 893 – 901. – ISSN 0097-8493

[316] Schönberger, Marius: Der professionelle Einstieg in die erfolgreiche App-Entwicklung. In: Aichele, Christian (Hrsg.) ; Schönberger, Marius (Hrsg.): *App4U*. Springer Fachmedien Wiesbaden, 2014, S. 87–131

[317] Seiger, Ronny ; Schlegel, Thomas: Test Modeling for Context-aware Ubiquitous Applications with Feature Petri Nets. In: *Proceedings of the Workshop on Model-based Interactive Ubiquitous Systems (MODIQUITOUS)*, 2012

[318] Sendall, Shane ; Kozaczynski, Wojtek: Model Transformation the Heart and Soul of Model-driven Software Development. 2003. – Forschungsbericht

[319] Sesia, Stefania ; Toufik, Issam ; Baker, Matthew: *LTE: The UMTS Long Term Evolution*. Wiley Online Library, 2009

[320] Shapiro, Samuel S. ; Wilk, Martin B.: An analysis of variance test for normality (complete samples). In: *Biometrika* 52 (1965), Nr. 3/4, S. 591–611

[321] Shull, Forrest ; Basili, Vic ; Boehm, Barry ; Costa, Patricia ; Lindvall, Mikael ; Port, Dan ; Rus, Ioana ; Tesoriero, Roseanne ; Zelkowitz, Marvin et al.: What We Have Learned About Fighting Defects. In: *Proceedings of the Eighth IEEE Symposium on Software Metrics* IEEE (Veranst.), 2002, S. 249–258

[322] Six to Start: *Zombies, Run!*. – URL https://play.google.com/store/apps/details?id=com.sixtostart.zombiesrunclient. – Zugriffsdatum: 2015-09-05

[323] Skvortzov, V.Y. ; Lee, Hyoung-Ki ; Bang, SeokWon ; Lee, YongBeom: Application of Electronic Compass for Mobile Robot in an Indoor Environment. In: *IEEE International Conference on Robotics and Automation*, April 2007, S. 2963–2970

[324] Sony Corporation: *Sony Smartwatch Manual*. – URL http://support.sonymobile.com/global-en/swr50/userguide/Wrist-gestures/. – Zugriffsdatum: 01.12.2015

[325] SourceForge Community: *Android GUITAR*. – URL http://sourceforge.net/apps/mediawiki/guitar/index.php?title=Android_GUITAR. – Zugriffsdatum: 12.01.2016

[326] SourceForge.NET: *CUnit*. – URL http://cunit.sourceforge.net. – Zugriffsdatum: 31.08.2015

[327] SPEDION GmbH: *SPEDION App*. – URL http://www.spedion.de/produkte/spedion-app/. – Zugriffsdatum: 31.08.2015

[328] Stachowiak, Herbert: *Allgemeine Modelltheorie*. Springer-Verlag, Wien, 1973

[329] Stahl, Thomas ; Voelter, Markus ; Czarnecki, Krzysztof: *Model-driven Software Development: Technology, Engineering, Management*. John Wiley & Sons, 2006

[330] Staines, Tony S.: Intuitive mapping of UML 2 Activity Aiagrams into Fundamental Modeling Concept Petri Net Diagrams and colored Petri nets. In: *Engineering of Computer Based Systems, 2008. ECBS 2008. 15th Annual IEEE International Conference and Workshop on the* IEEE (Veranst.), 2008, S. 191–200

[331] Statista Inc.: *Number of available apps in the Apple App Store from July 2008 to June 2015*. – URL http://www.statista.com/statistics/263795/number-of-available-apps-in-the-apple-app-store/. – Zugriffsdatum: 31.08.2015

[332] Statista Inc.: *Number of Smartphone Users Worldwide from 2012 to 2018 (in Billions)*. – URL http://www.statista.com/statistics/330695/number-of-smartphone-users-worldwide/. – Zugriffsdatum: 31.08.2015

[333] Statista Inc.: *Worldwide Mobile App Revenues from 2011 to 2017 (in Billion U.S. Dollars)*. – URL http://www.statista.com/statistics/269025/worldwide-mobile-app-revenue-forecast/. – Zugriffsdatum: 31.08.2015

[334] Stefanov, Veronika ; List, Beate ; Korherr, Birgit: Extending UML 2 Activity Diagrams with Business Intelligence Objects. In: Tjoa, AMin (Hrsg.) ; Trujillo, Juan (Hrsg.): *Data Warehousing and Knowledge Discovery* Bd. 3589. Springer Berlin Heidelberg, 2005, S. 53–63

[335] Steinberg, Dave ; Budinsky, Frank ; Merks, Ed ; Paternostro, Marcelo: *EMF: Eclipse Modeling Framework*. Pearson Education, 2008

[336] Strang, Thomas ; Linnhoff-Popien, Claudia: A Context Modeling Survey. In: *Workshop on Advanced Context Modelling, Reasoning and Management, UbiComp 2004 - The Sixth International Conference on Ubiquitous Computing, Nottingham/England*, 2004

[337] Sun, Lin ; Zhang, Daqing ; Li, Bin ; Guo, Bin ; Li, Shijian: Activity Recognition on an Accelerometer Embedded Mobile Phone with Varying Positions and Orientations. In: *Ubiquitous Intelligence and Computing*. Springer, 2010, S. 548–562

[338] Swing by Swing: *Golf GPS & Scorecard*. – URL `https://play.google.com/store/apps/details?id=com.swingbyswing`. – Zugriffsdatum: 2015-09-05

[339] Tanaka, Atau: Mobile Music Making. In: *Proceedings of the 2004 Conference on New Interfaces for Musical Expression* National University of Singapore (Veranst.), 2004, S. 154–156

[340] Tanaka, Atau ; Valadon, Guillaume ; Berger, Christophe: Social Mobile Music Navigation Using the Compass. In: *Proceedings of the International Mobile Music Workshop, Amsterdam*, 2007

[341] Tassey, Gregory: The Economic Impacts of Inadequate Infrastructure for Software Testing / National Institute of Standards and Technology. 2002. – Forschungsbericht

[342] Teixeira, Tania: Meet Marty Cooper – The Inventor of the Mobile Phone. In: *BBC News* (2010), April

[343] Thompson, Silvanus P.: *Philipp Reis, Inventor of the Telephone*. Arno Press, 1974

[344] Thummalapenta, S. ; Sinha, S. ; Singhania, N. ; Chandra, Satish: Automating Test Automation. In: *34th International Conference on Software Engineering (ICSE)*, 2012, S. 881–891

[345] Tsai, Ming-Chang ; Chou, Fu-Chiang ; Kao, Yih-Feng ; Yang, Kai-Cheng ; Chen, Mike: Polite Ringer II: A Ringtone Interaction System Using Sensor Fusion. In: *Proceedings of the 13th International Conference on Ubiquitous Computing*. New York, NY, USA : ACM, 2011 (UbiComp '11), S. 567–568

[346] Uber Technologies Inc.: *Uber*. – URL `https://www.uber.com/de/`. – Zugriffsdatum: 15.10.2015

[347] Utting, Mark ; Pretschner, Alexander ; Legeard, Bruno: A Taxonomy of Model-based Testing Approaches. In: *Software Testing, Verification and Reliability* 22 (2012), August, Nr. 5, S. 297–312

[348] Valmari, Antti: The State Explosion Problem. In: *Lectures on Petri Nets I: Basic models*. Springer, 1998, S. 429–528

[349] Vaseghi, Saeed V.: *Advanced Digital Signal Processing and Noise Reduction.* John Wiley & Sons, 2008

[350] Vicek, Charles ; McLain, Patricia ; Murphy, Midhael: GPS/Dead Deckoning for Vehicle Tracking in the "Urban Canyon" Environment. In: *Vehicle Navigation and Information Systems Conference, 1993., Proceedings of the IEEE-IEE* IEEE (Veranst.), 1993, S. 461–A

[351] Vieira, Marlon ; Leduc, Johanne ; Hasling, Bill ; Subramanyan, Rajesh ; Kazmeier, Juergen: Automation of GUI Testing Using a Model-driven Approach. In: *Proceedings of the 2006 International Workshop on Automation of Software Test.* New York, NY, USA : ACM, 2006 (AST '06), S. 9–14

[352] Vieira, Vaninha ; Holl, Konstantin ; Hassel, Michael: A Context Simulator as Testing Support for Mobile Apps. In: *Proceedings of the 30th Annual ACM Symposium on Applied Computing.* New York, NY, USA : ACM, 2015 (SAC '15), S. 535–541

[353] Vieira, Vaninha ; Tedesco, Patricia ; Salgado, Ana C.: Designing Context-sensitive Systems: An Integrated Approach. In: *Expert Systems with Applications* 38 (2011), Nr. 2, S. 1119–1138

[354] vimov: *iSimulate.* – URL `http://www.vimov.com/isimulate/`. – Zugriffsdatum: 10.11.2015

[355] Voas, Jeffrey ; Michael, J B. ; Genuchten, Michiel van: The Mobile Software App Takeover. In: *IEEE software* 29 (2012), Nr. 4, S. 25–27

[356] W3C: *Composite Capability/Preference Profiles (CC/PP).* – URL `http://www.w3.org/TR/2004/REC-CCPP-struct-vocab-20040115/`. – Zugriffsdatum: 12.10.2015

[357] Wall, Timothy: *Abbot Java GUI Test Framework.* – URL `http://abbot.sourceforge.net/doc/overview.shtml`. – Zugriffsdatum: 15.01.2016

[358] Want, Roy ; Hopper, Andy ; Falcao, Veronica ; Gibbons, Jonathan: The Active Badge Location System. In: *ACM Transactions on Information Systems (TOIS)* 10 (1992), Nr. 1, S. 91–102

[359] Wasserman, Anthony I.: Software Engineering Issues for Mobile Application Development. In: *Proceedings of the FSE/SDP Workshop on Future of Software Engineering Research* ACM (Veranst.), 2010, S. 397–400

[360] Whittaker, James et al.: What is Software Testing? And Why is it so Hard? In: *IEEE Software* 17 (2000), Nr. 1, S. 70–79

[361] Wiese, Jason ; Saponas, T. S. ; Brush, A. J. B.: Phoneprioception: Enabling Mobile Phones to Infer Where They Are Kept. In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems.* New York, NY, USA : ACM, 2013 (CHI '13), S. 2157–2166

[362] Wikitude GmbH: *Wikitude.* – URL `https://play.google.com/store/apps/details?id=com.wikitude`. – Zugriffsdatum: 26.08.2015

[363] Wiklund, Kristian ; Eldh, Sigrid ; Sundmark, Daniel ; Lundqvist, Kristina: Technical Debt in Test Automation. In: *Fifth International Conference on Software Testing, Verification and Validation (ICST)* IEEE (Veranst.), 2012, S. 887–892

[364] Williamson, John ; Murray-Smith, Roderick ; Hughes, Stephen: Shoogle: Excitatory Multimodal Interaction on Mobile Devices. In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* ACM (Veranst.), 2007, S. 121–124

[365] Wingfield, Nick: iPhone Software Sales Take Off: Apple's Jobs. In: *The Wall Street Journal* 11 (2008)

[366] Wohed, Petia ; Aalst, Wil M. P. van der ; Dumas, Marlon ; Hofstede, Arthur ; Russell, Nick: *Pattern-Based Analysis of the Control-Flow Perspective of UML Activity Diagrams.* S. 63–78. In: Delcambre, Lois (Hrsg.) ; Kop, Christian (Hrsg.) ; Mayr, Heinrich C. (Hrsg.) ; Mylopoulos, John (Hrsg.) ; Pastor, Oscar (Hrsg.): *Conceptual Modeling – ER 2005: 24th International Conference on Conceptual Modeling, Klagenfurt, Austria, October 24-28, 2005. Proceedings.* Berlin, Heidelberg : Springer Berlin Heidelberg, 2005

[367] Wohed, Petia ; Aalst, Wil M. van der ; Dumas, Marlon ; Hofstede, Arthur ; Russell, Nick: Pattern-based analysis of UML activity diagrams. In: *Beta, Research School for Operations Management and Logistics, Eindhoven* (2004)

[368] Wynne, Matt ; Hellesoy, Aslak: *The Cucumber Book: Behaviour-Driven Development for Testers and Developers.* Pragmatic Bookshelf, 2012. – ISBN 1934356808, 9781934356807

[369] Xamarin Inc.: *Calabash.* – URL `http://calaba.sh`. – Zugriffsdatum: 31.08.2015

[370] Xamarin Inc.: *Xamarin Test Cloud.* – URL `https://xamarin.com/test-cloud`. – Zugriffsdatum: 11.11.2015

[371] Xu, Dianxiang: A Tool for Automated Test Code Generation from High-level Petri Nets. In: *Applications and Theory of Petri Nets.* Springer, 2011, S. 308–317

[372] Yan, Jun ; Li, Zhongjie ; Yuan, Yuan ; Sun, Wei ; Zhang, Jian: BPEL4WS Unit Testing: Test Case Generation Using a Concurrent Path Analysis Approach. In: *17th International Symposium on Software Reliability Engineering (ISSRE)* IEEE (Veranst.), 2006, S. 75–84

[373] Yang, Wei ; Prasad, Mukul R. ; Xie, Tao: A Grey-Box Approach for Automated GUI-Model Generation of Mobile Applications. In: *Fundamental Approaches to Software Engineering.* Springer, 2013, S. 250–265

[374] Zandbergen, Paul A.: Accuracy of iPhone Locations: A Comparison of Assisted GPS, WiFi and Cellular Positioning. In: *Transactions in GIS* 13 (2009), Nr. s1, S. 5–25

[375] Zander, Justyna ; Dai, ZhenRu ; Schieferdecker, Ina ; Din, George: From U2TP Models to Executable Tests with TTCN-3 – An Approach to Model Driven Testing. In: Khendek, Ferhat (Hrsg.) ; Dssouli, Rachida (Hrsg.): *Testing of Communicating Systems* Bd. 3502. Springer Berlin Heidelberg, 2005, S. 289–303

[376] Zhang, Dongsong ; Adipat, Boonlit: Challenges, Methodologies, and Issues in the Usability Testing of Mobile Applications. In: *International Journal of Human-Computer Interaction* 18 (2005), Nr. 3, S. 293–308

[377] Zhang, Jack ; Sagar, Shikhar ; Shihab, Emad: The Evolution of Mobile Apps: An Exploratory Study. In: *Proceedings of the 2013 International Workshop on Software Development Lifecycle for Mobile*. New York, NY, USA : ACM, 2013 (DeMobile 2013), S. 1–8

[378] Zhao, Wei ; Zeng, Yi ; ZHANG, Li-wu: Generating Test Scenarios of Functional Test from UML Activity Diagrams. In: *Computer Engineering and Design* 22 (2006), S. 048

[379] Zhong, Nan ; Michahelles, Florian: Google Play is Not a Long Tail Market: An Empirical Analysis of App Adoption on the Google Play App Market. In: *Proceedings of the 28th Annual ACM Symposium on Applied Computing* ACM (Veranst.), 2013, S. 499–504

Selbstständigkeitserklärung

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbständig ohne unzulässige Hilfe Dritter verfasst, keine anderen als die angegebenen Quellen und Hilfsmittel benutzt und die Arbeit keiner anderen Stelle zu Prüfungszwecken vorgelegt habe.

Neuenhagen, den _____

Tobias Griebe