Entwicklung multipler Benutzerschnittstellen für eine Anwendung

Vom Fachbereich Informatik
der Technischen Universität Darmstadt
genehmigte

Dissertation

zur Erlangung des akademischen Grades Dr. rer. nat.

von

Dipl.-Phys. Alexander Behring

geboren in Frankfurt am Main

Referenten
Prof. Dr. Max Mühlhäuser (TU Darmstadt)
Prof. Dr. Detlef Zühlke (TU Kaiserslautern)

Tag der Einreichung: 27.04.2011
Tag der mündlichen Prüfung: 20.06.2011

Darmstadt 2012
Hochschulkennziffer D17

Danksagung

Diese Arbeit wäre nicht ohne die Unterstützung und Ermunterung durch meine Betreuer, Kollegen, Studenten, Familie und Freunde möglich gewesen. Ich bin Euch zu Dank verpflichtet.

Zuerst möchte ich mich bei meinem Betreuer, Max Mühlhäuser (TU Darmstadt), für das Vertrauen, mich in meinem Promotionsvorhaben zu unterstützen, und die wertvollen Hinweise zu meiner Arbeit bedanken. Auch bei Detlef Zühlke (DFKI) möchte ich mich für seine Tätigkeit als Zweitgutachter bedanken. Schließlich danke ich Felix Flentge für die Unterstützung als er zeitweilig mein Gruppenleiter war.

Ein großer Dank geht an meine Frau, die Höhen und Tiefer meiner Forschung direkt miterlebt hat und aushalten durfte. Auch meiner Familie möchte ich für die Unterstützung danken. Es ist gut zu wissen, dass Ihr da seid.

Für anregende Diskussionen und das freundliche, innovative Umfeld bedanke ich mich bei meinen Kollegen der Telekooperation und der RBG. Ein besonderer Dank gilt meinen Bürokollegen, deren Whiteboard ich regelmäßig mit Zeichnungen in Anspruch genommen habe; und speziell Andreas Petter, mit dem ich meine Zeit in der Forschung verbringen durfte und der diese Arbeit durch unsere häufigen Diskussionen stark bereichert hat.

Auch dem studentischen Team, welches unermüdlich an der Verwirklichung der Vision durch Mapache gearbeitet hat, bin ich zu Dank verpflichtet. Ein besonderer Dank geht dabei an Jannik Jochem, dessen Beiträge und Aktivität mich daran glauben ließen, dass die Vision Wirklichkeit wird.

Für die Unterstützung bei der Studie gilt mein Dank folgenden Firmen und Personen: B2M Software AG, optary consult GmbH, SAP AG, Sven Becker, Oliver Grübner, Anna Lewandowski, Marco Mähner und Dirk Songür.

Viele Menschen haben an der Verbesserung dieses Textes mitgewirkt. Für die vielen konstruktiven Kommentare und Diskussionen möchte ich mich bei meinen Kollegen in der TU Darmstadt und außerhalb bedanken: Birgit Behring, Gerd Behring, Sebastian Feuerstack (TU Berlin / UF São Carlos), Melanie Hartmann, Gerrit Meixner (DFKI), Max Mühlhäuser, Andreas Petter, Sebastian Ries, Dirk Schnelle-Walka und Jürgen Steimle.

Schließlich gilt mein Dank dem Bundesministerium für Bildung und Forschung, welches durch die Finanzierung der Projekte EMODE und SOKNOS diese Arbeit erst möglich gemacht hat.

Abstract

The ever increasing number of mobile devices as well as concepts like "Designing for Peak Experience" inevitably lead to *a quest for applications that provide user interfaces (UIs) for a growing number of different contexts of use*. The contexts of use differ in platforms and devices, user groups and their goals, as well as other boundary conditions such as the environment during interaction. This dissertation provides concepts that aid the developer in creating such Multi User Interfaces (MBS) and allow their execution.

Hereby, a first, essential contribution of this thesis is the *concept formation and elicitation of requirements*, which goes beyond the related work. Terms of the problem and solution domain were investigated in the context of related approaches; hereby, inconsistencies have been identified, especially in relation to the terms "Abstract User Interface (AUI)" and "Concrete User Interface (CUI)". This finding had substantial impact on the concepts developed in this thesis. In particular, the common dichotomy between AUI and CUI was dismissed in favor of an arbitrary number of increasingly concrete user interface descriptions. This novelty shapes the presented thesis and is reflected in the term "Abstraction Independent User Interface (UUI)".

According to the requirements, a concept was developed, which is comprised of *i)* an architecture pattern for MBS, *ii)* a domain specific language for describing MBS and *iii)* interactive as well as explorative support concepts. Hereby, modeling techniques for the UI structure (e.g., the layout for graphical UIs) are linked to programming techniques for the UI behavior. The *architecture pattern for MBS* is based on the pattern "Model View Controller (MVC)", which was extended to support multiple variants of a UI as well as to explicitly handle inheritance of behavior. Also developed in this thesis, the *domain specific language (DSL)* provides means for modeling the MBS variants and their refinement associations (inheritance) among each other. Thereby, the different variants of a MBS are ordered in a tree structure (refinement tree), allowing the propagation of a modification to arbitrary many variants.

Based on the architecture pattern and the architecture developed in this thesis, *support concepts* were devised focusing ease of use by the developer. Explorative support concepts make transparent the state of development of the MBS; they therefore visualize the refinement tree, as well as interfaces between behavior and structure. In contrast, interactive support concepts provide for

the modification of one or multiple variants of the MBS. The interpreter is pivotal, as it brings user interface models directly to interaction. The concept also provides for extending interpreters to WYSIWYG like editors. Finally, modular adaption concepts encapsulate specific user interface adaptations (e.g., scaling) into an easy to use form for the developer.

The concepts were applied in a research project with industry partners, by means of a *prototypical implementation* called Mapache. An infrastructure was developed, supporting the design and runtime. On top of it, a development environment was built and implemented in Eclipse. It allows for highly integrated development of Java based Multi-User-Interfaces.

Finally, the concepts developed and implemented were *evaluated through a use case of the project and a user study.* The use case showed that the requirements elicited are fulfilled by the approach developed. The user study, conducted in the form of "Cooperative Evaluation", resulted in a positive assessment of the approach and highlighted aspects that have to be considered when applying the approach in practice. As expected, the fundamentally new possibility to work on UIs at multiple levels of abstractions at the same time, proved to be utmost helpful when developing federated user interfaces. Overall, the presented thesis contributed to the general requirements of efficiency, ease of use, and consistency. The quality of the study itself goes beyond related work, in terms of practices applied, only professional developers of user interfaces were participating and the choice of the evaluation method was made explicit and transparent.

Zusammenfassung

Die stetig wachsende Zahl von mobilen Endgeräten und Konzepte wie "Designing for Peak Experience", führt unweigerlich zu einem Streben nach *Anwendungen, welche für immer mehr verschiedene Nutzungskontexte Benutzerschnittstellen (UIs) bereitstellen.* Die Nutzungskontexte unterscheiden sich in Bezug auf Plattformen und Geräte, Benutzergruppen und deren Ziele, sowie Randbedingungen wie z.B. die Umgebung während der Interaktion. Diese Dissertation stellt Konzepte bereit, welche den Entwickler bei der Erstellung solcher Multi-Benutzerschnittstellen (MBS) unterstützen und die Ausführung dieser ermöglichen.

Dabei liegt ein erster wesentlicher Beitrag der vorliegenden Arbeit in der *Begriffsbildung und Erhebung von Anforderungen*, wobei sie weit über verwandte Arbeiten hinausgeht. Begriffe der Problem- sowie der Lösungsdomäne wurden im Kontext der verwandten Arbeiten untersucht; dabei wurden Inkonsistenzen festgestellt, insbesondere im Zusammenhang mit den Begriffen "Konkrete Benutzerschnittstelle (CUI)" und "Abstrakte Benutzerschnittstelle (AUI)". Diese Erkenntnisse hatten wesentlichen Einfluss auf die in dieser Arbeit entwickelten Konzepte. Insbesondere wurde die herkömmliche Dichotomie zwischen AUI und CUI aufgehoben und durch eine beliebige Zahl von Stufen zunehmend konkreter Beschreibungen von Benutzerschnittstellen ersetzt. Diese Neuerung prägt die vorliegende Arbeit und schlägt sich im Begriff der "Abstraktionsunabhängigen Benutzerschnittstelle (UUI)" nieder.

Entsprechend den Anforderungen wurde ein Konzept entwickelt, welches aus *i)* einem Architekturmuster für MBS, *ii)* einer domänenspezifischen Sprache zur Beschreibung von MBS und *iii)* interaktiven sowie explorativen Unterstützungskonzepten besteht. Hierbei werden Modellierungstechniken für die UI-Struktur (bei z.B. grafischen UIs das Layout) mit Programmiertechniken für das UI-Verhalten verknüpft. Das *Architekturmuster für MBS* basiert auf dem Muster "Model View Controller (MVC)", welches um die Unterstützung mehrerer Varianten einer UI sowie die explizite Handhabung des Erbens von Verhalten erweitert wurde. Die im Rahmen der Arbeit ebenfalls entwickelte *domänenspezifische Sprache (DSL)* stellt Möglichkeiten zur Modellierung der MBS-Varianten und ihrer Verfeinerungsbeziehungen (Vererbung) untereinander bereit. Die verschiedenen Varianten einer MBS werden so in einer Baumstruktur (Verfeinerungsbaum) angeordnet, welche die Propagation einer Modi-

fikation auf beliebig viele Varianten ermöglicht.

Basierend auf dem Architekturmuster und der im Rahmen dieser Arbeit entwickelten Architektur wurden *Unterstützungskonzepte* entwickelt, welche auf einfache Nutzbarkeit durch den Entwickler ausgelegt sind. Explorative Unterstützungskonzepte machen den Entwicklungsstand der MBS transparent; sie visualisieren hierfür den Verfeinerungsbaum sowie Schnittstellen zwischen Verhalten und Struktur. Interaktive Unterstützungskonzepte dagegen ermöglichen die gleichzeitige Modifikation einer oder mehrerer Varianten der MBS. Zentral dabei ist der Interpreter, welcher Benutzerschnittstellenmodelle direkt zur Interaktion bringt; dies Konzept sieht auch den Ausbau des Interpreters zu einem WYSIWYG-artigen Editor vor. Modulare Adaptionskonzepte schließlich kapseln spezifische Anpassungen (z.B. Skalierungen) in einer einfach durch den Entwickler nutzbaren Weise.

Die Konzepte wurden im Rahmen eines Forschungsprojekt mit Industriepartnern konkret eingesetzt, mit Hilfe einer *prototypischen Realisierung, genannt Mapache*. Eine entwickelte Infrastruktur unterstützt hierfür die Lauf- und Entwicklungszeit. Die in Eclipse realisierte Entwicklungsumgebung baut auf ihr auf und ermöglicht eine hoch integrierte Entwicklung Java basierter Multi-Benutzerschnittstellen.

Abschließend wurden die entwickelten und realisierten Konzepte in einer *Fallstudie des Projektes sowie einer Nutzerstudie evaluiert*. Die Fallstudie zeigte, dass die erhobenen Anforderungen vom entwickelten Ansatz erfüllt werden. Die Nutzerstudie, durchgeführt in Form der sogenannten “Kooperativen Evaluation”, ergab eine positive Bewertung des Ansatzes und zeigte Themen auf, welche bei der Anwendung des Ansatzes in der Praxis beachtet werden müssen. Wie erwartet, erwies sich die grundlegend neue Möglichkeit, Benutzerschnittstellen auf unterschiedlichen Abstraktionsgraden gleichzeitig zu bearbeiten, als äußerst hilfreich für die Entwicklung föderierter Benutzerschnittstellen. Insgesamt wurden Fortschritte im Bereich der Anforderungen Effizienz, Nutzbarkeit und Konsistenz gemacht. Die Qualität der Studie selbst geht über verwandte Arbeiten hinaus, weil zum Einen nur berufsmäßige Entwickler von Benutzerschnittstellen teilnahmen und zum Anderen die Wahl der Evaluationsmethode klar und transparent dargelegt wurde.

Inhaltsverzeichnis

Kapitel 1

Einleitung

Wachsende Bedeutung von Benutzerschnittstellen: Die Erstellung der Benutzerschnittstelle ist ein *signifikanter Teil der Erstellung von Anwendungen.* Dies wurde schon in den frühen 1990er Jahren festgestellt: Aus Myers Studie zur Erstellung von Benutzerschnittstellen (Myers und Rosson 1992) geht hervor, dass bereits 1992 ein Anteil von 48% des Anwendungsprogramms auf die Benutzerschnittstelle bezogen ist, der von ihm ermittelte zeitliche Anteil der Implementierungsarbeit lag 1992 bei 50%.

Diese Arbeit befasst sich mit der Erstellung von Benutzerschnittstellen.

Seit der Untersuchung von Myers und Rosson im Jahre 1992 sind die Benutzerschnittstellen komplexer geworden und dementsprechend auch anspruchsvoller in ihrer Erstellung. So sticht z.B. Apple's iPhone mit seiner Benutzerschnittstelle hervor. Diese innovative und intuitive Benutzerschnittstelle wurde als ein Erfolgsfaktor von Apple's iPhone identifiziert (Laugesen und Yuan 2010). Dieses Beispiel zeigt, dass die *Investition in gute Benutzerschnittstellen auch ökonomisch sinnvoll* sein kann. Schon seit mehreren Jahrzehnten gibt es daher verschiedene Ansätze, die Entwicklung von Benutzerschnittstellen zu vereinfachen.

Entwicklung von Benutzerschnittstellen: Die Erstellung von Benutzerschnittstellen kann mit *verschiedenen Techniken* unterstützt werden. Myers, Hudson und Pausch (2000) geben einen Überblick über verschiedene Möglichkeiten, Benutzerschnittstellen zu entwickeln. Die zwei gängigsten Ansätze sind dabei zum Einen die Nutzung eines grafischen Editors und zum Anderen die Programmierung der Benutzerschnittstelle. *Grafische WYSIWYG*[1] *Editoren* sind weit verbreitet und werden z.B. im Microsoft Visual Studio und anderen Entwicklungswerkzeugen eingesetzt. Hierbei kann der Entwickler direkt die

[1] WYSIWYG – What You See Is What You Get

Darstellung der Benutzerschnittstelle manipulieren. Dagegen ist das *Programmieren einer Benutzerschnittstelle* mit Hilfe eines UI Toolkits eher unter der Programmiersprache Java gängig. Hierbei wird ein Programm geschrieben, welches dann zur Laufzeit die Benutzerschnittstelle erstellt. Die Erstellung läuft somit indirekt ab.

Neben diesen weit verbreiteten Techniken existiert auch die Gruppe der modellgetriebenen Ansätze. Diese bieten einen formaleren Rahmen und ermöglichen somit die Nutzung verschiedener Darstellungsformen sowie Einbeziehung zusätzlicher Informationsquellen bei der Erstellung einer Benutzerschnittstelle.

Diese Arbeit entwickelt modellgetriebene Techniken zur Beschreibung von Benutzerschnittstellen.

Im täglichen Leben nutzen wir bereits eine Vielzahl von verschiedenen Geräten und diese Diversifizierung der Endgeräte wird weiter zunehmen, da immer mehr Geräte auch als vollwertiger Ersatz eines Standard PCs vorgestellt werden (Seffah, Forbrig und Javahery 2004). Darüber hinaus stellt die Delphistudie 2010 (Münchner Kreis e.V. U. a. 2009) fest, dass mehr Lebensbereiche von Informations- und Kommunikationstechnologien durchdrungen werden. Es wird von einer umfangreicheren mobilen Nutzung des Internets ausgegangen.

Herausforderung der multiplen Nutzungskontexte einer Anwendung: Auch heute schon existieren Anwendungen mit multiplen Benutzerschnittstellen. Sie unterstützen hierbei meist unterschiedliche Geräte: Vom Standard Desktop über verschiedene mobile Endgeräte. Jede dieser unterschiedlichen Nutzungssituationen wird als *Nutzungskontext* bezeichnet. Bei den Anwendungen handelt es sich oft um Auskunft zu Fahrplänen von Bahn, Bus oder Flugzeugen. Aber auch weitergehende Interaktionen, wie z.B. die Möglichkeit zum Fahrscheinerwerb und die Nutzung als elektronische Bordkarte sind inzwischen in der Öffentlichkeit angekommen. *Bei der Entwicklung von Anwendungen stellt sich somit die Herausforderung, dass diese Benutzerschnittstellen für unterschiedliche Nutzungskontexte bereitstellen müssen.*

Diese Arbeit fokussiert auf Benutzerschnittstellen für multiple Nutzungskontexte.

Was den Nutzungskontext bestimmt ist sehr variabel und kann sich von Anwendung zu Anwendung unterscheiden. So kann ein wichtiger Faktor das genutzte Endgerät sein: vom Standard PC über eine Großbildwand bis hin zum Mobiltelefon. Aber auch der Nutzer selbst kann den Nutzungskontext bestimmen, so kann einem Experten eine Benutzerschnittstelle geboten werden mit der er sehr effizient arbeiten kann, wohingegen die Benutzerschnittstelle für einen Einsteiger eher einfach und mit mehr Erklärungen gehalten ist. Dix (2009) geht mit seinem Konzept "*Designing for Peak Experience*" noch weiter.

Er schlägt vor, statt Benutzerschnittstellen für größere Gruppen von Benutzern zu gestalten, diese auf sehr kleine Gruppen und sogar individuelle Benutzer anzupassen.

1.1 Beiträge dieser Arbeit

Die vorliegende Arbeit liefert *Beiträge zur Entwicklung multipler Benutzerschnittstellen für eine Anwendung* in mehreren Bereichen wie in Abbildung 1.1 illustriert. Als Grundlage dient eine tiefgehende Diskussion der *Begriffe und Anforderungen* (in Abbildung 1.1 blau dargestellt), auf welchen das *Lösungskonzept* (in 1.1 grün) aufbaut.

Ausgangspunkt des Lösungskonzeptes ist ein *Architekturmuster* als konzeptionelle Basis für die weiteren Beiträge aus dem Bereich Lauf- und Entwicklungszeit. Dabei werden Benutzerschnittstellen in den Dimensionen Struktur und Verhalten beschrieben. Für den Strukturaspekt wurde eine *domänenspezifische Sprache* (DSL) entwickelt, für das Verhalten ein Konzept auf Basis einer objektorientierten Programmiersprache. Die Entwicklungszeit wird mit *Unterstützungskonzepten* für Entwickler adressiert, welche Bezug auf das Architekturmuster nehmen; Die verschiedenen Bausteine des Lösungskonzeptes werden schließlich durch die *architektonische Integration* in Lauf- und Entwicklungszeit eng verzahnt und mit der Möglichkeit zur Erweiterung des Ansatzes ausgestattet.

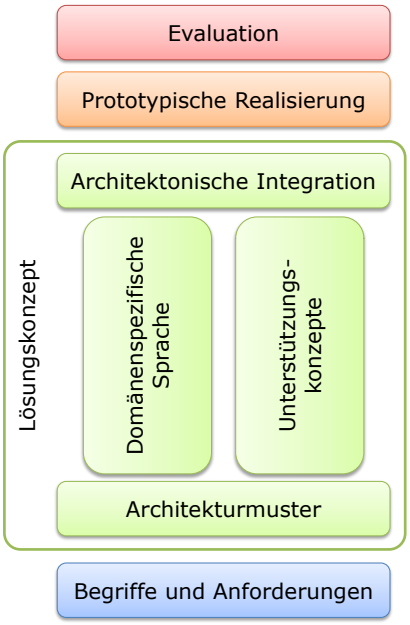


Abbildung 1.1: Aufbau der Beiträge dieser Arbeit.

Mit der darauf basierenden *prototypischen Realisierung* (in 1.1 orange) wurde die Anwendbarkeit des Ansatzes gezeigt und die Grundlage für die *Evaluation* (in 1.1 rot) gelegt. Diese schließt die notwendige Methodendiskussion mit ein und bewertet den Ansatz umfassend.

Alle Beiträge werden im Folgenden detaillierter beschrieben, wobei die **wesentlichen wissenschaftlichen Beiträge mit dem Symbol "☆"** gekennzeichnet sind.

Analyse des Standes der Wissenschaft (Teil I)

Begriffe und Anforderungen Im Rahmen der vorliegenden Arbeit wurde eine tiefgehendere Diskussion der Begriffe und Anforderungen geführt als in anderen, verwandten Arbeiten üblich. Diesem Teil kommt insofern besondere Bedeutung zu, als darin klar die ☆ Erfordernis herausgearbeitet wurde, von

der bisherigen Dichotomie in zwei Abstraktionsebenen (AUI vs. CUI) Abstand zu nehmen zu Gunsten einer abstraktionsunabhängigen Beschreibung (UUI).

Die ☆ erhobenen Anforderungen basieren auf umfangreichen Literaturstudien und Studien von verwandten Ansätzen. Dabei wurde deutlich mehr als in anderen Arbeiten der Entwickler selbst in den Mittelpunkt gestellt.

Konzept (Teil II)

Architekturmuster Die zentralen und notwendigen Elemente einer Multi-Benutzerschnittstelle werden durch das Architekturmuster festgelegt. Sie bilden somit einen Rahmen für die weiteren Bausteine des Lösungskonzeptes. Das ☆ entwickelte Architekturmuster basiert auf dem Architekturmuster "Model View Controller" (MVC), erweitert es aber in Hinblick auf die Einbindung mehrerer Varianten einer Benutzerschnittstelle.

Domänenspezifische Sprache (DSL) Zur Modellierung von Benutzerschnittstellen muss eine passende Modellierungssprache konstruiert werden. Die Domäne der im Rahmen dieser Arbeit ☆ entwickelten DSL ist die Beschreibung der Struktur multipler Benutzerschnittstellen. Sie ist (gemäß Anforderungen) Toolkit-unabhängig und leicht um neue Modellierungselemente erweiterbar. Mit ihr können mehrere Varianten einer Benutzerschnittstelle beschrieben und miteinander in Beziehung gesetzt werden, sodass eine Modifikation auf mehrere Varianten angewendet werden kann.

Unterstützungskonzepte Basierend auf den identifizierten Anforderungen wurden ☆ verschiedene Unterstützungskonzepte für den Entwickler zur Entwicklung von Multi-Benutzerschnittstellen konzipiert. Sie fokussieren die einfache Nutzbarkeit durch den Entwickler und erhöhen unter anderem die Transparenz sowie Flexibilität bei der Entwicklung. Ergebnis ist unter anderem ein Modellinterpreterkonzept, mit dem Benutzerschnittstellen zur Interaktion gebracht[2] werden und welches zu einem WYSIWYG-artigen Editor ausgebaut wurde. Weitere interaktive Unterstützungskonzepte erlauben die Modifikation der Benutzerschnittstelle, wohingegen explorative Unterstützungskonzepte es ermöglichen, den Entwicklungsstand der Benutzerschnittstelle zu überprüfen.

Architektonische Integration Die entwickelten Unterstützungskonzepte, das Erweiterungskonzept und die DSL bedürfen einer Architektur, welche sie auf Basis des Architekturmusters zusammenführt. Die ☆ konzipierte architektonische Integration unterstützt die erhobenen Anforderungen und sie erlaubt es, die Vorteile der vorgenannten Konzepte gemeinsam zu nutzen. Hierfür wurde die Architektur in drei großen Blöcken gestaltet: die zu entwickelnde

[2] Mangels existierender Begriffe wird der Prozess der Interpretation und anschließender Anzeige, zu-Gehör-Bringung etc. als "zur Interaktion bringen" bezeichnet.

Multi-Benutzerschnittstelle selbst, der Infrastrukturknoten, welcher sowohl zur Entwicklungs- als auch zur Laufzeit genutzt wird, und die Entwicklungsumgebung. Ein in diesem Rahmen entwickeltes Erweiterungskonzept ermöglicht die einfache und schnelle Erweiterung um einzelne Komponenten oder ganze Sprachen.

Realisierung und Evaluation (Teil III)

Prototypische Umsetzung Der hier vorgestellte Ansatz wurde prototypisch umgesetzt. Dabei wurde auf Seite der Entwicklerunterstützung die frei verfügbare Java Entwicklungsumgebung Eclipse erweitert. Die implementierten Infrastrukturkomponenten werden einerseits von dieser genutzt und wurden andererseits in die Laufzeitumgebung eines anwendungsgetriebenen Forschungsprojektes (SoKNOS) integriert. Mit Hilfe der umgesetzten Entwicklungsumgebung wurde eine Multi-Benutzerschnittstelle entwickelt, welche im Projektdemonstrator lief.

Evaluation Die Evaluation der vorliegenden Arbeit unterscheidet sich grundlegend von der verwandter Arbeiten, weil sie zum Einen eine ☆ ausführliche Diskussion der Methode und zum Anderen eine ☆ Nutzerstudie mit berufsmäßigen Entwicklern von Benutzerschnittstellen beinhaltet. Beide Punkte sind in den verwandten Arbeiten selten zu finden. Darüber hinaus wird eine Fallstudie aus dem anwendungsorientierten SoKNOS Projekt einbezogen, sodass in Summe die Praxisrelevanz und Übertragbarkeit des Ansatzes auf andere Nutzungssituationen gezeigt werden konnte.

1.2 Aufbau der Arbeit

Die vorliegende Arbeit ist in vier Teile gegliedert, die nahezu 1:1 den im Kapitel 1.1 vorgestellten Beiträgen entsprechen. Es ergibt sich folgende Struktur.

Analyse des Standes der Wissenschaft (Teil I): Nach Beleuchtung des historischen Kontextes der Arbeit (Kapitel 2), wird im Detail auf die Modellierung von Benutzerschnittstellen eingegangen, wobei auch Vor- und Nachteile von Modellierungsansätzen erarbeitet werden (Kapitel 3). Darauf folgt eine detaillierte Diskussion der Begriffe der Problem- und der Lösungsdomäne (Kapitel 4), wobei Inkonsistenzen aufgezeigt werden und abschließend die abstraktionsunabhängige Benutzerschnittstelle (UUI) eingeführt wird. Im Folgenden werden Anforderungen erarbeitet (Kapitel 5), auf Basis derer die verwandten Arbeiten schließlich bewertet werden (Kapitel 6).

Konzept (Teil II): Nach einem Überblick über das Lösungskonzept (Kapitel 7), wird das entwickelte Architektmuster eingeführt (Kapitel 8). Dabei werden auch Design-Entscheidungen besprochen und das Architekturmuster in Abgrenzung zu MVC und PAC diskutiert. Als Werkzeug zur Strukturmodellierung wird die domänenspezifische Sprache im Folgenden (Kapitel 9) eingeführt, wobei auf die Semantik, abstrakte Syntax und Wohlgeformtheitsregeln eingegangen wird. Die konkrete Syntax wird durch die Unterstützungskonzepte bestimmt, welche im Anschluss besprochen werden (Kapitel 10), und sich in explorative und interaktive Unterstützungskonzepte aufteilen lassen. Darauf wird die architektonische Integration besprochen (Kapitel 11), wobei auch auf das Erweiterungskonzept eingegangen wird, sowie die drei großen Blöcke der Architektur (Infrastrukturknoten, Multi-Benutzerschnittstelle und Entwicklungsumgebung). Eine abschließende Zusammenfassung (Kapitel 12) geht auch auf einen konzeptuellen Abgleich der Anforderungen und methodische Implikationen ein.

Realisierung und Evaluation (Teil III) Der Teil beginnt mit einer Beschreibung der prototypischen Realisierung auf Basis von Java (Kapitel 13). Dabei wird auf den Infrastrukturknoten und die Multi-Benutzerschnittstelle eingegangen. Die in Eclipse integrierte Entwicklungsumgebung wird im Anschluss behandelt (Kapitel 14) und im Detail auf die Editoren sowie die Verknüpfung von Lauf- und Entwicklungszeit eingegangen. Das Anwendungsbeispiel wird darauf eingeführt (Kapitel 15), wobei auf die Integration in das SoKNOS System eingegangen wird und Schlussfolgerungen daraus gezogen werden. Darauf baut auch die Fallstudie auf, welche im Folgenden behandelt wird (Kapitel 16). Es werden hierbei die entwickelten MBS-Varianten vorgestellt, die Durchführung einer Verfeinerung im Detail besprochen, sowie Ergebnisse und der Abgleich mit den Anforderungen auf Basis der Fallstudie festgehalten. Abschließend wird die Nutzerstudie vorgestellt (Kapitel 17), beginnend mit der Wahl der Evaluationsmethode, dann der Beschreibung des Vorgehens und schließlich der Vorstellung der wesentlichen Ergebnisse. Der Teil schließt ab mit einer zusammenfassenden Bewertung (Kapitel 18).

Fazit (Teil IV) Nach einer Zusammenfassung der Ergebnisse der Arbeit (Kapitel 19) wird auf mögliche Anschlussforschung eingegangen (Kapitel 20).

Anhang (Teil V) Im Anhang finden sich die genutzten Materialien zur Nutzerstudie (Kapitel A) sowie die Ergebnisse der Studie im Detail (Kapitel B).

1.3 Konventionen

Dieser Abschnitt gibt einen Überblick über die Konventionen, welche in der vorliegenden Arbeit genutzt werden.

1.3.1 Konventionen für Hervorhebungen

Begriffseinführungen Begriffe, welche im Text das erste Mal genutzt werden bzw. eingeführt werden, sind `mit fester Laufweite` hervorgehoben und über den Index auffindbar.

Definitionen Definitionen werden im Text geschrieben und durch eine Markierung auf dem Seitenrand hervorgehoben:

Definition Definitionsnummer: Text zur Definition des Begriffes.

Definition zu definierender Begriff

Zitate Zitate werden immer in "Anführungszeichen" gesetzt und *kursiv* geschrieben. Wichtige Zitate sind außerdem freigestellt.

Wichtige Begriffe Wichtige Aussagen und Begriffe eines Paragraphes werden *kursiv* gesetzt.

Thema Bezieht sich ein Paragraph auf ein spezielles Thema, so wird dieses **fett** hervorgehoben.

1.3.2 Begriffliche Konventionen

Des Weiteren werden verschiedene Begriffe festgelegt, um möglichen Mehrdeutigkeiten vorzubeugen.

`Benutzer`: Der Benutzer ist eine Person, welche mit der fertigen Benutzerschnittstelle interagiert.

`Nutzer`: Im Gegensatz zum Benutzer muss der *Nutzer eines Objektes* nicht unbedingt eine Benutzerschnittstelle benutzen, sondern kann auch andere Objekte nutzen. Es kann hiermit also sowohl der Benutzer sein, aber auch ein Entwickler, welcher ein Werkzeug nutzt. Auf Grund der leichten Verwechselbarkeit wird auf diesen Begriff weitgehend verzichtet.

`Klasse`: Mit Klasse ist eine Klasse im objektorientierten Sinne gemeint. Zum Beispiel die Klasse "JLabel" aus dem Java Swing Toolkit.

`Kategorie`: Im Gegensatz zum Begriff Klasse beschreibt der Begriff Kategorie eine Einteilung von (evtl. abstrakten) Elementen in verschiedene Gruppen. Diese muss nicht zwangsläufig mit Programmiersprachen zu tun haben. So können z.B. "Früchte" und "Gemüse" zwei Kategorien sein, in welche Objekte einsortiert werden können.

I

Analyse des Standes der Wissenschaft

Überblick über Teil I

Dieser Teil beschreibt den Stand der Wissenschaft. Hierzu wird nach einem kurzen historischen Abriss in Kapitel 2 die Herangehensweise der Modellierung in Kapitel 3 untersucht und abgegrenzt. Die grundlegenden Begriffe des Themengebietes werden daraufhin in Kapitel 4 eingeführt und diskutiert. Die Diskussionen werden hierbei ausführlicher geführt, da es bei den Begriffen Variationen bis hin zu Inkonsistenzen gibt.

Schließlich werden Anforderungen an ein Lösungskonzept zur Entwicklung mehrerer Benutzerschnittstellen für eine Anwendung in Kapitel 5 entwickelt. Die Diskussion der Anforderungen ist ein zentraler Punkt der vorliegenden Arbeit; sie schränkt die möglichen Lösungskonzepte ein. Wie auch die Diskussion der Begriffe hebt sich die Diskussion der Anforderungen von verwandten Arbeiten durch ihre größere Tiefe ab.

Basierend auf den Anforderungen werden am Ende des Teils in Kapitel 6 verwandte Arbeiten diskutiert und bewertet.

Kapitel 2

Historischer Kontext der Arbeit

Die vorliegende Arbeit baut auf bereits bestehenden Arbeiten auf, welche in Kapitel 6 detaillierter analysiert werden. Jenes Kapitel, wie auch die gesamte Betrachtung des Standes der Wissenschaft wird durch dies Kapitel 2 in einen historischen Kontext gesetzt. Dieser ist für ein tieferes Verständnis der Materie notwendig und bildet insbesondere auch die Grundlage für die Entwicklung der Anforderungen in Kapitel 5.

Benutzerschnittstellen (User Interfaces, UIs) entwickelten sich von anfänglich zeilenbasierten Terminals (wie z.B. VT100, IBM3270) hin zu komplexeren, grafischen Oberflächen (wie z.B. heutzutage Mac OS oder Microsoft Windows). Mit dieser Zunahme an Komplexität stellte sich auch die Frage, wie die Entwicklung der grafischen Benutzerschnittstellen unterstützt werden kann. UI Toolkits integrieren sich (auch heute noch) gut in genutzte Programmiersprachen und boten dem Entwickler eine erste Unterstützung (Szekely 1996) bei der Erstellung grafischer Benutzerschnittstellen. Einer der bekanntesten Vertreter dieser Generation ist das Garnet Toolkit von Myers u. a. (1990). Vereinfachten solche UI Toolkits schon die Programmierung der UI, zielten die folgenden Modelle auf mehr Unterstützung bei deren Erstellung ab. So wird aus Datenmodellen die Beschreibung der Struktur der Benutzerschnittstelle abgeleitet, indem zu den passende UI Elemente selektiert werden, wie z.B. in (Janssen, Weisbecker und Ziegler 1993). Die weitere Forschung (Puerta 1997, Schlungbaum 1996) brachte Domänenmodelle, wie z.B. in Trident (Vanderdonckt und Bodart 1993) hervor, welche die Beziehungen in der Domäne beschreiben können, und mündete in Anwendungsmodellen, mit welchen umfangreichere Verhaltensspezifikationen möglich sind (Puerta 1997).

Systeme, welche bei der Entwicklung der Benutzerschnittstelle mit Hilfe von Modellen den Entwickler unterstützen, wurden `User Interface Management Systeme` (`UIMS`) genannt. Bekannte UIMS-Vertreter dieser Zeit, wie z.B. *Mastermind* (Browne u. a. 1996, Szekely u. a. 1995) oder *MobiD* (Puerta 1997),

wurden von Pinheiro da Silva (2000) besonders zielführend untersucht. Er identifizierte dabei die folgenden essenziellen Modelle (mit semantischen Variationen) als zentral, welche auch von Van den Bergh und Coninx in einer viel zititerten Untersuchung (2004) genutzt wurden:

- *Anwendungsmodell:* Dieses beschreibt die für die Benutzerschnittstelle relevanten Eigenschaften der Anwendung.
- *Aufgaben-Dialog-Modell:* Es spezifiziert die mit der Anwendung durchführbaren Aufgaben und ihre wechselseitigen Beziehungen[1].
- *Abstraktes Benutzerschnittstellenmodell:* Dieses beschreibt die Benutzerschnittstelle und ihre Elemente in ihrer Struktur (Layout[2]) und ihr Verhalten in grober Granularität auf konzeptioneller Ebene.
- *Konkretes Benutzerschnittstellenmodell:* Hiermit wird die Benutzerschnittstelle in ihrer Struktur im Detail beschrieben.

Mitte bis Ende der 90er Jahre gab es grob zwei Ansätze, welche die UIMS verfolgten, wie Szekely resümiert (1996). Auf der einen Seite standen *spezifikationsbasierte Ansätze*, welche sich auf die Unterstützung des Entwicklers konzentrierten. Dem gegenüber standen *automatisierende Ansätze*, welche eine Verbesserung der Entwicklungseffizienz als ihr primäres Ziel ansahen.

Die UIMS-Ansätze konnten sich jedoch nicht durchsetzen und traten in den Hintergrund (Dan R. Olsen 2007). Unter anderem Myers, Hudson und Pausch (2000) sowie Calvary und Pinna (2008) sehen zentrale Gründe in der schlechten Nutzbarkeit der produzierten Benutzerschnittstellen. Darüber hinaus machte die Standardisierung auf das WIMP-Interaktionsparadigma mit einem Standard-PC, Maus und Tastatur größere Anpassungen an verschiedene Interaktionsgeräte überflüssig. Der alternative Ansatz so genannter `Interface Builder`, welche direkte, grafische Manipulation zur Erstellung grafischer Benutzerschnittstellen erlaubten, findet hingegen große Verbreitung.

Obwohl die Interface Builder die UIMS verdrängten, ging die Entwicklung der UIMS weiter. In der Folge wurde auf Aufgabenmodelle verstärkter Wert gelegt. Die *ConcurTaskTrees (CTT)*-Notation von Paternò, Mancini und Meniconi (1997) ist eine Weiterentwicklung von GOMS und UAN, welche sich temporale Operatoren aus LOTOS borgt. Sie hatte großen Erfolg und wurde von vielen weiteren Arbeiten als Grundlage genutzt (wie z.B. in Bergh und Coninx 2004, Feuerstack, Blumendorf und Albayrak 2007, Limbourg u. a. 2004). Ein dabei wohl wichtiger Faktor für den Erfolg von CTT war die einfach zu

[1] Die später in Kapitel 4.2 genauer beschriebenen Aufgaben- und Dialogmodelle sind in Pinheiro da Silva's Papier zur besseren Übersicht in eine Klasse einsortiert.

[2] in der gesamten Arbeit wird der Begriff Struktur genutzt, er kann für grafische UIs mit Layout synonym verwendet werden

nutzende, frei verfügbare Editierumgebung (CTTE) mit ihrem Transformationswerkzeug Teresa (Mori, Paternò und Santoro 2004)[3]. CTTE erlaubt die einfache Erstellung und Weiterverarbeitung von CTT-Modellen.

Zur gleichen Zeit, in der CTTE und Teresa entstanden, wurde das vielzitierte Cameleon-Referenz-Framework (Calvary u. a. 2003) entwickelt. Es beschreibt konzeptionell, wie modellbasierte Ansätze, welche Benutzerschnittstellen für mehrere Zielplattformen erstellen, aufgebaut sein können. Eine Teilmenge der dabei identifizierten Modelle liegt semantisch nahe bei denen aus der Arbeit von Pinheiro da Silva, welche bereits in diesem Kapitel beschrieben wurden. Das Cameleon-Referenz-Framework ist eine wichtige Arbeit auf dem Gebiete, welche heute noch immer zitiert wird. Gleichzeitig dokumentiert es eine Bewegung in der Motivation: weg von der Vereinfachung der Programmierung einer einzelnen Benutzerschnittstelle hin zur Fragestellung der Entwicklung multipler Benutzerschnittstellen für eine Anwendung. Diese Motivation liegt auch der vorliegenden Arbeit zu Grunde. Die verschiedenen Varianten der Benutzerschnittstelle sind hierbei für verschiedene *Nutzungskontexte* gedacht, welche sich nach Benutzer, Plattform und Umgebung klassifizieren lassen (Calvary u. a. 2003).

Nach dem beschriebenen Abflauen des Interesses an der modellgetriebenen Entwicklung von Benutzerschnittstellen in den 90er Jahren nimmt es seit den ersten Jahren des 21ten Jahrhunderts wieder zu. Calvary und Pinna sehen den Grund für das steigende Interesse an diesem Forschungsgebiet in der wachsenden Vielfalt der verfügbaren Interaktionsgeräte (2008), eine Annahme, die Myers, Hudson und Pausch auch schon trafen (2000). Die Verfügbarkeit verschiedener Interaktionsgeräte für breitere Bevölkerungsgruppen zählt daneben sicherlich auch zu den Einflussfaktoren, ebenso wie die immer mobileren Nutzungsformen des Internets, welche in der Zukunft auch weiter zunehmen werden (Münchner Kreis e.V. U. a. 2009). Dabei werden zunehmend maßgeschneiderte Lösungen für verschiedene Nutzungskontexte verlangt.

Der starke Fokus auf der Abdeckung möglichst vieler Nutzungskontexte brachte viele Arbeiten im Bereich der Automatisierung hervor (z.B. Calvary u. a. 2004, Nichols u. a. 2002, Stanciulescu u. a. 2005, Zaplata u. a. 2009). Die mit dem Fokus auf Automatisierung einhergehende Problematik ist evident: schon die UIMS wurden insbesondere auf Grund ihrer schlechten Nutzbarkeit nicht akzeptiert und von den Interface Buildern verdrängt. Diverse Arbeiten kritisieren daher auch die schlechte Nutzbarkeit und Ästhetik der produzierten Benutzerschnittstellen (z.B. Demeure u. a. 2008, Meskens u. a. 2008, Myers, Hudson und Pausch 2000)[4].

Technisch gesehen bewegt sich das Gebiet dabei auf die *Nutzung von Modellen zur Laufzeit (Models at Runtime)* zu. Hierbei stehen die verschiedenen, zur

[3] http://giove.isti.cnr.it/tools/CTTE

[4] Die Autoren in (Calvary u. a. 2008) benutzen in diesem Zusammenhang den interessanten Ausdruck “Fast Food UIs”.

Entwicklungszeit erstellten Modelle der Benutzerschnittstelle auch zur Laufzeit zur Verfügung. Dies steht im Kontrast zum konventionellen Vorgehen, die entwickelten Modelle für die Ausführung zu kompilieren, was mit einem Informationsverlust einhergeht. Die Möglichkeit, durch Modelle zur Laufzeit die Benutzerschnittstelle zu betreiben, wurde schon früher erkannt und genutzt, wie aus den Übersichten von Pinheiro da Silva (2000) sowie Szekely (1996) hervorgeht. Jedoch wird es erst in den letzten Jahren auf dem Gebiet der Entwicklung multipler Benutzerschnittstellen für eine Anwendung systematisch eingesetzt und entwickelt. Aktuelle Arbeiten untersuchen dabei, welche Modelle notwendig sind (Sottet, Calvary und Favre 2006), wie passende Metamodelle aufgebaut sind (Blumendorf u. a. 2008, Veit Schwartze 2009) und die Möglichkeiten der Modellierung zur Laufzeit (Lehmann, Blumendorf und Albayrak 2010). Die vorliegende Arbeit folgt diesem Vorgehen und nutzt Modelle zur Laufzeit.

Diese kurze geschichtliche Darstellung führt in den historischen Kontext der vorliegenden Arbeit ein. Auch zeigt sie Punkte auf, welche bei der Entwicklung von Lösungskonzepten beachtet werden müssen und Teil der Diskussion in Kapitel 5 sind: insbesondere der Satz genutzter Modelle und ihre Abstraktionsebenen sowie die Nutzbarkeit des Ansatzes für den Entwickler.

Kapitel 3

Modellierung

Der Begriff des `Modells` ist für viele Arbeiten auf dem untersuchten Themengebiet von zentraler Bedeutung und wird daher hier diskutiert. Spezifikationen von UIs, auch auf verschiedenen Abstraktionsebenen, werden hierbei als Modell bezeichnet. Die Diskussion von modellgetriebenen Techniken bildet auch die Grundlage für deren Einsatz im entwickelten Lösungskonzept (Teil II).

Der Modellbegriff wird dafür im Folgenden eingeführt um dann in Kapitel 3 auf Modellierung im Rahmen von Software-Entwicklung einzugehen. Das für die vorliegende Arbeit wichtige Merkmal der Trennung von abtrakter und konkreter Syntax wird in Kapitel 3.2 untersucht. Nach einer Diskussion der Vor- und Nachteile modellgetriebener Ansätze in Kapitel 3.3 wird abschließend in Kapitel 3.4 auf die semantische Tiefe von Modellierungsprimitiven eingegangen.

Stachowiak charakterisiert in seiner grundlegenden Arbeit (1973) ein Modell durch drei Eigenschaften, welche Ludewig (2003) aufgreift und weiter für den Bereich Softwareentwicklung diskutiert. Stachowiak charakterisiert ein Modell in drei Dimensionen:

1. Abbildung.

> "*Ein Modell ist immer ein Abbild von etwas, eine Repräsentation natürlicher oder künstlicher Originale, die selbst wieder Modelle sein können.*"

Modelle von Benutzerschnittstellen sind also Abbildungen derselben, wobei das "Original" erst nachträglich mit Hilfe des Modells erschaffen wird. Bei Ansätzen, welche mehrere Abstraktionsebenen durchlaufen, können wiederum abstraktere Spezifikationen Modelle von konkreteren Spezifikationen sein.

2. Verkürzung.

> "*Ein Modell erfasst nicht alle Attribute des Originals, sondern nur diejenigen, die dem Modellschaffer bzw. Modellnutzer relevant erscheinen.*"

Bei der Modellierung wird also nicht die gesamte Benutzerschnittstelle (oder das konkrete Modell) per se spezifiziert, sondern nur die Eigenschaften, welchen im jeweiligen Ansatz für die Schaffung der Benutzerschnittstelle (des konkreten Modells) benötigt werden. Der Übersetzungsprozess, d.h. eine Interpretation des Modells (Mellor und Balcer 2002), fügt die weiteren Informationen hinzu. Die für die Interpretation relevanten Eigenschaften müssen dabei dem Modell innewohnen. Letzteres beschreibt die folgende Eigenschaft: Pragmatismus.

3. Pragmatismus.

> "*Pragmatismus bedeutet soviel wie Orientierung am Nützlichen. Ein Modell ist einem Original nicht von sich aus zugeordnet. Die Zuordnung wird durch die Fragen "Für wen?", "Warum?" und "Wozu?" relativiert. Ein Modell wird vom Modellschaffer bzw. Modellnutzer innerhalb einer bestimmten Zeitspanne und zu einem bestimmten Zweck für ein Original eingesetzt. Das Modell wird somit interpretiert.*"

Das Modell wird also gezielt für bestimmte Teile des Erstellungsprozesses der Benutzerschnittstelle eingesetzt.

Es macht also Sinn, im Rahmen dieser (und der verwandten) Arbeit von Modellen zu sprechen, denn die Beschreibungen von Benutzerschnittstellen in der Literatur gehorchen im Allgemeinen den drei Kriterien von Stachowiak.

3.1 Grundlagen

Im Rahmen `modellgetriebener Softwareentwicklung` (`MDSD`, Model-Driven Software Development) wird Modellierung eingesetzt. Geht man vom abstrakten Stachowiak'schen Modellbegriff zur Anwendung eines Modells im Rahmen von Modellierung über, stellt sich die Frage, welche Aspekte hierfür definiert werden müssen. Stahl u. a. (2007) sowie Atkinson und Kühne (2003) sehen als zentrale Elemente einer Modellierungssprache vier Aspekte an: Semantik, abstrakte Syntax, konkrete Syntax, und Regeln zur Wohlgeformtheit.

Semantik: Die Semantik definiert die Bedeutung der einzelnen Sprachkonstrukte und ihrer Anordnung. Dies ist wichtig, damit Modelle, welche in

der Sprache beschrieben sind, auch genutzt[1] werden können (Mellor und Balcer 2002).

Abstrakte Syntax: Diese beschreibt die Sprachkonstrukte formell, sie legt die Grammatik der Sprache fest. Die abstrakte Syntax zielt darauf ab, Elemente mit einer gewünschten Semantik formell darstellbar zu machen; Sie beinhaltet allerdings noch nicht die konkrete Darstellung derselben.

Konkrete Syntax: Im Gegensatz zur abstrakten legt die konkrete Syntax (hier synonym zu Notation genutzt) fest, "wie" in der Sprache modelliert wird. Das heißt, mit welcher Repräsentation der Sprache ein Entwickler umgehen muss. Die Trennung in abstrakte und konkrete Syntax ist untypisch für klassische Programmiersprachen (siehe Folgekapitel 3.2).

Bei einer textuellen, konkreten Syntax kann z.B. jedem abstrakten Element ein Wort (im Sinne von Text) zugeordnet werden. Darüber können z.B. Textfragmente definiert werden, beispielsweise Klammern, welche Beziehungen zwischen Elementen ausdrücken können. Ist die konkrete Syntax dagegen z.B. grafisch, werden konkrete Symbole für die abstrakten Syntaxelemente definiert und Beziehungen zwischen Elementen zum Beispiel durch Pfeile oder Schachtelungen von Elementen dargestellt.

Regeln zur Wohlgeformtheit: Mit Hilfe der abstrakten Syntax lassen sich nicht beliebige Bedingungen, wie valide Modelle aussehen dürfen, ausdrücken. Daher können weitere Regeln zur Wohlgeformtheit der Modelle definiert werden.

Bei der `UML` (Unified Modeling Language) (Object Management Group 2007) wird hierzu oft die Object Constraint Language (OCL) der Object Management Group (2006b) herangezogen.

Die Grammatik für Modelle wird mit einem `Metamodell` beschrieben. Dies beinhaltet die vier vorgestellten Aspekte der Semantik, abstrakten und konkreten Syntax, sowie die Regeln zur Wohlgeformtheit. Korrekte Modelle sind folglich konform zu einem gegebenen Metamodell. Abbildung 3.1 illustriert diesen Zusammenhang.

Das Konzept, valide Modelle durch ein Metamodell zu charakterisieren, kann wiederum auch auf das Metamodell selbst angewendet werden. Ein Metamodell wird dann durch ein `Meta-Metamodell` beschrieben (analog wie die EBNF genutzt werden kann, um Grammatiken zu beschreiben). Dies erlaubt z.B. die Normierung der Austauschformate zwischen verschiedenen Werkzeugen, welche alle auf Basis des gleichen Meta-Metamodells arbeiten. Auch erlaubt ein Meta-Metamodell z.B. die Erstellung einer Transformationsmaschine, welche verschiedene Metamodelle versteht, die alle zum gleichen Meta-Metamodell

[1] Mellor et al. nennen die Nutzung "Interpretation", weil im Rahmen jeglicher Nutzung interpretiert werden muss, was mit dem vorliegenden Modell "gemeint" ist.

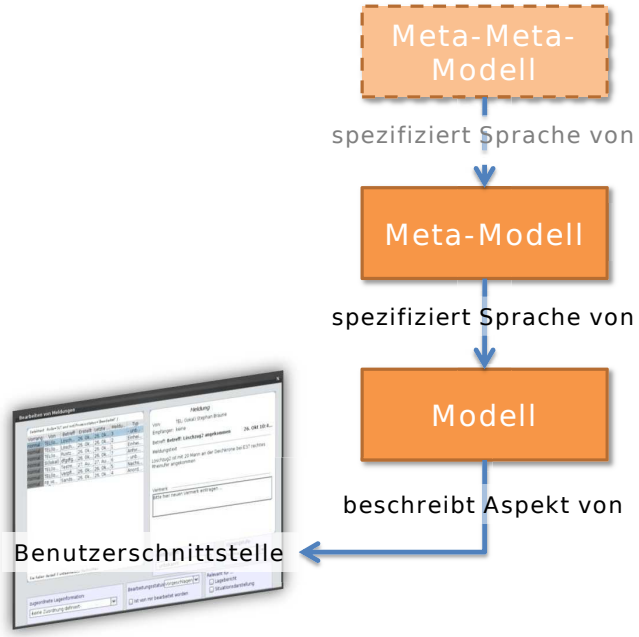


Abbildung 3.1: Ein Modell beschreibt einen bestimmten Aspekt einer Benutzerschnittstelle und ist konform zu einem Metamodell. Letzteres kann wiederum konform zu einem Meta-Metamodell (eine Sprache für Metamodelle) sein.

konform sind. Ein prominentes Beispiel für ein Metamodell ist die UML. Deren Meta-Metamodell ist die MOF (Meta Object Facility) (Object Management Group 2006a).

3.2 Trennung von abstrakter und konkreter Syntax

Wie der vorige Abschnitt nahe legt, sind die Begriffe des Metamodells und der Sprache eng verwandt. Jedoch unterscheiden sich klassische Ansätze von Programmiersprachen in einem Punkt essentiell von Ansätzen zur Modellierung: die Trennung der Sprache in abstrakte und konkrete Syntax ist für Modellierungsansätze ein deutlich wichtigeres Kriterium.

Eine Programmiersprache nutzt im Allgemeinen strukturierte Texte zur Spezifikation. Diese gehorchen einer Grammatik, welche z.B. in Bachus-Naur-Form vorliegt. Die Grammatik legt hierbei die abstrakte Syntax, also die Struktur, fest: Welche Elemente können wie miteinander in Beziehung gesetzt werden. Darüber hinaus wird durch sie aber auch die konkrete Syntax, also das Aussehen der Sprache, beschrieben. Die Wörter der Grammatik können in der textuellen Spezifikation genutzt werden. Beim Speichern einer Spezifikation wird diese als Text, also in ihrer konkreten Syntax, gespeichert, und damit

implizit auch die abstrakte.

In der Modellierung wird stark zwischen abstrakter und konkreter Syntax unterschieden, wobei die abstrakte Syntax die maßgebende ist. Ein Modell in abstrakter Syntax kann mehrere konkrete Syntaxvarianten (Darstellungen) haben (z.B. zwei grafische und eine textuelle). Solch konkrete Darstellungen bestimmen maßgeblich das "Gefühl", welches der Entwickler beim Nutzen der Sprache hat. Sie müssen jedoch als (zumindest teilweise) transient angesehen werden. So müssen sie sich bei Änderung der abstrakten Beschreibung mit anpassen. Weiter kann die abstrakte Beschreibung als maßgebende bezeichnet werden, da in ihr die Inhalte abgelegt sind. Unter anderen nutzt Markus Voelter dieses Kriterium zur Differenzierung zwischen klassischen Programmiersprachen und Modellierungsansätzen[2].

3.3 Vor- und Nachteile von Modellierung

Verschiedene Autoren, wie z.B. Hailpern und Tarr (2006), Pinheiro da Silva (2000), Stahl u. a. (2007), sehen die Möglichkeit abstraktere[3] Spezifikationen zu nutzen als großen Vorteil von Modellierungsansätzen. Ebenso wird die Möglichkeit zur Reduktion der Spezifikationskomplexität als Vorteil gesehen. Modelle bieten wohldefinierte Variationspunkte, somit muss nicht alles "ausprogrammiert" werden. Des Weiteren forciert ein Modellierungsansatz eine einheitliche Architektur aller Komponenten und erhöht somit die *Qualität und Wartbarkeit* des Produkts (Stahl u. a. 2007).

Eine höhere Effizienz durch Reduktion der "Tippzeit" wird von Stahl et al aber verneint. Ihrer Meinung nach wird dieser Zeitgewinn durch einen erhöhten Denkaufwand wieder egalisiert. Dies deckt sich mit Feststellungen anderer Autoren, die in der *Komplexität der Nutzung modellgetriebener Ansätze* einen Nachteil sehen (Hailpern und Tarr 2006, Pinheiro da Silva 2000). Hailpern und Tarr (2006) sehen weiter *Gefahren durch Round-Trip-Probleme*[4] und Redundanzen, welche durch Modellierungsansätze erzeugt werden. Aber auch beim Programmieren werden Code Doubletten erzeugt. Das Problem muss folglich durch adäquate Unterstützung des Entwicklers adressiert werden. Dies ist Teil der Anforderungen in Kapitel 5.

Letztendlich liefert aber ein gutes Metamodell auch eine *einheitliche Sprachdefinition* für ein Projekt bzw. eine Domäne. Diese Sprachdefinition findet zum Einen im Rahmen der Modellierung, zum Anderen auch für die verbale Kommunikation, in der auch die Sprachkonstrukte der Modellierung genutzt werden sollten (Stahl u. a. 2007), Anwendung.

Abschließend ist zu bemerken, dass modellgetriebene Techniken zur Er-

[2] Ausgeführt z.B. bei einem Vortrag vor der Mannheim Java User Group am 04.11.2009

[3] an Stelle von "abstrakt" wird in der Literatur oft der Begriff "deklarativ" genutzt

[4] D.h. die Änderung eines Artefakts impliziert Änderung von anderen Artifakten, weil sonst ein inkonsistenter Gesamtzustand entsteht.

stellung von Benutzerschnittstellen bei Adressierung der wachsenden Anzahl von Interaktionskontexten als vielversprechende Kandidaten gehandelt werden (Myers, Hudson und Pausch 2000, Szekely 1996).

3.3.1 Aktuelle Entwicklungen auf dem Feld der Modellierung

Klassische Modellierungsansätze fokussieren grafische Notationen als konkrete Syntax. Dabei gehen einige Vertreter sehr weit und beschreiben auch Algorithmen in grafischer Notation – z.B. Executable UML oder das SerCHo Projekt (Blumendorf u. a. 2008, CARE Technologies u. a. 2008, Mellor und Balcer 2002). Neuere Ansätze haben verdeutlicht, dass man nicht für alles grafische Notationen einsetzen kann[5] und nutzen textuelle Notationen ggf. in Kombination mit grafischen. Die Wahl der dem Entwickler *zur Verfügung stehenden Notationen* muss also mit Bedacht getroffen werden.

Ein Beispiel für die neueren Ansätze ist das Meta Programming System von JetBrains[6] , welches ein Metamodell für Java beinhaltet und eine textuelle Notation zur Modellierung anbietet. Auch die Intentional Domain Workbench von Intentional Software[7] ist hier anzuführen. Sie kombiniert grafische und textuelle Notationen einer Sprache Hand in Hand, ist aber leider nicht öffentlich verfügbar[8].

3.4 Semantische Tiefe der Modellierungsprimitive

Bei der klassischen Programmierung von Benutzerschnittstellen werden diese mit Hilfe von Toolkits meist indirekt erstellt. Dazu wird Code in einer generischen Programmiersprache (z.B. Java oder C) geschrieben, welcher die Benutzerschnittstelle dann zur Laufzeit erstellt.

In Microsoft's Visual Studio wird ein anderer Ansatz verfolgt. Ein `Interface Builder` (grafischer Editor) wird genutzt, mit Hilfe dessen man die Schnittstelle "bauen" kann. Im Hintergrund wird der passende (C#, ...) Quellcode zur Erstellung der Benutzerschnittstelle zur Laufzeit erzeugt. Im Gegensatz zum rein programmatischen Vorgehen wird der Entwickler mit Hilfe eines grafischen Werkzeugs unterstützt. Gemeinsam haben diese beiden Ansätze, dass die produzierten Artefakte die minimal notwendige Information zur Anzeige und Ausführung der Benutzerschnittstellen enthalten. Sie liegen am Ende immer indirekt als Programm, welches die Benutzerschnittstelle zur Laufzeit erzeugt, vor.

[5] Einen Einstieg in diese Diskussion liefert z.B. das Papier von Myers und Ko (2009) über die Zukunft der Benutzerschnittstellenprogrammierung.

[6] `http://www.jetbrains.com/mps/index.html`

[7] `http://www.intentsoft.com`

[8] Der Mantel des Schweigens wurde kürzlich durch eine Demo gebrochen, vgl. auch Fowler's Bliki Eintrag unter http://martinfowler.com/bliki/IntentionalSoftware.html.

Generische Sprachen (vor allem in der klassischen Programmierung), wie z.B. Java oder C#, bieten naturgemäß sehr allgemeine Konstrukte an, um eine breite Abdeckung von Problemdomänen zu erhalten. Ihre Semantik ist aber somit sehr allgemein: die Konstrukte sind *semantisch weniger tief*. Modellgetriebene Ansätze dagegen spezialisieren sich meist stärker auf eine Domäne und stellen meist stärker spezialisierte Sprachen zur Modellierung auf unterschiedlichen Abstraktionsebenen bereit, so z.B. für Aufgabenmodelle, Klassendiagramme oder konkrete Benutzerschnittstellenmodelle. Ihre (spezialisierten) Sprachkonstrukte sind somit auch semantisch tiefer als die allgemeiner Sprachen. Solche spezialisierten Sprachen werden `domänenspezifische Sprachen` (Domain Specific Languages, `DSL`) genannt.

Programmier-Frameworks werden in generischen Sprachen erstellt und bieten Konstrukte der jeweiligen Sprache an, welche ein bestimmtes Problem spezifischer adressieren. So bietet z.B. das EMF-Framework[9] Klassen mit Methoden an, welche das Laden und Speichern von Modellen vereinfachen. Im Gegensatz dazu bieten Modellierungsansätze eine echte, syntaktische Einschränkung mit einer domänenspezifischen Sprachsemantik. Das erlaubt z.B. auch, Wohlgeformtheitsregeln über die speziellere Semantik zu formulieren und damit statische Prüfungen zur Modellierungszeit vorzunehmen (Stahl u. a. 2007). Die Interpretation von domänenspezifischen Modellen erlaubt es somit gegenüber generischen Sprachen spezifischere Ergebnisse für den Kontext der Domäne zu liefern.

Da DSLs damit eine bessere `maschinenlesbare Semantik` als generische Sprachen haben, ist ihre Nutzung vorteilhaft im Bezug auf die Erstellung (semi-)automatischer Unterstützungswerkzeuge für Entwickler. Im Endeffekt stehen mit semantisch tieferen Sprachen der Entwicklungsumgebung (und bei Übernahme der Modelle in die Laufzeit auch der Laufzeitumgebung) mehr Informationen als bei klassischen Programmieransätzen zur Verfügung.

Die semantisch tieferen Konstrukte domänenspezifischer Sprachen zielen hierbei meist auf eine bessere Abbildung der Problem- oder Lösungskonzepte der Domäne ab. Dies bezieht sich auf die Pragmatismus-Eigenschaft von Modellen (vgl. Kapitel 3) und ermöglicht auch deren Verkürzung (die zweite Modelleigenschaft nach Stachowiak) im Gegensatz zu einer ausimplementierten Lösung in einer generischen Programmiersprache.

Die durch die semantisch tieferen Konstrukte zusätzliche Information kann gewinnbringend eingesetzt werden. Szekely sieht hierdurch großes Potenzial, um zusätzliche “UI Dienste” (semi-)automatisch zu erzeugen, z.B. ausgeklügelte Hilfe-Systeme (1996). Farenc, Liberati und Barthet sehen diese zusätzliche Information als notwendig an, um über einfache Regeln bei der automatischen Nutzbarkeitsprüfung hinaus zu kommen (1996). Des Weiteren schlagen verschiedene Autoren die Nutzung der zusätzlichen Information im Adaptionsprozess einer Benutzerschnittstelle an einen neuen Nutzungskontext vor, z.B.

[9] EMF – Eclipse Modeling Framework

Calvary u. a. (2002), Coutaz und Calvary (2008), Demeure u. a. (2006), Sottet u. a. (2008).

Die Tiefe der Semantik der Sprachkonstrukte stellt auch ein grundlegendes Problem bei der Integration von Software und Usability Engineering dar. Im Bezug zu dieser Arbeit wurde der Aspekt in (Behring, Petter und Mühlhäuser 2009) diskutiert. Hierbei wird argumentiert, dass semantisch tiefere Sprachen die Integration besser unterstützen können.

3.5 Zusammenfassung

Modellierung wurde über die Trennung von abstrakter und konkreter Syntax in Kapitel 3 eingeführt. Modellgetriebene Ansätze speichern hierbei die Inhalte in der abstrakten Spezifikation ab, wohingegen klassische Programmiersprachen die konkrete Notation abspeichern.

In den Kapiteln 3.3 und 3.4 wurden die Vor- und Nachteile von modellgetriebenen Ansätzen diskutiert. Die wichtigsten Aspekte modellgetriebener Ansätze sind:

- die durch eine forcierte Architektur erhöhte Qualität und Wartbarkeit,
- die Bereitstellung wohldefinierter Variationspunkte,
- der reduzierte Spezifikationsumfang, welcher durch die erhöhte Komplexität bei der Nutzung modellgetriebener Ansätze aber wieder egalisiert wird,
- die Gefahr durch Round-Trip-Probleme und Doubletten,
- die Bereitstellung einer einheitlichen Sprachdefinition (für die Modellierung und die Kommunikation),
- die größere semantische Tiefe und maschinenlesbare Semantik, welche vorteilhaft im Bezug auf die Erstellung (semi-) automatischer Unterstützungswerkzeuge ist, und
- der größere Informationsumfang, welcher für zusätzliche “UI Dienste” ausgenutzt werden kann.

Nach Abwägung der Vor- und Nachteile von Modellierung bzw. modellgetriebene Softwareentwicklung mit spezialisierten Sprachen wird der Schluss gezogen, dass sich ein modellgetriebener Ansatz zur Erstellung mehrerer Benutzerschnittstellen für eine Anwendung aus unterschiedlichen Gründen anbietet. Diese sind im Besonderen:

- Die Erstellung einer Benutzerschnittstelle umfasst viele Aspekte. Eine Trennung von abstrakter und konkreter Syntax erlaubt es, die verschiedenen Aspekte in der für den jeweiligen Aspekt sinnvollen konkreten Syntax zu bearbeiten.

- Gleichzeitig können diese verschiedenen Aspekte sich aber gegenseitig aufeinander beziehen, da sie auf einer einheitlichen abstrakten Sprache basieren.
- Spezialisierte Modelle mit klar definierten Variationspunkten und einer speziell auf die Domäne angepassten DSL reduzieren die Spezifikationskomplexität und fördern eine konsistente Systemarchitektur, was wiederum zu besserer Wartbarkeit führt.
- Schließlich definiert die DSL klare Schnittstellen und Artefakte, welche speziell auf die Eigenarten der Entwicklung von Benutzerschnittstellen zugeschnitten sind und die hierfür relevante Information enthalten.

Dennoch können nicht alle Aspekte des Systems durch spezialisierte Modellierungssprachen abgedeckt werden. Einige Bereiche, wie z.B. Berechnungen und Algorithmen, sind in generischen Sprachen sehr gut darstellbar. Eine Integration modellgetriebener Techniken und generischer Programmiersprachen ist somit erforderlich (Anforderung 5).

Kapitel 4

Begriffe und Definitionen

Nach der Einführung und Diskussion des Konzeptes der Modellierung im vorigen Kapitel 3 werden im Folgenden weitere grundlegende Begriffe für die untersuchte Domäne eingeführt und besprochen. Soweit sinnvoll möglich, werden dabei deutsche Begriffe genutzt. Sind englische Begriffe in der Forschergemeinschaft schon gut etabliert, wird auf eine deutsche Übersetzung verzichtet.

Die folgende Systematik legt offen, dass die Begriffe nicht durchgängig definiert und genutzt werden. Es werden im Weiteren verschiedene Sichtweisen auf die Begriffe eingeführt und ihre Unterschiede beleuchtet. Die vorliegende Arbeit liefert hier einen wichtigen Beitrag: Sie führt die Diskussion der zentralen Begriffe des Themengebiets tiefergehend als in bisherigen Arbeiten zum Thema.

Für die Diskussion der Begriffe werden diese aufgeteilt. Begriffe der Problemdomäne (Kapitel 4.1) sind unabhängig von konkreten Lösungsansätzen. Dagegen liegt Begriffen der Lösungsdomäne (Kapitel 4.2) ein bestimmter Lösungsansatz oder eine Klasse von Lösungsansätzen zu Grunde. Die identifizierten Inkonsistenzen zwischen abstrakter und konkreter Benutzerschnittstelle (AUI und CUI) werden schließlich in Kapitel 4.3 gelöst, indem diese Dichotomie durch den Begriff der abstraktionsunabhängigen Benutzerschnittstelle aufgehoben wird.

4.1 Begriffe der Problemdomäne

Dieser Abschnitt behandelt allgemeine Begriffe, unabhängig von konkreten Lösungsansätzen und -domänen.

4.1.1 Multi-Benutzerschnittstelle

Wird eine Anwendung mit mehreren Benutzerschnittstellen ausgestattet, so wird hierfür in der Literatur oft einer der Begriffe `MUI` (`Multiple User Interface`, `Multi-Benutzerschnittstelle`), `Multi-Presentation User Interface`

oder `Multi-Target UI` genutzt. Seffah, Forbrig und Javahery (2004) führen den Begriff MUI mit Fokus auf Geräten und dem Ziel des Informationszugriffs ein: "*"Multiple-User Interface" (MUI) refers to an interactive system that provides access to information and services using different computing platforms.*" Etwas generischer bzgl. der unterstützten Kontexte und Aufgaben der Schnittstelle sind Collignon, Vanderdonckt und Calvary (2008): "*A multi-presentation user interface is composed of a series of interconnected user interfaces for the same task to be carried out in different contexts of use.*"

Collignon, Vanderdonckt und Calvary machen die unterschiedlichen Geräte noch expliziter (2008): "*A Multi-target user interface is composed of a series of interconnected variations of the same user interfaces, but tailored for different targets or different contexts of use.*" Dabei wird in den beiden letzten Definitionen explizit hervorgehoben, dass die verschiedenen Versionen der Benutzerschnittstelle miteinander in Verbindung stehen.

Helms u. a. (2009) auf der anderen Seite kodieren in ihre Definition schon einen Teil der Umsetzung: "*[...] multiple user interfaces from a single model for multiple contexts of use, in particular multiple computing platforms, thus [...] multichannel user interfaces*". Es gibt somit ein zentrales Modell, welches im Besonderen auf Geräte fokussiert.

In der vorliegenden Arbeit soll der Begriff ohne Bezug zu möglichen Lösungen (z.B. ein zentrales Modell), Geräten oder konkreten Anwendungsszenarien (z.B. Service-Orientierung) definiert werden:

Definition Multi-Benutzerschnittstelle (MBS)

Definition 1: Eine **Multi-Benutzerschnittstelle (MBS)** ist eine Menge von Variationen einer Benutzerschnittstelle, welche auf unterschiedliche Nutzungskontexte zugeschnitten sind.

Dabei bezeichnet der Begriff `MBS-Variante` (Variante einer Multi-Benutzerschnittstelle) in der Folge eine der multiplen Benutzerschnittstellen der MBS, zugeschnitten auf einen bestimmten Nutzungskontext. Sie kann sich sowohl in ihrem Verhalten als auch ihrer Struktur von anderen MBS-Varianten unterscheiden. Dies wird auch unter Anforderung 5 in Kapitel 5.5 diskutiert.

Zentral ist hierbei der Begriff des Nutzungskontextes. Die in dieser Arbeit verwendete Definition orientiert sich an der Diskussion in (Calvary u. a. 2003). Dort wird ein Nutzungskontext in die Teile Benutzer, Plattform und Umgebung aufgeteilt. Ein Nutzungskontext kann jedoch nur im Bezug zu einer Bestimmung definiert werden (Crowley u. a. 2002), daher wird er im Rahmen dieser Arbeit in Bezug zur Entwicklung gesetzt, da dort die Ziele der Entwicklung einer MBS festgelegt werden:

Definition 2: Der **Nutzungskontext** einer MBS-Variante wird definiert als die Menge aller Annahmen, welche über die Nutzung der MBS-Variante im Rahmen ihrer Entwicklung getroffen werden. Die Annahmen klassifizieren sich insbesondere nach den intendierten Nutzerkreisen, den Zielplattformen und der Umgebung bei der Nutzung.

Definition
Nutzungskontext

Die vorliegende Arbeit macht somit keine genauen Vorgaben, was alles in einem Nutzungskontext enthalten sein muss. Vielmehr erlaubt sie, diesen problemspezifisch zu festzulegen (vgl. Kapitel 7 und 9.1). Insbesondere müssen nicht alle drei Teile (Nutzer, Plattform und Umgebung) enthalten sein.

Der Begriff `User Interface`, bzw. seine Abkürzung `UI` wird sehr oft in der Literatur verwendet. Er wird auch in dieser Arbeit synonym zum Begriff der Benutzerschnittstelle gebraucht.

4.1.2 Plastizität und Plastizitätsdomäne

Der Nutzungskontext, kann sich während der Nutzung einer Multi-Benutzerschnittstelle ändern. In diesem Zusammenhang wurde der Begriff der `Plastizität` (Plasticity) von Thevenin und Coutaz (1999) definiert als "*plasticity is the capacity of a user interface to withstand variations of both the system physical characteristics and the environment while preserving usability.*" Plastizität ist also die Fähigkeit einer Benutzerschnittstelle, sich Variationen im Nutzungskontext anzupassen unter Aufrechterhaltung der Nutzbarkeit der Benutzerschnittstelle.

Die `Plastizitätsdomäne` (Plasticity Domain) hingegen ist die Menge der Nutzungskontexte, für welche sich eine gegebene Benutzerschnittstelle plastisch verhält, in welchem sie also ihre Gebrauchstauglichkeit (Usability) behält (Calvary, Coutaz und Thevenin 2001). Die logisch folgende Diskussion, was Nutzbarkeit bedeutet, in welchen Granularitäten man Plastizität messen sollte und könnte sowie wie oft man eine Anpassung des UIs zulassen darf, sind nicht Bestandteil dieser Arbeit.

Der Begriff der Plastizität wurde später in Calvary u. a. 2004 überarbeitet, um auch Veränderungen in weiteren Teilen eines interaktiven Systems (z.B. Anwendungslogik) mit in den Plastizitätsbegriff einzubeziehen. Im Rahmen der vorliegenden Arbeit kann von der Plastizitätsdomäne einer konkreten MBS-Variante, aber der gesamten Multi-Benutzerschnittstelle gesprochen werden.

4.1.3 UI Toolkit

Wie im Kapitel 2 beschrieben, werden Toolkits eingesetzt, um den Entwickler bei der Erstellung von Benutzerschnittstellen zu unterstützen. Ein `UI Toolkit` (oder kurz Toolkit, deutsch "Werkzeugsatz") wird im Rahmen dieser Arbeit

definiert als ein Satz von *i)* Entwicklungswerkzeugen zur Erstellung und Verbesserung einer Benutzerschnittstelle, *ii)* grundlegenden Elementen, aus denen eine Benutzerschnittstelle aufgebaut werden kann (meist in Form von Konstrukten einer Sprache) und *iii)* Hilfsfunktionen und -programmen, welche durch den Entwickler zur Erstellung und Verbesserung eingesetzt werden können (meist in Form von APIs, Frameworks oder Bibliotheken). Dabei stellt ein Toolkit immer bestimmte Anforderungen an die Plattform, auf welcher es laufen soll. Auch hat es einen bestimmten Stil, Dinge darzustellen[1].

Basiert eine Benutzerschnittstelle auf einem Toolkit, benötigt es dieses (oder eine Laufzeitumgebung des Toolkits) auch zur Darstellung der Benutzerschnittstelle. Für die vorliegende Arbeit ist somit insbesondere wichtig, dass ein *Toolkit ein Ziel-System* beschreibt, auf welchem eine MBS-Variante aufsetzt. Des Weiteren haben die grundlegenden Elemente eines Toolkits zum Aufbau einer Benutzerschnittstelle eine Bedeutung, da sie im Rahmen der vorliegenden Arbeit zur Modellierung herangezogen werden (vgl. dazu Kapitel 7.3). *Ein Toolkit bildet somit eine Sprache für die Modellierung.* Umgekehrt ist aber nicht jede Sprache ein Toolkit (z.B. eine Sprache zur Aufgabenmodellierung). Beispiele für Toolkits sind Java Swing, spezielle Hardware Toolkits (wie in Kapitel 13.3.2 vorgestellt) oder auch HTML.

4.1.4 Erstellungs- und Modifikationsproblem

Im Rahmen der Entwicklung einer MBS können zwei grundlegende Problemfelder unterschieden werden: das der Erstellung und das der Modifikation der MBS. Das `Erstellungsproblem` bezieht sich auf die Herausforderungen, welche spezifisch für die Erstellung einer oder mehrerer neuer MBS-Varianten zu bewältigen sind. Im Gegensatz dazu hat das `Modifikationsproblem` die Modifikation einer oder mehrerer MBS-Varianten zum Inhalt.

Die Charakteristika des **Erstellungsproblems** sind:

- Die MBS-Varianten für die adressierten Nutzungskontexte müssen initial konzipiert werden.
- Die konzipierten MBS-Varianten müssen mit Hilfe der Entwicklungswerkzeuge umgesetzt werden.
- Bei der Erstellung müssen keine bestehenden MBS-Varianten modifiziert werden.
- Der Entwickler muss den Erstellungsprozess der MBS-Varianten kontrollieren können:

[1] Das Wort “darstellen” passt nur für grafische Benutzerschnittstellen, soll aber im übertragenen Sinne auch für andere Arten von Benutzerschnittstellen (akkustisch, olfaktorisch, ...) verstanden werden.

  - von welchen bestehenden Varianten werden die neuen MBS-Varianten abgeleitet? Und
  - in welcher Form werden sie abgeleitet?

- Das Ergebnis sind eine oder mehrere neu erstellte MBS-Varianten.

Dabei ist festzustellen, dass Lösungsansätze in diesem Bereich folglich meist auf die (semi)automatische Erzeugung von MBS-Varianten abzielen. Oft steht dabei die breitflächige (automatische) Abdeckung möglichst vieler Nutzungskontexte im Vordergrund.

Dagegen sind die Charakteristika des **Modifikationsproblem**s anders gelagert:

- Eine oder mehrere bestehende MBS-Varianten müssen aktualisiert werden.
- Zu tätigende Änderungen an bestehenden MBS-Varianten müssen identifiziert werden.
- Die Änderungen müssen mit Hilfe der Entwicklungswerkzeuge umgesetzt (formuliert) werden.
- Der Entwickler muss den Modifikationsprozess kontrollieren können, dabei insbesondere:
  - welche Zusammenhänge zwischen den verschiedenen MBS-Varianten ggf. beachtet werden müssen,
  - in welcher Weise die Änderung die Anbindung der Anwendungslogik beeinflusst, und
  - wie die anstehenden Modifikationen in einen gemeinsamen Plan passen (gegenseitige Beeinflussung).
- Das Ergebnis sind eine oder mehrere modifizierte MBS-Varianten.

Collignon, Vanderdonckt und Calvary erkennen in diesem Zusammenhang (2008), dass bisherige Ansätze ungenügende Verbindungen zwischen den verschiedenen UIs einer MBS-Anwendung herstellen. Diese könnten aber das Anbringen von Änderungen erleichtern, wie später im Konzept in Teil II eingeführt.

4.2 Begriffe der Lösungsdomäne

Um die verwandten Arbeiten zu diskutieren, ist Verständnis für einige grundlegende Begriffe der Lösungsdomäne unabdinglich. Wir orientieren uns dazu an den von Pinheiro da Silva (2000) eingeführten Modellen, welche in vielen weiteren Arbeiten, z.B. auch dem Cameleon Referenz-Framework (Calvary u. a.

2003), genutzt werden. Diese Modelle können im Laufe der Entwicklung einer Benutzerschnittstelle ineinander überführt werden wie in Abbildung 6.2 in Kapitel 6.3 zu verwandten Arbeiten dargestellt. Hierfür werden i.A. Transformationen eingesetzt. Darüber hinaus hat der Entwickler die Möglichkeit, die Modelle der Benutzerschnittstellen von Hand nachzubearbeiten. Diese Modelle werden im Folgenden diskutiert.

4.2.1 Aufgabenmodell

Das Konzept einer `Aufgabe` (englisch "Task") stammt aus der kognitiven Modellierung. Dort werden Aufgaben insbesondere im Rahmen von Aufgabenmodellen für die Analyse von Benutzerschnittstellen eingesetzt; z.B. mit Hilfe so genannter GOMS-Techniken (Card, Newell und Moran 2000). Im Gegensatz dazu beinhaltet ein `Aufgabenmodell` (englisch "Task Model"), wie es im Rahmen verwandter Arbeiten genutzt wird, die mit einer Benutzerschnittstelle möglichen bzw. zu erledigenden Aufgaben. Vor allem bei benutzerzentrierten Entwicklungsprozessen, z.B. dem Useware-Engineering-Prozess (Zühlke 2004), können Aufgabenmodelle[2] eingesetzt werden (Meixner und Görlich 2008), um die Tätigkeiten mit der Benutzerschnittstelle vor zu strukturieren. Bei Useware wird für diese Aufgabenmodellierung die Sprache *UseML* eingesetzt (Meixner und Görlich 2008, Reuther 2003). Eine Einführung zu Aufgabenmodellen mit Fokus auf der Erstellung von Benutzerschnittstellen findet sich z.B. in Limbourg und Vanderdonckt (2003), Paternò (2001).

Eines der meist zitiertesten Aufgabenmodelle ist ConcurTaskTrees (CTT) von Paternò, Mancini und Meniconi (1997), welches vielfach benutzt und erweitert wurde, z.B. von Bergh und Coninx (2004), Feuerstack, Blumendorf und Albayrak (2006), Limbourg (2004), Luyten u. a. (2003), Nóbrega, Nunes und Coelho (2005), Paternò, Santoro und Spano (2009). Aufgabenmodelle wie CTT haben einen *offenen Charakter*, d.h. sie formulieren Rahmenbedingungen für einen Ablauf – nicht den Ablauf selbst. Diese Eigenschaft ist für Aufgabenmodelle von zentraler Bedeutung. Daraus leitet sich ab, dass einem Aufgabenmodell in den allermeisten Fällen mehr als ein konkreter Ablauf der Bedienung der Benutzerschnittstelle (das `Dialogmodell`) (nicht eindeutig) zuzuordnen ist.

Diese Abbildung von Aufgaben- auf Dialogmodell ist nicht trivial. Um sie zu bewerkstelligen, nutzt CTT sog. Enabled Task Sets (Paternò und Santoro 2002). Eine Verallgemeinerung dieser Abbildung ist das von Puerta und Eisenstein (1999) eingeführte `Abbildungsproblem` (`Mapping-Problem`) zwischen Modellen. Verschiedene Arbeiten, welche sich diesem Problem widmen stellen Clerckx, Luyten und Coninx (2004) vor. Ihr eigener Ansatz für Dynamo-AID basiert auf ihrer vorigen Arbeit (Luyten u. a. 2003) und nutzt die Verwandtschaft zwischen Aufgaben aus, um State-Transition Networks aus CTT Bäumen zu erstellen.

[2] Meixner und Görlich referenziert im Rahmen von Useware auf das Aufgabenmodell (aus historischen Gründen) als `Benutzungsmodell`.

4.2.2 Abstrakte Benutzerschnittstelle (AUI)

Ein Aufgabenmodell wird in den meisten Ansätzen auf eine `Abstrakte Benutzerschnittstelle` (englisch: "Abstract User Interface", `AUI`) abgebildet. Es ist das AUI Modell, in dessen Definition sich die verwandten Arbeiten am stärksten unterscheiden. Die verschiedenen Definitionen haben dabei auch Bestandteile ohne gegenseitige Überschneidungen oder direkte Vergleichsmöglichkeiten.

Herangehensweise und Abstraktion ist bei verschiedenen Autoren unterschiedlich gelagert. So sehen Calvary, Coutaz und Thevenin (2001), Calvary u. a. (2002) das AUI Modell als isomorph zum Aufgabenmodell an. Diese Isomorphie beinhaltet eins-zu-eins Beziehungen zwischen den AUI Modellelementen und den einzelnen Aufgaben als auch deren Enthalten-Beziehungen. Vanderdonckt und Bodart (1993) dagegen sehen nicht diese enge Bindung zum Aufgabenmodell. Vielmehr beschreiben sie das AUI als Abstraktion der konkreten Benutzerschnittstelle (CUI).

Eine weitere Beschreibung nahe an der konkreten Benutzerschnittstelle liefern Schneider und Cordy (2001). Sie legen das AUI auf grafische Benutzerschnittstellen fest. Dies wird im Ansatz genutzt, um auf AUI Ebene Verhaltensbeschreibungen zu hinterlegen. Ähnlich erkennen auch Kavaldjian u. a. (2007), dass ein AUI abhängig von der genutzten Modalität ist, weil (beispielsweise) der verfügbare Platz auf dem Bildschirm mit einfließt, welcher essentiell die Auswahl der darzustellenden Komponenten und deren Hierarchie beeinflusst. Botterweck (2006) beschreibt daher ein AUI Modell, für welches Ausprägungen für verschiedene Nutzungskontexte erstellt werden können. Im Rahmen von Useware kommt schließlich DISL (Dialog and Interface Specification Language) von Mueller, Schaefer und Bleul (2004) als Sprache für die AUI zum Einsatz (Meixner und Görlich 2009). DISL ist eng mit UIML (Helms u. a. 2008) verwandt (besprochen in Kapitel 6.6).

Des Weiteren werden User Interface Patterns (vgl. auch Kapitel 20 zum Ausblick) betrachtet, um den Entwicklungsprozess zu verbessern, z.B. in (Breiner u. a. 2010, Luyten u. a. 2005, Seissler, Meixner und Breiner 2010).

Struktur und Konstituenten differieren je nach Autor. Einige Autoren (Calvary, Coutaz und Thevenin 2001, Calvary u. a. 2002) beschreiben das AUI Modell abstrakt als eine Struktur von `Workspaces` (Arbeitsflächen). Ein solch abstrakter Workspace ist dabei ein Platzhalter für ein Element der Benutzerschnittstelle (z.B. ein Knopf oder Eingabefeld).

Eine konkreteres Konzept von Vanderdonckt und Bodart (1993) befüllt das AUI Modell mit `abstrakten Interaktionsobjekten` (Abstract Interaction Object, `AIO`). Ein solches AIO wird über seine Eigenschaften definiert. So hat es keine (grafische) Erscheinung, d.h. es kann nicht direkt in einer Benutzerschnittstelle angezeigt werden; dafür ist ein CIO (Concrete Interaction

Object) aus dem konkreten Benutzerschnittstellenmodell nötig. Ein AIO kann 0...N verschiedene CIOs haben, welche es in einer Benutzerschnittstelle repräsentieren. Das AIO ist somit eine Klassifizierung von CIOs. Des Weiteren kann es einfach sein (d.h. von einem bestimmten Typ, z.B. “Eins aus N Auswahl”) oder komplex (zusammengesetzt aus unter-AIOs).

Szekely (1996) beschreibt AIO Elemente etwas anders ausdifferenziert. AIOs repräsentieren feingranulare Interaktionsaufgaben (d.h. interaktive Elemente, z.B. Auswahl 1-aus-N). Daneben sind Information Elements Platzhalter für darzustellende Daten, sowohl konstante, als auch dynamische. Presentation Units schließlich sind eine Abstraktionen von Fenstern, welche darzustellende Elemente gruppieren.

Anders Demeure u. a. (2006) und Limbourg (2004), welche ein AUI Modell, bestehend aus abstrakten Containern, beschreiben. Diese Container wiederum können weitere Container oder abstrakte Komponenten enthalten. Dabei haben abstrakte Komponenten Facetten, wie z.B. Eingabe, Ausgabe oder Kontrolle.

Navigationsbeziehungen zwischen AUI Workspaces werden im am Aufgabenmodell orientierten Ansatz von Calvary, Coutaz und Thevenin (2001), Calvary u. a. (2002) durch die Abbildung aus den Aufgaben in das AUI Modell mit übernommen. Dagegen werden von anderen Autoren (Behring u. a. 2007, Vanderdonckt und Bodart 1993) keine Navigationsbeziehungen eingeführt.

4.2.2.1 Zusammenfassung

Zusammenfassend lässt sich feststellen, dass die untersuchten Arbeiten unterschiedliche Schwerpunkte in ihren Definitionen einer AUI setzen. Diese sind im Folgenden kurz zusammengefasst.

- *Herangehensweise:* es gibt es Ansätze, welche sehr eng am Aufgabenmodell orientiert sind, aber auch Arbeiten, welche ein AUI als Abstraktion einer konkreten Benutzerschnittstelle sehen.
- *Abstraktion:* neben Ansätzen, welche ein AUI betonen, das unabhängig von der genutzten Modalität ist, existieren auch Arbeiten, welche es konkret auf eine (z.B. grafische) Modalität fixieren und eine stärkere Abhängigkeit zwischen Modalität und AUI zu Grunde legen.
- *Struktur:* Alle Ansätze beinhalten eine Form der Charakterisierung (Klassifizierung) der Elemente des AUIs, sowie die Möglichkeit, AUI Elemente hierarchisch anzuordnen.
- *Konstituenten:* Die möglichen Typen von AUI Elementen der verschiedenen Ansätze sind unterschiedlich und stark vom jeweiligen Einsatzgebiet des AUI Modells abhängig. Insbesondere unterscheidet sich auch die Anzahl der beschriebenen Typen und deren Abstraktionsgrad (z.B. “Presentation Unit” gegenüber “Eins aus N Auswahl”).

- *Navigationsbeziehungen:* Es existieren sowohl Ansätze, welche Navigationsbeziehungen im AUI Modell beschreiben, als auch solche, die diese nicht in das AUI Modell einbringen.

Auf Grund der starken Unterschiede in der Nutzung des AUI Begriffes, verzichtet die vorliegende Arbeit darauf, eine eigene Definition zu entwickeln oder eine bestehende zu übernehmen. Im Lichte der untersuchten Arbeiten ist insbesondere die Abstraktionsebene unklar, auf welcher ein solcher, zu definierender Begriff abzielen sollte. Stattdessen wird nach Besprechung der konkreten Benutzerschnittstelle im Folgenden die abstraktionsunabhängige Benutzerschnittstelle (UUI) in Kapitel 4.3 eingeführt.

4.2.3 Konkrete Benutzerschnittstelle (CUI)

Zur `konkreten Benutzerschnittstelle` (Concrete User Interface, `CUI`) findet sich weit weniger Literatur. Zentral ist hier wieder die Arbeit von Vanderdonckt und Bodart (1993). Dort beschreiben die Autoren `konkrete Interaktionsobjekte` (Concrete Interaction Objects, `CIO`), die Elemente eines CUI Modells, als "*grafische Objekte zur Eingabe und Anzeige von Daten mit Bezug zur Benutzerschnittstellenaufgabe*"[3]. CIOs werden auch `Widgets` (vom englischen Window Gadget) oder physikalische `Interaktoren` genannt.

Die zwei Hauptaspekte eines CIOs sind dabei ihre (grafische) Repräsentation und Verhalten (Intention, Ziel). Ersteres ist CIO spezifisch, wohingegen letzteres als AIOs (abstrakte Interaktionsobjekte, vgl. voriger Abschnitt) herausfaktorisiert wird. UIML (Helms u. a. 2008) zum Beispiel geht diesen Weg, indem die Verhaltensbeschreibung auf Basis abstrakter Objekte vorgenommen wird, welche dann auf die CIOs abgebildet werden.

Abstraktion Ähnlich zur AUI ist die Nutzung des CUI-Begriffes in der Literatur nicht konsistent. Bei Paternò, Santoro und Spano (2009) z.B. legen die CUI-Elemente die grafische Repräsentation ("Knopf") fest und sind damit abhängig von der Modalität (grafisch, akustisch, etc.), aber Toolkit unabhängig ("HTML Knopf", "Java Swing Knopf"). Im Gegensatz dazu sind die CUI Elemente von Blumendorf, Feuerstack und Albayrak (2006) sehr wohl Toolkit abhängig. Analog dazu ist aber die Nutzung des AUI-Begriffes von Schneider und Cordy (2001). Dort ist nicht das CUI sondern das AUI Modell abhängig von der Modalität (grafisch, akustisch, etc.) und unabhängig vom verwendeten Toolkit.

4.2.3.1 Zusammenfassung

Ähnlich zur abstrakten Benutzerschnittstelle im vorigen Kapitel, lassen sich Inkonsistenzen bei der Definition des Begriffes der konkreten Benutzerschnitt-

[3] Übersetzung aus dem Englischen

stelle (CUI) in der untersuchten Literatur feststellen. Die wichtigsten Merkmale und Differenzen sind im Folgenden zusammengefasst.

- **Direkte Wahrnehmbarkeit:** Konstituenten des CUI Modells stellen Elemente dar, welche durch den Nutzer direkt wahrgenommen werden können.
- **Abstraktion:** In allen Arbeiten ist das CUI Modell konkreter als das AUI Modell. Jedoch behandeln es einige Arbeiten als abhängig von einem gewählten Toolkit, wohingegen für andere Arbeiten das CUI Modell unabhängig vom Toolkit ist.

Aus ähnlichen Gründen wie beim Begriff der abstrakten Benutzerschnittstelle (inkonsistente Nutzung, unklare Abstraktionsebene), wird auch für die konkrete Benutzerschnittstelle auf die eigene Definition oder Übernahme einer bereits existierenden Definition des Begriffes verzichtet. Stattdessen wird im folgenden Kapitel die Dichotomie zwischen AUI und CUI aufgelöst um eine tragende Definition zu erhalten.

4.3 Abstraktionsunabhängige Benutzerschnittstelle (UUI)

Die Begriffe der abstrakten (AUI) und konkreten (CUI) Benutzerschnittstelle wurden in in den Kapiteln 4.2.2 respektive 4.2.3 diskutiert und Inkonsistenzen in der Literatur identifiziert. Zusammenfassend wurde festgestellt, dass es verschiedene Herangehensweisen an das AUI Modell gibt (Orientierung am Aufgabenmodell gegenüber Abstraktion vom CUI Modell) und unterschiedliche Abstraktionsebenen betont werden (unabhängig gegenüber abhängig von einer Modalität). Ein ähnliches Bild bietet sich beim CUI Modell, bei welchem insbesondere unterschiedliche Abstraktionsebenen betont werden (unabhängig gegenüber abhängig von einem Toolkit).

Aufgrund dieser Inkonsistenzen wird keine Definition für AUI und CUI gegeben bzw. aus existierenden Arbeiten übernommen. Vielmehr lässt die inkonsistente Nutzung der Begriffe in der Literatur den Schluss zu, dass an Stelle von AUI und CUI ein anderer Blickwinkel gewählt werden muss. Insbesondere fällt auf, dass der betonte Abstraktionsgrad von AUI und CUI in der Literatur variiert. Es liegt daher nahe, diesen je nach Fragestellung passend zu wählen.

So erscheint es für den praktischen Gebrauch sinnvoll (vgl. Kapitel 16.1), eine generische („abstrakte") Benutzerschnittstelle unter Nutzung eines konkreten UI-Toolkits (z.B. Swing) erstellen zu können. Diese kann dann für verschiedene konkretere Nutzungskontexte (ebenfalls im Swing-Toolkit) verfeinert werden.

Neben dieser Form der Abstraktion ist es mit der Dichotomie von AUI und CUI schwierig, bestimmte Gestaltungsfreiheiten zu geben. So wäre es wünschenswert, die Strukturierung (z.B. Gruppierung von Elementen) und den

Dialogfluss der Interaktion abhängig von der Modalität oder sogar den konkreten Interaktionselementen zu wählen. In einer grafischen UI hat z.B. die Wahl des konkreten Interaktionselements starken Einfluss auf den Platzbedarf. Man könnte die Gestaltungsentscheidung treffen, in einem Dialog ein großes Listenfeld einzusetzen, weil die zu tätigende Auswahl wesentlich ist. Daneben lässt man aber aufgrund des geringen, verfügbaren Bildschirmplatzes eine weitere Funktion weg, da diese im konkreten Zusammenhang nicht wesentlich (sondern nur wünschenswert) ist und über einen anderen Weg auch erreichbar ist.

Führt man den Gedanken konsequent weiter, so muss es möglich sein, die Struktur, den Dialogfluss und die konkreten Interaktionsobjekte gemeinsam zu wählen und nicht in getrennten Abstraktionsebenen. Zumindest muss der Entwickler entscheiden können, was er wann abstrahiert. Daher gestalten auch einige Ansätze das AUI-Modell zumindest Modalitäts-abhängig.

In Kapitel 5.1 werden aus den hier besprochenen Herausforderungen Anforderungen abgeleitet. Im Vorgriff auf diese werden nun zwei zentrale Begriffe der vorliegenden Arbeit definiert.

Definition 3: Eine **abstraktionsunabhängige Benutzerschnittstelle** (abstraktions**u**nabhängige UI, `UUI`) ist eine Benutzerschnittstelle, welche Elemente beliebiger Abstraktion beinhalten kann. Sie charakterisiert sich durch folgende Eigenschaften:

- Eine UUI enthält Interaktionsobjekte.
- Interaktionsobjekte innerhalb einer UUI können von unterschiedlicher Abstraktion sein. Besteht das UUI aus Elementen mehrerer Toolkits, so darf aber jeder Teilbaum der UUI als Ganzes wiederum nur Elemente aus einem Toolkit enthalten.
- Die Abstraktionsebene(n) einer UUI werden passend zum vorliegenden Modellierungsproblem gewählt.

Definition Abstraktionsunabhängige Benutzerschnittstelle

Die gegebene Definition einer UUI löst somit die Dichotomie zwischen AUI und CUI auf und erlaubt, beliebige Abstraktionsebenen zu nutzen. Insbesondere ist auch die Mischung von Toolkits möglich, was für die Modellierung föderierter Benutzerschnittstellen förderlich ist.

Als Konstituenten einer UUI werden im Folgenden Interaktionsobjekte definiert. Die Definition übernimmt die Möglichkeit der Schachtelung und Klassifizierung von Elementen und verknüpft diese miteinander.

Definition 4: Ein **Interaktionsobjekt** (Interaction Object, `Interaktor`) definiert sich (analog zu AIO und CIO) über folgende zu erfüllende Eigenschaften:

- Ein Interaktionsobjekt ist ein Element aus einer UUI.
- Spezielle Charakteristika können einem Interaktionsobjekt mit Hilfe eines Klassifizierungsmechanismus zugeordnet werden.
- Interaktionsobjekte können Container-Charakter haben und somit geschachtelt werden, soweit ihre Charakteristika dies zulassen.
- Ein Interaktionsobjekt kann Nutzdaten (zur Darstellung, Eingabe, etc.) enthalten.
- Zur Interaktion muss ein Interaktionsobjekt interpretiert werden (im Rahmen einer Transformation zu Code oder direkt zur Interaktion).

Definition
Interaktionsobjekt

4.4 Zusammenfassung

Dieses Kapitel führte Begriffe der Problem- (Kapitel 4.1) und der Lösungsdomäne (Kapitel 4.2) ein. Die von dieser Arbeit beigetragene, umfassende Betrachtung stellt dabei verschiedene Positionen von verwandten Arbeiten zu den Begriffen abstrakte und konkrete Benutzerschnittstelle (AUI und CUI) dar. Dabei konnte eine inkonsistente Verwendung der Begriffe festgestellt werden, insbesondere im Bezug auf die Abstraktionsebenen, welche vom AUI bzw. CUI Modell betont werden. Daher und aufgrund des wünschenswerten, größeren Gestaltungsspielraumes wurde in Kapitel 4.3 die Dichotomie zwischen AUI und CUI durch die Definition der abstraktionsunabhängigen Benutzerschnittstelle (UUI) aufgelöst, welche im Rahmen dieser Arbeit zur Anwendung kommt.

Im folgenden Kapitel werden die identifizierten Herausforderungen aufgegriffen und daraus Anforderungen abgeleitet.

Kapitel 5

Anforderungen

Um für die Problemstellung, welche in Kapitel 1 beschrieben wurde, ein Lösungskonzept zu erarbeiten werden in diesem Kapitel Anforderungen an Ansätze zur Entwicklung von Multi-Benutzerschnittstellen diskutiert und erhoben. Die Anforderungen setzten auf dem historischen Kontext der Arbeit (Kapitel 2) sowie der Diskussion der Begriffe und Definitionen (Kapitel 4) auf. Dort erarbeitete Grundlagen werden im Rahmen der Anforderungsdiskussion vertieft und ergänzt.

Im Rahmen der Entwicklung der Anforderungen wird zur Argumentation auch auf verwandte Arbeiten eingegangen. Dies ist ein Vorgriff auf das folgende Kapitel 6, in welchem die entwickelten Anforderungen zur Bewertung verwandter Arbeiten genutzt werden. Anschließend wird das Lösungskonzept dieser Arbeit in Teil II auf ihrer Grundlage entwickelt. In Teil III wird schließlich die Erfüllung der Anforderungen durch das Lösungskonzept überprüft. Zudem wird der Anspruch erhoben, dass die vorgestellten Anforderungen auch über diese Arbeit hinaus Gültigkeit haben und sie werden tiefergehend als bei anderen Arbeiten diskutiert.

Der Aufbau des Kapitels folgt den sechs erhobenen Anforderungen an ein Lösungskonzept, welche jeweils in weiteren Unteranforderungen detailliert werden. Die sechs Anforderungen beziehen sich auf Abstraktionen (Anforderung 1 in Kapitel 5.1), Erweiterbarkeit (Anforderung 2 in Kapitel 5.2), Modellierungsdetailgrad (Anforderung 3 in Kapitel 5.3), Einfache Nutzbarkeit (Anforderung 4 in Kapitel 5.4), Integration Struktur und Verhalten (Anforderung 5 in Kapitel 5.5) und Modifikation (Anforderung 6 in Kapitel 5.6). Mit der Zusammenfassung in Kapitel 5.7 werden alle erhobenen Anforderungen und Unteranforderungen nochmals gelistet.

5.1 Anforderung 1: Abstraktionen

In Kapitel 4.2 wurde deutlich, dass die Begriffe der abstrakten und konkreten Benutzerschnittstelle schwierig zu definieren sind und in der Literatur nicht

konsistent verwendet werden. Hierbei differieren die verschiedenen Ansätze stark hinsichtlich des Abstraktionsniveaus von AUI und CUI. Der Grad der Abstraktion der Benutzerschnittstellenmodelle wird dabei in den unterschiedlichen Ansätzen *je nach Fragestellung* passend gewählt:

> **Anforderung 1**: Information (zu Struktur und Verhalten) kann auf einem für die Aufgabe passenden Abstraktionsniveau gegeben werden.

Anzahl der Abstraktionsebenen Sieht man den Sinn der verschiedenen Modelle darin, die Aspekte Struktur, Verhalten, Wahl der Interaktoren und Wahl der Implementierungssprache zu trennen, stellt sich die Frage nach dem Nutzen. Die verschiedenen Teilaspekte beeinflussen sich stark gegenseitig, ein Wasserfall-Prozess, welcher die Aspekte derart trennt, dass sie der Reihe nach betrachtet werden (z.B. Dialogmodellierung im AUI vor dem Design der grafischen Benutzerschnittstelle im CUI) wäre also nicht per se angebracht. Auch ist die Möglichkeit zur Wiederverwendung von Informationen für andere Nutzungskontexte bei diesem Aufbau in Frage zu stellen. Die Modelle müssen für jeden neuen Nutzungskontext in einen vollständigen Satz neuer Modelle überführt werden. Wünschenswert wäre es aber stattdessen, je *nach Bedarf, beliebige, passende Zwischenebenen* einziehen zu können.

Diese verschiedenen Ebenen lassen sich auch in der Literatur finden. So wählen Lin und Landay (2008) mit Ihrem Ansatz "Damask" zwei Abstraktionsebenen: gerätespezifische und geräteunabhängige Ebene. Elemente der unabhängigen Ebene werden in allen gerätespezifischen Varianten der Benutzerschnittstelle gezeigt.

Eine solche *Herauslösung von gemeinsamer "Information"* für verschiedene Nutzungskontexte wird in verschiedenen Ansätzen durch den Einsatz der Entwurfsmuster Faktorisierung bzw. Dekoration (Gamma u. a. 1995) verfolgt. In den Arbeiten von Bergh und Coninx (2004) sowie Souchon, Limbourg und Vanderdonckt (2002) werden mehrere Aufgabenmodelle für unterschiedliche Nutzungskontexte bereitgestellt: ein "Super"-Aufgabenmodell faktorisiert gemeinsame Aspekte heraus. Mori, Paternò und Santoro (2004) dagegen nutzen einen Filterungsansatz, um ein generisches Aufgabenmodell in Nutzungskontext spezifische Aufgabenmodelle zu überführen. Souchon, Limbourg und Vanderdonckt (2002) diskutieren die verschiedenen Möglichkeiten, mehrere Nutzungskontext spezifische Aufgabenmodelle zu verschmelzen. Botterweck (2006) schließlich beschreibt die Nutzung eines allgemeinen in Verbindung mit mehreren Nutzungskontext spezifischen AUI Modellen, welche den Restriktionen der jeweiligen Zielplattformen Rechnung tragen.

Es lässt sich somit feststellen, dass verschiedene Ansätze verschiedene Zwischenstufen nutzen, um die jeweils relevanten Abstraktionsebenen abzudecken. Auch die `Model Driven Architecture` (`MDA`) verfolgt dieses Ziel (Koch, Uhl und Weise 2002). Hierbei werden Anwendungen über mehrere Abstraktionsebenen für verschiedene Plattformen verfeinert, sodass Information, welche für

mehrere Nutzungskontexte relevant ist in abstrakteren Modellen gehalten wird. Es wird die Unteranforderung formuliert:

> *Unteranforderung 1.1*: Die Anzahl der Abstraktionsebenen kann je nach Anwendungsfall gewählt werden.

Natur der Abstraktion Zum Anderen soll auch die Natur der Abstraktion frei *für den Anwendungsfall wählbar* sein. Mit Natur der Abstraktion ist hierbei gemeint, wovon in einem abstrakteren Modell abstrahiert wird. Abstrahiert werden kann beispielsweise von Toolkits (wie bei vielen AUI Modellen der Fall) oder aber von anderen Eigenschaften des Nutzungskontextes, wie beispielsweise Gerät, Umgebung oder einer Benutzerrolle.

Lin und Landay (2008) zum Beispiel wählen die Art der Abstraktion unterschiedlich zu den in Kapitel 4.2 vorgestellten Ebenen von AUI und CUI. Stattdessen sind ihre geräteunabhängige und geräteabhängige Ebene beide sehr konkret - es lassen sich konkrete Interaktionsobjekte (Eingabefelder, Auswahlboxen, etc.) auf beiden Ebenen platzieren und anpassen. Somit kann auf den konkreten Anwendungsfall eingegangen werden, wenn z.B. nur grafische Benutzerschnittstellen geplant sind. Darüber hinaus kann der Entwickler durch (z.B.) das WYSIWYG-Paradigma gezielt unterstützt werden, im Gegensatz zur Nutzung abstrakter Modelle (vgl. Anforderung 4). Das nutzbare Toolkit ist für beide Ebenen das gleiche. Ähnlich in SoKNOS (vgl. Kapitel 15.1), wo verschiedenen Swing-basierte Benutzerschnittstellen einer Anwendung erstellt wurden. Diese variieren je nach Bildschirmgröße, verfügbarer Hardware und Benutzerrolle. Dagegen könnte eine Kinoticket Anwendung, oder ein Museumsführer, wie z.B. der von Mori, Paternò und Santoro (2004), viele verschiedene Geräte und Toolkits unterstützen. Folglich wird formuliert:

> *Unteranforderung 1.2*: Die Natur der Abstraktionen ist nicht fixiert, sondern kann je nach Anwendungsfall gewählt werden.

Dies Unteranforderung korrespondiert mit dem spezifikationsbasierten Ansatz von Szekely (1996), hinter welchem die Philosophie steckt, dem Entwickler eine möglichst passende Sprache (Werkzeuge) zur Formulierung von Benutzerschnittstellen an die Hand zu geben – i.G. zum primären Ziel der Automatisierung. Dies ist ebenfalls der Ansatz von domänenspezifischen Sprachen (vgl. Stahl u. a. (2007) und Kapitel 3.4) und wird von Dan R. Olsen (2007) in seinem wichtigen Papier zu "UI Systems Research" hauptsächlich unter `Expressive Match` diskutiert. Mit dem Begriff wird die passende Ausdrucksstärke der Sprache im Bezug auf das Problemfeld beschrieben, d.h. die Nähe der Ausdrücke zu Konzepten aus der Problem- und Lösungsdomäne.

Ähnliches fordern Myers, Pane und Ko (2004) in ihrem Papier über `Natural Programming` (natürliche Programmierung). Sie zielen darauf ab, dass Entwickler sich möglichst natürlich ausdrücken können: Ein subtiler Unterschied zu Olsen, denn Myers et al. fordern nicht die Nähe der Ausdrücke zur Problemdomäne, sondern zum mentalen Modell den Entwicklers.

5.2 Anforderung 2: Erweiterbarkeit

Ein Ansatz muss für die Umsetzung von Anforderung 1 auch die passenden Möglichkeiten bereithalten, neue Sprachen (z.B. zur Aufgabenmodellierung, Abdeckung einer komplett anderen Abstraktionsebene oder eines neuen Toolkits[1]) einzubinden und technischen Weiterentwicklungen gegenüber offen zu sein. Auch in kommerziellen Werkzeugen ist die Erweiterbarkeit wichtig, so können in Visual Studio und Netbeans neue (eigene) Elemente zur Konstruktion der Benutzeroberfläche mit aufgenommen werden und XAML[2]

Dies führt zu Anforderung 2:

> **Anforderung 2**: Die Erweiterung des Ansatzes auf neue Sprachen ist leicht möglich.

Die Innovationsgeschwindigkeit der Domäne macht eine leichte Erweiterbarkeit notwendig, um mit der Entwicklung Schritt halten zu können. Wäre diese nicht gegeben, fiele der Ansatz wahrscheinlich dem "Moving Target Problem" (Myers, Hudson und Pausch 2000) zum Opfer (bevor er greift, ist er schon nicht mehr aktuell). Die XHTML-Initiative z.B. hat diese Notwendigkeit für das Web erkannt und hat deshalb XHTML initiiert[3], welches sich leicht mit weiteren XML-Sprachen erweitern lässt.

Im gewählten Ansatz muss damit insbesondere gefordert werden:

> *Unteranforderung 2.1*: Keine Sprachen sind im Ansatz fest kodiert.

5.3 Anforderung 3: Modellierungsdetailgrad

Wichtig bei der Modellierung ist auch die Frage nach dem Detailgrad der Modellierung. Das heißt, wie stark soll von der zu erstellenden Benutzerschnittstelle abstrahiert werden, oder wie viel Einfluss soll der Entwickler auf die Feinheiten der zu erstellenden Benutzerschnittstelle haben? Grundsätzlich haben Luyten u. a. (2005) festgestellt, dass Designer sehr wohl die Präferenz haben, Benutzerschnittstellen selbst zu gestalten, um "ansehnlichere" Ergebnisse zu erhalten. Es wird also folgende Anforderung formuliert.

> **Anforderung 3**: Das Look and Feel[4] jeder MBS-Variante kann manuell im Detail angepasst werden.

[1] Zur Beziehung zwischen einer Sprache und einem Toolkit sei auf Kapitel 4.1.3 verwiesen.
[2] Teil der Windows Presentation Foundation, vgl. http://msdn.microsoft.com/de-de/library/ms747122.aspx
[3] http://www.w3.org/TR/xhtml1
[4] Der gängige Term Look and Feel wird genutzt, wobei auch "exotischere" Modalitäten wie Sprache gemeint sind.

Manuelle Anpassung Automatische Ansätze bergen großes Potenzial um die Effizienz in der Entwicklung zu verbessern, aber die von ihnen generierten Benutzerschnittstellen sind anfällig für Nutzbarkeitsprobleme (Usability-Probleme) und lassen ästhetische Qualitäten vermissen[5] (Demeure u. a. 2008, Meskens u. a. 2008, Myers, Hudson und Pausch 2000).

Generell schlechte Ästhetik bis zu schlechter Nutzbarkeit bei vollautomatischen Transformationen identifizieren z.B. auch Demeure und Calvary (2008), Demeure u. a. (2008), Meskens u. a. (2008), Myers, Hudson und Pausch (2000). Calvary u. a. (2008) formulieren hierbei, dass die automatische Generierung nur für simple Varianten – das "Fast Food" unter den Benutzerschnittstellen – Sinn macht: "*Pure automatic UI generation is appropriate for simple (not to say simplistic, "fast-food") UI's.*"

Auf der anderen Seite stellen Meskens u. a. (2008) fest, dass es von Vorteil ist, neue MBS-Varianten zu generieren und im Nachhinein von einem Entwickler modifizieren zu lassen. Dieser Trend wurde auch durch Calvary u. a. erkannt, und sie erweiterten das Cameleon Referenz-Framework (vgl. Kapitel 4.2), um manuelle Anpassung von Benutzerschnittstellen zu unterstützen (2004). Dies formuliert auch die folgende Anforderung:

> *Unteranforderung 3.1*: Jede MBS-Variante kann manuell angepasst werden (auch Generierte).

Modifikation von Details der Benutzerschnittstelle Die Konsequenz aus Unteranforderung 3.1 ist es, dem Entwickler auch die Möglichkeit zu verleihen, alle Aspekte (auch plattformspezifische) einer Benutzerschnittstelle modifizieren können. Dies ist z.B. bei mobilen Endgeräten sehr wichtig – sie können sich in der Bedienung stark unterscheiden, und die Plattformspezifika haben somit einen starken Einfluss auf die Nutzbarkeit der Benutzerschnittstelle.

> *Unteranforderung 3.2*: Die Details der verschiedenen Benutzerschnittstellen sind modifizierbar und nicht auf einen gemeinsamen Nenner von verschiedenen Toolkits beschränkt.

Ein Fokus rein auf den Gemeinsamkeiten verschiedener Toolkits ermöglicht die einfache Portierbarkeit der Benutzerschnittstelle, weil alle zu portierenden Eigenschaften und Elemente auf allen Plattformen unterstützt werden. Auch ermöglicht er eine einfachere Generierung von Benutzerschnittstellen, birgt aber Gefahren im Hinblick auf Aussehen und Usability. Darüber hinaus beschränkt er die Ausdrucksmacht und macht damit den Fehler, den *Entwickler von der konkreten Benutzerschnittstelle zu isolieren.* Doch genau diese Isolation ist laut Myers, Hudson und Pausch (2000) einer der Gründe, warum UIMS sich nicht gegen Interface Builder (vgl. Kapitel 2) durchgesetzt haben. Den Entwicklern ist es wichtig, die Kontrolle über die "*low-level pragmatics of the interactions*

[5] Die Schwere wiegt natürlich je nach Definition von "ästhetisch", sowie Grad und Werkzeuge der Automatisierung.

look and feel" zu haben (Myers, Hudson und Pausch 2000). Auch für Szekely (1996) ist eine Konsequenz aus den Arbeiten der UIMS bis 1996, dass mehr Wert auf die Möglichkeit gelegt werden sollte, alle Features der UIs modifizieren zu können – egal wie feingranular bzw. niedrig in der Abstraktion diese sind.

5.4 Anforderung 4: Einfache Nutzbarkeit

Zentral für die Akzeptanz eines Ansatzes ist, dass die anvisierte Nutzergruppe diesen auch einsetzen kann. Die passende Anforderung wird im Folgenden in mehrere Unteranforderungen detailliert.

> **Anforderung 4**: Der Ansatz ist für einfache Nutzbarkeit durch den Entwickler konzipiert.

Transparenz des Systemzustands Verschiedene Autoren, so z.B. Luyten u. a. (2008), Myers, Hudson und Pausch (2000), identifizieren *Unvorhersagbarkeit* bei der Nutzung von einigen Ansätzen als ein großes Problem. Laut Myers et al. war Unvorhersagbarkeit auch ein großes Problem von UIMS. Dabei musste der Entwickler abstrakte Modelle modifizieren, wobei die Auswirkungen der Modifikationen auf die zu entwickelnde Benutzerschnittstelle für den Entwickler nicht klar waren[6].

Grundlegend für bessere Vorhersagbarkeit ist, dass ein System seinen Zustand (bzw. den der entwickelten Benutzerschnittstelle) dem Entwickler transparent macht.

> *Unteranforderung 4.1*: Der Zustand der Multi-Benutzerschnittstelle sowie mögliche Auswirkungen von Aktionen des Entwicklers werden dem Entwickler transparent gemacht.

Übereinstimmung editiertes Artefakt und Endergebnis Ein verwandtes Problem beim Editieren von Benutzerschnittstellen wird von verschiedenen Autoren diskutiert. Das Ziel von "Natural Programming" (natürliches Programmieren) (Myers, Pane und Ko 2004) ist es, dass die Nutzer einer Programmiersprache (Entwickler, Endnutzer etc.) ihre Ideen so ausdrücken können sollten, wie sie darüber denken. Das heißt, dass die Programmierumgebungen möglichst gut auf das mentale Modell ihrer Nutzer abgestimmt sind. Eng hiermit verwandt ist das Prinzip der *Direktheit*, welches in "direct manipulation UIs" eine zentrale Bedeutung hat (Myers, Pane und Ko 2004).

Bei der direkten Manipulation werden Dinge immer direkt, d.h. nicht via andere Artefakte, modifiziert. Interface Builder, welche die Gestaltung von Benutzerschnittstellen durch direkte Manipulation ermöglichen, setzten sich gegen UIMS durch (vgl. Kapitel 2 zum historischen Kontext). Letztere zwangen den Entwickler zur Modifikation abstrakter Modelle. Jedoch auch Luyten u. a.

[6] was jedoch nicht heißt, dass sie indeterministisch wären

(2008) haben die Erfahrung gemacht, dass viele MBS-Werkzeuge schwer zu nutzen sind, weil eben diese Lücke zwischen mentalem Modell des Entwicklers und der Präsentation des Werkzeugs zu groß war. Abstrakte Beschreibungen isolieren den Designer; und deren Beziehungen zur konkreten Benutzerschnittstelle sind oft unklar (Meskens u. a. 2008, Myers, Hudson und Pausch 2000).

Das `WYSIWYG`-Konzept – What You See Is What You Get – ist eine Ausprägung dieser Direktheit. Es ist ein Paradigma für Editoren, bei denen das Produkt genau so aussieht, wie das im Editor bearbeitete Artefakt: die grafische Benutzerschnittstelle wird direkt manipuliert. Insbesondere dies ermöglicht ein experimentelles Lernen, sowie direkteres Feedback an die Entwickler (Selic 2003).

Auch viele der heutigen Benutzerschnittstellen bedienen sich des WYSIWYG Konzeptes – im einfachen Fall z.B. Schieberegler, welche sich durch Ziehen mit der Maus modifizieren lassen, anstelle Eingabe eines textuellen Wertes. Angewandt auf die Modellierung von Benutzerschnittstellen garantiert das Direktheitsprinzip, dass Entwickler nicht auf abstrakten Artefakten arbeiten und somit von den konkreten Benutzerschnittstellen isoliert sind. Es wirkt den von Myers und anderen Autoren identifizierten Problemen entgegen und mindert die Barriere zu Nutzung, sowie dass Probleme durch Unvorhersagbarkeit entstehen.

> *Unteranforderung 4.2*: Die Übereinstimmung zwischen dem editierten Artefakt und der resultierenden Benutzerschnittstelle ist so groß wie möglich.

Integrierte Werkzeugkette Im Rahmen dieser Arbeit gewonnene Erfahrungen, insbesondere aus dem EMODE-Projekt (Domene u. a. 2007), haben gezeigt, dass die Nutzung einer zersplitterten Werkzeugkette mit Problemen behaftet ist. Diese Anforderung geht somit über die Anforderung eines zentralen Repositories für Interaktionsdaten (Puerta und Eisenstein 2002) hinaus, als dass die Daten nicht (nur) zusammen abgelegt, sondern auch integriert nutzbar sind. Industrielle Werkzeuge verfolgen ebenfalls diesen Ansatz, so bieten z.B. Microsoft Visual Studio und Eclipse eng integrierte Entwicklungsumgebungen an. Eine solche Integration ist auch im Rahmen von Forschungsarbeiten interessant, weil Entwickler das Zusammenwirken interessanter Unterstützungstechniken erfahren können, wie z.B. der Verhaltensansicht und dem Editor (vgl. Teil II zu den Konzepten). Dies formuliert die folgende Anforderung.

> *Unteranforderung 4.3*: Die Werkzeugkette zur Entwicklung ist gut integriert, und nicht aus einzelnen, disjunkten Programmen zusammengesetzt.

Kurze Iterationszyklen (Flexibilität) Schnellere Exploration von Designalternativen ist durch kurze Iterationszyklen in der Entwicklung möglich

(Newman u. a. 2003, Szekely, Luo und Neches 1992). Ein solcher Rapid Prototyping Ansatz reduziert die Entwicklungsviskosität (Dan R. Olsen 2007), da Entwickler Änderungen an der Benutzerschnittstelle schneller bewerten können, und ermöglicht daher bessere Benutzerschnittstellen (Myers, Hudson und Pausch 2000). Dies ist insbesondere wichtig, da das Testen der Benutzerschnittstelle ein wichtiger Teil des des UI-Design-Prozesses ist (Meskens, Luyten und Coninx 2010).

Insbesondere agile Methoden werden durch solch kurze Iterationszyklen besser unterstützt, da Änderungen z.B. nicht in einem wasserfallartigen Prozess durch verschiedene Prozess-Stufen laufen müssen.

Die Möglichkeit zur schnellen Iteration ist somit ein wichtiges Merkmal und wird in der folgenden Anforderung festgehalten.

> *Unteranforderung 4.4*: Es werden kurze Iterationszyklen zur Entwicklung unterstützt (Flexibilität).

Prüfung statischer Semantik Diese schnelle Iteration muss auch für die Entwicklung von Verhalten (vgl. Anforderung 5) möglich sein. Dies ist bisher jedoch nicht der Fall, wie von Myers u. a. (2008) in einer Studie eruiert.

Eine Möglichkeit, Fehler zu reduzieren, schnelle Rückmeldung an Entwickler zu geben und die Iterationszyklen zu verkürzen ist die Prüfung der Artefakte zur Entwicklungszeit. Würde dies erst zur Kompilier- oder Laufzeit geschehen, können Fehler zu langwierigen Debug Zyklen führen (Domene u. a. 2007, Stahl u. a. 2007). Daher muss die Modellierungsumgebung die statische Semantik der Artefakte prüfen.

> *Unteranforderung 4.5*: Die statische Semantik der Artefakte wird während der Modellierung geprüft.

5.5 Anforderung 5: Integration Struktur und Verhalten

`Struktur` und `Verhalten` einer Benutzerschnittstelle sind "die zwei Seiten des Look and Feel". Feel (Verhalten) impliziert auch einen temporalen Aspekt "wie sich die Benutzerschnittstelle anfühlt", welcher über deren "Look" (Struktur) hinaus geht. Einige Ansätze betrachten das Verhalten der Benutzerschnittstelle nicht explizit, z.B. UsiXML (Limbourg u. a. 2004) oder Teresa (Mori, Paternò und Santoro 2004). Andere Ansätze tun dies und abstrahieren es in ihrem AUI Modell, wie bei Schneider und Cordy (2001), oder beschreiben es in einer Art Aufgaben- bzw. Dialogmodell wie in SerCHo (Blumendorf u. a. 2008), was jedoch das Anpassen des Verhaltens später im Entwicklungsprozess erschwert. Dies trifft auch auf UIML (Helms u. a. 2008) zu, welches wiederum eine Beschreibung von Struktur und Verhalten auf einer gemeinsamen, abstrakten Ebene erlaubt.

> **Anforderung 5**: Struktur und Verhalten der Multi-Benutzerschnittstellen werden gemeinsam (und nicht sequentiell getrennt) formuliert.

Gemeinsame Anpassung Struktur und Verhalten Dabei ist die Gestaltung des Verhaltensaspekts wichtig, wenn nicht nur statische Benutzerschnittstellen erstellt werden sollen, sondern der Benutzer interaktiv mit der Benutzerschnittstelle arbeiten kann (Myers u. a. 2008). Park, Myers und Ko (2008) resümieren weiter, dass die Arbeit der Designer sich immer mehr in Richtung Gestaltung der Interaktivität der Benutzerschnittstelle verlagert. Dies wird auch von Myers u. a. (2008) in einer Studie untermauert. Sie zeigen, dass der zeitliche Anteil, welcher von Designern für die Gestaltung des Verhaltens gebraucht wird, größer ist als der für das Aussehen.

Analog zu Anforderung 1 muss es auch möglich sein, Verhalten auf beliebigen Ebenen zu spezifizieren. So erkennen Schneider und Cordy (2001), dass Verhalten allein auf AUI Niveau zu spezifizieren keine Lösung ist, bieten aber keinen Ausweg an. Ein rein wasserfallartiges Vorgehen bietet sich ohnehin nicht an, denn Myers u. a. (2008) erkennen in einer Studie, dass Verhalten in einem explorativen Prozess entsteht. Dabei gibt es gibt kein fixiertes Verhaltenskonzept, wenn mit der Entwicklung einer Benutzerschnittstelle begonnen wird.

Eine Schnittstelle wird durch beides, Verhalten und Struktur, bestimmt. In der Verfeinerung über mehrere Abstraktionsebenen muss also auch beides betrachtet werden (Behring u. a. 2008):

> *Unteranforderung 5.1*: In jeder Verfeinerung der Multi-Benutzerschnittstelle ist eine gemeinsame Anpassung der Struktur und des Verhaltens möglich.

Synchronisation Struktur und Verhalten Daneben muss Unteranforderung 4.4 – die Forderung nach kurzen Iterationszyklen – auch Beachtung finden.

> *Unteranforderung 5.2*: Das einfache und schnelle Wechseln sowie das Synchronisieren zwischen der Bearbeitung von Struktur und Verhalten ist möglich.

Programmatische Spezifikation von Verhalten Ein weiteres Ergebnis der Studie von Myers u. a. (2008) mit 203 Teilnehmern ist, dass das von Designern gewünschte Verhalten i.A. sehr komplex und divers ist. Es wurde lt. Studie bei 78% der Befragten von Programmierern erstellt. Myers et al. schließen, dass vollständige Programmiermöglichkeiten hierfür zur Verfügung stehen müssen und einfache Werkzeuge, welche “Standard-Verhalten” zur Auswahl stellen, nicht ausreichend sind.

> *Unteranforderung 5.3*: Das Verhalten ist komplett programmatisch spezifizierbar.

Klares Architekturmuster Dabei erfordert die Koordination zwischen dem Verhalten und der Struktur der Benutzerschnittstelle besondere Sorgfalt. Es muss für den Entwickler klar sein, wie die beiden Teile ineinander greifen. Ebenso kann ein Ansatz nur umgesetzt und umfassend getestet werden, wenn klar ist, wie die Benutzerschnittstelle mit der Anwendungslogik kommuniziert – auch in verschiedenen Nutzungskontexten.

Die Konsequenz ist, ein klares Architektur- oder Anbindungsmodell zu formulieren, welches die Schnittstelle zwischen Benutzerschnittstelle und der sonstigen Anwendung beschreibt. Pinheiro da Silva (2000) identifiziert die fehlende Integration der Benutzerschnittstelle mit der darunter liegenden Anwendung bei vielen Ansätzen als Problem. Aktuelle Arbeiten auf dem Gebiet, wie z.B. UsiXML (Limbourg und Vanderdonckt 2009, Limbourg u. a. 2004), bieten hierzu keine expliziten Lösungen an oder wählen eine einfache Web-zentrierte Umsetzung. Hierbei erstellt der Ansatz die über das Web ausgelieferten Dateien, wie z.B. bei SerCho (Blumendorf, Feuerstack und Albayrak 2006) oder CTT/Teresa (Mori, Paternò und Santoro 2004).

> *Unteranforderung 5.4*: Dem Ansatz liegt ein klares Architekturmuster zu Grunde.

5.6 Anforderung 6: Modifikation

Viele Ansätze, insbesondere die mit Fokus auf der Abdeckung möglichst vieler Nutzungskontexte, zielen mit ihrer Unterstützung auf die Erstellung von mehreren UIs ab. Aber die systematische Unterstützung der Modifikation bestehender Benutzerschnittstellen ist, insbesondere vor dem Hintergrund agiler Prozesse, auch sehr wichtig. Nur wenige Ausnahmen bedenken dies, z.B. die Arbeiten zu Damask von Lin und Landay (2008) durch ein einfaches 2-Stufen-Modell mit Vererbung, oder die von Sukaviriya u. a. (2007) mit Hilfe von Hinweisen zu modifizierten Elementen für den Entwickler. Grundlegend hierfür ist, dass die verschiedenen Elemente auch miteinander in Relation gesetzt werden können, wie unter anderem von Puerta und Eisenstein (2002) explizit als Anforderung formuliert.

> **Anforderung 6**: Die Modifikation existierender MBS-Varianten wird explizit unterstützt.

Unterstützung zur Modifikation Die Herausforderungen und Motivationen zwischen Erstellung und Modifikation sind anders gelagert, wie in Kapitel 4.1 beschrieben. Stehen dabei mehrere Artefakte miteinander in einer Verfeinerungsbeziehung, so läuft man immer in die Gefahr von Round-Trip-Problemen[7] und Redundanzen.

[7] d.h. die Änderung eines Artefakts impliziert Änderung von anderen Artefakten, weil sonst ein inkonsistenter Gesamtzustand entsteht

Im Kontext von Modellierung werden dabei z.B. die Historien und das Zusammenspiel von Transformationen wichtig. Genauer beleuchten Hailpern und Tarr (2006), welche Gefahren hier bestehen. Aber auch bei der klassischen Programmierung können diese Probleme auftreten. Die Anbringung von Modifikationen muss also durch adäquate Unterstützung des Entwicklers adressiert werden.

> *Unteranforderung 6.1*: Das Anbringen von Modifikationen an existierenden MBS-Varianten wird systematisch unterstützt.

Kontrolle der Modifikation Und in Kombination mit Anforderung 3 sowie Unteranforderung 4.1 ergibt sich eine weitere Anforderung:

> *Unteranforderung 6.2*: Die Kontrolle darüber, wie Modifikationen auf die Multi-Benutzerschnittstellen angewendet werden, liegt beim Entwickler.

5.7 Zusammenfassung

In diesem Kapitel wurden verschiedene Anforderungen für einen Lösungsansatz zur Entwicklung von MBS entwickelt. Die erarbeiteten sechs Anforderungen gliedern sich dabei in weitere Unteranforderungen auf. Anforderungen sowie Unteranforderungen, welche in diesem Kapitel entwickelt wurden, werden in der folgenden Tabelle 5.1 zusammengefasst. Auf Basis dieser Anforderungen werden im folgenden Kapitel die verwandten Arbeiten bewertet und schließlich das Konzept zur Unterstützung in Teil II entwickelt.

Anforderung	**Inhalt**
Abstraktion (Kapitel 5.1)	
Anforderung 1	Information (zu Struktur und Verhalten) kann auf einem für die Aufgabe passenden Abstraktionsniveau gegeben werden.
. . . Unteranforderung 1.1	**Fall-spezifische Ebenen:** Die Anzahl der Abstraktionsebenen kann je nach Anwendungsfall gewählt werden.
. . . Unteranforderung 1.2	**Fall-spezifische Abstraktionsnatur:** Die Natur der Abstraktionen ist nicht fixiert, sondern kann je nach Anwendungsfall gewählt werden.
Erweiterbarkeit (Kapitel 5.2)	
Anforderung 2	Die Erweiterung des Ansatzes auf neue Sprachen (Toolkits) ist leicht möglich.

Anforderung	Inhalt
... Unteranforderung 2.1	**Keine Sprachenhardkodierung:** Es sind keine Sprachen (Toolkits) fest im Ansatz einkodiert.
Modellierungsdetailgrad (Kapitel 5.3)	
Anforderung 3	Das Look and Feel jeder MBS-Variante kann manuell im Detail angepasst werden.
... Unteranforderung 3.1	**Varianten Modifizierbar:** Jede MBS-Variante kann manuell angepasst werden (auch Generierte).
... Unteranforderung 3.2	**Keine Detailbeschränkung:** Die Details der verschiedenen Benutzerschnittstellen sind modifizierbar und nicht auf einen gemeinsamen Nenner von verschiedenen Toolkits beschränkt.
Einfache Nutzbarkeit (Kapitel 5.4)	
Anforderung 4	Der Ansatz ist für einfache Nutzbarkeit durch den Entwickler konzipiert.
... Unteranforderung 4.1	**Transparenz MBS Zustand:** Der Zustand der Multi-Benutzerschnittstelle sowie mögliche Auswirkungen von Aktionen des Entwicklers werden dem Entwickler transparent gemacht.
... Unteranforderung 4.2	**Direktes Editieren:** Die Übereinstimmung zwischen dem editierten Artefakt und der resultierenden Benutzerschnittstelle ist so groß wie möglich.
... Unteranforderung 4.3	**Integrierte Werkzeugkette:** Die Werkzeugkette zur Entwicklung ist gut integriert, und nicht aus einzelnen, disjunkten Programmen zusammengesetzt.
... Unteranforderung 4.4	**Flexibilität:** Es werden kurze Iterationszyklen zur Entwicklung unterstützt (Flexibilität).
... Unteranforderung 4.5	**Semantikprüfung:** Die statische Semantik der Artefakte wird während der Modellierung geprüft.

Anforderung	Inhalt
Integration Struktur und Verhalten (Kapitel 5.5)	
Anforderung 5	Struktur und Verhalten der Multi-Benutzerschnittstellen werden gemeinsam (und nicht sequentiell getrennt) formuliert.
... Unteranforderung 5.1	**Gemeinsame Anpassung:** In jeder Verfeinerung der Multi-Benutzerschnittstelle ist eine gemeinsame Anpassung der Struktur und des Verhaltens möglich.
... Unteranforderung 5.2	**Synchronisation:** Das einfache und schnelle Wechseln sowie das Synchronisieren zwischen der Bearbeitung von Struktur und Verhalten ist möglich.
... Unteranforderung 5.3	**Programmierbares Verhalten:** Das Verhalten ist komplett programmatisch spezifizierbar.
... Unteranforderung 5.4	**Klares Architekturmuster:** Dem Ansatz liegt ein klares Architekturmuster zu Grunde.
Modifikation (Kapitel 5.6)	
Anforderung 6	Die Modifikation existierender MBS-Varianten wird explizit unterstützt.
... Unteranforderung 6.1	**Modifikationskonzept:** Das Anbringen von Modifikationen an existierenden MBS-Varianten wird unterstützt.
... Unteranforderung 6.2	**Kontrolle Modifikationsverteilung:** Die Kontrolle darüber, wie Modifikationen auf die Multi-Benutzerschnittstellen angewendet werden, liegt beim Entwickler.

Tabelle 5.1: Im Rahmen der Arbeit erhobenen Anforderungen.

Kapitel 6

Verwandte Arbeiten

Die Analyse des Standes der Wissenschaft begann in Kapitel 2 mit einen groben Überblick über das Themengebiet und damit, die vorliegende Arbeit in einen historischen Kontext zu setzen. In Kapitel 4 wurden Begriffe der Problem- und Lösungsdomäne diskutiert und wie verwandte Arbeiten diese einsetzen. Das vorliegende Kapitel geht auf die verwandten Arbeiten in mehr Detail ein und bewertet diese. Dazu wird als thematische Grundlage die transformationsgetriebene Adaption von Benutzerschnittstellen in Kapitel 6.1 betrachtet. Diese Technik ist zwar orthogonal zum vorliegenden Ansatz, aber, da von vielen verwandten Arbeiten eingesetzt, für das Verständnis der verwandten Arbeiten unabdingbar. Die Bewertung der verwandten Arbeiten findet an Hand der in Kapitel 5 entwickelten Anforderungen statt. Hierfür werden die möglichen Erfüllungsgrade in Kapitel 6.2 beschrieben.

Es folgt das bei der Betrachtung der verwandten Arbeiten zentrale Cameleon Referenz Framework, welche für fast alle verwandten Arbeiten einen Rahmen setzt. Daher werden die Ansätze nicht alle im Detail diskutiert, dafür aber das Cameleon Framework in Abschnitt 6.3 detailliert beschrieben und bewertet. Das dort Diskutierte dient somit als "Obergrenze" für alle Cameleon basierten Ansätze. Daneben wird für den einfacheren Vergleich auch eine Cameleon basierte Notation in Abschnitt 6.3.1 entwickelt.

Viele Ansätze nutzen Transformationen zur Überführung der Modelle ineinander, welche in Kapitel 6.1 diskutiert werden. Die verwandten Arbeiten werden daraufhin in den folgenden drei Kapiteln, gemäß ihrer Einordnung je nach Fokus in eine von drei Klassen, betrachtet. Auf Grund der Komplexität der betrachteten Ansätze ist es nur bedingt möglich, diese in eine Klassifikation zu bringen. Auch erschwert die Tatsache, dass Ansätze oft konzeptionell erweitert werden könn(t)en, um zusätzliche Anforderungen zumindest zu einem rudimentären Grad zu erfüllen, deren Einordnung und Bewertung. Die getroffene Einordnung der Ansätze in eine der drei Klassen ist somit nicht immer eindeutig.

Die drei Klassen zur Einordnung sind im Einzelnen: *i)* Abstrakte Spra-

chen (Kapitel 6.4), welche auf die dynamische Generierung der konkreten Benutzerschnittstelle setzen, dagegen ermöglichen *ii)* Struktur-adaptive Ansätze (Kapitel 6.5) das (nachträgliche) Editieren der konkreten Benutzerschnittstelle. Weitere *iii)* adaptive, rekombinierende und andere Ansätze sind in Kapitel 6.6 beleuchtet. Die Betrachtung der verwandten Arbeiten schließt mit einer zusammenfassenden Bewertung in Kapitel 6.7.

6.1 Transformationsgetriebene Adaption von UIs

Viele der verwandten Arbeiten nutzen die Technik der Transformationsgetriebenen Adaption von Benutzerschnittstellen. Dies trifft insbesondere für Arbeiten aus dem Bereich abstrakte Sprache in Kapitel 6.4 zu. Aber auch Struktur-adaptive Arbeiten aus Kapitel 6.5 nutzen Transformationen. Ein Grundverständnis der Technik ist für die Betrachtung der verwandten Arbeiten somit notwendig und wird in diesem Kapitel vermittelt. Zum vorliegenden Ansatz ist die beschriebe Technik orthogonal – sie kann mit dem Lösungskonzept aus Teil II kombiniert werden.

Bei der Nutzung von Transformationen steht die automatische Erstellung von MBS-Varianten im Vordergrund. Szekely benutzt hierfür den Begriff "automatisches UI Design" (Szekely 1996). Die Ansätze fokussieren oft auf Domänen- und Aufgabenmodelle und Generieren AUI und CUI Modelle. Interessanter Weise haben die betrachteten automatischen Ansätze, wie Cameleon auch, immer fix definierte Abstraktionsebenen (in Konflikt mit Anforderung 1). Dies ist aber stimmig, denn sie müssen die Transformationen zwischen den Ebenen definieren.

Untergruppen der Ansätze lassen sich nach *i) Einbettung der Transformation* (Integriert in einem Werkzeug – ja/nein) und *ii) Offenheit der Einbettung* (wird eine proprietäre Sprache genutzt, sind sie fest einkodiert oder austauschbar) bilden. Als eine Art Referenzimplementierung für TDDUI (Transformation Driven Development of User Interfaces) und das Cameleon Framework (vgl. Abschnitt 6.3) kann UsiXML bezeichnet werden.

Herausforderungen Transformationen sind sehr gut wiederverwendbar und oft sehr generisch. Dies sichert eine breite Anwendbarkeit, zwingt aber meist den Entwickler dazu, abstrakte Modelle zu modifizieren, welche schwierig zu interpretieren sind und an schlechter Vorhersagbarkeit leiden (Luyten u. a. 2008). *Abstrakte Beschreibungen isolieren den Entwickler* von der erzeugten Benutzerschnittstelle und deren Beziehungen zum konkreten UI ist oft unklar (Meskens u. a. 2008, Myers, Hudson und Pausch 2000).

Die Generizität bedeutet auch, dass der Einzelfall nicht detailliert betrachtet wird. Sie leiden also unter einem *Trade Off zwischen Generizität und Benutzbarkeit*. So erzeugen automatische Ansätze meist schlecht nutzbare Benutzerschnittstellen, was umfassender in Szekely's Überblick (Szekely 1996) disku-

tiert ist. Ihnen wird generell eine schlechte Ästhetik zugeschrieben (Demeure und Calvary 2008, Demeure u. a. 2008, Meskens u. a. 2008, Myers, Hudson und Pausch 2000).

Dennoch müssen Transformationen alle möglichen Fälle in einem gegebenen Rahmen abdecken – mindestens um ihre Anwendbarkeit zu gewährleisten. Je generischer (= wiederverwendbarer) eine Transformation sein soll, desto mehr Fälle müssen bedacht werden. Es ergibt sich also ein *Zusammenhang zwischen Wiederverwendbarkeit und Komplexität.*

Ergeben sich Änderungen an Modellen, muss eine Transformation erneut ausgeführt werden. Traces sind eine Technik von Transformationsmaschinen, modifizierte Elemente zu erkennen. So ermöglichen sie es prinzipiell, dass manuelle Änderungen an Modellen nicht überschrieben werden. Dennoch scheint es nicht sinnvoll, generisch zu entscheiden, was in jedem Fall von einer Re-Transformationen überschrieben wird und was nicht. Diese Entscheidung sollte vielmehr je nach Anwendungsfall und Art der Modifikation getroffen werden. Somit besteht ein *Trade Off zwischen Wiederverwendbarkeit und Möglichkeit zur Re-Transformation.*

Begegnung der Herausforderungen Dem Problem der schlecht nutzbaren Benutzerschnittstellen wird oft durch nachträgliches Editieren der erzeugten UIs begegnet. Dies kann seinerseits jedoch auch zu einem Problem führen, welches Szekely als "*Post Editing Problem*" bezeichnet. Manuelle Änderungen an einer Benutzerschnittstelle können dabei durch ein erneutes Generieren überschrieben werden. Die Lösung von z.B. Pederiva u. a. (2007), die manuellen Änderungen an der Generierten UI zu formalisieren und "wieder abzuspielen" (Replay), ist Szekely's Meinung nach nur in einfachen Fällen möglich. Als Ausweg bietet sich hier ggf. die Nutzung semiautomatischer Transformation unter Einbeziehung des Entwicklers an.

Auch dem Problem der Unvorhersagbarkeit kann begegnet werden. Frühzeitige Erstellung von Prototypen und kurze Iterationszyklen, wie z.B. in der agilen Entwicklung üblich, reduzieren die Asynchronität zwischen Modell und erstellten Artefakt. Jedoch darf dies nicht mit "echtem" WYSIWYG Editieren gleichgesetzt werden. Des Weiteren werden User Interface Patterns (vgl. auch Kapitel 20 zum Ausblick) als Möglichkeit gehandelt, die Nutzbarkeit der automatisch erstellten Benutzerschnittstellen zu verbessern (Breiner u. a. 2010, Seissler, Meixner und Breiner 2010).

Es lässt sich resümieren, dass Transformationen gut für die automatische Erstellung von Benutzerschnittstellen geeignet sind, aber komplex in ihrer Handhabung sind. Sie einzusetzen bringt Vorteile, muss aber bedacht geschehen und eine gute Entwicklerunterstützung scheint hierbei essentiell. Sie können z.B. mit dem Ansatz der vorliegenden Arbeit kombiniert werden (wie in Abschnitt 12.2.1) beschrieben.

6.2 Erfüllungsgrade der Anforderungen

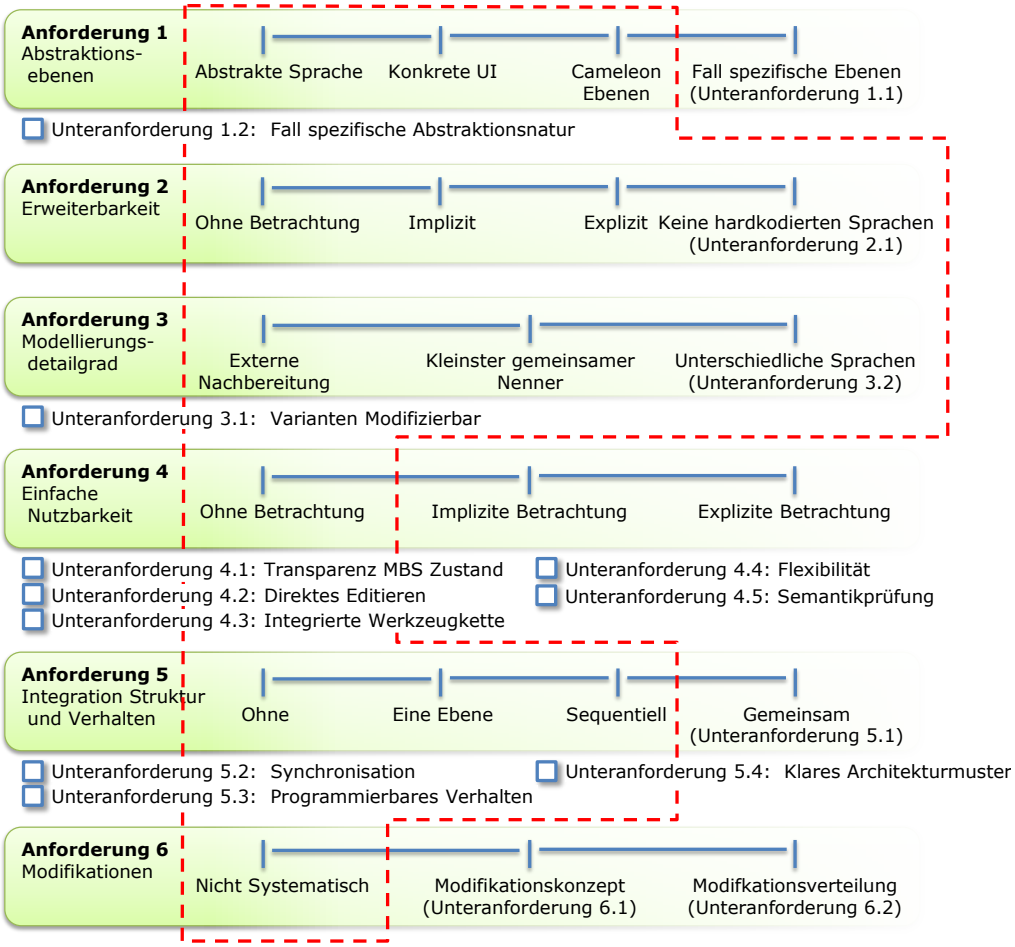


Abbildung 6.1: Erfüllungsgrade der erhobenen Anforderungen – Unteranforderungen stehen teilweise orthogonal zur übergreifenden Bewertung. Die Reichweite des Cameleon Frameworks, als der zentralen verwandten Arbeit, ist rot gestrichelt eingezeichnet.

Abbildung 6.1 illustriert verschiedene Erfüllungsgrade der in Kapitel 5 erarbeiteten Anforderungen. Diese werden in den nächsten Kapiteln (6.3 ff.) genutzt, um die besprochenen verwandten Arbeiten zu bewerten. Hierbei wurde zur einfacheren Vergleichbarkeit eine eindimensionale Skala gewählt. Mehrere Unteranforderungen stehen daher orthogonal zu den Erfüllungsgraden der Anforderungen.

Die Erfüllungsgrade orientieren sich sehr eng an den in Kapitel 5 besprochenen Anforderungen. Zusätzliche Bemerkungen bzgl. der Abbildung auf die Erfüllungsgrade werden bei Bedarf im Folgenden noch gegeben. Sie sind auf

eine alphabetische **Skala** von G bis A abgebildet worden, wobei G den niedrigsten Erfüllungsgrad und A den höchsten Erfüllungsgrad bedeutet. Die Wahl der Skala geschah derart, dass sowohl drei- als auch vierstellige Abstufungen erfasst werden können. Dementsprechend können Anforderungen mit drei möglichen Bewertungen abgebildet werden auf G, D, A, wohingegen Anforderungen mit vier möglichen Bewertungen abgebildet werden auf G, E, C, A.

Einige Unteranforderungen (bspws. 1.1) sind durch bestimmte Erfüllungsgrade der darüber liegenden Anforderung mit erfüllt. Diese sind in der Übersicht in Abbildung 6.1 und in den Zusammenfassungen nicht separat aufgeführt. Die weiteren, in der Abbildung mit einer Checkbox gezeigten Unteranforderungen (wie z.B. Unteranforderung 1.2), werden bei Erfüllung in der zusammenfassenden Tabelle bei der bewerteten Arbeit explizit gelistet.

Anforderung 1 (Abstraktionsebenen): Ansätze können zur Abdeckung multipler Nutzungskontexte *nur eine abstrakte Sprache* (nur AUI) nutzen (Abschnitt 6.4) oder (zusätzlich) das Editieren der *konkreten UI* erlauben (CUI + ggf. AUI). Beziehen sie des Weiteren die in Kapitel 4.2 vorgestellten (festen) Abstraktionsebenen Aufgabenmodell, AUI und CUI ein, so unterstützen sie die *Cameleon Framework Ebenen*. Als höchste Ausbaustufe können *beliebige, Anwendungsfall spezifische Ebenen* unterstützt werden.

Anforderung 2 (Erweiterbarkeit): Neben Ansätzen mit *einkodierten* Sprachen, welche sich nicht erweitern lassen, ist eine *implizite Erweiterung* über das Erstellen neuer Abbildungsregeln (z.B. bei transformationsbasierten Vorgehen) möglich. *Explizite Erweiterung* schließt ein, dass zum Einen die Modellierungssprache selbst Mechanismen vorsieht, aber auch Editoren zur Verfügung stehen, welche mit den Erweiterungen umgehen können (wie z.B. bei kommerziellen Werkzeugen oft der Fall). In der höchsten Ausbaustufe sind *keine Sprachen im Ansatz hardkodiert*, sodass komplett neue Sprachen eingebunden werden können (wie z.B. Jetbrains Meta Programming System, vgl. Kapitel 6.6).

Anforderung 3 (Modellierungsdetailgrad): Ansätze mit generischen (*kleinster gemeinsamer Nenner*) Sprachen bieten eine bessere Unterstützung als Ansätze mit rein *externer Nachbereitung*. Ist Anforderung 1 zum Grad E (Konkrete UI) erfüllt, wird i.A. auch das Editieren der konkreten UI mit spezifischen, *unterschiedlichen Sprachen* unterstützt, sodass der Entwickler direkt die zu präsentierenden Benutzerschnittstellen im Detail modifizieren kann.

Anforderung 4 (Einfache Nutzbarkeit): Diese Anforderung ist stark nutzungs- sowie umsetzungsspezifisch. Es kann daher nicht die Nutzbarkeit per se untersucht werden, vielmehr wird betrachtet, ob ein Ansatz die Nutzbarkeit in Betracht zieht bzw. ob er das explizit tut.

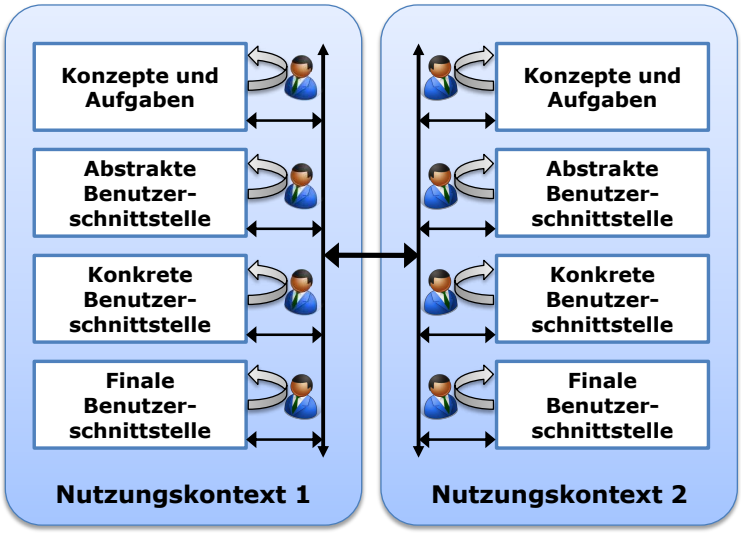


Abbildung 6.2: Skizze der in Cameleon genutzten Modelle nach (Calvary u. a. 2004). Abstraktere Modelle sind weiter oben angeordnet, die einzelnen Modelle können ineinander überführt werden (symbolisiert durch die Pfeile).

Anforderung 5 (Integration Struktur und Verhalten): Einige Ansätze betrachten den Aspekt gar nicht, bieten also *keine Integration* an. Ein erster Schritt darüber hinaus ist, neben der Möglichkeit die UI-Struktur zu beschreiben, *eine Ebene* zur Beschreibung des Verhaltens anzubieten (z.B. ein allgemeines Dialogmodell). Im Rahmen einer *sequentiellen Bearbeitung* von Struktur und Verhalten können darüber hinaus auch mehrere Ebenen von Verhaltensspezifikationen genutzt werden, wobei die beiden Aspekte sequentiell bearbeitet werden (beim Forward--Engineering i.A. zuerst das Verhalten und dann die Struktur). Schließlich kann ein Ansatz auch die *gemeinsame Beschreibung* ermöglichen, wobei auf jeder der mehreren Verfeinerungsebenen Struktur und Verhalten gemeinsam angepasst werden können.

Anforderung 6 (Modifikationen): Viele Ansätze betrachten den Aspekt der Anbringung von Modifikationen *nicht systematisch*, wie es im Rahmen eines *Modifikationskonzeptes* geschieht. Darüber hinaus kann der Entwickler auch noch aktiv Kontrolle über die *Verteilung einer Modifikation* auf die verschiedenen MBS-Varianten übernehmen.

6.3 Cameleon Reference Framework

Das Cameleon Referenz Framework ist als de-facto Standard zur Einordnung von bestehenden Arbeiten zu sehen. Hervorgegangen ist es aus dem Camaleon

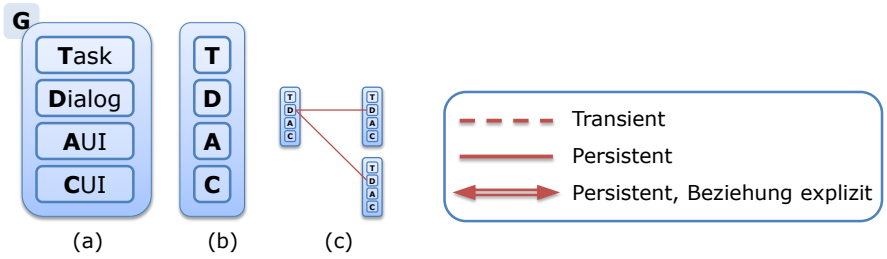


Abbildung 6.3: Notation zum Vergleich der verwandten Arbeiten. Globale Modelle (Teilbild *a*) werden wie in Teilbild *c* zu sehen zu konkreteren Modellen (Teilbild *b*) verfeinert. Die dabei erstellen Beziehungen zwischen den Verfeinerungen sind wie rechts zu sehen klassifiziert.

Projekt[1] und es wurde in (Calvary, Coutaz und Thevenin 2001, Calvary u. a. 2003) veröffentlicht. Erweiterungen wurden in (Calvary u. a. 2002) und später in (Calvary u. a. 2004) publiziert. Es basiert auf den in Kapitel 4.2 eingeführten Begriffen Aufgabenmodell, AUI und CUI, wie in Abbildung 6.2 illustriert. Darüber hinaus wird noch eine `finale Benutzerschnittstelle` (englisch: "Final User Interface", `FUI`) beschrieben. Diese stellt das "ausführbare" UI in der endgültigen Laufzeitumgebung dar, im Gegensatz zum CUI, welches noch in der Entwicklungsumgebung dargestellt wird (Calvary u. a. 2003).

In vielen Arbeiten wird es als Grundlage genutzt bzw. zur Einordnung verwandter Arbeiten eingesetzt, so z.B. von Florins (2006), Limbourg (2004). Darüber hinaus (vgl. Kapitel 2 zur Geschichte) ist es eng verwandt mit den älteren Frameworks von Szekely (1996) und Pinheiro da Silva (2000). Diese zentrale Arbeit konsolidiert also einen großen Teil der verwandten Arbeiten auf ein gemeinsames Framework, mit Hilfe dessen sie untersucht und verglichen werden können.

6.3.1 Cameleon-basierte Notation zum Vergleich

Da das Cameleon Framework eine sehr zentrale Position unter den verwandten Arbeiten einnimmt und sehr umfassend ist, wird es als Basis für eine Notation zum Vergleich der Arbeiten herangezogen. Die Notation zeigt die verfügbaren Verfeinerungen und deren Beziehungen untereinander auf. Die *Legende zur Notation* ist in Abbildung 6.3 zu sehen: *(a)* zeigt global fixierte Aspekte, welche einmal per Anwendung beschrieben werden. *(b)* zeigt eine Verfeinerung für einen Nutzungskontext mit Beschreibungen, welche speziell für diesen erstellt wurden. Schließlich illustriert *(c)* die Verfeinerung. Werden dabei zwei Verfeinerungen gezeigt (wie im Bild), so können an Stelle der zwei auch beliebig viele erstellt werden.

[1] http://giove.isti.cnr.it/projects/cameleon.html

Die **Pfeilarten** beschreiben, in welcher Form die Verfeinerungen untereinander vernetzt sind. *Transiente Beziehungen* signalisieren, dass eine Beschreibung eines Aspektes "on the fly" (z.B. zur Laufzeit) generiert wird und nicht zur Bearbeitung zur Verfügung steht. *Persistente Beziehungen* werden erstellt, wenn die Beschreibungen (Modelle) an beiden Enden persistent sind (im Gegensatz zu transient) und diese Beschreibungen vom Entwickler bearbeitet werden können. *Persistente, explizite Beziehungen* können darüber hinaus auch selbst vom Entwickler modifiziert werden.

6.3.2 Funktionsweise und Abdeckung der Anforderungen

Abstraktionsebenen Zusammenfassend beschreibt das Cameleon Framework eine MBS mit Hilfe der in Abschnitt 4.2 eingeführten Aufgaben-, AUI und CUI Modelle. Die Art der Abstraktionen sind dabei fixiert (im *Widerspruch zu Unteranforderung 1.2*). Dennoch ist, wie in Kapitel 4.2 diskutiert, die genaue Definition der verschiedenen Modelle nicht klar.

Hierzu können die Modelle mit Hilfe von Transformationen (vgl. Diskussion in Abschnitt 6.1) ineinander überführt werden. Dies kann innerhalb eines Nutzungskontextes (vgl. Abbildung 6.2) geschehen (Abstraktion und Konkretisierung: vertikal in der Abbildung), es kann aber auch ein Modell in einen neuen Nutzungskontext überführt werden (Kontextwechsel: horizontal in der Abbildung). Durch die Verkettung von horizontalen und vertikalen Transformationen kann – zumindest theoretisch – eine beliebig komplexe *Struktur von Modellen für verschiedenen Nutzungskontexte* erstellt werden. Somit ist die Anzahl der Abstraktionsebenen theoretisch nicht fixiert (*Unteranforderung 1.1 teilweise adressiert*). Diese beliebig komplexen Strukturen werden auch in keiner dem Autor bekannten Arbeiten betrachtet. Folglich stellt sich die Frage nach der Machbarkeit, ein solch komplexes Netz beliebiger Ebenen in Cameleon zu verwalten. In Vorgriff auf Kapitel 7 bzw. 9.1.3 sei zum Vergleich erwähnt, dass die vorliegende Arbeit dies explizit mit einer Baumstruktur unterstützt.

Modellierungsdetailgrad, Erweiterbarkeit und einfache Nutzbarkeit Camelon implementierende Arbeiten werden nicht auf bestimmte Toolkits bzw. deren gemeinsamen Nenner beschränkt (*Unteranforderung 2.1 und Unteranforderung 3.2 adressiert*). Die Modelle können in unterschiedlichen Sprachen verfasst sein. Jedes Modell kann im Konzept des Frameworks von Transformationen generiert und von Hand nachbereitet werden (*Unteranforderung 3.1 adressiert).* Jedoch wird die notwendige Unterstützung für Entwickler im Framework nicht diskutiert (*Anforderung 4*). Cameleon-basierte Ansätze, wie z.B. Teresa, UsiXML und SerCHo (welche in den folgenden Kapiteln besprochen werden) adressieren diesen wichtigen Aspekt im Allgemeinen aber.

Integration Struktur und Verhalten Das Verhalten kann auf der Ebene des Aufgabenmodells und des AUI Modells angepasst werden, was jedoch

konzeptionell vor der Strukturspezifikation (AUI und CUI Modell) liegt (*Anforderung 5 nur sequentiell, Unteranforderung 5.1 nicht erfüllt*). Weitere Angaben zur Verhaltensspezifikation werden vom Framework nicht gemacht, auch nicht, ob zusätzliche, programmatische Beschreibung von Verhalten möglich ist (*Unteranforderung 5.3*). Auf die Synchronisation von Struktur und Verhalten wird nicht speziell eingegangen. Allein die vorgeschlagene Nutzung von Modell zu Modell Transformationen, um die Beziehungen zwischen verschiedenen MBS-Varianten zu pflegen ist recht komplex und eher für die Erstellung von Benutzerschnittstellen als für deren Aktualisierungen geeignet (vgl. dazu Diskussion in Abschnitt 6.1). Dies ist jedoch nicht ausreichend und kann zu Synchronisationsproblemen führen (*Unteranforderung 5.2 nicht erfüllt*). Des Weiteren macht das Framework keine Angaben bzgl. einer Umsetzung zur Laufzeit und des dabei zu nutzenden Architekturmusters (*Unteranforderung 5.4* nicht adressiert). Dies ist abhängig vom implementierenden Ansatz.

Modifikation Die Modifikation von abgeleiteten MBS-Varianten über Transformationen wird in den untersuchten Arbeiten nicht systematisch betrachtet. Es wird kein Modifikationskonzept vorgelegt, daher ist *Anforderung 6 nicht erfüllt.*

Anforderung	1	2	3	4	5	6
Cameleon Referenz Framework	*C*	*A*	*A* 3.1	*G*	*C*	*G*

6.4 Abstrakte Sprache

Hiermit werden Ansätze zusammengefasst, bei denen eine abstrakte Sprache genutzt wird, um die MBS-Varianten für alle unterstützten Nutzungskontexte zu beschreiben. Es existiert eine AUI Beschreibung, welche zur Laufzeit für den konkreten Nutzungskontext in eine CUI Beschreibung übersetzt wird. Ziel ist es somit, die Intention ("Semantik") der UI (AUI) für alle Nutzungskontexte festzulegen, die Präsentation ("Syntax") der UI (CUI) aber dynamisch erzeugen zu lassen. Somit lassen sich abstrakte Sprachen in das Cameleon Framework einordnen.

Es wird der größte gemeinsame Teiler aller anvisierten Nutzungskontexte in der abstrakten Sprache beschrieben, aber auf die Spezifika der Nutzungskontexte nicht eingegangen. Dies kann mit Hilfe von Annotationen verbessert werden, wodurch eine leichte Spezialisierung der abstrakten Beschreibung möglich ist, z.B. durch Transcoding Techniken oder CSS. Die Mächtigkeit ist allerdings durch die Indirektion der Annotationen stark beschränkt. Und somit *hat die Klasse große Probleme mit Anpassung der Struktur und des Verhaltens an einen gegebenen Nutzungskontext.*

Konsequenter Weise ist die Benutzerschnittstelle nicht im Detail und nicht manuell modifizierbar (*Anforderung 3 nicht erfüllt*). Auch kann nur auf einer

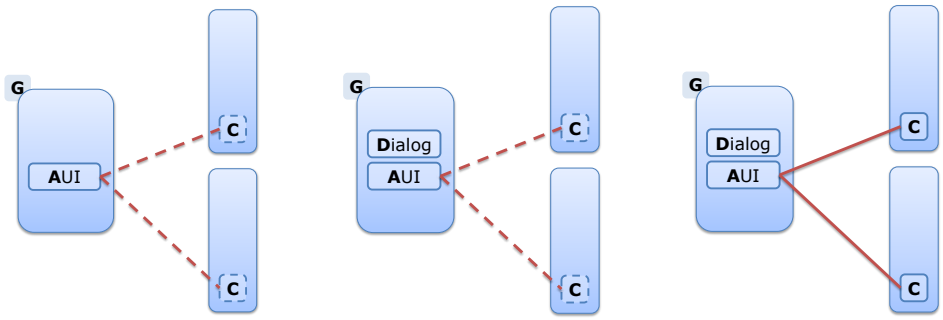


Abbildung 6.4: Schema für PUC und XForms alleine (links), XForms mit Petrinetzen und dem Ansatz von Zaplata (mitte), sowie XForms mit Petrinetzen und CSS (rechts).

Abstraktionsebene gearbeitet werden (*Anforderung 1 nicht erfüllt*).

Auf der anderen Seite bietet der Ansatz *extrem gute Portierbarkeit*, da nur eine AUI formuliert werden muss, welche automatisch in eine passende CUI überführt wird. Auch skaliert er gut mit der Anzahl der Nutzungskontexte, weil nur einmal gegen die abstrakte Sprache programmiert werden muss und keine Anpassung von Verhalten für spezielle Nutzungskontexte nötig bzw. erlaubt ist. Des Weiteren weisen die Benutzerschnittstellen auf Grund der einen zentralen Beschreibung eine hohe Konsistenz auf.

Teilweise existieren abstrakte Sprachen, welche auch Verhaltensbeschreibungen betrachten, so z.B. von Freudenstein u. a. (2008) oder Zaplata u. a. (2009). Typische Vertreter der abstrakten Sprachen sind im Folgenden skizziert.

Personal Universal Controller (PUC) Der Personal Universal Controller (PUC) wurde von Nichols und Myers (2009), Nichols u. a. (2002) entwickelt (vgl. Abbildung 6.4). Der Ansatz erlaubt die abstrakte Spezifikation von Gerätefunktionalität, Zustandsvariablen und Aktivierungsbedingungen. Hieraus generiert er zur Laufzeit Benutzerschnittstellen für unterschiedliche Interaktionsgeräte. Er hat – wie alle abstrakten Sprachen – daher seine Stärke im Bereich der Portabilität. PUC soll hierbei als Beispiel dienen, so gibt es weitere Ansätze, welche sich explizit mit dem Problem von User Interfaces für Fernbedienungen befassen, wie z.B. die auch die ISO-Norm (ISO 24752 2011) zur "Universal Remote Control" (**URC**).

Da nur eine Beschreibungsebene angeboten wird, ist Anforderung 1 somit nicht erfüllt. Die Erweiterbarkeit (Anforderung 2) ist durch Implementierung neuer Transformationen implizit gegeben. Anforderung 3 ist nicht erfüllt, da generierte Benutzerschnittstellen nicht explizit nachbearbeitet werden können.

Anforderung 4 wird im Ansatz nicht explizit betrachtet, lediglich die Umsetzung lässt auf implizite Betrachtung von Unteranforderung 4.4 schließen. Eine

integrierte Beschreibung des Verhaltens ist mit PUC nicht möglich (Anforderung 5). Schließlich sind Modifikationen (Anforderung 6) nicht systematisch adressiert.

Anforderung	1	2	3	4	5	6
PUC	*G*	*E*	*G*	*G* 4.4	*G*	*G*

XForms mit Petrinetzen und CSS XForms[2] (vgl. Abbildung 6.4) selbst erlaubt die Spezifikation von Formularen und Regeln für in ihnen einzugebende Daten auf abstrakter Ebene, ohne Verhaltensbeschreibung und ohne konkrete Strukturinformation (Layout) und fällt somit in die Kategorie "Abstrakte Sprache". In Verbindung mit **Petrinetzen**, wie im Ansatz von Freudenstein u. a. (2008), können darüber hinaus Web basierte Dialoge beschrieben werden. Zusätzlich lässt sich CSS einsetzen, um das konkrete Aussehen der Formulare zu gestalten (vgl. Abbildung 6.4, welche die Unterschiede darstellt).

Diese Dreierkombination erlaubt daher eine abstrakte Beschreibung der Benutzerschnittstelle und (mit passenden Werkzeugen) auch eine Beschreibung der konkreten Benutzerschnittstelle, Anforderung 1 ist somit zum Grad *E* erfüllt. Erweitern (Anforderung 2) lässt es sich implizit durch die Implementierung eines neuen XForms Interpreters. Anforderung 3 ist über die Nutzung von CSS auf einen gemeinsamen Nenner der auf den Zielplattformen unterstützten Toolkits beschränkt.

Die Art der Bearbeitung und damit Anforderung 4 hängt vom gewählten (kommerziellen) Werkzeug ab und ist somit nicht bewertbar. In der Kombination mit Petrinetzen wird eine Ebene der Verhaltensbeschreibung unterstützt (Anforderung 5). Modifikationen (Anforderung 6) sind jedoch nicht systematisch adressiert.

Anforderung	1	2	3	4	5	6
XForms	*E*	*E*	*D*	n.b.	*E* 5.4	*G*

AUI Ansatz von Zaplata Beim AUI Ansatz von Zaplata (Zaplata u. a. 2009) (vgl. Abbildung 6.4) wird die Migration von Prozessen auf verschiedene Plattformen unterstützt. Der Fokus des Papiers liegt auf der Kombination einer Workflow Engine mit automatischer UI Erzeugung. Somit kann der Benutzer an einem Geschäftsprozess weiterarbeiten, unabhängig vom genutzten Interaktionsgerät. Die abstrakte Sprache beschreibt die Plattform unabhängige Benutzerschnittstelle, wofür das Teresa Werkzeug (vgl. Kapitel 6.5) genutzt wird.

Der Ansatz bietet zwei Beschreibungsebenen an: eine für die Abstrakte UI sowie eine für die Prozessbeschreibung. Die konkrete UI ist somit nicht betrach-

[2] http://www.w3.org/MarkUp/Forms

tet Anforderung 1. Wie bei den anderen Ansätzen zu abstrakten Sprachen, lässt sich dieser Ansatz auch über die Implementierung eines Interpreters implizit erweitern (Anforderung 2). Anforderung 3 ist nicht erfüllt, da maximal eine externe Nachbearbeitung der Schnittstelle möglich ist.

Die Nutzbarkeit wird vom Ansatz nicht betrachtet (Anforderung 4), sieht man von der Nutzung vom Teresa Werkzeug ab, welches aber in einem anderen Kapitel bewertet wird. Durch die Prozessbeschreibung wird eine Ebene der Verhaltensbeschreibung unterstützt (Anforderung 5). Zur Ausführung dieser wird eine grobe Architekturbeschreibung gegeben (Unteranforderung 5.4). Eine systematische Adressierung von Modifikationen (Anforderung 6) ist jedoch nicht gegeben.

Anforderung	1	2	3	4	5	6
AUI-Zaplata	*G*	*E*	*G*	*G*	*E*	*G*

6.5 Struktur-adaptive Ansätze

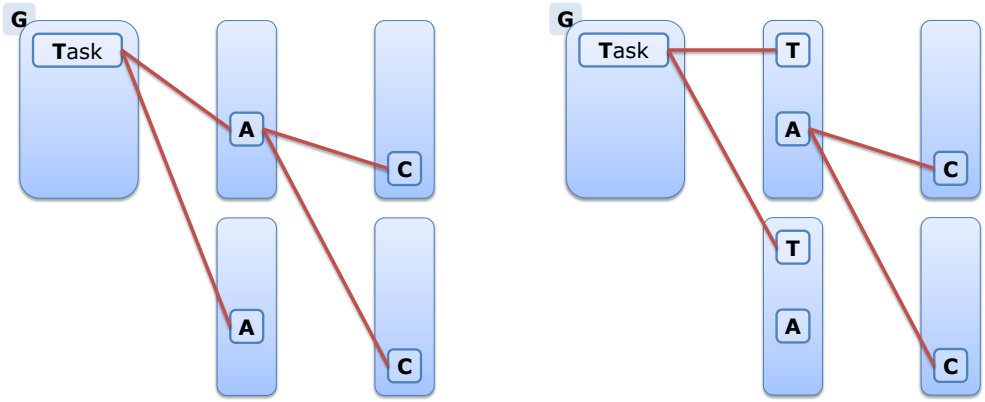


Abbildung 6.5: Schema für UsiXML (links) sowie Teresa (rechts).

Struktur-adaptive Ansätze erlauben es, für Nutzungskontexte essentiell unterschiedliche Strukturen (Layouts) zu produzieren. Im Gegensatz zu abstrakten Sprachen können diese vielfältigere Formen annehmen, da sie auch mehr Anpassung an den jeweiligen Nutzungskontext erlauben. Daher kann jedoch nicht davon ausgegangen werden, dass man gegen eine einzige, abstrakte Schnittstelle programmieren kann. Im Bezug auf die Nutzungskontexte skalieren Struktur-adaptive Ansätze somit weniger gut als abstrakte Sprachen.

Struktur-adaptive Ansätze werden oft in Verbindung mit einem Aufgabenmodell eingesetzt. Hierbei kann man die Ansätze weiter charakterisieren, ob sie das Aufgabenmodell *filtern*, sodass ein abgeleitetes Aufgabenmodell speziell für einen gegebenen Nutzungskontext erzeugt wird. Und ob sie ein *Dialogmodell*

beinhalten, welches die Interaktion beschreibt (zum Unterschied zu Aufgabenmodellen, siehe Abschnitt 4.2.1).

Typische Vertreter der Struktur-adaptiven Ansätze sind:

UsiXML UsiXML[3] (Limbourg 2004, Limbourg u. a. 2004, Limbourg u. a. 2004, Stanciulescu u. a. 2005) kann als *Referenzimplementierung des Cameleon Frameworks* betrachtet werden (vgl. Abbildung 6.5). In den Arbeiten von Limbourg wurden Graph Grammatiken genutzt, um die verschiedenen Modelle durch Transformationen ineinander zu überführen. Essentiell für UsiXML ist, dass die vielen verfügbaren Werkzeuge ein gemeinsames Speicherformat haben, über welches sie verbunden sind. Der darauf aufbauende **Plasticity-domain Editor** (Collignon, Vanderdonckt und Calvary 2008) erweitert die Arbeiten um die Möglichkeit zu beschreiben, in welchen Nutzungskontexten, welche MBS-Varianten zu nutzen sind.

UsiXML unterstützt die Cameleon Ebenen (Anforderung 1), wobei die unterstützte Modellierungssprache fest kodiert ist und Erweiterbarkeit nicht betrachtet wird (Anforderung 2). Dafür stellt UsiXML Modelltyp spezifische Werkzeuge zur Bearbeitung der konkreten UI bereit, welche nicht auf einen gemeinsamen Nenner verschiedener Modelle beschränkt sind (Anforderung 6); auch lassen sich alle MBS-Varianten bearbeiten (Unteranforderung 3.1).

Die Nutzbarkeit für den Entwickler wird nicht explizit betrachtet, wohl aber implizit bei den verschiedenen Werkzeugen, so ist direktes Editieren möglich (Unteranforderung 4.2). Die verschiedenen Werkzeuge sind jedoch nicht integriert (Unteranforderung 4.3). UsiXML bietet (gemäß Cameleon) die sequentielle Bearbeitung von Struktur und Verhalten (Anforderung 5), geht aber auf deren Zusammenspiel und eine Laufzeit nicht ein. Modifikationen werden nicht systematisch betrachtet (Anforderung 6).

Anforderung	1	2	3	4	5	6
UsiXML	*C*	*E*	*A* 3.1	*D* 4.2	*C*	*G*

Teresa Einer der erfolgreichsten Ansätze ist Teresa (vgl. Abbildung 6.5) von Mori, Paternò und Santoro (2004), Paternò und Santoro (2003), welches ebenfalls im Cameleon Projekt (wie das Camelon Framework) entstand. Teresa nutzt zur Aufgabenmodellierung die Sprache *ConcurTaskTrees (CTT)* (vgl. Abschnitt 4.2.1), entwickelt von Paternò, Mancini und Meniconi (1997). Der Ansatz stellt den wohl ausgereiftesten Editor für Aufgabenmodelle bereit (vgl. Kapitel 2).
Unter anderem UsiXML setzt eine erweiterte Version von CTT zur Aufgabenmodellierung ein (Limbourg 2004). Aber auch andere Ansätze, wie z.B. *Contextual CTT* (**CCTT**) von Bergh und Coninx (2004) bauen auf CTT auf.

[3] http://www.usixml.org

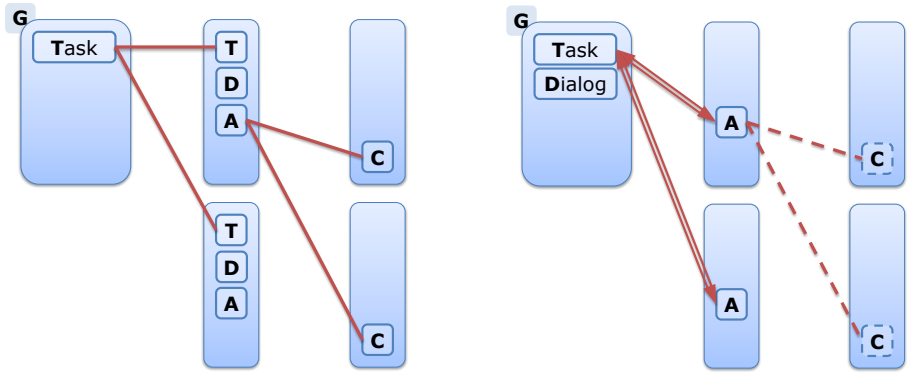


Abbildung 6.6: Schema für SerCHo (links) sowie Maria (rechts).

Letzterer führt sogenannte Context Aufgaben ein, um Untermodelle speziell auf bestimmte Nutzungskontexte zuzuschneiden.

Wie UsiXML unterstützt auch Teresa die Cameleon Ebenen (Anforderung 1). Dabei können neue CUI Modelltypen durch Erstellung einer neuen Transformation implizit integriert werden (Anforderung 2). Die generierten konkreten Benutzerschnittstellen können (losgelöst von Teresa) über einen externen Editor bearbeitet werden (Anforderung 3).

Der Teresa Editor ist im Vergleich zu anderen verwandten Arbeiten recht ausgereift und implizit wird somit auch die Nutzbarkeit betrachtet (Anforderung 4). Er integriert die verschiedenen Editoren für Aufgaben und AUI Modellierung, sowie Transformationen (Unteranforderung 4.3). Wie UsiXML bietet Teresa die sequentielle Bearbeitung von Struktur und Verhalten (Anforderung 5), geht aber auf deren Zusammenspiel und eine Laufzeit nicht ein. Modifikationen werden nicht systematisch betrachtet (Anforderung 6).

Anforderung	1	2	3	4	5	6
Teresa	*C*	*E*	*G*	*D* 4.3	*C* 5.4	*G*

SerCHo Im Rahmen des SerCHo Projektes (Feuerstack, Blumendorf und Albayrak 2006) wurde ein Agenten basierter Ansatz (vgl. Abbildung 6.6) gewählt, bei dem Aufgabenmodelle ausgeführt werden können (Blumendorf u. a. 2008) und zum prototypischen Bau multimodaler Interaktion eingesetzt werden (Feuerstack, Blumendorf und Albayrak 2007).
Ein mehrstufiger Multikanal Ansatz genannt MASP (Blumendorf 2009), welcher neben den Aufgabenmodellen über eine AUI, eine Koordinationsebene und CUI vermittelt (Anforderung 1), wird zur Laufzeit genutzt und ermöglicht auch die Synchronisation multimodaler UIs untereinander (Blumendorf, Feuerstack und Albayrak 2006). Die konkreten Benutzerschnittstellen werden dabei mit Hilfe von Verteilungs-Modellen auf multiple Endgeräte aufgeteilt und

jeweils geformt. Constraint Solver können hierbei kontext- und anwendungsspezifische Layout Regeln zur Laufzeit evaluieren und die UI entsprechend anpassen (Feuerstack u. a. 2008). Der Entwickler kann durch Bereitstellung von XSLT-Transformationen und JET-Templates Einfluss auf diesen Prozess nehmen (Anforderung 3). Eine Erweiterbarkeit des Ansatzes auf weitere Sprachen steht nicht im Fokus, ist aber auf Grund der mehrstufigen Metamodellhierarchie prinzipiell möglich (Anforderung 2).
Das Aufgabenmodell ist die zentrale Ablage für die Verhaltensspezifikation (Anforderung 5) und wird von einer Laufzeitkomponente interpretiert. Blumendorf (2009), Blumendorf, Lehmann und Albayrak (2010) beschreiben das Meta-Metamodell des Ansatzes und Lehmann, Blumendorf und Albayrak (2010) stellen verschiedene integrierte Werkzeuge (Unteranforderung 4.3) zum Modellieren zur Laufzeit (Unteranforderung 4.4) vor. Hiermit ist es unter anderem möglich, den Ortskontext auch WYSIWYG-artig zur Laufzeit zu modifizieren (Unteranforderung 4.2). Dennoch kann der Ansatz auf Grund der Vielfalt von Modellen und ihren mannigfaltigen Querbezügen als der Komplexeste der verwandten Arbeiten bezeichnet werden, was von den Autoren nicht konkret adressiert wird (Anforderung 4).

Anforderung	1	2	3	4	5	6
SerCho	*C*	*E*	*A* 3.1	*G* 4.2 4.3	*E* 5.4	*G*

Maria Maria (vgl. Abbildung 6.6) ist der Nachfolger des bereits vorgestellten Teresa. Grundlegende Konzepte, wie die Orientierung am Aufgabenmodell wurden übernommen und werden im Folgenden nicht erneut bewertet (Anforderung 1, Anforderung 2, Anforderung 6), doch beinhaltet Maria Verbesserungen, auf welche eingegangen wird. Im Gegensatz zu Teresa lassen sich mit Maria (Web) Dienste mit Knoten in einem Aufgabenmodell assoziieren, sodass das Aufgabenmodell mit der Anwendungslogik verbunden ist (Unteranforderung 5.4). Eine zentrale Neuerung ist auch die Laufzeit für Web-basierte MBS, welche deren Ausführung ermöglicht. Die im Rahmen von Maria beschriebene Transformationstechnik (Paterno', Santoro und Spano 2009) ist eine Verbesserung von Teresa und erlaubt generische, aber auch Instanz spezifische Regeln. Hierdurch können, ähnlich wie bei der **formalen Verschönerung** (Pederiva u. a. 2007), MBS-Varianten indirekt (d.h. über Regeln, nicht WYSIWYG) detailliert ausgestaltet werden (Anforderung 3).

Anforderung	1	2	3	4	5	6
Maria	*C*	*E*	*A* 3.1	*D* 4.3	*C* 5.4	*G*

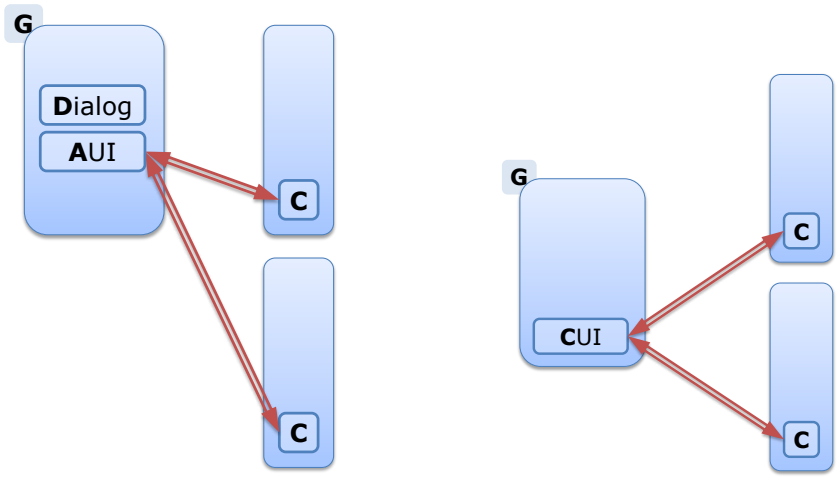


Abbildung 6.7: Schema für UIML (links) sowie Damask (rechts).

Weitere Ansätze Neben den besprochenen Ansätzen gibt es weitere Arbeiten, welche sich mit dem Problem der Entwicklung multipler Schnittstellen befassen. Diese sind hier nicht explizit besprochen, da deren relevante Konzepte von anderen verwandten Arbeiten abgedeckt wurden. **XIML** beispielsweise (Puerta und Eisenstein 2001, 2002) unterstützt im Wesentlichen die Cameleon-Ebenen und setzt einen Fokus auf dem Nutzungskontext (vgl. Definition 2). Puerta, Micheletti und Mak (2005) beschreiben hierfür auch Werkzeug-Unterstützung.

6.6 Adaptive, Rekombinierende und weitere Ansätze

Die Ansätze im letzten Abschnitt fokussierten insbesondere auf die Cameleon Abstraktionsebenen (und erfüllen daher Anforderung 1 nicht). Darüber hinaus sind sie nur begrenzt erweiterbar (Anforderung 2). Die im Folgenden beschriebenen Ansätze dagegen sehen die *Erweiterbarkeit als ein zentrales Element.*

UIML Ähnlich wie einige Ansätze bei abstrakten Sprachen, erlaubt UIML[4] (Helms u. a. 2009) das Formulieren einer abstrakten Benutzerschnittstelle (vgl. Abbildung 6.7). Jedoch wird diese mit ein oder mehreren Abbildungsmodellen versehen, welches jedes abstrakte Element auf ein konkretes Element mit klar beschriebenen Eigenschaften abbildet (Anforderung 1). Somit ist es zum Einen möglich, verschiedene MBS-Varianten für unterschiedliche Nutzungskontexte zu erstellen und sie zum Anderen detailliert auszugestalten (Anforderung 3). UIML kann durch das Erstellen neuer Interpreter mit zugehörigen “Peers” (zur

[4] http://www.uiml.org

Abbildung der UIML Konstrukte) systematisch erweitert werden, wobei keine Sprache hart in den Ansatz einkodiert ist (Anforderung 2). Darüber hinaus ist eine eventbasierte Aktionssprache in UIML integriert, mit welcher einfache Operationen der UI beschrieben werden können (Anforderung 5). Eine systematische Unterstützung zur Modifikation von MBS steht nicht im Fokus von UIML (Anforderung 6).
Verschiedene Arbeiten setzen auf UIML auf und erweitern die zentrale Arbeit, da der Ansatz sehr ausgereift ist: Aktuell wird an UIML Version 4 im Rahmen eines OASIS Committee Drafts[5] gearbeitet (Helms u. a. 2008). Zum Beispiel erlaubt **Gummy** (Meskens u. a. 2008) das Erstellen verschiedener MBS-Varianten mit WYSIWYG Unterstützung, wobei dem Entwickler basierend auf den existierenden MBS-Varianten ein Vorschlag für die neue Variante gemacht wird (Anforderung 4). Gummy lässt dabei die (grafische) Benutzerschnittstelle auf dem Zielgerät ausgeben und stellt dem Entwickler ein Faksimile dieser zum Bearbeiten (WYSIWYG) zur Verfügung (Meskens, Luyten und Coninx 2009) (Unteranforderung 4.2). **Jelly** (Meskens, Luyten und Coninx 2010) ist eine Weiterentwicklung von Gummy und ermöglicht, konkrete Interaktionselemente (z.B. eine Kombobox) direkt von einer Plattform (z.B. Windows Mobile) auf eine andere Plattform (z.B. Adobe Flex) zu kopieren. Die kopierten Elemente sind fortan verknüpft und der Entwickler wird unterstützt, indem ihr Inhalt durch das "linked-editing" Konzept beim Bearbeiten konsistent gehalten wird (Anforderung 6).
DISL (Dialog and Interface Specification Language) dagegen erweitert UIML im Hinblick auf die Dialogbeschreibung (Mueller, Schaefer und Bleul 2004), wobei insbesondere parallele Zustandsübergänge unterstützt werden (Schaefer, Bleul und Mueller 2006). Generell wird bei DISL ein starker Fokus auf die abstrakte Benutzerschnittstelle gelegt, da DISL insbesondere Modalitätsübergreifend konzipiert ist. Im Rahmen von Useware wird es daher auch für die Beschreibung der abstrakten Benutzerschnittstelle eingesetzt (Meixner und Görlich 2009).
LiquidApps[6] schließlich ist eine kommerzielle, integrierte Umgebung für die Gestaltung von Benutzerschnittstellen und basiert auf UIML. Neben der Möglichkeit, wie bei Gummy und Jelly konkrete Benutzerschnittstellen direkt zu editieren, können Anforderungen sowie das Aufgabenmodell im gleichen Werkzeug bearbeitet werden. Das Werkzeug bietet auch Traceability: Verknüpfungen zwischen den verschiedenen Anforderungen, Aufgaben und Interaktionselementen können angelegt und untersucht werden.

Anforderung	1	2	3	4	5	6
UIML	*E*	*A*	*A* 3.1	*D* 4.2	*E* 5.4	*D*

[5] http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=uiml
[6] http://liquidapps.harmonia.com/features

Damask Damask (Lin und Landay 2002, 2008) ist ein Werkzeug zum Skizzieren von konkreten Benutzerschnittstellen (CUIs). Es ermöglicht dem Entwickler, UI Elemente auf einer von zwei Ebenen zu platzieren (vgl. Abbildung 6.7): Entweder auf einer Nutzungskontext spezifischen oder einer für alle Nutzungskontexte gemeinsamen Ebene (Anforderung 1, wobei beide als CUI zu charakterisieren sind. Bei der Modellierung kommt nur das eingebaute Toolkit zum Einsatz, eine Erweiterung wird nicht behandelt (Anforderung 2). Das Toolkit selbst bietet eine generische Sprache für grafische Benutzerschnittstellen (Anforderung 3).
Werden Elemente auf der gemeinsamen Ebene abgelegt, erscheinen sie in allen Nutzungskontexten, auch im WYSIWYG-Editor (Anforderung 4, Unteranforderung 4.2). Änderungen auf der gemeinsamen Ebenen werden dabei auch sofort in allen Konkretisierungen sichtbar (Anforderung 6). Verhalten kann mit dem Ansatz nicht beschrieben werden (Anforderung 5).

Anforderung	1	2	3	4	5	6
Damask	*E*	*G*	*D* 3.1	*A* 4.2	*G*	*D*

UML-basierte Ansätze Neben den besprochenen Arbeiten sollte nicht unerwähnt bleiben, dass auch eine Reihe von Ansätzen auf Basis der UML existieren. Wisdom (Nunes und Cunha 2000) beispielsweise adaptiert die Sprache CTT (vgl. Kapitel 4.2.1) zur Aufgabenmodellierung, wohingegen UMLi (Pinheiro da Silva und Paton 2003) Aktivitätsdiagramme adaptiert, um Abläufe zu beschreiben. UWE (Knapp u. a. 2007) fokussiert auf Web-Anwendungen und erlaubt es, Navigationsstrukturen und darauf ablaufend Prozesse zu beschreiben. Allen drei Ansätzen ist gemein, dass sie insbesondere die Modellierung der abstrakten Benutzerschnittstelle (AUI) betrachten, wobei Wisdom und UMLi dies etwas generischer tun (UWE orientiert sich hier an Web-Anwendungen). Diese Ansätze werden hier jedoch nicht weiter diskutiert, da ihre für den Kontext dieser Untersuchung relevanten Eigenschaften durch andere, bereits betrachtete Ansätze abgedeckt sind.

Spricht man speziell über Erweiterbarkeit und domänenspezifische Sprachen, müssen auch das Language Workbench Konzept und das Meta Programing System erwähnt werden. Beide zielen nicht auf die Modellierung von Benutzerschnittstellen ab, bieten aber wichtige konzeptionelle Beiträge.

Meta Programing System Das Meta Programing System (MPS)[7] ist eine Entwicklungsumgebung, welche sich das Modellierungsparadigma der strikten

[7] http://www.jetbrains.com/mps/index.html

Trennung von abstrakter und konkreter Syntax (vgl. Kapitel 3.2) zu Eigen macht. Der Entwickler modelliert textuell, als würde er programmieren, jedoch schreibt er ein Modell. Das heißt, dass nicht ein Text, sondern die abstrakte Syntax des Programmes abgespeichert wird. Dies nutzt MPS aus, um dynamische Erweiterungen der Syntax (domänenspezifische Sprachen) zuzulassen, welche in sogenannten Sprachbibliotheken abgelegt und sofort verwendet werden können.

Language Workbench Der Begriff Language Workbench wurde maßgeblich von Martin Fowler geprägt. In seinem bekannten Bliki beschreibt er eine Language Workbench[8] als eine Entwicklungsumgebung, in welcher die Entwickler ihre Programmiersprache frei erweitern können. Diese Erweiterungen sollen sich komplett in die bestehenden Sprachen integrieren. Dabei werden erstellte Programme in ihrer abstrakten Syntax abgespeichert. Insofern ist auch das MPS eine Language Workbench[9].
Fowler sieht durch die Nutzung der abstrakten Syntax als zentrales Format (im Gegensatz zum Speichern von Textdateien) vor allem die Möglichkeit, Sprachen und Editoren zu schaffen, welche über die bisherigen Konzepte (textuelle Spezifikation oder grafisches Modellieren mit Kästen und Pfeilen) hinausgehen.

Rekombinationstechniken Einen ganz anderen Weg gehen Rekombinationstechniken. Hierbei werden einzelne Inhalte zu kompletten Benutzerschnittstellen rekombiniert. Die Wahl, welche Inhalte für eine bestimmte MBS-Variante genutzt werden und in welcher Anordnung hängt dabei vom Nutzungskontext ab.

Auf Grund der Herangehensweise, bei der eine MBS-Variante nicht verfeinert, sondern kombiniert wird, können Rekombinationsansätze Anforderung 1 nicht erfüllen. Die Modifikationen im Detail (Anforderung 3) ist prinzipiell möglich, würde aber dem grundlegenden Gedanken, fertige Inhalte zu nutzen, widersprechen. Die Integration von Struktur und Verhalten (Anforderung 5) erschöpft sich bei den zwei vorgestellten Ansätzen darin, dass beide Internetseiten erzeugen. Die Server Seite der Anwendung muss separat erstellt werden. Schließlich unterstützen sie Modifikationen (Anforderung 6) rudimentär, da sich Änderungen an Inhalten in allen Darstellungen, welche das jeweilige Inhaltselement zeigen, widerspiegeln.

RIML (Ziegert, Lauff und Heuser 2004) erlaubt es, wiederverwendbare Inhalte auf einer Seite anzuordnen. Dafür werden horizontale und vertikale Gruppierungsmechanismen zur Verfügung gestellt, sowie Paginierungsmöglichkeiten gegeben. **D3ML** (Göbel u. a. 2006) erweitert dies im Hinblick auf multimodale Benutzerschnittstellen, unter anderem um sequentielle (zeitliche) Ordnung,

[8] http://martinfowler.com/articles/languageWorkbench.html
[9] Was Martin Fowler auch bestätigt: http://martinfowler.com/articles/mpsAgree.html

Selektionsausdrücke zur Wahl der passenden Inhalte und Annotationen zu z.B. Grammatiken.

6.7 Zusammenfassende Bewertung der verwandten Arbeiten

Die diskutierten verwandten Arbeiten sind in Tabelle 6.1 nochmals zusammengefasst. Im oberen Teil der Tabelle sind Ansätze aus dem Bereich der abstrakten Sprachen zu finden, welche einen klaren Fokus auf Automatisierung haben. Es ist zu erkennen, dass diese insgesamt schlechter im Bezug auf die Anforderungen abschneiden. Struktur-adaptive und weitere Arbeiten (die untere Hälfte der Tabelle) sind im Allgemeinen besser im Hinblick auf die Anforderungen zu bewerten. Jedoch bestehen erkennbare Defizite insbesondere im Bereich der Abstraktionsebenen, der Erweiterbarkeit, der Modifikation und oft auch im Bereich der Einfachheit der Nutzung, wie im Folgenden kurz zusammengefasst.

Viele der Ansätze orientieren sich stark am Cameleon Framework. Meist haben sie einen Fokus auf der Abdeckung möglichst vieler Nutzungskontexte und sind daher oft transformationsbasiert (vgl. Abschnitt 6.1). Sie sind daher komplex zu nutzen – der Entwickler steht nicht im Fokus des Konzeptes (Anforderung 4).

Auch werden die Beziehungen zwischen verschiedenen MBS-Varianten für den Entwickler nicht explizit gemacht. Das heißt auch, dass das Modifikationsproblem (Anforderung 6) nicht adressiert wird. Abstraktionsebenen und Art der Abstraktion, welche zur Verfügung stehen sind i.A. vordefiniert und können nicht auf den jeweiligen Anwendungsfall angepasst werden (Anforderung 1).

Eine Formulierung des Verhaltens wird in den meisten verwandten Ansätzen, wenn, nur auf abstrakteren Ebenen unterstützt. Dies geschieht im Allgemeinen nicht zusammen mit der UI Struktur (Anforderung 5). Auch wird die (programmatische) Spezifikation von Verhalten nur selten diskutiert und so gut wie nie ein konkretes, umfassendes Architekturmuster präsentiert.

Viele Ansätze sind nicht leicht erweiterbar, weil Konzepte in Sprachen fest einkodiert sind (Anforderung 2). Dagegen erlauben viele Ansätze (insbesondere Struktur-adaptive, Abschnitt 6.5) das detaillierte Anpassen der Benutzerschnittstelle (Anforderung 3).

Es kann somit resümiert werden, dass bestehende Arbeiten nicht die in Kapitel 5 erhobenen Anforderungen erfüllen, dies trifft insbesondere auf Anforderung 5 zu. Der im Teil II eingeführte Lösungsansatz dieser Arbeit jedoch wurde auf Basis dieser Anforderungen entwickelt und zielt darauf ab, diese zu erfüllen. Dafür wird in Kapitel 17.3 untersucht, in wie weit das im Folgenden vorgestellte Lösungskonzept dies auch bewerkstelligt.

	Arbeit	**Anforderung**					
		1	2	3	4	5	6
	Cameleon Referenz Framework	*C*	*A*	*A* 3.1	*G*	*C*	*G*
Abstrakte Sprache	PUC	*G*	*E*	*G*	*G* 4.4	*G*	*G*
	XForms[10]	*E*	*E*	*D*	n.b.	*E* 5.4	*G*
	AUI-Zaplata	*G*	*E*	*G*	*G*	*E*	*G*
Struktur-adaptive	UsiXML	*C*	*E*	*A* 3.1	*D* 4.2	*C*	*G*
	Teresa	*C*	*E*	*G*	*D* 4.3	*C* 5.4	*G*
	SerCho	*C*	*E*	*A* 3.1	*G* 4.2 4.3	*E* 5.4	*G*
	Maria	*C*	*E*	*A* 3.1	*D* 4.3	*C* 5.4	*G*
Weitere	UIML	*E*	*A*	*A* 3.1	*D* 4.2	*E* 5.4	*D*
	Damask	*E*	*G*	*D* 3.1	*A* 4.2	*G*	*D*

[10] Bewertung in Verbindung mit Petrinetzen und CSS

Tabelle 6.1: Tabellarische Zusammenfassung der Diskussion der verwandten Arbeiten.

II

Konzept

Überblick über Teil II

Teil II stellt das im Rahmen der Arbeit entwickelte Lösungskonzept zur Entwicklung von Multi-Benutzerschnittstellen (MBS) sowie der dazu passenden Entwickerunterstützung vor. Das Konzept basiert auf den Anforderungen, welche in Kapitel 5 detailliert erarbeitet wurden (eine Übersichtstabelle findet sich in Kapitel 5.7 auf Seite 49). Die sechs Anforderungen sind:

Anforderung 1 – **Abstraktionen:** Informationen (zu Struktur und Verhalten) können auf einem für die Aufgabe passenden Abstraktionsniveau gegeben werden.

Anforderung 2 – **Erweiterbarkeit:** Die Erweiterung des Ansatzes auf neue Sprachen ist leicht möglich.

Anforderung 3 – **Modellierungsdetailgrad:** Das Look and Feel jeder MBS-Variante kann manuell bis ins Detail angepasst werden.

Anforderung 4 – **Einfache Nutzbarkeit:** Der Ansatz ist für einfache Nutzbarkeit durch den Entwickler konzipiert.

Anforderung 5 – **Integration Struktur und Verhalten:** Struktur und Verhalten der Benutzerschnittstellen werden gemeinsam (und nicht sequentiell getrennt) formuliert.

Anforderung 6 – **Modifikation:** Die Modifikation existierender Benutzerschnittstellen wird explizit unterstützt

Der Teil II der Arbeit ist wie folgt aufgebaut. Nach dem Überblick über das Lösungskonzept in Kapitel 7 wird in Kapitel 8 das Architekturmuster zur Entwicklung von Multi-Benutzerschnittstellen (MBS) im Detail vorgestellt. Es greift mit der domänenspezifischen Sprache (DSL) zur Modellierung der Struktur von Benutzerschnittstellen (Kapitel 9) ineinander. Das Kapitel zur DSL beschreibt deren Semantik, abstrakte Syntax und Wohlgeformtheitsregeln. Die konkrete Syntax der Sprache wird von den Unterstützungskonzepten bestimmt, mit denen der Entwickler interagiert. Diese werden in Kapitel 10 vorgestellt und bauen auf dem Architekturmuster und der DSL auf. Schließlich wird in Kapitel 11 eine Beschreibung der architektonischen Integration der vorgestellten Konzepte gegeben. Abgeschlossen wird die Vorstellung des Konzeptes in Kapitel 12 mit einer Zusammenfassung und einem konzeptuellen Abgleich mit den Anforderungen.

Zur Lektüre dieses Teils II werden insbesondere die Begriffsdefinitionen aus Kapitel 4 vorausgesetzt. Ebenso wird die Vertrautheit mit dem in Kapitel 3 eingeführten Konzept zur Modellierung mit Abgrenzung zur Programmierung angenommen.

Kapitel 7

Überblick über das Lösungskonzept

Der Teil II befasst sich mit dem im Rahmen dieser Arbeit entwickelten Lösungskonzept. Dabei gibt das vorliegende Kapitel einen Gesamtüberblick darüber und beschreibt es im Groben. Die weiteren Kapitel des Teils II vertiefen dann seine einzelnen Bestandteile.

Die Entwicklung des Lösungskonzepts orientierte sich an den in Kapitel 5 erhobenen Anforderungen. Das Konzept selbst ist in Abbildung 7.1 dargestellt und besteht im Wesentlichen aus zwei Teilen, *i)* dem Architekturmuster mit der Modellierungssprache, sowie *ii)* den Unterstützungskonzepten für Entwickler. Ein Überblick über diese beiden Teile wird in den Kapiteln 7.1 respektive 7.2 gegeben. Im Anschluss wird auf den für den Ansatz wichtigen Aspekt eingegangen, wie ein Modell zur Interaktion gebracht wird (Kapitel 7.3) und schließlich in Kapitel 7.4 das für den weiteren Verlauf genutzte laufende Beispiel eingeführt.

7.1 Architekturmuster und Modellierungssprache

Die dem vorliegenden Ansatz zu Grunde liegende Vorgehensweise ist die **iterative Verfeinerung** von MBS-Varianten für Nutzungskontexte (Unteranforderung 4.4). Der Entwickler kann zur *Verfeinerung*, ausgehend von einer bestehenden Variante, eine neue MBS-Variante für einen neuen Nutzungskontext erstellen (vorgestellt in Kapitel 9.1). *Eine Benutzerschnittstelle kann so je nach Anwendungsfall über mehrere Varianten verfeinert werden*, wie in Anforderung 1 gefordert. Bei einer solchen Verfeinerung kann eine MBS-Variante von genau einer MBS Ausgangsvariante ableiten; d.h. Mehrfachvererbung ist nicht möglich. Somit entsteht durch das iterative Vorgehen eine Baumstruktur, welche *Verfeinerungsbaum* genannt wird, wie in Abbildung 7.1 im rechten (roten) Teil illustriert. Dabei sind Nutzungskontexte als Boxen und die Verfeinerungen als Pfeile zwischen den Boxen dargestellt.

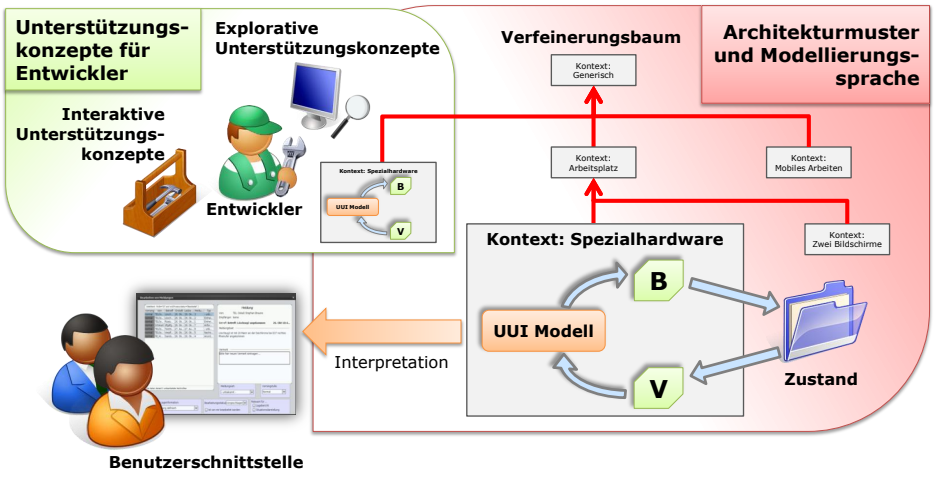


Abbildung 7.1: Überblick über das Lösungskonzept dieser Arbeit. Die Unterstützungskonzepte basieren auf der entwickelten Modellierungssprache und dem für MBS passend entworfenen Architekturmuster.

Die **Benutzerschnittstelle** zu einem Nutzungskontext besteht wiederum **aus mehreren Konstituenten**, wie in Abbildung 7.1 innerhalb einer Nutzungskontextbox dargestellt. Zentral dabei ist das vom Entwickler erstellte Modell der der Benutzerschnittstelle "UUI-Modell" (die orangene, längliche Box innerhalb des Nutzungskontext, vorgestellt in Kapitel 9.1.3), welches die Struktur (das Layout) der UI beschreibt. Durch Verhaltens- und Beobachter*fragmente* (als grüne Box mit Aufschrift "V" respektive "B" im Nutzungskontext abgebildet und vorgestellt in Kapitel 8.2) wird das Verhalten (die Dynamik) der Benutzerschnittstelle beschrieben. Diese Konstituenten können für jede Verfeinerung der Benutzerschnittstelle manuell angepasst werden (Unteranforderung 3.1). Eine solche Anpassung kann gemeinsam für Struktur und Verhalten durchgeführt werden (Unteranforderung 5.1). Dennoch greifen alle Fragmente – unabhängig aus welchem Nutzungskontext – auf den gleichen MBS Zustand (im Bild ganz rechts dargestellt) zu, um den Zustand der MBS zu lesen bzw. zu modifizieren, sowie ihn über Wechsel zwischen MBS-Varianten zu konservieren. Die strikte Trennung in die drei Konstituenten unterstützt das Vererben von Verhalten und Struktur, wie in Kapitel 8 beschrieben, und sorgt für eine klare und strikte Architektur (Unteranforderung 5.4).

Zur Unterstützung von Modifikationen (Anforderung 6) sind die unterschiedlichen UUI-Modelle durch **Verfeinerungsbeziehungen** untereinander verknüpft. Ebenso werden ihre Bestandteile (die Interaktionsobjekte, vorgestellt in Kapitel 9.1.1) miteinander verbunden. Entlang dieser Struktur der Verfeinerungsbeziehungen kann eine Modifikation auf mehrere MBS-Varianten angewendet werden (vorgestellt in Kapitel 9.2.3). Des Weiteren werden *entlang dieser Beziehungen Fragmente geerbt*, welche in Verfeinerungen modifiziert, hin-

zugefügt oder wieder entfernt werden können (beschrieben in Kapitel 8.2).

Soll eine Benutzerschnittstelle für einen Nutzungskontext **zur Interaktion gebracht** werden, so wird das entsprechende UUI-Modell interpretiert (vgl. folgendes Kapitel 7.3). Die passenden Fragmente werden aktiviert und können auf Interaktionsevents reagieren, indem sie den Zustand ändern und zur Ausgabe das UUI-Modell wiederum modifizieren.

7.2 Unterstützungskonzepte für Entwickler

Die Arbeit des Entwicklers bei der Erstellung einer MBS wird durch die im Rahmen dieser Arbeit konzipierten Unterstützungskonzepte (Kapitel 10) erleichtert. Diese machen die MBS in ihrer Struktur und Entwicklungsstand dem Entwickler transparent (Anforderung 4), Zusammenhänge werden visualisiert und Modifikationen an der MBS gezielt unterstützt (Anforderung 6). Die Unterstützungskonzepte können in zwei Kategorien unterteilt werden. Zum Einen die *explorativen Unterstützungskonzepte* (Kapitel 10.2), welche die Untersuchung der MBS in verschiedenen Dimensionen gestatten, mit dem Ziel, deren Entwicklungsstand dem Entwickler transparent zu machen. Zum Anderen die *interaktiven Unterstützungskonzepte* (Kapitel 10.1), welche verschiedenste Modifikationen zulassen und den Entwickler bei diesen gezielt unterstützen.

Durch die Ausnutzung der Eigenschaften des Architekturmusters und der Modellierungssprache können die Unterstützungskonzepte gezielter dem Entwickler helfen, als es ohne ein striktes Architekturmuster der Fall wäre. So wird z.B. die Struktur des Verfeinerungsbaumes ausgenutzt (Kapitel 10.1.4) oder die Schnittstelle zwischen Benutzerinteraktion und Verhaltensfragmenten dargestellt (Kapitel 10.2.2).

7.3 Vom Modell zur Interaktion

Unter anderem auf Grund der Tatsache, dass zu einer abstrakten mehrere konkrete Syntax eingebunden werden können und weil verschiedene Aspekte verknüpft werden können, wurde für diese Arbeit ein modellgetriebener Ansatz gewählt (eine ausführliche Diskussion findet sich in Kapitel 3). Mit Hilfe der domänenspezifischen Modellierungssprache werden Strukturen von Benutzerschnittstellen beschrieben. Modelle der Benutzerschnittstelle können aber im Gegensatz zu Code einer Programmiersprache nicht direkt ausgeführt werden – sie müssen erst interpretiert werden.

Es stehen drei Möglichkeiten zur Verfügung, Modelle von Benutzerschnittstellen zur Interaktion zu bringen (Pinheiro da Silva 2000, Szekely 1996):

1. Erzeugung von Programmcode in einer generischen Programmiersprache, welcher die Benutzerschnittstelle zur Laufzeit erzeugt, ggf. mit Hilfe von Toolkits.

2. Erzeugung von Konfigurationscode, welcher ein UI System zur Laufzeit konfiguriert, oder

3. die Interpretation der UI Modelle zur Laufzeit.

Die Ansätze unterscheiden sich maßgeblich darin, welche (und wie viel) Information zur Laufzeit zur Verfügung steht und wie stark die Artefakte standardisiert sind.

Die ersten beiden Optionen fokussieren stark auf den technischen Aspekt, eine UI möglichst effizient zur Interaktion zu bringen. In beiden Fällen besteht das Ziel nicht darin, **möglichst umfassende Information aus der Entwicklungszeit auch zur Laufzeit zur Verfügung zu stellen**, wie es bei der letzten Option der Fall ist. Daher wird das Konzept der *Modelle zur Laufzeit* in aktuellen Ansätzen auch stärker verfolgt. So sehen Sottet, Calvary und Favre (2006) und Vanderdonckt u. a. (2008) hierin eine Möglichkeit, mehr Information zur Adaption bereit zu halten. Dieses "mehr an Information" wurde bereits durch Szekely (1996) auch als ein Vorteil von Modellen beschrieben, wenn auch mehr mit einem Fokus auf die Entwicklung der UI selbst. Paterno', Santoro und Spano verfolgen ebenfalls diesen Ansatz. Dabei schreiben sie: "*keeping the models alive at runtime to make the design rationale available*" (Paterno', Santoro und Spano 2009).

Die **Integration von Lauf- und Entwicklungszeit** halten Calvary und Pinna (2008) nach dem erneuten Aufleben der UIMS (vgl. Kapitel 2) für die aktuelle Herausforderung. Die Nutzung von Modellen zur Laufzeit ist eine Möglichkeit, dies zu unterstützen. Hierdurch können Änderungen direkt evaluiert werden, denn es müssen nicht erst intermediäre Artefakte erzeugt werden (z.B. Programmcode), die zur Evaluation des Ergebnisses erst ausgeführt werden müssen. Dies wiederum resultiert in kürzeren Iterationszyklen (Unteranforderung 4.4). So wird Modellinterpretation z.B. auch im Mozilla Webbrowser eingesetzt, bei welchem die XUL (XML User Interface) Beschreibungen der Benutzerschnittstelle zur Anzeige interpretiert werden.

Weitere Vorteile für die Ziele dieser Arbeit liegen in der potenziell **besseren Werkzeugunterstützung durch Modelle zur Laufzeit**. Da keine intermediären Artefakte erzeugt werden müssen, können alle Werkzeuge parallel auf das oder die Modelle zugreifen. Somit sind "instantane Aktualisierungen" aller aktiven Werkzeuge möglich, weil Modelländerungen sofort interpretiert werden können. Es entfallen dabei vorgeschaltete Analyseschritte, wie z.B. das Parsen von Programmcode, welcher zum Einen aufwändig ist und zum Anderen in Werkzeug spezifischen intermediären Artefakten resultiert, da jedes Werkzeug diesen Schritt separat für sich durchführt. Daher sind weniger Konsistenzprobleme zu erwarten, weil weniger Artefakte synchronisiert werden müssen.

Ein oft angeführtes Argument gegen diesen Ansatz ist die **Geschwindigkeit bei der Modellinterpretation**. Aber wie später beschrieben (Kapitel

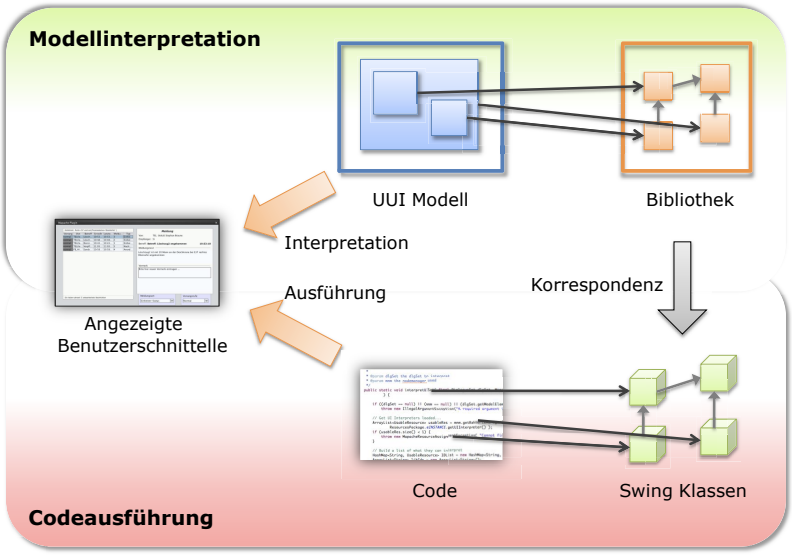


Abbildung 7.2: Gegenüberstellung der Interpretation eines Benutzerschnittstellenmodells (oben) versus der Programmierung und Ausführung des Codes zur Anzeige der Benutzerschnittstelle (unten). Swing wird hier als beispielhaftes Toolkit verwendet.

15.2), ist kein Geschwindigkeitsverlust bei der Interaktion zu bemerken[1]. Auf Grund der skizzierten Vorteile nutzt der vorliegende Ansatz auch das Konzept der Interpretation von Modellen zur Laufzeit.

7.3.1 Abgrenzung zum herkömmlichen Vorgehen

Dieser Abschnitt dient zur Illustration, wie das in dieser Arbeit genutzte Konzept der Interpretation von Modellen der Benutzerschnittstelle im Vergleich zum herkömmlichen Vorgehen der Programmierung von Benutzerschnittstellen arbeitet. Die essentiellen Unterschiede (aber auch Gemeinsamkeiten) illustriert Abbildung 7.2.

Die Grundlage beider Ansätze ist die Nutzung eines Toolkits (in der Abbildung als Beispiel Swing, rechts unten). Beim herkömmlichen Vorgehen werden die Toolkit Klassen aus dem Code referenziert (unterer Teil der Abbildung). Der Code enthält hierbei Befehle, welche die Benutzerschnittstelle bei Ausführung konstruieren. Beim herkömmlichen Vorgehen wird die Benutzerschnittstelle somit *implizit* beschrieben.

Anders beim Vorgehen dieser Arbeit (oberer Teil der Abbildung). Hier enthält eine Modellierungsbibliothek (siehe Abschnitt 11.4) Elemente, welche

[1] Wohlgemerkt wird hier von Modellen der UI gesprochen. Bei Modellen, welche z.B. eingebettete Systeme beschreiben, Echtzeitanforderungen stellen oder wenig Ressourcen zur Verfügung haben kann dies sehr wohl zum Problem werden.

die Klassen des Toolkits modellieren. Ein UUI-Modell (vgl. Abschnitt 9.1.3) beschreibt die Benutzerschnittstelle. Dabei werden Interaktionsobjekte (siehe Abschnitt 9.1.1) eingesetzt, welche durch die Elemente aus der Modellierungsbibliothek klassifiziert werden. Zur Laufzeit interpretiert ein Interpreter (Abschnitt 10.1.1) das Modell der Benutzerschnittstelle und bringt sie somit zur Interaktion.

Unabhängig, ob dabei der Weg über die Programmierung oder den Interpreter gewählt wird, steht am Ende die interagierte Benutzerschnittstelle (links in der Abbildung zu sehen). Diese setzt sich aus `UIKomponenten` zusammen. Eine UIKomponente ist eine Instanz einer Toolkit Klasse, z.B. die JLabel Klasse in Swing. UIKomponenten (als Instanzen von Klassen im Objekt orientierten Sinne) haben Felder (Eigenschaften) und Methoden, über welche ihre Interaktion (z.B. das Aussehen) gesteuert werden kann. Darüber hinaus sind sie eine Quelle für Interaktionsevents, welche vom Programm oder dem Interpreter aufgefangen und verarbeitet bzw. weitergegeben werden. Sie sind also nahe verwandt zu konkreten Interaktionsobjekten (CIOs), sind aber nicht im Modell sondern in der interagierten UI zu finden.

7.4 Laufendes Beispiel

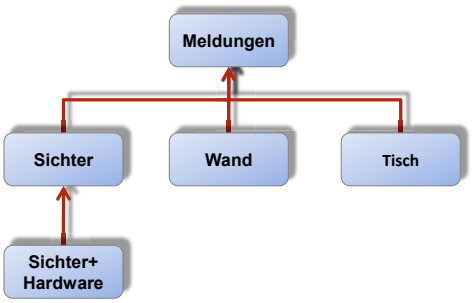


Abbildung 7.3: Die verschiedenen MBS-Varianten des laufenden Beispiels aus dem Anwendungsfall SoKNOS und ihre Beziehungen untereinander.

Der Teil II nutzt das im Folgenden eingeführte Beispiel zur Illustration einer MBS. Es ist aus dem SoKNOS-Projekt entnommen und ist das Fallbeispiel aus der prototypischen Umsetzung in Kapitel 15.1. Eine detailliertere Einführung in SoKNOS und die erstellte Multi-Benutzerschnittstelle ist dort zu finden. Zusammenfassend ist SoKNOS ein IT-System, welches Krisenstäbe unterstützt und mit Hilfe von Plugins um Funktionalität erweitert werden kann.

Das laufende Beispiel ist die Multi-Benutzerschnittstelle für die Meldungsverarbeitung in SoKNOS. Diese wurden im Rahmen der vorliegenden Arbeit prototypisch für Testzwecke und Demonstrationen implementiert. Für die Meldungsverarbeitung stehen, wie in Abbildung 7.4 zu sehen, fünf MBS-Varianten bereit, welche für verschiedene Nutzungskontexte gestaltet wurden.

Meldungen: diese Variante steht allen Stabsmitgliedern zur Verfügung und beinhaltet generische Funktionalität zur Arbeit mit Meldungen.

Sichter: die Sichter Rolle wird mit Hilfe dieser Variante bei der Bearbeitung ihrer primären Aufgabe, dem Weiterleiten von Meldungen, unterstützt.

Die Unterstützung findet durch eine grafische UI statt, welche zusätzliche Funktionalität zur schnellen Adressierung beinhaltet.

Sichter+Hardware: die Verfeinerung der Sichter Variante ist eine föderierte Benutzerschnittstelle mit einem grafischen Anteil und einer Spezialhardware. Sie lagert die zusätzliche Adressierungsfunktionalität der Sichter Variante auf die Spezialhardware aus und passt den grafischen Teil an den erweiterten Bildschirmplatz an.

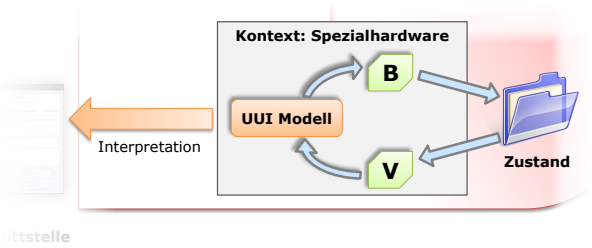
Wand: eine auf große, hochauflösende Bildschirmwände angepasste MBS-Variante wurde hier realisiert.

Tisch: neben einer Anpassung der UI an die größere Auflösung des Tisches, wird zusätzliche Funktionalität auf Grund der Bedienung mit Hilfe eines Stiftes bereitgestellt.

In Kapitel 16.1 werden die einzelnen Varianten detaillierter vorgestellt.

Kapitel 8

Architekturmuster



Ein zentrales Element des Lösungskonzeptes (Übersicht in Kapitel 7) ist das Architekturmuster. Es beschreibt, wie eine Multi-Benutzerschnittstelle (MBS) aufgebaut ist, wo die Schnittstellen zwischen ihren einzelnen Komponenten sind, und wie diese zusammenarbeiten. Es arbeitet mit der im folgenden Kapitel 9 vorgestellten domänenspezifischen Sprache (DSL) zusammen und bildet die Grundlage für die in Kapitel 10 vorgestellten Unterstützungskonzepte; insofern, als dass diese auf dem Architekturmuster basieren.

Das Architekturmuster ist als Zyklus (siehe Abbildung 8.1) angelegt, seine Abschnitte werden in diesem Kapitel beschrieben. Ausgehend vom Benutzer umfasst er die folgenden Abschnitte: Das UUI-Modell wird für die Interaktion mit dem Benutzer interpretiert (Kapitel 8.1). Seine Interaktionen werden mittels Verhaltensfragmenten (Kapitel 8.2) in Änderungen des MBS Zustands (Kapitel 8.3) umgesetzt, welche wiederum mittels Beobachterfragmenten (Kapitel 8.4) in einer Änderung des UUI-Modells resultieren (und damit die Interaktion ändern). Zentrale Design Entscheidungen, die auch für die anderen Teile des Lösungskonzeptes Relevanz haben, werden im darauf folgenden Kapitel 8.5 detailliert. Den Abschluss bildet eine Diskussion des vorgestellten Architekturmusters in Kapitel 8.6, wobei auch der Bezug zu MVC hergestellt wird.

8.1 Benutzer und Benutzerschnittstelle

Aus der Fülle der MBS-Varianten wird eine Variante durch den Interpreter (vgl. Kapitel 10.1.1) dem Benutzer zur Interaktion gebracht. Der Benutzer interagiert mit dieser Benutzerschnittstelle, wie in Abbildung 8.1 im oberen Teil illustriert. Dabei nimmt die Benutzerschnittstelle die Eingaben des Benutzers

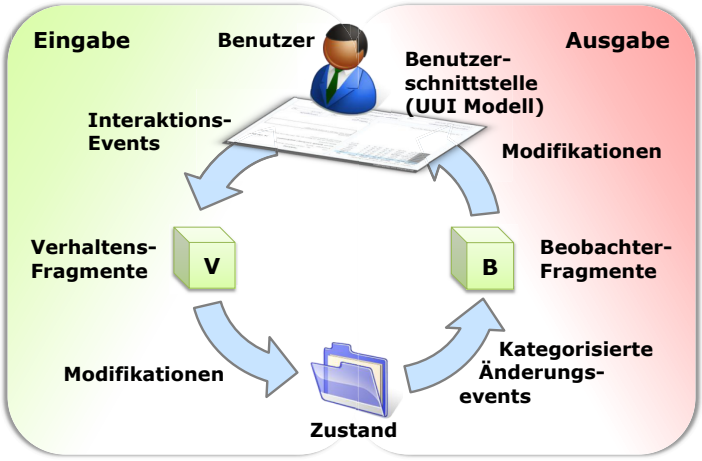


Abbildung 8.1: Architekturmuster für Multi-Benutzerschnittstellen. Ein- und Ausgabe sind scharf in zwei Kanäle (linke, respektive rechte Seite des Kreises) getrennt. Abschnitt 8.5.1 erläutert dazu die wichtigen Designentscheidungen.

entgegen und erzeugt im Verlaufe der Interaktion *Interaktionsevents*. Dieser Event Ansatz ist gängig, und harmoniert gut mit der Abstraktion der Interaktionsobjekte (vgl Kapitel 9.1 zur Semantik).

8.2 Verhaltensfragmente

Die programmatische Spezifikation von Verhalten (Unteranforderung 5.3) findet in *Verhaltensfragmenten* (in Abbildung 8.1 links dargestellt) statt. Sie beschreiben das Verhalten als Reaktion auf Interaktionsevents, welche von Elementen der Benutzerschnittstelle "geworfen" werden.

Die Anbindung von Verhaltensfragmenten muss *berücksichtigen*, dass *i)* die einzelnen MBS-Varianten sich signifikant in ihrer Struktur (Layout) unterscheiden und *ii)* man das Verhalten in jeder Verfeinerung anpassen darf (Unteranforderung 5.1). Die Verarbeitung der Interaktionsevents muss somit in jeder Verfeinerungsstufe modifizierbar sein. Abschnitt 8.5.1 detailliert wie die scharfe Trennung von Ein- und Ausgabe und eine starke Modularisierung der Fragmente dies ermöglichen. Auf Grund dieser scharfen Trennung zwischen Ein- und Ausgabe kann nicht (wie sonst oft üblich) im Zuge der Eingabebehandlung eine Aktualisierung der betroffenen UI Komponenten durchgeführt werden. Hierfür werden Änderungskategorien und Beobachterfragmente eingesetzt (vgl. Abschnitte 8.3 respektive 8.4).

Die Elemente dieser Modularisierung sind die Verhaltensfragmente. Auf jeder Verfeinerungsebene sind die bekannten CRUD Operationen[1] mit einem

[1] CRUD – Create, Read, Update, Delete, die basischen Operationen bei Datenbanksystemen.

Fragment möglich. Das heißt ein Fragment kann

- hinzugefügt,
- entfernt oder
- verändert werden.

Verhaltensfragmente werden auf Grund von Interaktionsevents eines bestimmten Typs, welche von bestimmten Interaktionsobjekten geworfen wurden, aktiviert. Diese Tupel (Interaktionsobjekt, Interaktionsevent Typ) werden standardmäßig bei der Verfeinerung geerbt, müssen aber modifiziert werden. Zum Einen wird das Interaktionsobjekt durch seine Verfeinerung (identifiziert auf Grund der Verfeinerungsbeziehungen, vgl. Abschnitt 9.1.3) ersetzt. Zum Anderen wird der Interaktionsevent Typ angepasst, wenn das Toolkit des Klassifizierers des Interaktionsobjektes sich in der Verfeinerung geändert hat. So wurde im laufenden Beispiel beim Übergang von der Variante “Sichter” zur Variante “Sichter+Hardware” der Klassifizierer von “Swing JButton” zu “Hardware Knopf” geändert. Beide Anpassungen geschehen automatisch. Die Anpassung des Event Typs ist in Kapitel 9.1.6 detaillierter diskutiert.

8.3 Zustand

Der Zustand der MBS ist in Abbildung 8.1 unten dargestellt. Er ist für alle MBS-Varianten identisch und die gemeinsame Datengrundlage. Dahinter liegt die Anwendungslogik für Berechnungen, Datenbankzugriffe etc. (im Bild nicht dargestellt).

Verhaltensfragmente verändern den Zustand der MBS durch Modifikation seiner Eigenschaften. Solche Änderungen am Zustand werden in anwendungsspezifische *Änderungskategorien* eingeteilt, um darauf reagieren zu können (Abschnitt 8.5.2 führt hierzu u.a. detaillierter aus, warum solche Änderungskategorien die passende Granularität haben). Das Architekturmuster gibt keine konkreten Änderungskategorien vor, sondern unterstützt die Kategorisierung per se. So gehören im laufenden Beispiel u.a. “Meldungsliste geändert”, “Meldung geändert” und “Empfängerliste geändert” dazu. Hierbei beinhaltet die Kategorie “Meldung geändert” unter anderem die Änderung des Betreffs der aktuellen Meldung, aber auch der Änderung der Empfängerliste.

Modifiziert ein Verhaltensfragment den Zustand, so feuert der Zustand im Anschluss ein *Änderungsevent* mit der entsprechenden Änderungskategorie. Beobachterfragmente können sich auf eine oder mehrere dieser Kategorien subskribieren und werden beim Feuern eines Änderungsevents der passenden Kategorie aktiviert, sodass sie Aktualisierungen im UI vornehmen können.

8.4 Beobachterfragmente

Beobachterfragmente beobachten den im letzten Abschnitt eingeführten Zustand der MBS und Aktualisieren auf Grund von Änderungen an diesem die interagierte Benutzerschnittstelle, wie im rechten Teil von Abbildung 8.1 illustriert. Sie werden hierzu, wie die Verhaltensfragmente, programmatisch spezifiziert (Unteranforderung 5.3). Darüber hinaus werden sie – wie Verhaltensfragmente – beim Verfeinern geerbt und erlauben die gleichen CRUD-Operationen (Hinzufügen, Modifizieren oder Entfernen). Abschnitt 8.5.1 detailliert wie die scharfe Trennung von Ein- und Ausgabe (in Verhaltens- und Beobachterfragmente) und eine starke Modularisierung der Fragmente die Verfeinerung ermöglichen.

Im laufenden Beispiel gibt es ein Beobachterfragment, welches den Text der entsprechenden UI Elemente bei Änderungen der Empfängerliste aktualisiert. Hierzu modifiziert es die entsprechenden Interaktionsobjekte und deren Eigenschaften, was sofort durch den Interpreter zur Interaktion gebracht wird (vgl. Kapitel 7.3).

8.5 Designentscheidungen zum Architekturmuster

Da das Architekturmuster zentral für das präsentierte Lösungskonzept ist, behandelt dieses Kapitel die wichtigsten Designentscheidungen, welche zum vorgestellten Architekturmuster geführt haben.

8.5.1 Auftrennung und Modularisierung von Ein- und Ausgabe

Die Ein- und Ausgabe ist im vorliegenden Architekturmuster stark modularisiert und streng getrennt. Verhaltensfragmente (Abschnitt 8.2) übernehmen die Verarbeitung von Eingaben, wohingegen Beobachterfragmente (Abschnitt 8.4) die Ausgabe übernehmen. In dieser Form dürfen Verhaltensfragmente keine Änderungen am UI direkt vornehmen. Diese muss über eine Änderung des Zustands und die Abstraktion der Änderungskategorien (Abschnitt 8.5.2) geschehen – also den unteren Teil des in Abbildung 8.1 dargestellten Kreises.

Die Begründung liegt in der Sicherung der Lauffähigkeit der Benutzerschnittstelle in der Verfeinerung. Denn bei der Verfeinerung kann das UUI-Modell signifikant modifiziert werden. Würden geerbte Verhaltensfragmente direkt die Ausgabe (also das UUI-Modell) modifizieren, könnte dies zu Fehlern führen. Durch Trennung der Ein- und Ausgabefragmente wird die Architektur strukturierter und damit sicherer bzgl. Verfeinerungen.

Müssen bei einer Verfeinerung dann Änderungen am UUI-Modell durchgeführt werden, ist, auf Grund der starken Modularisierung, die Wahrscheinlichkeit ist höher, dass Fragmente wiederverwendet werden können. Somit müssen

nur wenige kleinere Fragmente bei der Anpassung einer verfeinerten MBS-Variante modifiziert werden. Darüber hinaus werden die einzelnen Fragmente in ihrer Größe reduziert und damit übersichtlicher.

8.5.2 Konzept der Änderungskategorien

Änderungskategorien ermöglichen, dass Beobachterfragmente auf bestimmte Änderungen des Zustands reagieren und damit die Benutzerschnittstelle aktualisieren können. Ein kanonischer Ansatz dies zu ermöglichen wäre die Subskription auf Teile des Zustands, welcher durch die Eingaben verändert wird. Dies wurde untersucht und für nicht handhabbar befunden; insbesondere, weil der Ansatz generisch genug sein muss, um für alle Konstrukte des Zustands zu greifen.

Bei einer solchen Umsetzung angelehnt an *EMF*[2], kann man sich bei jedem Objekt für Änderungsnotifizierungen subskribieren. Die Erfahrung zeigt, dass dies recht umständlich sein kann, da man sich für jedes Objekt auf jede gewünschte Änderungsart (Wertänderung, Objekt Hinzufügen, Objekt Entfernen etc.) registrieren muss. Dies kann weiter je nach beobachtetem Datentyp auch unterschiedlich funktionieren. Zusätzlich müssen alle Objekte des Zustands ein einheitliches[3] Subskriptionsschema nutzen. Schließlich ergibt sich noch die Frage, welcher Teil in der Architektur diese Subskriptionen aktuell hält, wenn ein Element in den Zustand hinzugefügt oder aus dem Zustand entfernt wird.

Alternativ wäre die Realisierung mit Hilfe einer Abfragesprache, z.B. XQuery[4] artig. Mit ihrer Hilfe könnte man Unterbäume und Elemente aus Bäumen selektieren und somit den (als Baum interpretierten Zustand) abfragen. Im Gegensatz zum EMF Ansatz lässt sich somit generisch formulieren, welche Elemente bzgl. Änderungen beobachtet werden sollten. Jedoch können Elemente aus dem Zustand über beliebige und mehrere Pfade erreichbar sein. Es stellt sich hier die Frage, welcher genutzt werden soll.

Es lässt sich resümieren, dass beide Arten von Ansätzen das Problem haben, dass man die Objekte, auf die subskribiert werden soll, und deren Objektstruktur genau bekannt sein müssen. Insbesondere müssen folgende Fragen beantwortet werden:

- Wo findet man das Objekt auf welches man sich subskribieren will?
- Auf welche Eigenschaften des Objektes muss man sich subskribieren?
- Was passiert, wenn Objekte hinzugefügt oder entfernt werden?

Beide Ansätze bergen zwar Potenzial, sind aber auch komplex in der Nutzung. Vor dem Hintergrund, dass der hier diskutierte Mechanismus nicht das

[2] EMF – Eclipse Modeling Framework. Siehe http://www.eclipse.org/emf.

[3] Man kann argumentieren, dass die Schemata vom Prinzip her heterogen sein dürfen, dann würden aber die Vorzüge einer einheitlichen Behandlung verloren gehen.

[4] http://www.w3.org/TR/xquery

zentrale Element der Architektur ist, für das ein signifikanter Einarbeitungsaufwand zu rechtfertigen wäre, widerspricht dies der einfachen Nutzbarkeit (Anforderung 4).

Aus der Erfahrung mit den vorgestellten Möglichkeiten wird abgeleitet, dass die Granularität, auf welcher man sich auf Änderungen subskribieren kann, reduziert werden muss. Also nicht auf eine bestimmte Eigenschaft eines speziellen Objektes, sondern abstrakter auf ganze Gruppen (Kategorien) von Änderungen. Diese müssen je nach Anwendung definierbar sein.

8.6 Diskussion des Architekturmusters

Im Folgenden werden wichtige Punkte bei der Nutzung des Architekturmusters diskutiert.

Beobachterfragmente modifizieren Interaktionsobjekte, um dem Benutzer eine Zustandsänderung darzustellen[5]. Dabei muss bedacht werden, dass in einer verfeinerten MBS-Variante *Interaktionsobjekte gelöscht* werden können. Eine adäquate Prüfung im Code des Beobachterfragments ist somit erforderlich. Dies kann nicht mit Verhaltensfragmenten passieren, da der Zustand für alle MBS-Varianten gleich ist.

Das vorgestellte *Architekturmuster ist sehr restriktiv*, vor allem im Hinblick auf Fragmente und die klar getrennte Nutzung von Interaktions- und Änderungsevents. Dies macht es einfacher nutzbar für Unterstützungskonzepte, sodass diese den Entwickler gezielter unterstützen können. Auch gibt es klare Richtlinien zur Umsetzung (Unteranforderung 5.4). Interessanter Weise hat nur ein Proband der Studie auf die strengen Vorgaben Bezug genommen. Selbst dieser war aber nicht pauschal abgeneigt, sondern eher interessiert daran, wie in Kapitel B.2.1 beschrieben.

Das hier vorgestellte Architekturmuster basiert auf, unterscheidet sich aber essentiell von, MVC. Die folgenden Abschnitte grenzen das in dieser Arbeit entwickelte Architekturmuster von MVC und PAC, zwei bekannten Mustern für Benutzerschnittstellen, ab.

8.6.1 Abgrenzung zu MVC

MVC (Model-View-Controller) wurde für Smalltalk entwickelt (Reenskaug 2003) [6] und ist inzwischen ein Standardmuster geworden (Gamma u. a. 1995). MVC fokussiert auf die Trennung von Anwendungs-(Domain)Code und der Darstel-

[5] Das Wort "darstellen" passt nur für grafische Benutzerschnittstellen, soll aber im übertragenen Sinne auch für andere Arten von Benutzerschnittstellen (akkustisch, olfaktorisch, ...) verstanden werden.

[6] Eine gute Zusammenfassung einiger UI Architekturstile mit Fokus auf MVC und daraus entwickelter Konzepte von Martin Fowler ist auf seinem Bliki zu finden: http://martinfowler.com/eaaDev/uiArchs.html.

lung, sodass Änderungen an der Darstellung den Anwendungscode nicht beeinflussen.

Um seiner Aufgabe gerecht zu werden, legt MVC generische *Bestandteile* der Architektur und ihr Zusammenspiel fest: Das Modell (Model), die Ansicht (View), und den Controller. Das Architekturmuster der vorliegenden Arbeit dagegen besteht aus den Bestandteilen Modell, Beobachterfragmenten, UUI-Modell und Verhaltensfragmenten. In beiden Arbeiten beschreibt die Ansicht respektive das UUI-Modell die Struktur einer MBS-Variante und der Controller respektive die Beobachterfragmente werden für die Interpretation der Eingabeereignisse genutzt. Auch das Modell übernimmt in beiden Arbeiten die gleiche Funktion, jedoch wird die *Schnittstelle zur Ausgabe* (in MVC über das Observer Pattern und den View) bei der vorliegenden Arbeit mit dem Konzept der Änderungskategorien stärker standardisiert und für das Anbinden multipler Benutzerschnittstellen auf unterschiedlichen Abstraktionsebenen vorbereitet.

Als weiteres, wichtiges Unterscheidungsmerkmal beschreibt die vorliegende Arbeit ein *Konzept für Multiple Benutzerschnittstellen und das Vererben von Struktur (View) und Verhalten (Controller)* im Detail (vgl. auch Kapitel 9). So können beispielsweise Fragmente im Rahmen der Vererbung erstellt (hinzugefügt), modifiziert oder gelöscht werden (CRUD-Operationen). MVC dagegen legt kein analoges Konzept vor. Aber erst diese klare Festlegung sorgt für eine einheitliche Basis, auf derer Unterstützungskonzepte für MBS konzipiert werden können.

8.6.2 Abgrenzung zu PAC

PAC (Presentation-Abstraction-Control) wurde 1987 als ein Modell zur Erleichterung der Implementierung von Benutzerschnittstellen entwickelt (Coutaz 1987). Ähnlich zum vorliegenden Ansatz und MVC werden Anwendungscode und Präsentation getrennt, jedoch ist bei PAC der *Controller* allein für die Koordination zwischen Abstraktion (im vorliegenden Ansatz der Zustand) und Präsentation (im vorliegenden Ansatz das UUI-Modell) zuständig. Dagegen setzt der vorliegende Ansatz auf die strikte Trennung von Ein- und Ausgabemechanismen zur Vereinfachung der Vererbung.

Der wichtigste Unterschied zwischen PAC und dem vorliegenden Ansatz liegt im Agenten-Charakter von PAC. Zu Erstellung einer Benutzerschnittstelle werden mehrerer PAC-Tripel, jedes mit den Bestandteilen Präsentation, Abstraktion und Controller, miteinander verschaltet (über ihre Controller). Jedes PAC-Tripel ist dabei für einen Teil der Präsentation und einen Teil der Verbindung zum Anwendungscode (Abstraktion) zuständig. Dies macht PAC zu einem guten Kandidaten für die Konstruktion von hoch adaptiven Schnittstellen. Jedes Triple kann mehrere Arten der Präsentation bereitstellen, wie im Comets-Ansatz von Calvary u. a. (2004) beschrieben. Es fehlt bei PAC jedoch (wie bei MVC) ein Konzept, verschiedene MBS-Varianten zu strukturieren und zu vernetzen, welches die Konzipierung von Unterstützungskonzepten zur Ent-

wicklung von MBS erst ermöglicht.

8.7 Zusammenfassung

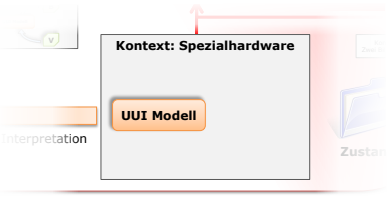
In diesem Kapitel wurde ein Architekturmuster für Multi-Benutzerschnittstellen (MBS) eingeführt, d.h. wie die Teile einer MBS aufgebaut sind und zusammenwirken (Unteranforderung 5.4). Dabei wurde insbesondere die Schnittstelle zwischen Struktur (Layout) und Verhalten (Anforderung 5) und wie sich die Verfeinerung von Struktur und Verhalten in Artefakten niederschlägt (Unteranforderung 5.1) definiert. Das Architekturmuster beschreibt, wie die modellierte Benutzerschnittstelle mit dem programmierten Verhalten zusammenarbeitet (Unteranforderung 5.3, Unteranforderung 5.4). Insofern ist das Architekturmuster zentral und auch die Grundlage für die folgenden Kapitel des Lösungskonzeptes. Die in diesem Zusammenhang relevanten Designentscheidungen wurden ausgeführt (Kapitel 8.5) und abschließend das Architekturmuster diskutiert (Kapitel 8.6).

Das Architekturmuster basiert auf MVC, forciert aber eine strikte Trennung von Ein- und Ausgabeverarbeitung. Tiefgreifender als zu MVC ist der Unterschied zu PAC, da der vorliegenden Ansatz nicht (wie PAC) auf eine Agentenstruktur setzt. Darüber hinaus fehlt sowohl bei MVC als auch bei PAC ein Konzept um verschiedene MBS-Varianten (mit UI-Struktur und Verhalten) zu strukturieren und zu vernetzen. Erst dies macht die Konzipierung von Unterstützungskonzepten zur Entwicklung von MBS möglich. Das vorliegende Architekturmuster ist somit besser auf die Erfordernisse zur Umsetzung einer Multi-Benutzerschnittstelle angepasst.

Es bleibt anzumerken, dass das in dieser Arbeit entwickelte Architekturmuster im Rahmen von Kapitel 11 in eine Architektur konkretisiert wird, indem unterstützende Komponenten und Schnittstellen beschrieben werden. Dies erst ermöglicht die Bereitstellung von Unterstützungskonzepten für die konkrete Architektur. Ein Abgleich des Architekturmusters mit den Anforderungen findet (zusammen mit den anderen Konzepten dieser Arbeit) in Kapitel 12 statt. Die Umsetzung und Evaluation sind dagegen in Teil III beschrieben.

Kapitel 9

Domänenspezifische Sprache (DSL)



Eine MBS-Variante teilt sich, wie im Überblick über das Lösungskonzept in Kapitel 7 beschrieben, in 3 Konstituenten auf: Verhaltensfragmente, Beobachterfragmente und ein UUI-Modell. Die Fragmente wurden in Kapitel 8 behandelt. Dies Kapitel beschreibt die Sprache des UUI-Modells zur Beschreibung der Struktur (des Layouts) einer Benutzerschnittstelle. Zusammen mit dem Rest des Architekturmusters bildet sie die Grundlage für die Konzipierung der Unterstützungskonzepte zur Entwicklung von Multi-Benutzerschnittstellen in Kapitel 10. Kapitel 11 schließlich beschreibt eine konkrete Architektur, in welcher die Umsetzung der DSL ein Teil ist.

Die Gliederung des Kapitels folgt den Bestandteilen einer DSL, welche in Kapitel 3 bereits eingeführt wurden. Die *Semantik* (Kapitel 9.1) beschreibt, welche Konstrukte es in der Sprache gibt und wie diese zu interpretieren sind. Die formelle Beschreibung dieser Sprachkonstrukte im Sinne einer Grammatik übernimmt die *abstrakte Syntax*, vorgestellt in Kapitel 9.2. In Kapitel 9.3 schließlich werden *Regeln zur Wohlgeformtheit* vorgestellt, welche die Formulierung von Bedingungen bzgl. der Validität von Modellen erlauben.

Die *konkrete Syntax* der Sprache legt fest, wie sich deren Benutzung aus Sicht des Entwicklers “anfühlt”. Sie wird in diesem Kapitel jedoch nicht vorgestellt, da sie stark Werkzeug spezifisch ist und somit durch die Unterstützungskonzepte bestimmt wird, welche in Kapitel 10 vorgestellt werden.

9.1 Semantik

Wie in Kapitel 3 eingeführt, beschreibt die Semantik einer Sprache die Bedeutung der einzelnen Sprachkonstrukte und ihrer Anordnung.

Ableitung der Sprachkonstrukte Die einzelnen Sprachkonstrukte lassen sich wie folgt ableiten und charakterisieren. Im Rahmen des Lösungskonzeptes wird das UUI-Modell zur Modellierung einer Benutzerschnittstelle eingesetzt. Dabei soll die Modellierung (Anforderung 2) erweiterbar auf neue Sprachen sein und (Unteranforderung 2.1) keine Sprache fest einkodiert haben. Es ist somit ein "Platzhalter-"Konstrukt erforderlich, welches verschiedene Bedeutungen (im Sinne einer Klassifizierung) zugewiesen bekommen kann: *Interaktionsobjekte* (Definition 4 in Kapitel 4.3, beschrieben in Abschnitt 9.1.1) werden zur Modellierung eingesetzt und durch *Interaktionsobjektklassifizierern* (Abschnitt 9.1.2) klassifiziert.

Somit kann eine MBS-Variante modelliert werden. In einer Modelldatei werden aber verschiedene Varianten beschrieben und miteinander in Beziehung gesetzt. Der für die Abgrenzung einer Variante notwendige Container ist die *UUIBox* (Abschnitt 9.1.3). Eine UUIBox korrespondiert damit zu einem Nutzungskontext. Die dabei gewünschte Unterstützung von Modifikationen (Anforderung 6, detailliert in Abschnitt 9.1.5) wird durch *Verfeinerungsbeziehungen* (Abschnitte 9.1.3 und 9.1.4) ermöglicht, welche Verbindungen zwischen Modellelementen herstellen.

Modellierungkonzept Für die in UUIBoxen modellierten MBS-Varianten werden keine Einschränkungen bzgl. der Abstraktionsebenen (Unteranforderung 1.1) und der Natur der Abstraktion (Unteranforderung 1.2) von Varianten gemacht. Das Konzept macht somit keine Einschränkungen bezüglich den unterstützten Nutzungskontexten. Konsequenter Weise werden die modellierten Benutzerschnittstellen als *abstraktionsunabhängige Benutzerschnittstellen (UUI)* bezeichnet, wie in Kapitel 4.3 in Definition 3 festgehalten.

Erstellt der Entwickler eine neue Variante, so leitet er diese manuell oder mit einem Werkzeug (vgl. Kapitel 10.1.4.2) von einer schon bestehenden Variante der MBS ab – er *verfeinert* diese. Die so verfeinerte Variante kann dann frei bearbeitet werden, um sie dem neuen Nutzungskontext anzupassen (Unteranforderung 3.1). Dabei können Interaktionsobjekte hinzugefügt, entfernt und ihre Eigenschaften modifiziert werden. Mit Hilfe der Interaktionsobjektklassifizierer werden die Interaktionsobjekte einer Klasse zugewiesen (z.B. PushButton), wodurch auch ihre Eigenschaften bestimmt sind.

Zusammen mit der neuen Verfeinerung werden *Verfeinerungsbeziehungen* zwischen der neuen Verfeinerung und der ursprünglichen Variante angelegt. Diese Beziehungen werden sowohl für die UUIBoxen, als auch für ihren Konstituenten, die Interaktionsobjekte, erstellt. Letztere dokumentieren, welches Interaktionsobjekt welches abstraktere Interaktionsobjekt verfeinert und werden für die Propagation von Modifkationen (Anforderung 6) ausgenutzt. Mehrfache Anwendung des Verfeinerungsschrittes resultiert in einem Baum von MBS-Varianten, dem Verfeinerungsbaum, wie in Abbildung 7.4 (Kapitel 7.4) und Abbildung 9.1 illustriert.

Modifikationen werden explizit unterstützt (Anforderung 6, Unteranforderung 6.1). Um eine Modifikation auf mehrere MBS-Varianten anzuwenden, wird sie entlang der Verfeinerungsbeziehungen propagiert. Der Entwickler kann somit durch Veränderung der Verfeinerungsbeziehungen explizit auf die Propagation von Modifikationen Einfluss nehmen (Unteranforderung 6.2).

Die verschiedenen Konzepte werden im Folgenden weiter detailliert.

9.1.1 Interaktionsobjekte

Ein `Interaktionsobjekt` (Definition 4 in Kapitel 4.3) repräsentiert eine beliebige Form der Interaktion zwischen Benutzer und der Anwendung. Es muss zwangsweise mit Hilfe von Interaktionsobjektklassifizierern (vorgestellt im folgenden Abschnitt) auf eine konkrete Interaktionsform (d.h. eine Klasse aus einem Toolkit) festgelegt werden.

Das Interaktionsobjekt wird durch Interpretation (vgl. Kapitel 7.3) zur Interaktion gebracht. Auf Grund von Benutzerinteraktionen (z.B. Eingaben oder Drücken eines Knopfes) kann der Interpreter dann Interaktionsevents werfen, welche spezifisch für den Interaktionsobjektklassifizierer sind. Interaktionsobjekte können weiter spezifische Eigenschaften haben, welche unter anderem die Wahrnehmung der Interpretation durch den Benutzer stark beeinflussen kann.

Interaktionsobjekte können auf verschiedenen Abstraktionsstufen auftreten. In Form von konkreten Interaktionsobjekten kann der Benutzer ihre Interpretation direkt wahrnehmen und mit ihr interagieren – beispielsweise ein grafischer Knopf, ein Ton oder ein Knopf auf einer Spezialhardware. Dagegen können sie als abstrakte Interaktionsobjekte (z.B. Auswahl-1-aus-N oder Aktion-anstoßen) nicht direkt vom Benutzer wahrgenommen werden, sondern bilden eine Zwischenabstraktion im MBS.

Wie in gängigen Toolkits üblich, ist es möglich, *Interaktionsobjekte zu schachteln*. Das heißt, dass in ein Interaktionsobjekt mehrere weitere Interaktionsobjekte eingebettet werden können. Zum Beispiel können JPanels aus dem Swing Toolkit JLabels und JTextFields enthalten. Das Interaktionsobjekt, welches das JPanel modelliert enthält die Interaktionsobjekte, welche die JLabels und JTextFields modellieren.

Interaktionsobjekte sind die grundlegenden Bausteine, welche im vorliegenden Ansatz zur Modellierung von Benutzerschnittstellen eingesetzt werden.

9.1.2 Interaktionsobjektklassifizierer

Ähnlich wie in UML (Object Management Group 2005) muss Interaktionsobjekten ein Klassifizierer (`Interaktionsobjektklassifizierer`) zugewiesen werden. Der Klassifizierer kann z.B. eine HTML Knopf Klasse oder Swing JPanel Klasse modellieren und ist in einer Bibliothek eingeordnet, wie in Abbildung 7.2 in Kapitel 7.3.1 illustriert. Durch die Klassifizierung werden auch die Eigenschaften bestimmt, welche ein Interaktionsobjekt haben kann.

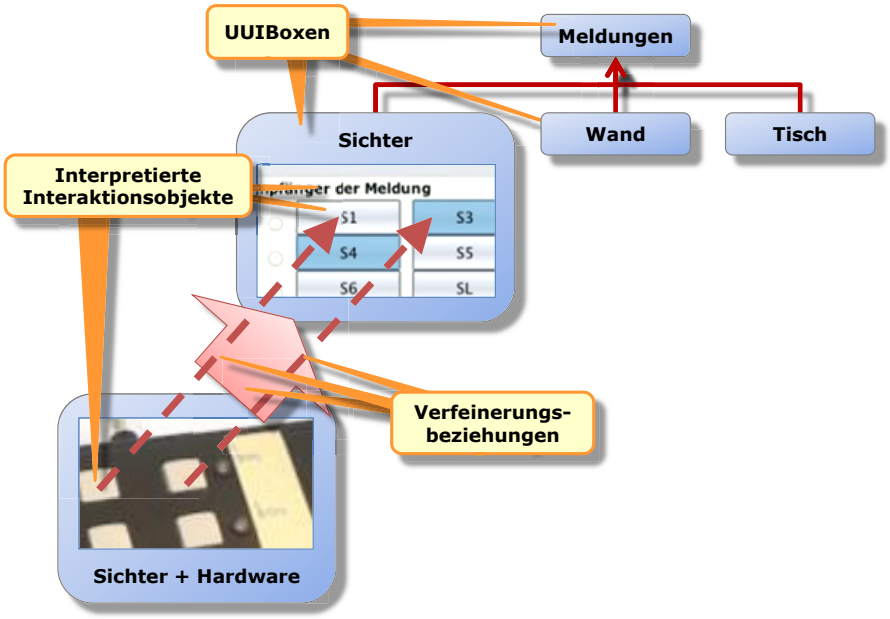


Abbildung 9.1: Die in der DSL zentralen Konzepte illustriert am laufenden Beispiel.

Interaktionsobjektklassifizierer sind der Einfachheit halber in Bibliotheken organisiert, was eine einfacher Erweiterbarkeit des Ansatzes ermöglicht (Anforderung 2), wie in Kapitel 11.4 beschrieben. Es gibt dabei keinen vordefinierten Satz von Klassifizierern, welche im Ansatz integriert sind (Unteranforderung 2.1).

Eine Menge von Klassifizierern (sie können auf mehrere Bibliotheken aufgeteilt sein) kann folglich zu einer Sprache – einem Vokabular (z.B. ein Toolkit: Swing, VoiceXML oder XForms) zusammengefasst werden. Dabei wird auch (analog zu gängigen UI Toolkits) die Vererbungsbeziehung zwischen Klassifizierern (die `Klassifiziererhierarchie`) modelliert. Ein spezialisierter Klassifizierer erbt dabei alle Attribute seines generalisierten Klassifizierers. Diese gängige Praxis der Klassifizierung und Strukturierung eines Satzes von Klassifizierern in ein Vokabular wird z.B. auch in UIML verwendet (Helms u. a. 2008).

In der vorliegenden Arbeit wird begrifflich zwischen *Spezialisierung* respektive *Generalisierung* im Kontext der Klassenhierarchie auf der einen Seite und *Verfeinerung* respektive *Abstraktion* im Kontext der Verfeinerung von Benutzerschnittstellen unterschieden.

9.1.3 Verfeinerung und UUIBoxen

Die an einen gegebenen Nutzungskontext angepasste Benutzerschnittstellenvariante wird durch das Arrangement von Interaktionsobjekten innerhalb eines UUIBox Elementes definiert. Die UUIBox ist hierbei ein Container, welcher zu einem bestimmten Nutzungskontext korrespondiert.

Zur Spezialisierung einer Benutzerschnittstelle für einen neuen Nutzungskontext wird eine abstraktere MBS-Variante verfeinert. So wird im laufenden Beispiel (vgl. Abbildung 9.1) die MBS-Variante "Sichter" von der Variante "Sichter+Hardware" verfeinert. Dabei hat der Entwickler die grafischen Knöpfe in Knöpfe auf einer Spezialhardware umgewandelt (ein konkretes Beispiel ist in Kapitel 16.2 zu finden). Wiederholte Durchführung von Verfeinerungen resultiert in einem Baum von Benutzerschnittstellenvarianten dem `Verfeinerungsbaum`. Im Baum sind abstraktere Varianten folglich weiter oben und konkretere Varianten weiter unten im Baum angeordnet. Das oberste Element wird *Wurzel* genannt. Auch ist die umgekehrte Entwicklung möglich: die Abstraktion (im Gegensatz zur Verfeinerung). Dabei muss einzig gewährleistet sein, dass der Baum danach auch noch eine eindeutige Wurzel hat. Es ist anzumerken, dass gemäß dem Konzept der abstraktionsunabhängigen Benutzerschnittstelle (vgl. Definition 3) und beliebig vieler Abstraktionsebenen (vgl. Anforderung 1), die verfeinernde Variante nicht unbedingt die Bedeutung einer konkreten Benutzerschnittstelle (CUI) haben muss.

Das Verfeinerungskonzept erlaubt es, MBS-Varianten auf beliebigen Abstraktionsebenen (Unteranforderung 1.1) zu erstellen. Dabei werden keine Einschränkungen gemacht, von was abstrahiert werden soll (z.B. Gerät, Toolkit, Benutzerrolle); dies kann *je nach Anwendungsfall* festgelegt werden (Unteranforderung 1.2). Der Verfeinerungsbaum nimmt somit unterschiedliche Formen an – je nachdem, wie vom Anwendungskontext verlangt.

Die Beziehungen, welche die verschiedenen MBS-Varianten (UUIBoxen) im Verfeinerungsbaum miteinander verbinden, werden `Verfeinerungsbeziehungen` genannt. Sie dokumentieren die Verfeinerung der MBS für einen Nutzungskontext. Neben der Verbindung von UUIBoxen wird weitere Information benötigt, um eindeutig feststellen zu können, welches Interaktionsobjekt in der Verfeinerung welches Interaktionsobjekt in der abstrakteren Variante verfeinert. Daher sind die Interaktionsobjekte auch über Verfeinerungsbeziehungen miteinander verbunden, wie in Abbildung 9.1 durch gestrichelte (rote) Pfeile dargestellt. Werden die Beziehungen durch Pfeile illustriert, sind die *Pfeile immer von der verfeinernden zur abstrakteren Version* gezeichnet. Bei der Ableitung einer Benutzerschnittstelle für einen neuen Nutzungskontext werden diese Beziehungen initial angelegt.

Abbildung 9.1 zeigt weiter die hier besprochenen Aspekte im Zusammenhang. Die Knoten des Verfeinerungsbaumes sind UUIBox Elemente. Ihre Konstituenten, Interaktionsobjekte, werden wie die UUIBox Elemente auch bei der Verfeinerung mit Verfeinerungsbeziehungen verbunden.

9.1.4 Kategorisierung der Verfeinerungsbeziehung

Gemäß dem Konzept der Verfeinerungsbeziehung (vgl. vorigen Abschnitt 9.1.3) stehen über sie zwei gegebene Interaktionsobjekte oder UUIBoxen A, B in der MBS miteinander in Verbindung. Diese Beziehung kann in eine der vier folgenden Kategorien eingeteilt werden.

Verfeinerung: Hierbei ist A (transitiv) eine Verfeinerung von B. Das heißt, dass man von B aus durch Navigation *nur in Richtung der Verfeinerung* im Verfeinerungsbaum entlang der Verfeinerungsbeziehungen auf A trifft.

Abstraktion: Gegenüber der Verfeinerung ist hier A (transitiv) eine Abstraktion von B. Das heißt, dass man von B aus durch Navigation *nur in Richtung der Abstraktion* im Verfeinerungsbaum entlang der Verfeinerungsbeziehungen auf A trifft.

Verwandt: Stehen die Elemente A, B nicht im Verfeinerungsbaum rein via Abstraktion oder Verfeinerung transitiv miteinander in Beziehung, sondern über ein gemeinsames Element C, so werden die Elemente als verwandt bezeichnet. Dabei kann man von A nach B navigieren indem man im Verfeinerungsbaum *in Richtung der Verfeinerung und auch in Richtung der Abstraktion* der Verfeinerungsbeziehungen folgt.

Nicht in Beziehung: Trifft keiner der vorigen Fälle zu, so stehen die Elemente A, B nicht miteinander in Beziehung. Bei UUIBoxen kann dies nicht vorkommen, denn sie sind die obersten Elemente der Modellierung, sie repräsentieren die verschiedenen Varianten (die nach und nach durch Verfeinerung entstanden sind) der MBS und stehen somit miteinander in Verbindung.

Man bemerke, dass Verfeinerung und Abstraktion zueinander inverse Konzepte sind. Sie sind zwei verschiedene Sichten auf ein und dieselbe Beziehung zwischen zwei Elementen. Sie treten somit immer paarweise auf.

9.1.5 Modifikationen von Elementen

Die Kernidee zur Unterstützung von Modifikationen (Anforderung 6) ist, dass eine Modifikation auf mehreren MBS-Varianten angewendet werden kann (welche im Verfeinerungsbaum miteinander verbunden sind). Mit Hilfe der Verfeinerungsbeziehungen kann eine Modifikation auf mehrere Benutzerschnittstellen angewendet werden, indem sie an den Verfeinerungsbeziehungen entlang propagiert wird (Unteranforderung 6.1). Dabei kann der Entwickler durch Modifikation der Verfeinerungsbeziehungen Einfluss auf die Propagation von Modifikationen nehmen (Unteranforderung 6.2).

Eine Modifikation kann sich *i)* auswirken auf die Eigenschaften eines Interaktionsobjektes, oder *ii)* Interaktionsobjekte hinzufügen oder entfernen. Änderungen von Eigenschaften werden propagiert, wie durch den Entwickler mit

Hilfe der Verfeinerungsbeziehungen festgelegt. Dabei können diese Beziehungen als Konsistenzbedingungen aufgefasst werden: sind Elemente durch Verfeinerungsbeziehungen verbunden, so werden Änderungen in allen MBS-Varianten übernommen – die Verfeinerungsbeziehungen beschreiben, welche Teile der MBS konsistent gehalten werden. Die zweite Form der Modifikation, das Hinzufügen und Entfernen von Interaktionsobjekten ist von anderer Natur. Sie wird in der vorliegenden Arbeit nicht durch die Sprache, sondern durch interaktive Unterstützungskonzepte umgesetzt, wie z.B. in Kapitel 10.1.4 beschrieben.

Basierend auf der vorgestellten Sprache kann die Propagation einer Änderung somit passiv oder aktiv erfolgen.

Definition 5: Werte für Eigenschaften von Interaktionsobjekten werden von der im Sinne der Verfeinerung abstrakteren Versionen des Interaktionsobjektes (**passiv**) übernommen. Diese Übernahme findet genau dann statt, wenn das Interaktionsobjekt keinen eigenen Wert für die Eigenschaft spezifiziert.

Dies bedeutet umgekehrt, dass das Setzen eines Eigenschaftswertes an einem Interaktionsobjekt auch zur Wertänderung der Eigenschaft bei verfeinernden Interaktionsobjekten führen kann, vorausgesetzt, sie spezifizieren keinen eigenen Wert für die Eigenschaft.

Definition Passive Propagation

Die Vererbung geschieht *direkt, ohne, dass Werkzeuge eingreifen müssen*. Damit können während der Propagation auch keine Modellelemente neu erstellt oder entfernt werden.

Die passive Propagation ist also auf einen begrenzten Satz von Anwendungsmöglichkeiten beschränkt und findet nur auf Eigenschaften statt. Dagegen kann die aktive Propagation gut für komplexe Änderungen genutzt werden und wird mit Hilfe semiautomatischer oder automatischer Unterstützung durchgeführt. Die aktive Propagation erlaubt insbesondere das Hinzufügen und Entfernen von Modellelementen und ist somit auf mehr Änderungen anwendbar als die passive Propagation. Wichtig bei beiden Formen ist die Abbruchbedingung. Bei der passiven Propagation war sie der gesetzte Eigenschaftswert. Bei der aktiven muss diese Bedingung separat durch das Werkzeug oder den Nutzer des Werkzeugs (Entwickler) gesetzt werden.

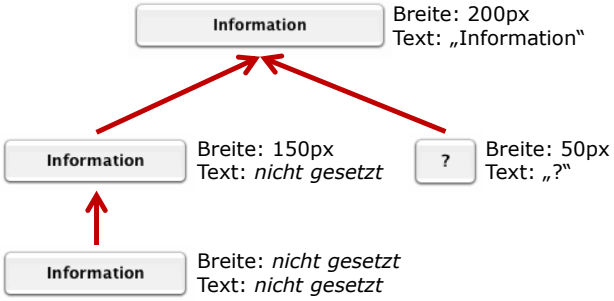


Abbildung 9.2: Beispiel zur passiven Propagation von Eigenschaftswerten. Die (roten) Pfeile symbolisieren Verfeinerungsbeziehungen. Die Texte an den Elementen illustrieren Wertzuweisungen, welche am jeweiligen Interaktionsobjekt modelliert sind. Wird dabei ein Wert nicht zugewiesen (“nicht gesetzt”), so wird der jeweilige Wert (durch die passive Propagation) vom abstrakteren Interaktionsobjekt übernommen. Der jeweils neben dem Text abgebildete Knopf ist das Ergebnis.

Definition Aktive Propagation

Definition 6: Beliebige Modifikationen werden mit Hilfe von Werkzeugen **aktiv** in mehrere MBS-Varianten propagiert. Das jeweilige Werkzeug bedient sich der Verfeinerungsbeziehungen, um die passenden Zielobjekte für die Modifikation zu identifizieren. Im Gegensatz zur passiven Propagation können *beliebige Modifikationen* propagiert werden (Werkzeug spezifisch). Dafür muss durch das Werkzeug oder den Nutzer des Werkzeugs (Entwickler) die Abbruchbedingung für die Anwendung der Modifikation spezifiziert werden.

9.1.5.1 Verankerung der passiven Propagation in der DSL

Aus der Definition der passiven Propagation folgt, dass wenn ein Wert für eine Eigenschaft gesetzt ist, dieser zum Standardwert für die Eigenschaft bei verfeinernden Interaktionsobjekten wird. Somit kann man Eigenschaften von Interaktionsobjekten in zwei Zustände einteilen.

lokal gesetzt: Die Eigenschaft wurde lokal am jeweiligen Interaktionsobjekt gesetzt und nutzt keinen Standardwert.

verfeinert: Die Eigenschaft hat im gegebenen Interaktionsobjekt keinen Wert lokal zugewiesen bekommen, sie übernimmt damit automatisch den Standardwert.

Dies ist in Abbildung 9.2 illustriert. Das oberste Element hat Werte für zwei Eigenschaften gesetzt: Text und Breite. Beim Betrachten der zweiten Ebene

Modifikation	UUIBox	Interaktionsobjekt	Eigenschaft
Einfügen	nicht sinnvoll	aktiv	nicht möglich
Löschen	nicht sinnvoll	aktiv	nicht möglich
Veränderung	nicht sinnvoll	aktiv	aktiv / passiv

Tabelle 9.1: Zeigt die Form der *Propagationsunterstützung*, eine Modifikation einer Art (Zeilen) auf Elemente eines Typs (Spalten) anzuwenden.

wird in der linken Verfeinerung der Text übernommen und die Breite etwas schmaler gesetzt, was das Bild des Knopfes widerspiegelt. In der rechten Verfeinerung werden dagegen beide Eigenschaftswerte lokal gesetzt, wie auch am daneben abgebildeten Knopf zu sehen.

9.1.5.2 Klassifikation der Modifikationen

Es wurden die möglichen Kombinationen zwischen Element Typen und Arten von Modifikationen untersucht. Tabelle 9.1 zeigt dazu zusammenfassend, welche Arten je nach Kombination sinnvoll sind. Die einzelnen Zeilen werden im Folgenden detaillierter diskutiert.

UUIBoxen repräsentieren die Nutzungskontexte, sind also nur die Container für die Spezifikation der Interaktion, tragen aber selbst nicht zur Interaktion bei. Darüber hinaus macht es keinen Sinn, mehrere UUIBoxen gleichzeitig anzulegen, da diese im Rahmen der Verfeinerung spezifisch für den jeweiligen Nutzungskontext nacheinander abgeleitet und angepasst werden. Eine Modifikation parallel auf mehrere UUIBoxen anzuwenden ist somit nicht sinnvoll.

Interaktionsobjekt können sinnvoll in ein oder mehreren UUIBoxen hinzugefügt oder gelöscht werden. Geht das Löschen noch mit einer einfachen Unterstützung, welche einfach alle Verfeinerungen entfernt, so ist das Hinzufügen nicht ganz trivial. Eine Vorgehensweise zur automatischen Propagation des Hinzufügens eines neuen Interaktionsobjektes in mehreren UUIBoxen wird in Abschnitt 10.1.4.1 entwickelt.

Eine Veränderung am Interaktionsobjekt, welche auf anderen Verfeinerungen übertragen werden kann ist insbesondere die Klassifizierung. Auf Grund der Problemstellung, welche im folgenden Kapitel 9.1.6 untersucht wird, bietet es sich hierfür an, eine aktive Unterstützung zu konzipieren.

Eigenschaften eines Interaktionsobjektes können weder eingefügt, noch gelöscht werden – sie werden gesetzt (verändert). Ihre Werte können über passive Propagation auf Verfeinerungen übertragen werden. Im Rahmen der abstrakten Syntax wird in Kapitel 9.2.3 ein solches Konzept zur Übertragung der Eigenschaftswerte vorgestellt. Aktive Konzepte sind darüberhinaus auch möglich.

Verfeinerung	Klassifikation	
	Bleibt gleich	**Ändert sich**
Auf ein Ziel	Klassifizierer erhaltende 1 zu 1 Verfeinerung vgl. 9.1.6.1	Klassifizierer veränderliche 1 zu 1 Verfeinerung vgl. 9.1.6.2
Auf mehrere Ziele	nicht sinnvoll	Klassifizierer veränderliche 1 zu N Verfeinerung vgl. 9.1.6.2

Tabelle 9.2: Die möglichen Kombinationen von Klassifizierung und Verfeinerung.

9.1.6 Verfeinerung im Zusammenspiel mit Klassifizierern

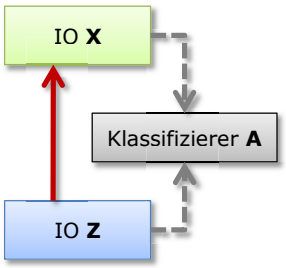


Abbildung 9.3: Klassifizierung in Verbindung mit Verfeinerung: der Klassifizierer bleibt erhalten.

Die Kombination aus den Konzepten Klassifizierung und Verfeinerung liefert eine Problemstellung, welche getrennt diskutiert werden muss. Im Fokus steht hierbei, wie die Bindung an Interaktionsevents (vgl. Abschnitt 8.2), sowie die Nutzung von Eigenschaften und Methoden durch Beobachterfragmente (vgl. Abschnitt 8.4) bei der Verfeinerung beeinflusst wird.

Tabelle 9.2 gibt einen Überblick über die möglichen Kombinationen der beiden Konzepte. Die Kombination der Klassifizierer erhaltenden 1 zu N Verfeinerung wird nicht weiter beleuchtet, da sie nicht sinnvoll ist: Zum Einen kann sie immer als Spezialfall der Klassifizierer veränderliche 1 zu N Verfeinerung aufgefasst werden und zum Anderen gewinnt man durch die Erhaltung der Klasse nichts, was eine getrennte Betrachtung sinnvoll macht, da in jedem Fall eine Aufspaltung bzw. Zusammenfassung der Werte von Nöten ist. Die verbleibenden drei Kategorien sind in Abbildungen 9.3, 9.4 und 9.5 illustriert und werden im Weiteren diskutiert.

9.1.6.1 Klassifizierer erhaltende, 1:1 Verfeinerung

Wie dargestellt in Abbildung 9.3 verfeinert ein Interaktionsobjekt Z ein abstrakteres Interaktionsobjekt X. Dabei ist der Klassifizierer des verfeinernden und des abstrakten Interaktionsobjektes der gleiche (Klassifizierer A). Insofern ändert sich auch nichts an den Interaktionsevents, welche vom Interaktionsobjekt geworfen werden. Es treten daher bei diesem Fall keine besonderen Probleme auf.

9.1.6.2 Klassifizierer veränderliche, 1:1 Verfeinerung

Dargestellt in Abbildung 9.4 ist die Klassifizierer veränderliche 1 zu 1 Verfeinerung. Hierbei verfeinert ein Interaktionsobjekt Z ein abstrakteres Interaktionsobjekt X und ändert dabei den Klassifizierer von A nach B. Dies geschieht z.B. auch im laufenden Beispiel (Kapitel 7.4) bei der Verfeinerung von "Sichter" nach "Sichter+Hardware". Dabei werden Interaktionsobjekte, welche Java JButtons modellieren durch Interaktionsobjekte, welche Hardware[1] Buttons modellieren ersetzt. Dies ist auch in Abbildung 9.1 auf Seite 98 illustriert.

Es können prinzipiell zwei Unterfälle unterschieden werden. *i)* Die beiden Klassifizierer stehen miteinander innerhalb der Klassifiziererhierarchie in Beziehung (vgl. Abschnitt 9.1.2) und *ii)* die beiden Klassifizierer sind nicht miteinander über die Klassifiziererhierarchie verwandt.

Im Falle, dass die **beiden Klassifizierer über die Klassifiziererhierarchie verbunden** sind, können erneut Unterfälle unterschieden werden.

B ist Subtyp zu A: Für die Verfeinerung wird eine Spezialisierung des Klassifizierers genutzt. Alle Eigenschaften, welche A hat, liegen somit auch bei B. Alle Events, welche von A definiert wurden, werden auch von B unterstützt. Es ist daher keine gesonderte Behandlung erforderlich.

B ist Supertyp zu A: Für die Verfeinerung wird eine Generalisierung des Klassifizierers genutzt. Dabei werden von A eingeführte Eigenschaften von B nur teilweise unterstützt. Ebenso kann A Interaktionsevents definiert haben, welche vom Basisklassifizierer B nicht unterstützt werden. Daher müssen zum Einen die Zugriffe durch Fragmente auf das Element angepasst werden, sodass nur existierende Eigenschaften genutzt werden. Zum Anderen ist die Bindung an Interaktionsevents zu überprüfen, sodass die Interaktionsevents des generalisierten Klassifizierers verwendet werden.

A und B stehen *indirekt* in Vererbungsbeziehung: Hierbei sind die beiden Klassifizierer nicht durch eine direkte Vererbungsbeziehung verbunden, aber Spezialisierung des gleichen Supertyps C. Das heißt, für die Verfeinerung wird eine andere Ausprägung der Interaktion mit ähnlicher Semantik genutzt. Die vom gemeinsamen Supertyp C eingeführten Eigenschaften und Interaktionsevents sind kompatibel und können weitergenutzt werden. Die weiteren, für B und A spezialisierten Eigenschaften und Interaktionsevents bedingen Anpassungen in Fragmenten und Eventbindungen wie im Fall ohne Klassifiziererhierarchie (im Folgenden behandelt).

[1] Das Hardware Toolkit, welches im Rahmen dieser Arbeit als Fallstudie entwickelt wurde, wird in Kapitel 13.3.2 vorgestellt.

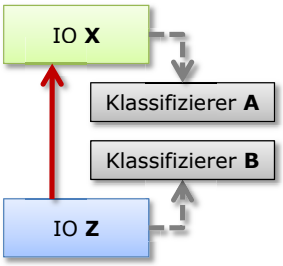


Abbildung 9.4: Klassifizierung in Verbindung mit Verfeinerung: der Klassifizierer wird geändert.

Sind die **Klassifizierer nicht über die Klassifiziererhierarchie verbunden**, wenn z.B. in der Verfeinerung ein anderes Toolkit genutzt wird, muss eine Abbildung zwischen den Eigenschaften und Interaktionsevents vorgenommen werden. Fragmente und Bindung an Interaktionsevents müssen passend modifiziert werden. Lösungsansätze hierzu können von händischer Abbildung, über die gezielte Abbildung einzelner Toolkits aufeinander, bis hin zu elaborierten Techniken des Ontology Matchings, z.B. von Doan u. a. (2004), gehen. Jedoch ist die konkrete Ausarbeitung dieser Abbildung auf Grund des Umfangs nicht Teil der vorliegenden Arbeit. Daher wurde aus Überlegungen der Umsetzbarkeit und Anwendbarkeit im Rahmen des Fallbeispiels eine Schnittstelle konzipierte. Diese erlaubt es, Komponenten einzubinden, welche die Interaktionsevents einzelner Toolkits aufeinander abbilden können.

9.1.6.3 Klassifizierer veränderliche 1:N Verfeinerung

Dargestellt in Abbildung 9.5 ist die Klassifizierer veränderliche 1 zu N Verfeinerung. Neben der Herausforderung der Änderung des Klassifizierers, wie sie im vorigen Abschnitt bereits beschrieben wurde, wird das Interaktionsobjekt aufgespalten. Dies kann z.B. eine funktionale Spaltung eines Datumsfeldes in getrennte Felder für Tag, Monat, Jahr sein oder die Duplizierung einer Wertauswahl auf mehrere parallele Eingabemöglichkeiten z.B. über ein Textfeld und einen Schieberegler.

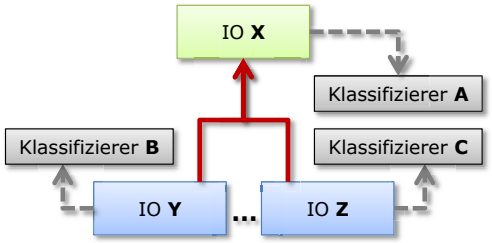


Abbildung 9.5: Klassifizierung in Verbindung mit Verfeinerung: das Interaktionsobjekt wird geteilt und der Klassifizierer ändert sich.

Zusätzlich zu den Fragestellungen bei nicht über die Hierarchie verbundenen Klassifizierern (vgl. voriger Abschnitt 9.1.6.2), treten weitere Fragen nach der Zuordnung auf. So muss spezifiziert werden, an welche Interaktionsevents welches verfeinerten Interaktionsobjektes die Verhaltensfragmente in der Verfeinerung gebunden werden. Ebenso muss das Setzen von Eigenschaften auf die verschiedenen verfeinerten Interaktionsobjekte aufgeteilt wird. Die detaillierte Behandlung dieses Aspekts geht auf Grund ihres Umfanges über die vorliegende Arbeit hinaus. Hierfür müssen unter anderem Fragen zur Sinnhaftigkeit der Automatisierung dieses Aspektes geklärt werden und in wie weit z.B. User Interface Patterns (Borchers 2000a) oder ähnliche Arbeiten zur Lösung beitragen können.

9.2 Abstrakte Syntax

Wie in Kapitel 3 eingeführt, beschreibt die Abstrakte Syntax einer Sprache die Sprachkonstrukte formell, sie legt die Grammatik der Sprache fest. Die abstrakte Syntax zielt darauf ab, Elemente mit einer gewünschten Semantik (voriges Kapitel 9.1) formell darstellbar zu machen; Sie beinhaltet allerdings noch nicht die konkrete Darstellung derselben. Wie in der Einleitung zum Kapitel 9 DSL beschrieben, wird die konkrete Syntax durch die Unterstützungskonzepte bestimmt, welche in Kapitel 10 vorgestellt werden.

Im Rahmen dieser Arbeit wurde die abstrakte Syntax mit Hilfe des Werkzeugs Enterprise Architect[2] ausmodelliert und anschließend in Java implementiert (siehe Kapitel 13.2.1), um sie danach als abstrakte Syntax für die prototypische Implementierung (Teil III) nutzen zu können.

Im Folgenden wird **nicht die komplette, umgesetzte abstrakte Syntax** wiedergegeben. Die Beschreibung ist auf den Kern fokussiert, da viele Teile der Umsetzung inhaltlich nicht interessant sind. So sind einige "organisatorische Konstrukte" zu erstellen, wie z.B. Identifizierer für Modellierungselemente, sodass sie durch Modellierungswerkzeuge eindeutig referenziert werden können. Weiter müssen zusätzliche abgeleitete Eigenschaften[3] eingeführt werden, welche den Werkzeugen eine korrekte und verständliche Betitelung der Modellierungselemente gegenüber dem Entwickler ermöglichen.

Die **Gestaltung der abstrakten Syntax** folgt dem Prinzip der Modularisierung – die Sprache wird in funktionale Einheiten zerlegt. So wird z.B. die Referenzierung über Namen mit Hilfe der Objekte BenanntesElement und Namensraum eingeführt. Diese beiden Objekte werden aber nicht genutzt, um andere Funktionen zu realisieren (wie z.B. die Klassifizierung). Mit Hilfe von Vererbungsbeziehungen werden die einzelnen Funktionen miteinander kombiniert.

Im Folgenden werden verschiedene Ausschnitte aus der abstrakten Syntax vorgestellt. Die Ausschnitte sind nach Funktionen gruppiert und in Abbildungen illustriert. Die UML-artige **Notation der Abbildungen** ist wie folgt: Die Elemente werden als gelbe Kästen dargestellt und ihre Beziehungen als graue benannte Pfeile mit nicht ausgefüllten Pfeilenden. Die Vererbungsbeziehungen zwischen Elementen hingegen sind als unbenannte Pfeile mit gelb ausgefüllten Pfeilenden notiert. Haben Elemente direkte Superklassen und diese Superklassen sind nicht im gleichen Diagramm dargestellt, so werden die Superklassen in kursiver Schrift rechts oben im Element angeführt (wie z.B. in Abbildung 9.7 die Superklasse Namensraum am Element Klassifizierer).

[2] http://www.sparxsystems.com/products/ea
[3] Eine abgeleitete Eigenschaft berechnet ihren Wert aus den Werten anderer Eigenschaften.

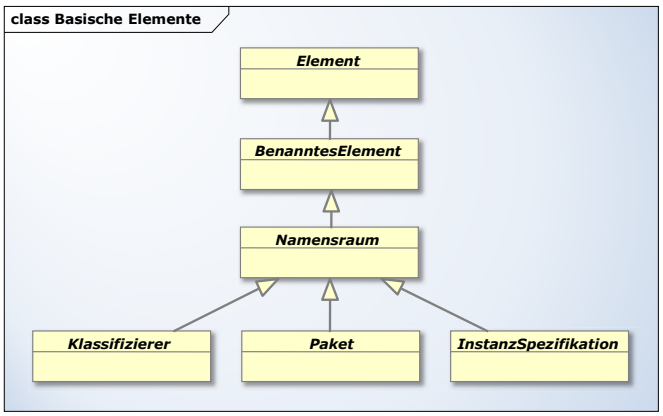


Abbildung 9.6: Grundlegende Elemente der DSL und ihre Vererbungsbeziehungen.

9.2.1 Grundlegende Elemente

Abbildung 9.6 zeigt die grundlegenden Elemente der abstrakten Syntax. Die Objekte Element, BenanntesElement, Namensraum und Paket sind häufig genutzte Sprachkonstrukte, welche typische Bedürfnisse adressieren.

Element Zur einfacheren Handhabung der Syntax wird ein Wurzelelement eingeführt, von welchem alle anderen Sprachkonstrukte erben. Modellierungswerkzeuge können sich somit immer auf dies generische Element zurückziehen.

BenanntesElement und Namensraum Eine Referenzierung über vollqualifizierte Benennung ist ein gängiger Ansatz (z.B. in Java, C# und im DNS), welcher auch im Rahmen dieser Arbeit eingesetzt wird. Hierfür werden benannte Elemente eingeführt, welche einen Namen tragen. In Kombination mit Namensraum Elementen, welche selbst wieder benannte Elemente sind, ist eine vollqualifizierte Referenzierung von Elementen über die Namenshierarchie möglich. Dabei ist der Namensraum ein Element, welches benannte Elemente aufnehmen kann (über Verschachtelung, vgl. Abschnitt 9.2.2).

Der Namensraum ist für viele andere Elemente das Basiselement, da ab hier Verschachtelung und Namen basierte Referenzierung möglich ist.

Klassifizierer und InstanzSpezifikation Klassifizierer und InstanzSpezifikationen führen das notwendige Klassifizierungskonzept in die abstrakte Syntax ein (vgl. Abschnitt 9.1.2). Ein Klassifizierer ermöglicht die Klassifikation von Instanzen (Instanzspezifikationen). Dabei können Eigenschaften am Klassifizierer spezifiziert werden, welche an einer Instanzspezifikation gesetzt werden können, wie in Abschnitt 9.2.3 ausgeführt.

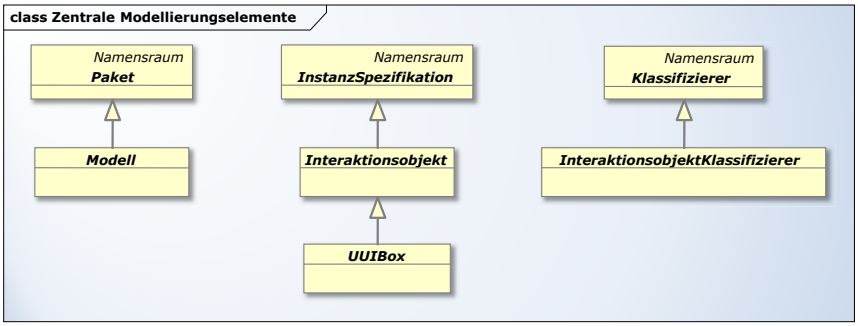


Abbildung 9.7: Zentrale Modellierungselemente und ihre Vererbungsbeziehungen zu grundlegenden Elementen.

Interaktionsobjekt und -klassifizierer Interaktionsobjekte sind die Konstituenten von unabhängigen Benutzerschnittstellenmodellen (UUIBoxen), wie in Definition 4 in Kapitel 4.3 definiert und im Kapitel 9.1.1 zur Semantik beschrieben. Sie können mit Hilfe von Interaktionsobjektklassifizierern klassifiziert werden, weshalb sie von der InstanzSpezifikation und Interaktionsobjektklassifizierer vom Klassifizierer erben.

Pakete und Modell Pakete sind ebenfalls ein typisches Konstrukt gängiger (Programmier)sprachen. Sie sind generische Container, welche zur Ordnung von Elementen genutzt werden können. Sie bedienen sich der Konzepte des Namensraum Elementes. Ein Modell ist ein spezielles Paket, welches das Containerelement für die Modellierung der gesamten MBS darstellt. In ihm sind somit die UUIBoxen enthalten, welche die einzelnen Nutzungskontexte repräsentieren.

UUIBox UUIBoxen sind Elemente, welche abstraktionsunabhängige Benutzerschnittstellen (UUIs, Definition 3 in Kapitel 4.3, Semantik in Kapitel 9.1.3 beschrieben) repräsentieren. Betrachtet man die Schachtelung von Interaktionsobjekten bei der Modellierung einer UI, so stellen sie den äußersten Container. Als Container müssen sie vom Element Namensraum erben. Für eine einfachere Handhabung in den Werkzeugen und da sie auch mit Hilfe von Verfeinerungsbeziehungen verbunden werden müssen, erben sie vom konkreteren Interaktionsobjekt, an welchem Verfeinerungsbeziehungen angelegt werden können (vgl. folgender Abschnitt).

9.2.2 Verschachtelung und Verfeinerungsbeziehungen

Die Verschachtelung von Elementen und ihre Verfeinerungsbeziehung wird in Abbildung 9.8 dargestellt.

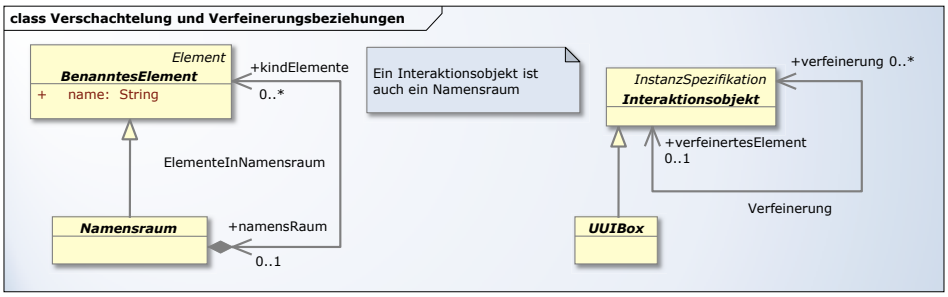


Abbildung 9.8: Abstrakte Syntax zur Verschachtelung und Verfeinerungsbeziehungen. Benannte Elemente liegen in Namensräumen. Interaktionsobjekte können verfeinert werden.

9.2.2.1 Verschachtelung

Die Verschachtelung bildet die Struktur der Benutzerschnittstelle ab (z.B. das eine Textbox in einem Panel liegt) und ist ein gängiges Konzept in UI Toolkits (z.B. Swing, HTML, C#). In der vorliegenden Arbeit ist dies durch die Namensraumhierarchie realisiert. Mit Hilfe der Beziehung *ElementeInNamensraum* können mehrere Elemente einem Namensraum zugeordnet werden. Da sie somit "im Namensraum" sind, ist die Beziehung als Komposition angelegt – ein Element ist einem Namensraum also echt zugehörig (Kardinalität 0..1).

Ein Namensraum kann beliebig viele Elemente enthalten (Kardinalität 0..*). Da ein Namensraum auch ein benanntes Element ist, können Namensräume wiederum geschachtelt werden. Es bildet sich somit eine Baumstruktur der `Verschachtelung` über Namensräume. Benannte Elemente erhalten über die Namensraum Schachtelung vollqualifizierte Namen.

9.2.2.2 Verfeinerung

Wie in Kapitel 9.1 zur Semantik eingeführt, können Interaktionsobjekte verfeinert werden. Die *Verfeinerungsbeziehung* erlaubt es, einem Element mehrere Verfeinerungen zuzuweisen (Kardinalität 0..*). Dabei kann ein Element nur ein anderes Element verfeinern (Kardinalität 0..1), was der Baumstruktur der MBS-Varianten entspricht (vgl. Kapitel 9.1.3). Der Umgang mit Eigenschaften im Rahmen der Verfeinerung wird im folgenden Kapitel 9.2.3 beschrieben.

9.2.2.3 Zusammenwirken von Verschachtelung und Verfeinerung

Verschachtelung und Verfeinerung bilden beide Baumstrukturen aus. Die Verschachtelung hat direkte Auswirkungen auf die interagierte Benutzerschnittstelle, da sie die Sortierung der Komponenten darin bestimmt. Dagegen betrifft die Verfeinerung rein die Beziehungen zwischen verschiedenen Varianten der Benutzerschnittstelle.

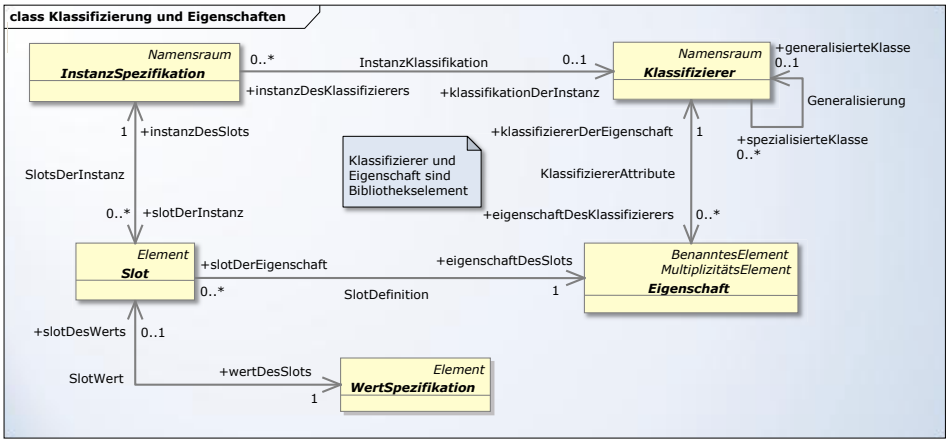


Abbildung 9.9: Abstrakte Syntax zur Klassifikation. Instanzspezifikationen können über einen Klassifizierer getypt werden und erhalten somit Eigenschaften, welchen mit Hilfe von Slots Werte zugewiesen werden können.

Die Baumstrukturen sind aber nicht komplett voneinander unabhängig, denn es darf sich die Verschachtelungsordnung beim Verfeinern nicht invertieren. Dies führt die Wohlgeformtheitsregel Verschachtelungskonsistenz in Abschnitt 9.3.6 aus.

9.2.3 Klassifizierung und Eigenschaften

Abbildung 9.9 zeigt die Klassifikation von Elementen und die Behandlung von Eigenschaften.

Klassifizierer und Eigenschaften Ein Klassifizierer klassifiziert Instanzspezifikationen über die Beziehung *InstanzKlassifikation*. Dabei kann eine Instanz nur einen Klassifizierer haben (Kardinalität 0..1), aber ein Klassifizierer kann mehrere Instanzen klassifizieren.

Eine Instanz hat Eigenschaften, da ein (ihr) Klassifizierer über die Beziehung *KlassifiziererAttribute* Eigenschaften zugeordnet bekommt. Dabei muss jede Eigenschaft genau einem Klassifizierer zugeordnet sein (Kardinalität 1). Ein Klassifizierer kann mehrere solcher Eigenschaften haben (Kardinalität 0..*). Sind Klassifizierer in einer Vererbungshierarchie angeordnet (Beziehung *Generalisierung*), so erben spezialisierende Klassifizierer die Eigenschaften ihrer Generalisierungen.

Abschließend sei erwähnt, dass die Beziehungen *InstanzKlassifikation* und *SlotDefinition* nur in eine Richtung navigierbar sind. Hierdurch können Klassifizierer und Eigenschaften in Bibliotheken abgelegt werden und unabhängig von konkreten Modellen behandelt werden (vgl. Kapitel 9.2.4).

Zuweisung von Eigenschaftswerten Die Zuweisung von Eigenschaftswerten ist analog zu UML (Object Management Group 2005) realisiert. Ein Slot verknüpft einen Wert mit einer Eigenschaft und der Instanz, deren Eigenschaft den Wert zugewiesen bekommt. Dafür referenziert der Slot genau eine Eigenschaft (Kardinalität 1) über die Beziehung *SlotDefinition*, wobei auf eine Eigenschaft bei mehreren Instanzen auch mehrere Slots zeigen können (Kardinalität 0..*).

Die Instanz, welche einen Eigenschaftswert bekommt, wird über *SlotsDerInstanz* referenziert. Ein Slot muss hierbei genau einer Instanz zugeordnet sein (Kardinalität 1), wobei eine Instanz mehrere Werte (und damit Slots) für unterschiedliche Eigenschaften haben kann (Kardinalität 0..*). Die Einschränkung "unterschiedlich" ist jedoch über die abstrakte Syntax nicht abbildbar und wird später als Wohlgeformtheitsregel in Abschnitt 9.3.4 eingeführt.

Der Wert schließlich wird über die Beziehung *SlotWert* referenziert. Ein Slot muss genau eine Wertspezifikation referenzieren (Kardinalität 1), und im Gegenzug muss diese maximal einem Slot zugeordnet sein. Dass eine Wertspezifikation genau einem Slot zugewiesen sein muss, widerspricht der Intuition (man denke an ein komplexes Objekt, welches für mehrere Eigenschaften referenziert werden soll). Komplexe Werte werden jedoch nicht direkt als Wertspezifikation abgelegt, sondern werden von dieser wiederum nur referenziert. Mit Hilfe der Abstraktion der Wertspezifikation können also unterschiedliche Arten der Wertzuweisung realisiert werden, welche als Spezialisierungen der Wertspezifikation definiert sind.

Passive Propagation Wie in Definition 5 in Kapitel 9.1.5 beschrieben, können Werte von Eigenschaften im Rahmen der Verfeinerung *passiv propagiert* werden. Ein verfeinerndes Element übernimmt dabei einen Eigenschaftswert automatisch, ohne dass am Modell Änderungen gemacht werden müssen; der Wert selbst ist nur ein mal spezifiziert. Es liegt nahe, den Eigenschaftswert des abstrakten Elements zum Standardwert für die Verfeinerungen zu machen.

Der Wert einer Eigenschaft wird (wie zuvor beschrieben) durch den Wert, welcher über einen Slot zugewiesen ist, bestimmt. Ist der Wert nicht zugewiesen, wird der Standardwert der Eigenschaft genutzt. Verfeinert das Interaktionsobjekt aber ein anderes Element, so wird – statt des Standardwertes der Eigenschaft – der Eigenschaftswert an dem abstrakten Element herangezogen. Dies wird rekursiv im Verfeinerungsbaum nach oben verfolgt, bis ein Wert gefunden wird oder ein Element nicht mehr die Verfeinerung eines Anderen ist. Hierdurch kommt der Mechanismus aus, ohne dass zusätzliche Verfeinerungsbeziehungen zwischen Eigenschaften oder Wertspezifikationen angelegt werden müssen. Es folgt, dass wenn ein Wert gesetzt ist, dieser damit der Standardwert für die jeweilige Eigenschaft für alle Verfeinerungen des Elementes ist.

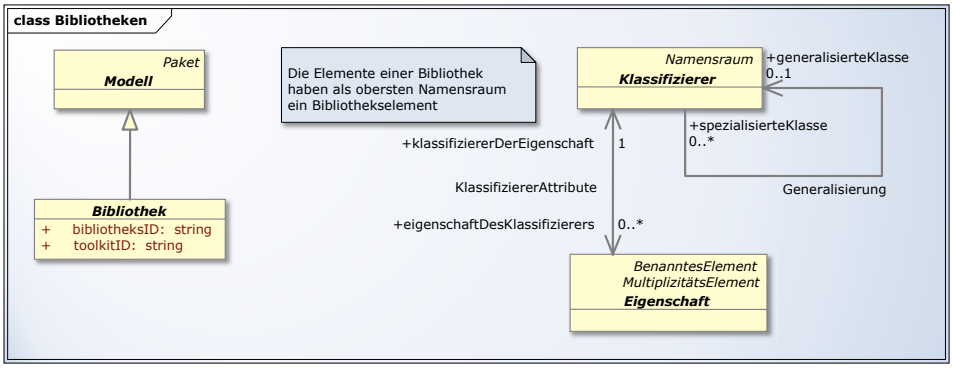


Abbildung 9.10: Bibliothekselement und Elemente, welche in der Bibliothek abgelegt werden.

9.2.4 Bibliotheken und ihre Elemente

Die Möglichkeit zur Erweiterung (Anforderung 2) bedingt, dass neue Interaktionsobjektklassifizierer für die Modellierung eingeführt werden können. Soll hierbei auch die Wiederverwendbarkeit der Klassifizierer sichergestellt werden, müssen diese (zusammen mit ihren Eigenschaftsdefinitionen) unabhängig von den UUI-Modellen organisiert werden. Als Ordnungskonstrukt wird hierbei das Element Bibliothek verwendet. Um die Objekte aufnehmen zu können und da sie neben dem UUI-Modell als eigenständiges Modell bestehen soll, wird die Bibliothek vom Element Modell abgeleitet.

Erweitert man den Ansatz, wird unterschieden (vgl. auch Kapitel 11.4) zwischen der Erweiterung einer bestehenden Sprache (Toolkit, z.B. mit eigenen Konstrukten) und der Erweiterung des Ansatzes mit einer komplett neuen Sprache (Toolkit). Dies ist notwendig, denn Elemente eines Toolkit (z.B. Swing) harmonieren miteinander (die Elemente können gemischt werden) und können zusammen zur Interaktion gebracht werden. Elemente unterschiedlicher Toolkits können dagegen nicht miteinander gemischt werden. Bibliotheken haben hierfür das Attribut “Toolkit ID”, über welche das Toolkit identifiziert wird und mehrere Bibliotheken einem Toolkit zugeordnet werden können. Über diesen Mechanismus kann ein Entwickler also auch seine eigenen Elemente zur Modellierung nutzen, welche ein existierendes Toolkit erweitern. Da aber somit ein Toolkit auf mehrere Bibliotheken verteilt sein kann, muss die Generalisierungsbeziehung zwischen Klassifizierern unidirektional sein, sodass der generalisierte Klassifizierer nichts von seinen Spezialisierungen weiß.

Werden in einer MBS-Variante verschiedene Toolkits eingesetzt, wie im laufenden Beispiel in der Verfeinerung “Sichter+Hardware” der Fall, so müssen unterschiedliche, Toolkit spezifische Teilbäume innerhalb der UUIBox modelliert werden. Die Bedingung an diese Teilbäume beschreibt die Wohlgeformtheitsregel zur Bibliothekskonsistenz in Abschnitt 9.3.3.

9.3 Regeln zur Wohlgeformtheit

Wie in Kapitel 3 beschrieben, lassen sich mit Hilfe der abstrakten Syntax nicht beliebige Bedingungen, wie valide Modelle aussehen dürfen, ausdrücken. Daher können weitere Regeln zur Wohlgeformtheit der Modelle definiert werden. In einer Modellierungsumgebung können diese z.B. eingesetzt werden, um Prüfungen der Semantik durchzuführen (Unteranforderung 4.5). Dies Kapitel beschreibt Wohlgeformtheitsregeln, welche in Verbindung mit der im vorigen Kapitel vorgestellten abstrakten Syntax zum Einsatz kommen.

9.3.1 Erlaubte Schachtelung

In der abstrakten Syntax wurden UUIBoxen als Spezialisierung von Interaktionsobjekten eingeführt. Dies würde rein syntaktisch aber auch eine Schachtelung einer UUIBox in einem Interaktionsobjekt oder in einer anderen UUIBox erlauben. Daher wird formuliert, dass das Elternelement (bzgl. der Schachtelung) einer UUIBox nur ein Paket oder eine Spezialisierung eines Paketes sein darf.

Weiter muss sichergestellt werden, dass Interaktionsobjekte nicht außerhalb einer UUIBox erscheinen. Daher wird formuliert, dass alle Interaktionsobjekte, welche keine UUIBox sind, innerhalb eines anderen Interaktionsobjektes oder innerhalb einer UUIBox liegen müssen.

9.3.2 Vollständigkeit

Damit ein Benutzerschnittstellenmodell vollständig ist, muss all seinen Elementen (Interaktionsobjekte, welche keine UUIBox sind) ein Klassifizierer zugeordnet sein. Diese Regel könnte man prinzipiell auch mit Hilfe von Kardinalitätsangaben über die abstrakte Syntax formulieren. Da das Element UUIBox aber vom Interaktionsobjekt erbt, selbst aber keinen Klassifizierer benötigt, wird diese Regel benötigt.

Des Weiteren dürfen UUIBoxen keinen Klassifizierer zugeordnet haben – diese Zuordnung wäre nicht sinnvoll.

9.3.3 Bibliothekskonsistenz

Wird eine Benutzerschnittstelle interpretiert und dargestellt (vgl. Kapitel 7.3), so können Sprachen (Toolkits) nicht gemischt werden (vgl. Abschnitte 9.2.4 und 11.4.1). Die passende Regel, welche dies formuliert, besteht aus zwei Unterregeln: *i)* die Bibliothek selbst darf nur Elemente eines Toolkits enthalten; Dies ist notwendig, da eine Bibliothek eindeutig einem Toolkit zuordnenbar ist. Und *ii)* alle Elemente eines Unterbaumes einer UUIBox müssen zu einer Bibliothek gehören, sodass bei der Interpretation immer ein kompletter Unterbaum herangezogen werden kann. Nur wenn beide Unterregeln erfüllt sind, hält auch die Bibliothekskonsistenzregel.

Unterregel i) Die vorgestellte abstrakte Syntax formuliert, dass alle Elemente einer Bibliothek innerhalb eines Bibliothekelements über die Beziehung *ElementeInNamensraum* gesammelt werden (vgl. Kapitel 9.2.4). Die erste Unterregel ist somit erfüllt, wenn alle diese Elemente unterhalb des Bibliothekselements tatsächlich zu einem Toolkit gehören. Um dies zu erfüllen, muss die Bibliothek selbst korrekt spezifiziert sein, was nicht auf Ebene des Modells geprüft werden kann, sondern im Ermessensspielraum des Menschen liegt.

Unterregel ii) Die zweite Unterregel ist erfüllt, wenn alle Teilbäume der UUIBoxen jeweils nur Unterelemente enthalten, welche jeweils alle zum gleichen Toolkit gehören. Dies bedeutet z.B., dass in einer UUIBox, wie in Abbildung 10.1 in Kapitel 10.1.1 gezeigt, zwei (d.h. mehr als ein) Teilbäume existieren können: Einer für ein grafisches UI, dessen oberste Komponente zum Swing-Toolkit gehört, auf der anderen Seite ein Teilbaum für ein Spezialhardware UI, dessen oberste Komponente zum Hardware Toolkit gehört. Solche Unterbäume beginnen immer direkt in der Ebene unter einer UUIBox, da sonst Teilbäume existieren würden, welche gegen Unterregel ii verstoßen.

9.3.4 Eigenschaftsauflösbarkeit

Im Rahmen der abstrakten Syntax (Abschnitt 9.2.3) wurde die Struktur einer Wertzuweisung für Eigenschaften beschrieben. Jedoch konnten zwei Bedingungen zur Sicherung der Konsistenz bei der Wertzuweisung an einem Interaktionsobjekt nicht festgehalten werden. Zum Einen dürfen nur Eigenschaftswerte für Eigenschaften des referenzierten Klassifizierers gesetzt werden (Unterregel i). Zum Anderen muss auch sichergestellt werden, dass eine Eigenschaft nicht mehrfach einen Wert zugewiesen bekommt (Unterregel ii). Die Regeln können an Hand Abbildung 9.9 auf Seite 111 nachvollzogen werden.

Unterregel i) Da die Modellierung von Eigenschaften in der vorgestellten abstrakten Syntax an UML angelehnt ist, muss auch eine ähnliche Regel formuliert werden, welche die Auflösbarkeit einer Eigenschaft im Bezug auf den Klassifizierer der InstanzSpezifikation beschreibt. Diese Regel verhindert, dass z.B. für einen HTML Knopf ein Wert für die Eigenschaft “text” modelliert wird, wobei das referenzierte Eigenschaftselement aber zum Klassifizierer für JAbstractButton gehört. Oder bildlich gesprochen: wenn man in Abbildung 9.8 vom Slot aus links und rechts herum zum Klassifizierer geht, muss für einen gegebenen Slot auf beiden Wegen der gleiche Klassifizierer erreicht werden.

Dazu wird die UML Regel, Seite 127 (Object Management Group 2005) angepasst zu: die einen Slot definierenden Eigenschaft (Beziehung *SlotDefinition*) muss eine Eigenschaft des Klassifizierers (Beziehung *KlassifiziererAttribute*) sein, welcher der Klassifizierer (Beziehung *InstanzKlassifikation*) der Instanz-Spezifikation ist, der der Slot zugeordnet ist (Beziehung *SlotDerInstanz*).

Unterregel ii) Zur Erfüllung der zweiten Unterregel muss sichergestellt sein, dass eine Eigenschaft nicht mehrfach einen Wert zugewiesen bekommt. Somit wird formuliert, dass zu einer Instanzspezifikation keine zwei unter-

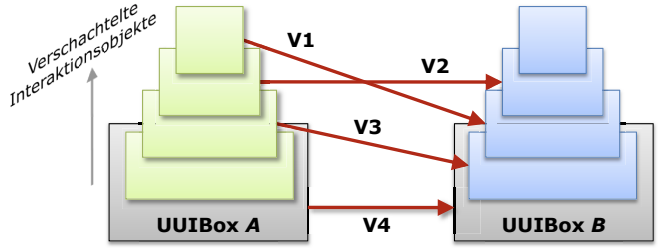


Abbildung 9.11: Illustration einer nicht validen Verschachtelung; die Verfeinerungsbeziehungen V1 und V2 kreuzen sich.

schiedliche Slots existieren dürfen, welche auf das gleiche Eigenschaftselement verweisen.

9.3.5 Zirkuläre Verfeinerungen

Damit das präsentierte Konzept funktioniert, muss die Baumstruktur der Verfeinerungen erhalten bleiben. Dies bedeutet vor allem, dass keine zirkulären Verfeinerungsbeziehungen entstehen bzw. modelliert werden dürfen. Daher gilt: für jedes Interaktionsobjekt A, welches von Interaktionsobjekt B verfeinert wird, darf A nicht die (transitive) Verfeinerung von B sein. `Transitiv` hat hierbei die gängige Bedeutung, dass wenn Interaktionsobjekt A durch Interaktionsobjekt B verfeinert wird, welches wiederum durch Interaktionsobjekt C verfeinert wird, so ist C transitiv auch eine Verfeinerung von A.

9.3.6 Verschachtelungskonsistenz

Bei der Verfeinerung einer UUIBox dürfen Verschachtelungshierarchien nicht invertiert werden, damit die inhaltliche Konsistenz gewahrt bleibt. Eine solche Invertierung würde bedeuten, dass eine Teilmenge plötzlich zur Übermenge wird. Zum Beispiel könnte eine Empfängerliste innerhalb einer Nachricht liegen. Nach einer Verfeinerung darf jedoch nicht die Nachricht innerhalb der Empfängerliste liegen. Im Bezug auf die Verfeinerung bedeutet dies, dass ein Kindelement eines Elements nach der Verfeinerung nicht dessen Elternelement sein kann.

Auf der anderen Seite ist das Ein- und Ausbetten eines Elementes erlaubt. Zum Beispiel kann die Empfängerliste in einer Verfeinerung aus der Nachricht herausgezogen sein, weil es besonders wichtig für den avisierten Nutzungskontext ist. Ebenso erlaubt sein muss das Hinzufügen und Entfernen von Elementen in der Verschachtelungshierarchie (ein weiteres Element Nachrichtenkopf zur Gruppierung von Empfänger, Betreff, etc. könnte z.B. hinzugefügt werden).

Die Regel wird somit wie folgt formuliert. Verfeinert eine UUIBox B eine UUIBox A (transitiv), werden die Elemente (Interaktionsobjekte) X innerhalb A betrachtet. Für alle in X geschachtelten Elemente Y wird überprüft, ob

deren (transitive) Verfeinerungen Y' in B auch wirklich nicht Elternelemente der Verfeinerung X' von X in B sind. Denn wäre dies der Fall, würden sie, wie in Abbildung 9.11 illustriert, die Verfeinerungsbeziehungen des verletzenden Elementes und von X kreuzen (Pfeile V1 und V2 in der Abbildung).

9.4 Zusammenfassung

Dies Kapitel 9 stellte die domänenspezifische Sprache (DSL) des in der vorliegenden Arbeit entwickelten Lösungskonzeptes vor. Sie dient zur Beschreibung der Struktur von Benutzerschnittstellen. Die Sprache ist auf das Architekturmuster aus Kapitel 8 abgestimmt und besteht aus den Teilen Semantik (Kapitel 9.1), abstrakte Syntax (Kapitel 9.2) und Wohlgeformtheitsregeln (Kapitel 9.3). Die konkrete Syntax wird durch die Unterstützungskonzepte bestimmt, welche im folgenden Kapitel 10 eingeführt werden.

Die vorgestellte Sprache differenziert sich in mehrfacher Hinsicht von anderen Arbeiten:

Unterstützte Abstraktionen: Ein zentrales Differenzierungsmerkmal zu anderen Arbeiten sind die unterstützten Abstraktionen (Anforderung 1), welche mit dem vorliegenden Ansatz beliebig gewählt werden können. Dies gilt sowohl für die Anzahl der Ebenen (Unteranforderung 1.1), als auch für die Natur der Abstraktion (Unteranforderung 1.2). Vergleichbare Arbeiten erlauben nur zwei Abstraktionsebenen: Eine AUI und eine CUI (vgl. Kapitel 4.2 zur Begriffsbildung und Vergleich, sowie Kapitel 6 für verwandte Arbeiten). Darüber hinaus wird bei verwandten Arbeiten meist die Natur der Abstraktion (im Allgemeinen von einem bestimmten Toolkit oder Gerät) festgelegt. Der vorliegende Ansatz dagegen erlaubt die freie Wahl von beidem je nach Anwendungsfall. Es können beliebige Abstraktionsebenen genutzt werden und frei gewählt werden, von was zu abstrahieren ist (auch das Toolkit, aber z.B. auch die Nutzerrolle).

Betrachtung des Modifikationsproblems: Ein weiteres wichtiges Differenzierungsmerkmal ist die Betrachtung des Modifikationsproblems, denn große Teile der verwandten Arbeiten konzentrieren sich nur auf das Erstellungsproblem – die initiale Erstellung einer Benutzerschnittstelle. Dabei liegen die zu betrachtenden Fragestellungen anders, wie in Abschnitt 4.1.4 beschrieben. Damask und UIML/Jelly (Diskussionen in Abschnitt 6.6) bilden hierbei eine Ausnahme. Damask adressiert das Modifikationsproblem, löst es allerdings nur für zwei Abstraktionsebenen mit fester Abstraktionsnatur. UIML/Jelly erlaubt, dass die Inhalte von Interaktionselementen konsistent gehalten werden. Die Lösung der vorliegenden Arbeit stellt eine Lösung für das Modifikationsproblem über beliebige Abstraktionsebenen und Abstraktionsnatur bereit. Der Entwickler wird durch passive Propagation zwischen beliebigen Eigenschaften der Interaktionselemente auf beliebigen Abstraktionsebenen unterstützt (die aktive

Propagation ist Teil der im kommenden Kapitel entwickelten Unterstützungskonzepte), er kann diese durch die Verfeinerungsbeziehungen kontrollieren und gezielt beeinflussen (Unteranforderung 6.2).

Erweiterbarkeit: Darüber hinaus wurde die Erweiterbarkeit (Anforderung 2) in den betrachteten verwandten Arbeiten nur von UIML konkret adressiert (Cameleon beschränkt sich auf abstrakte Aussagen). UIML (Diskussion in Abschnitt 6.6) und die in der vorliegenden Arbeit entwickelte Sprache hingegen nutzt die *Bibliotheks-Metapher*. Dies hat den Vorteil einer stabilen und einfachen Kernsprache, gepaart mit maximaler Flexibilität im Bezug auf Erweiterbarkeit (Atkinson und Kühne 2005). Gegenüber UIML bietet der vorliegende Ansatz die Kombination der Erweiterbarkeit mit beliebigen Abstraktionsebenen und -Natur, sowie einer Lösung des Modifikationsproblems.

Ein Abgleich der Sprache mit den Anforderungen findet (zusammen mit den anderen Konzepten dieser Arbeit) in Kapitel 12 statt. Die Umsetzung und Evaluation sind dagegen in Teil III beschrieben.

Kapitel 10

Unterstützungskonzepte

Wie im Überblick über das Lösungskonzept (Kapitel 7) beschrieben, wurde in den vorangegangenen Kapiteln das Architekturmuster für Multi-Benutzerschnittstellen (MBS) vorgestellt (Kapitel 8) und eine domänenspezifische Sprache (DSL) für die Modellierung der Struktur der verschiedenen Varianten der Benutzerschnittstelle eingeführt (Kapitel 9). Das folgende Kapitel befasst sich mit Unterstützungskonzepten für Entwickler, welche ihm bei der Entwicklung von MBS helfen.

Insbesondere *bestimmen die im Folgenden vorgestellten Unterstützungskonzepte die konkrete Syntax* der in Kapitel 9 vorgestellten DSL. Wie in Kapitel 3 eingeführt, legt die konkrete Syntax (hier synonym zu Notation) fest, “wie” in der Sprache modelliert wird. Das heißt, mit welcher Repräsentation der Sprache ein Entwickler umgehen muss. Da die Unterstützungskonzepte ihren Wert maßgeblich bei der Interaktion mit dem Entwickler zeigen, findet ihre Bewertung im folgenden Teil III zur Realisierung und Evaluation statt.

Die Unterstützungskonzepte werden im Folgenden in zwei Kategorien eingeteilt. *Interaktive Unterstützungskonzepte*, vorgestellt in Kapitel 10.1, erlauben die Modifikation der MBS. Dagegen zielen *explorative Unterstützungskonzepte*, wie in Kapitel 10.2 vorgestellt, darauf ab, dem Entwickler die MBS, ihre Struktur und ihre Zusammenhänge transparent zu machen.

10.1 Interaktive Unterstützungskonzepte

Interaktive Unterstützungskonzepte erlauben es, dem Entwickler die MBS zu modifizieren, beispielsweise mit Editoren. Sie erlauben damit auch das aktive Propagieren von Änderungen, wie in Kapitel 9.2.3 eingeführt.

Als Erstes wird In Kapitel 10.1.1 der für den Ansatz zentrale Modellinterpreter eingeführt. Er bildet auch die Basis für die darauf eingeführten Editoren. Schließlich werden zwei spezialisierte Konzepte eingeführt: das Konzept modularer Adaptionen (Kapitel 10.1.3), welches spezielle Anpassungen in einer einfach zu nutzenden Form kapselt und Verfeinerungsbaum basierte Konzepte (Kapitel 10.1.4), welche u.a. zur Propagation einer Änderung auf verschiedene MBS-Varianten nutzbar sind.

10.1.1 Modellinterpreter

Wie in Abschnitt 7.3 beschrieben, müssen Modelle der Benutzerschnittstelle interpretiert werden, um sie zur Interaktion zu bringen. Das passende Unterstützungskonzept eines Modellinterpreters wird in diesem Kapitel entwickelt und vorgestellt. Die Entscheidung zum Ansatz der Interpretation und Abgrenzung zu einer programmierten Benutzerschnittstelle wurde in Kapitel 7.3 erörtert.

Ein Interpreter bringt Elemente einer UUIBox zur Interaktion. Hierfür ließt und interpretiert er ein UUI-Modell und interagiert passend zur Interpretation mit dem Benutzer. Er ist somit eine Abbildung von Elementen der UUIBox auf die aktive, vom Benutzer wahrgenommene Benutzerschnittstelle. Da die Interpretation Toolkit spezifisch ist, kann nicht jeder Interpreter alles anzeigen. Umgekehrt können aber mehrere Interpreter gleichzeitig eine UUIBox bedienen und unterschiedliche Elemente daraus anzeigen, wie in Abbildung 10.1 zu sehen (vgl. auch Regel zur Bibliothekskonsistenz 9.3.3).

Je nach gewünschter Abstraktionsebene, welche durch das verwendete Toolkit definiert wird, *legt der Interpreter somit eine konkrete Syntax für die Sprache fest*: Die zur Interaktion gebrachte UI. Diese ist vorerst nur eine Syntax zum Lesen, ohne Möglichkeiten, Änderungen am Modell vornehmen zu können.

Eingaben des Nutzers müssen gemäß Architekturmuster (Kapitel 8) in Aktivitäten von Verhaltensfragmenten umgesetzt werden. Dazu werden Eingaben vom Interpreter zurück in das Modell geschrieben und passende Interaktionsevents erzeugt. Letztere werden der MBS in einem Toolkit agnostischen Format übermittelt. Somit findet die Kopplung des Interpreters mit anderen Teilen der MBS maßgeblich über Modellaktualisierungen und weitergeleitete Interaktionsevents statt (wie im Architekturmuster beschrieben). Dies Konzept ist abstrakt

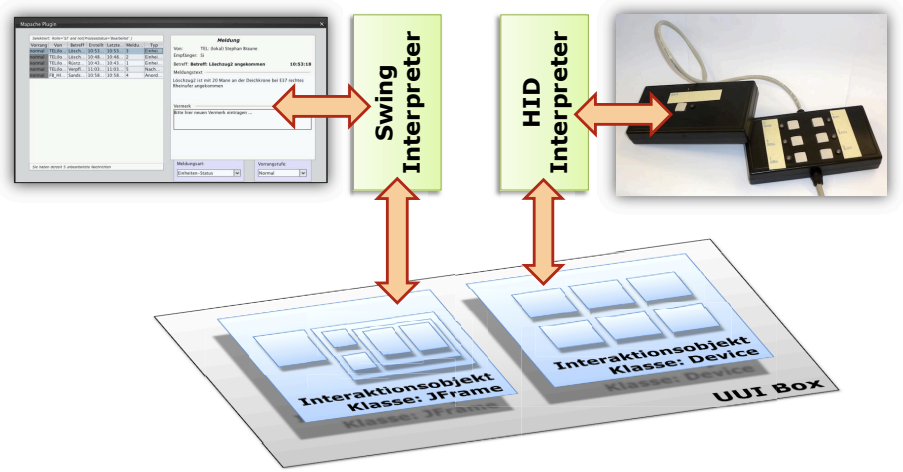


Abbildung 10.1: Interpretiertes UUI-Modell einer föderierten Benutzerschnittstelle aus dem laufenden Beispiel, bestehend aus einem grafischen Teil (Swing) und einem Teil Spezialhardware. Zwei Interpreter greifen parallel auf eine UUI-Box zu, um verschiedene Teile der Benutzerschnittstelle zur Interaktion zu bringen.

genug, um verschiedene Toolkit basierte Interaktionsformen abzubilden, da die Abstraktion im UUI-Modell ein Interaktionsobjekt und nicht ein konkretes Elemente eines bestimmten Toolkits (z.B. Swing) ist. Das heißt, dass kein Toolkit fest in der DSL einkodiert ist (Unteranforderung 2.1).

10.1.1.1 Komponenten des Interpreters

Abbildung 10.2 skizziert die Komponenten eines Interpreters. Der Interpreter wird zur Interpretation an ein Interaktionsobjekt einer UUIBox (`Wurzelelement`) gebunden, welches er dann zur Interaktion bringt. Dabei muss der Entwickler auch auf die echten, vom Interpreter erstellten, Objekte der Benutzerschnittstelle zugreifen könne, z.B. für Kompatibilität mit anderen Frameworks, welche beispielsweise Java Swing Objekte erwarten. Der Interpreter stellt die hierfür passende Schnittstelle bereit.

Werden für einen Anwendungsfall spezielle UI Komponenten (z.B. ein Swing JPanel mit Möglichkeit zur Meldungszuordnung) geschrieben, so muss der Interpreter ermöglichen, das Toolkit mit diesen speziellen Komponenten zu erweitern (vgl. Kapitel 11.4 zur Erweiterung). Hierfür muss er eine Schnittstelle zur Aktualisierung der verfügbaren Komponenten bereitstellen – z.B. die Möglichkeit, ihm einen Classloader zum Laden der Komponenten mitzugeben. Der Classloader wird an die **Komponente zur Instanziierung** übergeben, welche zur Startzeit aktiviert wird. Diese instanziiert die UI, welche zur Interaktion gebracht wird, gemäß Modell. Dafür wird das Modell rekursiv eingelesen und die

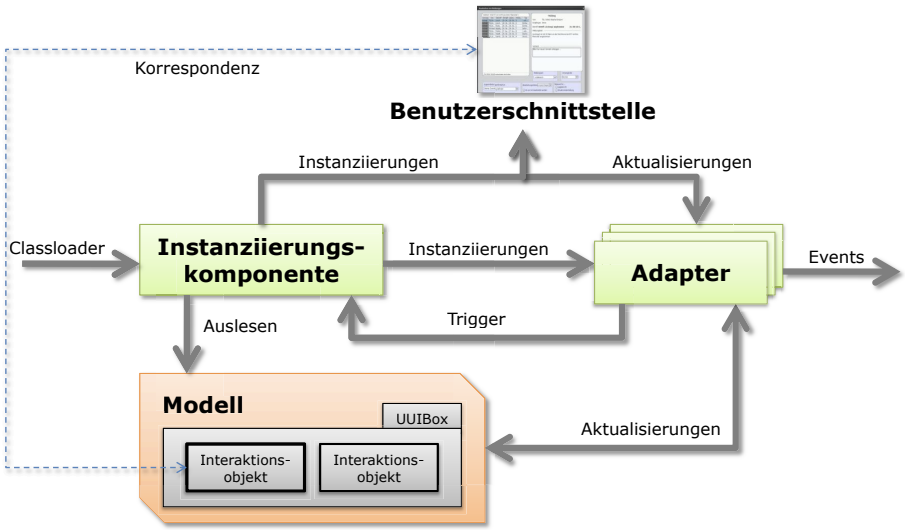


Abbildung 10.2: Komponenten eines Interpreters und deren Zusammenwirken. Nach der Instanziierung übernehmen Adapter die Kopplung der zur Interaktion gebrachten Benutzerschnittstelle mit dem Modell.

jeweils passende Komponente instanziiert. Für die instanziierten Komponenten werden jeweils Adapter erschaffen, welche mit der instanziierten Komponente und den Interaktionsobjekten aus dem UUI-Modell verbunden werden.

Die **Adapter** verbinden ein bis mehrere Modellelemente mit ein bis mehreren Komponenten aus der zur Interaktion gebrachten Benutzerschnittstelle. Verfolgt wird ein Observer und ein Adapter Architekturmuster. Adapter observieren Änderungen an Benutzerschnittstelle und Benutzerschnittstellenmodell, sowie Interaktionsevents, welche von Benutzerschnittstellenkomponenten gefeuert werden. Änderungen an der zur Interaktion gebrachten UI werden über die Adapter sofort in das Modell geschrieben. Dagegen werden Änderungen am UUI-Modell durch die Adapter sofort zu Änderungen an der zur Interaktion gebrachten UI umgesetzt. Ist dabei eine Instanziierung notwendig, wird hierfür die Instanziierungskomponente bemüht. Von Komponenten der zur Interaktion gebrachten UI geworfene Interaktionsevents werden an die Verarbeitung für Verhaltensfragmente weitergeleitet.

10.1.1.2 Adaptertypen

Wie ausgeführt, verbinden Adapter ein bis mehrere Modellelemente mit ein bis mehreren Komponenten aus der zur Interaktion gebrachten Benutzerschnittstelle. Zur Erfüllung dieser Aufgabe werden drei grundlegende Adaptertypen im Folgenden beschrieben, welche zum Einsatz auch kombiniert werden können (und im Rahmen der Umsetzung, vgl. Kapitel 14.3 auch kombiniert wurden).

Containeradapter: Innerhalb von Interaktionsobjekten mit Container Charakter (z.B. klassifiziert als Swing JPanel) können weitere Interaktionsobjekte liegen. Daher muss die ElementeInNamensraum-Beziehung (vgl. abstrakte Syntax in Kapitel 9.2.2) überwacht werden und ggf. neue Interaktionsobjekte zur Interaktion gebracht, bzw. Gelöschte entfernt werden.

Komponentenadapter: Interaktionsobjekte haben Eigenschaften, welche die Eigenschaften der Elemente in der zur Interaktion gebrachten Benutzerschnittstelle bestimmen. Das heißt, dass die Eigenschaftswerte zwischen den Interaktionsobjekten im UUI-Modell und den jeweils korrespondierenden Komponenten der zur Interaktion gebrachten Benutzerschnittstelle synchron gehalten werden müssen. Bezogen auf die abstrakte Syntax bedeutet dies die Überwachung der Slots und Wertspezifikationen des Interaktionsobjektes auf Veränderung (vgl. abstrakte Syntax in Kapitel 9.2.3). Daneben muss der Adapter sich bei "seinen" Komponenten in der zur Interaktion gebrachten UI registrieren, um auf Änderung ihrer Eigenschaftswerte reagieren zu können.

Reflexionsadapter: Adapter können fest für ein oder mehrere Interaktionsobjektklassifizierer entwickelt werden, sodass nur Interaktionsobjekte unterstützt werden, welche durch einen dieser Klassifizierer klassifiziert sind. Es besteht aber auch die Möglichkeit, Adapter generisch zu entwickeln, sodass auch Interaktionsobjektklassifizierer unterstützt werden, welche beim Design des Adapters noch nicht zur Verfügung standen. Sie unterstützen damit eine einfache Erweiterbarkeit der Sprache, da keine speziellen Adapter für jede Komponente erstellt werden müssen.
Solche Adapter greifen reflektiv auf die Komponenten in der zur Interaktion gebrachten UI zu (daher der Name), da deren Klasse (im objektorientierten Sinn) bei Erstellung des Adapters noch nicht bekannt war. Dies wird auch in Kapitel 11.4 konzeptionell und in Kapitel 13.3.2 im Bezug auf die prototypische Umsetzung beschrieben.

10.1.2 Modellinterpretierende Editoren

Wie in Kapitel 5.4 definiert, beschreibt Unteranforderung 4.2, dass Teil des Lösungsansatzes Editoren sein sollten, welche eine direkte Modifikation der Benutzerschnittstelle gemäß dem WYSIWYG Paradigma ermöglichen. Anders formuliert sollte das Modell "wie echt" interagiert werden mit der zusätzlichen Möglichkeit, daran Modifikationen anbringen zu können. Es bietet sich also an, die Interpreter aus dem vorigen Kapitel hierfür heranzuziehen, da sie genau auf passendem Abstraktionsniveau arbeiten

Der Interpreter wird um Editierfunktionalität erweitert, sodass er das UUI-Modell modifizieren kann. Unterschiedliche Editoren können auf verschiedenen Abstraktionsebenen arbeiten. Zum Beispiel kann ein Editor für ein spezielles

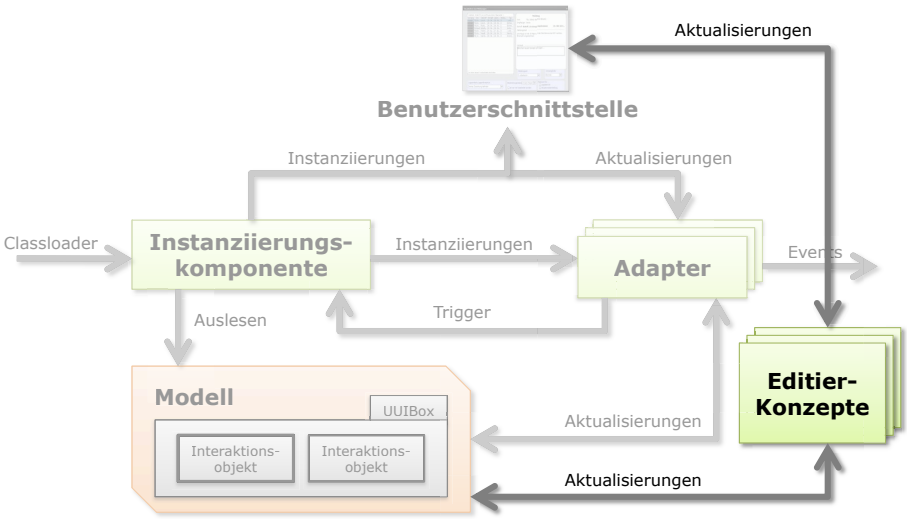


Abbildung 10.3: Erweitertes Interpreterkonzept zum Editieren von Benutzerschnittstellenmodellen.

Toolkit (beispielsweise Swing) oder generisch, komplett Toolkit unabhängig arbeiten. Die durch die Editierfunktionalität unterstützten Toolkits müssen eine Untermenge der Toolkits sein, welche der Interpreter, auf dem die Editierfunktionalität aufbaut, unterstützt.

Das im vorigen Kapitel vorgestellte Konzept für den Interpreter (vgl. Abbildung 10.2) muss hierfür nicht verändert, nur erweitert werden. Editierkonzepte werden eingebaut, welche, passend zu der zur Interaktion gebrachten UI, bereitgestellt werden. Dies kann z.B. für Swing basierte UIs ein Auswahlrahmen mit Möglichkeit zur interaktiven Größenänderung und Verschieben sein (Kapitel 14.3 für eine prototypische Umsetzung des Editors). In der Praxis und Literatur stehen ausreichend WYSIWYG basierte Editierkonzepte zur Verfügung.

Da der Entwickler neben dem Interpreter nun auch mit den Editierkonzepten arbeitet, bestimmen *Interpreter und Editierkonzepte zusammen die konkrete Syntax der DSL.*

10.1.2.1 Erweitertes Interpreterkonzept

Das Konzept muss es folglich ermöglichen, mit dem Entwickler zu interagieren und Modelländerungen durchzuführen. Die Editierkonzepte fügen in der zur Interaktion gebrachten Benutzerschnittstelle Elemente für das Editieren hinzu. Dies geschieht für den Interpreter transparent, wodurch die Editierfunktionalität von der grundlegenden Funktionalität, das UI anzuzeigen (in diesem Kontext also die Vorschau), getrennt ist und somit auch getrennt gewartet und entwickelt werden kann. Abbildung 10.3 zeigt ein um solche Editierfunktiona-

lität erweitertes Interpreterkonzept.

Der Entwickler interagiert mit den zusätzlichen Editierelementen, welche die Editierkonzepte in der interagierten UI hinzugefügt haben. Das Ergebnis dieser Interaktion ist eine Modifikation der UI, was durch das jeweilige Editierkonzept direkt in eine Modelländerung umgesetzt wird. Da der Interpreter agnostisch bzgl. der Quelle von Modelländerungen ist, wird er diese aufgreifen und die zur Interaktion gebrachte UI entsprechend anpassen.

Dies wird am Beispiel eines Editierkonzepts zum Verschieben und Verändern der Komponentengröße illustriert (vgl. prototypische Implementierung in Kapitel 14.3). Vom Editierkonzept wird ein Rahmen mit 4 "Griffen" um die selektierte Komponente gezeichnet. Wird an einem der Griffe mit der Maus gezogen, um die Komponente in der Größe zu verändern, so schreibt das Editierkonzept die neue Größe der Komponente in das UI Modell. Der Interpreter erkennt die Modelländerung und aktualisiert die Größe der selektierten UI Komponente.

10.1.3 Modulare Adaptionskonzepte

Für große Eingriffe in das Benutzerschnittstellenmodell (komplexe Adaptionen, wie z.B. Skalieren von größeren Teilen der UI), eignen sich die Editierkonzepte, welche sich in das Modell der Benutzerschnittstelle zeitweise einbetten, nur bedingt. Änderungen von vielen Elementen können einfacher bei ausgeschaltetem Interpreter en-block durchgeführt werden. Zusätzliche Interaktion mit dem Nutzer (z.B. für eine Parameterisierung) ist in einer separaten UI (und nicht eingebettet in die modellierte UI) ebenfalls einfacher zu realisieren. Daher wurde ein Konzept entwickelt, welches mehrere bzw. komplexe Modifikationen gekapselt als modulare Adaptionen zur Verfügung stellt. Dies ermöglicht es, eine solche Adaption besser zu unterstützen und somit für den Entwickler einfacher nutzbar zu machen (Anforderung 4).

Die Komponenten des Ansatzes sind dargestellt in Abbildung 10.4. Kern ist die **Adaptionskomponente**. Sie kapselt die zu unterstützende Adaption. Dabei kann die Adaption Makro basiert sein und das Makro insbesondere mit konkreten Werten (z.B. Skalierungsfaktoren) parametrisieren. Direkte Änderungen am UUI-Modell können durch die Adaptionskomponente auch vorgenommen werden.

Zum Festlegen von Adaptionsparametern und Einstellungen muss die Adaptionskomponente mit dem Entwickler interagieren. Da es sich um komplexere Anpassungen handelt, welche mehrere Komponenten betreffen können, würde eine Integration dieser Interaktion in andere Werkzeuge die Komplexität für den Entwickler erhöhen (konträr zu Anforderung 4). Daher wird eine dezidierte UI genutzt: Die Adaptionsschnittstelle. Dabei sollen Änderungen durch und Einstellungen für die Adaptionskomponente dem Entwickler möglichst transparent gemacht werden (Unteranforderung 4.1). Hierfür wird eine Vorschau genutzt (Unteranforderung 4.2) bzw. relevante Elemente im UI hervorgehoben.

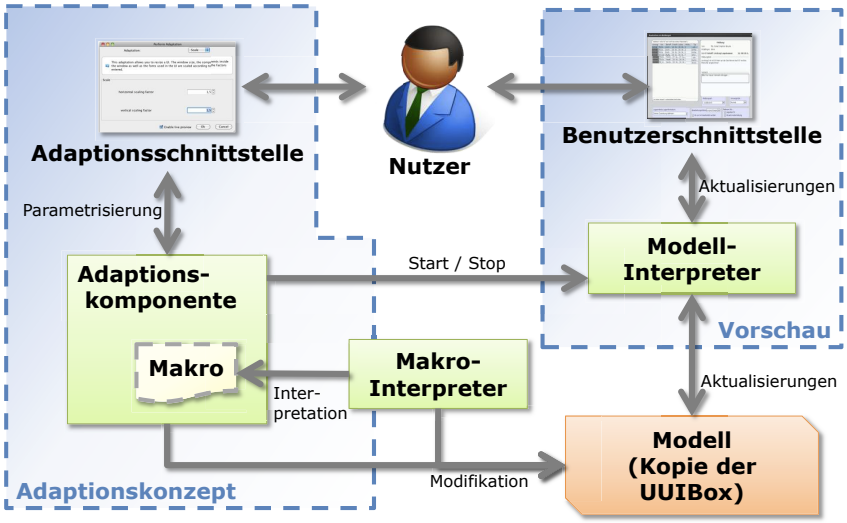


Abbildung 10.4: Elemente der Unterstützung durch modulare Adaptionskonzepte. Der fachlich relevante Teil wird in einer Adaptionskomponente gekapselt (ggf. Makro basiert), der Interpreteransatz liefert die Möglichkeit zur Vorschau von Änderungen.

Eine **Vorschau** der Anpassung ist sinnvoll, um Auswirkungen von Änderungen dem Entwickler möglichst transparent darzustellen. Die Vorschau wird über einen Interpreter realisiert, welcher ein modifiziertes UUI-Modell interpretiert. Hierzu wird im Vorfeld eine Kopie der UUIBox angelegt, sodass durch einfaches Löschen der Ursprungszustand wiederhergestellt werden kann. Jede Änderung am Modell wird (wie in Kapitel 10.1.1 beschrieben) direkt in eine Änderung der zur Interaktion gebrachten UI umgesetzt, sodass Entwickler Änderungen direkt evaluieren können. Die Vorschau (d.h. der Interpreter) wird je nach Bedarf gestartet oder beendet.

10.1.3.1 Implikationen

Hier wird der *Vorteil von Interpretern ausgespielt*. Jegliche Änderung kann direkt evaluiert werden und es sind keine Umwege über Zwischenartefakte wie UI Code nötig. Mögliche Probleme durch Inkonsistenzen werden damit ausgeschlossen. Darüber hinaus vereinfacht die dezidierte Adaptionsschnittstelle die Nutzung durch den Entwickler.

Durch die Kapselung ist weiter keine Erstellung einer spezialisierten Vorschau Komponente per Adaption nötig, stattdessen müssen nur das Adaptionskonzept und die Adaptionsschnittstelle erstellt werden – die für die Adaption fachlich relevanten Komponenten werden klar herausgelöst. Über den zur Verfügung gestellten Makrointerpreter wird das Erstellen von Adaptionen unterstützt.

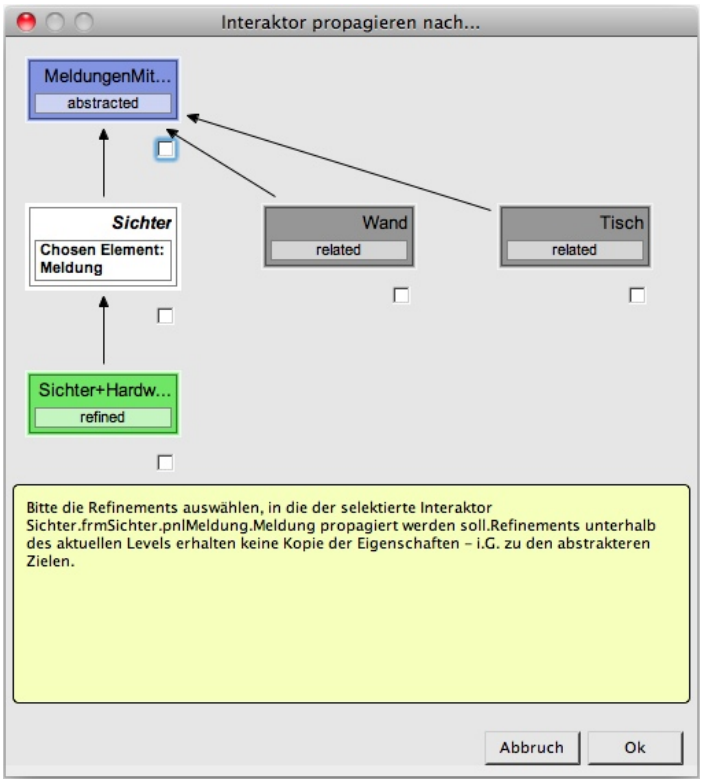


Abbildung 10.5: Verfeinerungsbaum basierte Auswahl der Ziele für die Propagation.

In der prototypischen Umsetzung wurden gängige Anpassungen, welche z.B. bei der Migration von UIs von Arbeitsplatzrechnern auf Großbildwände gemacht werden müssen, unterstützt, wie in Kapitel 14.3.3 beschrieben. Daneben wurde eine spezielle Adaptionsschnittstelle zur direkten Nutzung des Makrointerpreters für den Entwickler erstellt.

10.1.4 Verfeinerungsbaum basierte Unterstützungskonzepte

Verschiedene Unterstützungskonzepte fokussieren sich auf den *Verfeinerungsbaum* (eingeführt in Kapitel 9.1.3). Im Folgenden Kapitel 10.1.4.1 wird ein Konzept zur Propagation von Modifikationen im Baum vorgestellt. Das anschließende Kapitel 10.1.4.2 beschäftigt sich mit der Erstellung von neuen Verfeinerungen und Abstraktionen im Baum.

10.1.4.1 Propagation von Änderungen

Für einige Konzepte steht die Übertragung einer Modifikation auf mehrere MBS-Varianten im Mittelpunkt. Hierdurch sollen vor allem dem Entwickler Modifikationen an UIs (Unteranforderung 6.1) möglichst einfach gemacht wer-

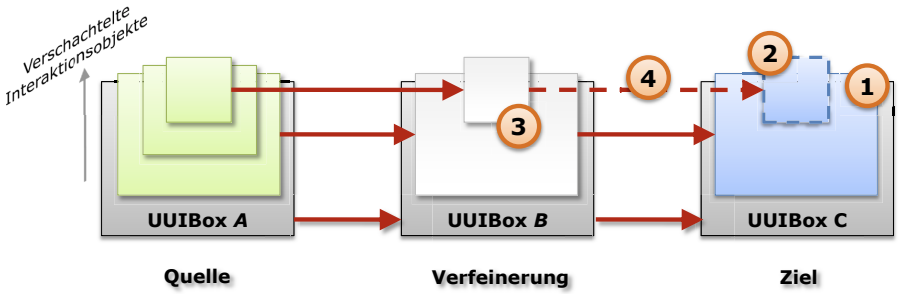


Abbildung 10.6: Illustration der Schritte zur Propagation eines neuen Interaktionsobjektes in ein oder mehrere UUIBoxen. (Rote) Pfeile sind Verfeinerungsbeziehungen, gestrichelte Objekte werden bei der Propagation erstellt.

den (Anforderung 4). Das hier beschriebene Unterstützungskonzept stellt für diese Kategorie von Unterstützung ein konzeptionelles Gerüst bereit.

Einschränkung der Zielvarianten Alle solche Unterstützungskonzepte (wie z.B. das Löschen eines Elementes propagieren, Klassifizierung der Interaktionsobjekte in mehreren Varianten ändern, ein neues Interaktionsobjekt propagieren, multiple Modifikationen propagieren, ...) benötigen vom Entwickler die Information, in welche Zielvarianten die Änderung zu übertragen ist. Dabei kann die Auswahl der Zielvarianten eigeschränkt sein, z.B. auf Grund von Inkompatibilitäten des Toolkits, bereits getätigter Modifikation oder nicht unterstützte Propagationsrichtung (Abstraktion statt Verfeinerung). Innerhalb dieser Rahmenbedingungen müssen die UUIBoxen gewählt werden, in die eine Änderung übertragen werden soll.

Hierfür wird die schon vorhandene Visualisierung des Verfeinerungsbaumes (vgl. Kapitel 10.2.1) genutzt, um dem Entwickler einen möglichst hohen Wiedererkennungswert (Anforderung 4) zu bieten und die Auswahl transparent zu gestalten (Unteranforderung 4.1). Dies ist in Abbildung 10.5 gezeigt.

Hinzufügen eines Interaktionsobjektes in mehreren MBS-Varianten Eine solche Modifikation ist das Hinzufügen eines Interaktionsobjektes in mehreren MBS-Varianten. Dabei sind konzeptionell zwei Probleme zu lösen: Es muss zum Einen beachtet werden, dass *i)* in Verfeinerungen Interaktionsobjekte hinzugefügt oder weggenommen worden sein könnten. Dies erschwert die Positionierung des propagierten Interaktionsobjektes innerhalb der Verschachtelungshierarchie. Zum Anderen kann es sein, dass *ii)* eine UUIBox, welche im Verfeinerungsbaum zwischen der Quell- und Zielbox liegt, keine Verfeinerung des Elements beinhaltet. Sie muss also übersprungen werden.

Die Zielvarianten sind hierbei auf die Verfeinerungen der aktuellen UUIBox beschränkt.

Das Vorgehen wird illustriert in Abbildung 10.6: Das innerste Interaktionsobjekt aus UUIBox A soll nach UUIBox C propagiert werden.

1. Es wird das erste Elternelement (bzgl. Verschachtelung) des abstrakten (zu verfeinernden) Interaktionsobjektes gefunden, welches in der Ziel-UUIBox C eine Verfeinerung hat.

2. Innerhalb dieses wird das neue Interaktionsobjekt erstellt (in Abbildung gestrichelt dargestellt).
   Anschließend wird das Element gesucht, zu welchem von dem neu erstellten Elemente aus eine Verfeinerungsbeziehung (Abstraktion) erstellt werden soll.

3. Dazu wird die erste UUIBox gesucht, welche eine Verfeinerung des Quellelementes aus UUIBox A enthält und in der Verfeinerungshierarchie über der Ziel UUIBox C steht.
   Das Ergebnis im Beispiel ist ein Element in UUIBox B.

4. Die Verfeinerungsbeziehung wird dann zwischen dem verfeinernden Element in der gefunden UUIBox B und dem neu erstellten Element in UUIBox C angelegt.

10.1.4.2 Erstellung von Verfeinerungen und Abstraktionen

Eine neue MBS-Variante muss bei ihrer Erstellung in den Baum der Verfeinerungen eingefügt werden und durch Verfeinerungsbeziehungen mit den anderen UUIBoxen und ihren Interaktionsobjekten verknüpft werden. Das Erstellen von neuen Varianten (Abstraktion und Verfeinerung) im Verfeinerungsbaum inkl. der notwendigen Verfeinerungsbeziehungen gehört somit zur grundlegenden Unterstützung für den Entwickler. Konzeptionell leiten neue Varianten von schon bestehenden ab (vgl. Kapitel 9.1.3). Eine Kopie der schon bestehenden Variante als Ausgangspunkt für weitere Modifikationen bereitzustellen, ist somit eine gute Grundlage für die weiteren Nutzungskontext spezifischen Modifikationen.

Neue Varianten können alternativ auch mit Hilfe von Transformationen erstellt werden. Diese gibt es in verschiedener Komplexität von einfachen, welche nur die Klassifikation von Interaktionsobjekten ändern, bis hin zu optimierenden und Constraint lösenden Transformationen. Die methodischen Implikationen hierzu werden in Kapitel 12.2.1 weiter beleuchtet.

Verfeinerung Zur Verfeinerung möchte der Entwickler eine UUIBox unter einer bestehenden UUIBox im Baum einhängen. Dafür wird eine neue UUIBox erstellt und über eine Verfeinerungsbeziehung mit der AusgangsUUIBox verbunden. Interaktionsobjekte in der ZielUUIBox werden gemäß Verschachtelung

und mit gleichem Namen (und anderen nicht Toolkit spezifischen Eigenschaften), wie in der AusgangsUUIBox erstellt. Zwischen dem jeweiligen Ausgangsinteraktionsobjekt und dem neu Erstellten wird eine Verfeinerungsbeziehung angelegt. Auf Grund der passiven Propagation (Definition 5 in Kapitel 9.1.5) erbt das neu erstellte Interaktionsobjekt somit alle Toolkit spezifischen Eigenschaften des Ausgangselementes, sowie dessen Verhalten.

Abstraktion Der umgekehrte Weg ist das Anlegen einer Abstraktion für eine bestehende UUIBox. Es muss dabei darauf geachtet werden, dass nach Erstellung der neuen, abstrahierten UUIBox dennoch nur eine Wurzel im Verfeinerungsbaum vorhanden ist. Nach Anlegen der neuen UUIBox und Verbinden dieser zur AusgangsUUIBox über eine Verfeinerungsbeziehung, werden äquivalente Interaktionsobjekte, gleich zum Vorgehen bei der Verfeinerung, in der neu erstellten Box erzeugt.

Da aber die Verfeinerungsbeziehung in diesem Falle umgekehrt wirkt, müssen die Toolkit spezifischen Eigenschaften (vgl. Kapitel 9.2.3 für die abstrakte Syntax) und Fragmentzuordnungen (vgl. Kapitel 8.2 für die Zuordnungstupel und 8.4 für Beobachterfragmente) von den Ausgangsinteraktionsobjekten zu den neu erstellten Interaktionsobjekten verschoben werden. Denn so erben die "alten" Interaktionsobjekte korrekt von der neu erstellten Abstraktion.

Hierfür wird zum Einen die Beziehung zwischen Slot und Interaktionsobjekt angepasst. Der Slot und das Wertspezifikationsobjekt müssen in den neuen Namensraum verschoben werden. Zum Anderen müssen die Fragmentzuordnungen angepasst werden. Dafür werden die Zuordnungstupel auf die neu erstellte Variante umgeschrieben und die Interaktionsobjekte in den Tupel durch die neu erstellten ersetzt. Beobachterfragmente werden auch auf die neu erstellte Variante umgeschrieben und müssen beim Modifizieren von UUI-Modellen die abstraktere Variante berücksichtigen.

10.2 Explorative Unterstützungskonzepte

Interaktive Unterstützungskonzepte, wie im vorigen Kapitel vorgestellt, erlauben die interaktive Modifikation der MBS durch den Entwickler. Im Gegensatz dazu zielen explorative Unterstützungskonzepte auf die Exploration der MBS, ihrer Struktur und Zusammenhänge ab: Der Zustand der MBS dem Entwickler transparent gemacht werden (Unteranforderung 4.1). Explorative Unterstützungskonzepte erlauben dabei keine Modifikation der MBS.

10.2.1 Verfeinerungsansicht

Die verschiedenen Varianten der MBS sind in einer Baumstruktur angeordnet, wie in Kapitel 9.1.3 beschrieben. Ziel der Verfeinerungsansicht ist es, diese zentrale Struktur der MBS-Varianten dem Entwickler transparent zu machen (Unteranforderung 4.1). Dazu wurde die in Abbildung 10.7 zu sehende Darstellung des Verfeinerungsbaumes entwickelt.

Die Darstellung orientiert sich am Aufbau des Baumes: UUIBoxen sind zentrale Elemente und als Boxen dargestellt. Ebenso sind deren Verfeinerungsbeziehungen wichtig, welche als Pfeile zwischen den Boxen dargestellt sind. Um dem Entwickler Orientierung zu geben, wie seine aktuellen Arbeiten im Verfeinerungsbaum zu verorten sind, wird eine *Farbcodierung* der einzelnen Boxen genutzt, um deren jeweilige Beziehung zur aktuell editierten UUIBox anzugeben. Das Farbschema nutzt die Kategorisierung der Verfeinerungsbeziehungen, welche in Kapitel 9.1.4 vorgestellt wurde: Das Bezugselement (die aktuell editierte UUIBox) ist weiß dargestellt. Abstraktere Varianten sind blau eingefärbt, verfeinernde Varianten dagegen grün. Varianten, welche nicht in direkter Verfeinerungsbeziehung stehen sind grau gefärbt.

Mit der Visualisierung können nicht nur UUIBoxen, sondern auch einzelne Interaktionsobjekte exploriert werden. Das im Editor selektierte Interaktionsobjekt (im Abbildung 10.7 mit dem Namen *btnS6*) wird im Baum in der selektierten UUIBox dargestellt. Zu den anderen MBS-Varianten wird angezeigt, ob das selektierte Interaktionsobjekt dort analoge Elemente hat, die mit ihm über Verfeinerungsbeziehungen verbunden sind (jeweils in den inneren Boxen dargestellt). Ist kein verwandtes Element vorhanden, wird “n/a” ausgegeben.

Die Visualisierung wird zusammen mit anderen der vorgestellten Unterstützungskonzepte eingesetzt. So haben alle Komponenten in der Entwicklungsumgebung eine gemeinsame, synchronisierte Selektion, welche auch von der Verfeinerungsansicht angezeigt wird (vgl. Kapitel 11.3). Des Weiteren dient sie auch dazu, MBS-Varianten zum Editieren zu öffnen, die Verfeinerung und Ab-

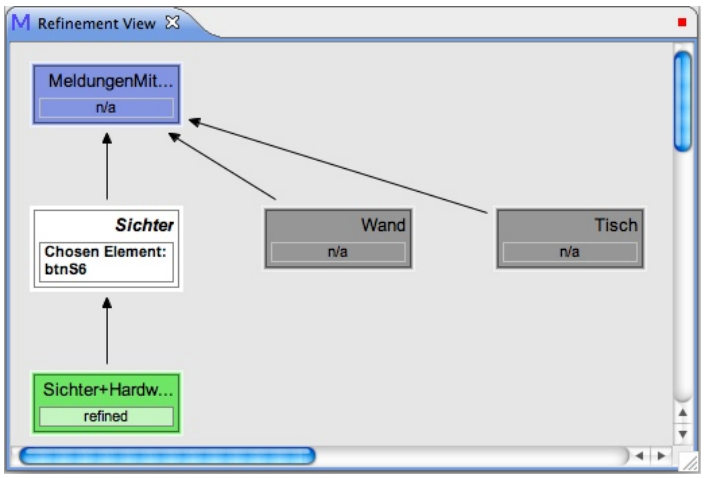


Abbildung 10.7: Die Verfeinerungsansicht ermöglicht es dem Entwickler, einen schnellen Überblick über die Struktur der MBS-Varianten zu bekommen. Des Weiteren ist es möglich, einzelne Interaktionsobjekte im Bezug auf ihre Verfeinerungen zu untersuchen.

straktion von UUIBoxen (vgl. Kapitel 10.1.4.2) anzustoßen, die Adaption einer UUIBox mit Hilfe der Adaptionskonzepte anzubieten (vgl. Kapitel 10.1.3) oder die gewählte UUIBox testweise zur Interaktion zu interpretieren (vgl. Kapitel 10.1.1). Das direkte Starten dieser Aktionen über die Visualisierung wird zwecks einfacher Nutzbarkeit (Anforderung 4) unterstützt. Die Verfeinerungsansicht trägt damit zur besseren Integration der verschiedenen Werkzeuge bei (Unteranforderung 4.3).

10.2.2 Visualisierung des aktiven Sets von Fragmenten

Im Rahmen des Architekturmusters (Kapitel 8) wurde der Zusammenhang von Benutzerschnittstellenmodell und den Verhaltens- sowie Beobachterfragmenten (im Folgenden als Fragmente bezeichnet) beschrieben. Grundsätzlich können in verschiedenen MBS-Varianten unterschiedliche Fragmente vorkommen. In Verfeinerungen können Fragmente übernommen, neu erstellt, modifiziert oder entfernt werden (vgl. Kapitel 8.2). Der Entwickler wird durch das vorgestellte Konzept bei der Analyse unterstützt, welche Fragmente in einer gegebenen MBS-Variante aktiv sind. Dies ist vor allem vor dem Hintergrund einer gängigen Kritik am MVC Architekturmuster wichtig, dass die Nutzung des Observer Patterns (Beobachter-Musters) es signifikant erschwert, die Zusammenhänge eines Programms zu erkennen – Maier, Rompf und Odersky (2010) besprechen hierzu Probleme, ebenso wie Martin Fowler[1]. Auf Grund des strikten Architekturmusters der vorliegenden Arbeit ist es jedoch möglich, eine Visualisierung zu konzipieren, welche die Einbindung der Observer transparent macht.

[1] http://martinfowler.com/eaaDev/OrganizingPresentations.html

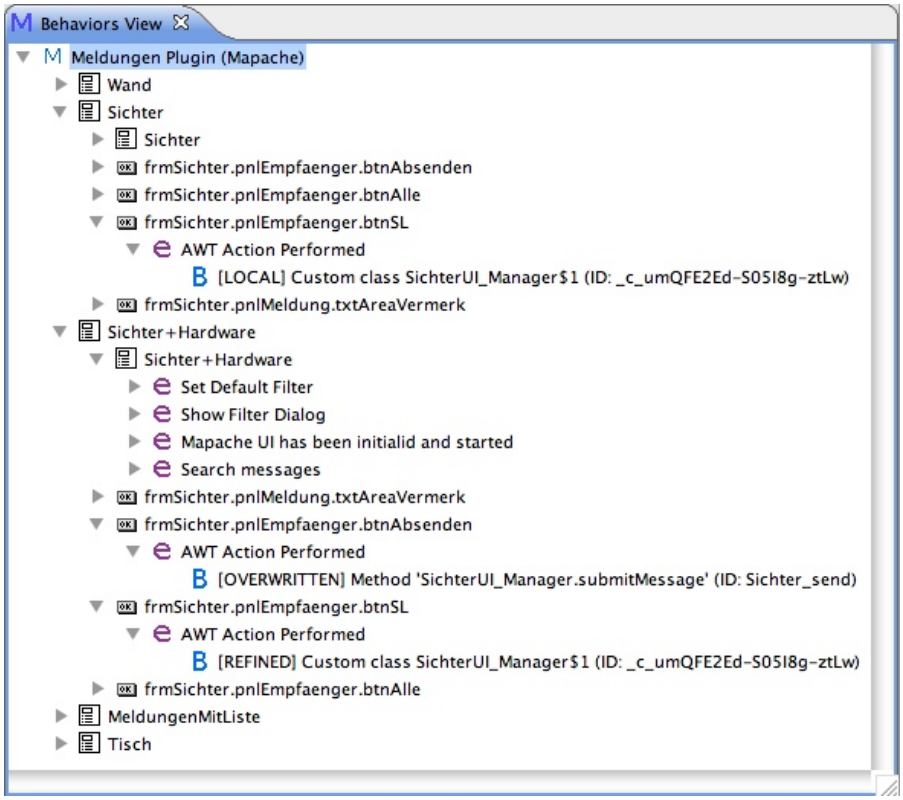


Abbildung 10.8: Darstellung der Verhaltensfragmente des laufenden Beispiels. Die Markierungen [Local], [Overwritten] und [Refined] an Fragmenten beschreiben, welche Fragmente in der jeweiligen MBS-Variante neu eingeführt, überschrieben, respektive unverändert übernommen wurden. Man erkennt, dass das Verhalten des Knopfes btnSL im Sichter hinzugefügt und von Sichter+Hardware übernommen wird.

Strukturierung der Darstellung *Verhaltensfragmente* sind über Tupel aus Interaktionsobjekt und Interaktionsevent angebunden (vgl. Abschnitt 8.2). Dabei ist zu beachten, dass mehrere Fragmente an das gleiche Tupel Interaktionsobjekt und Interaktionsevent gebunden sein können. Die Interaktionsevents können ferner auch von Extern sein, z.B. aus einem umgebenden Framework stammen. Dazu dient die UUIBox als Quelle (Interaktionsobjekt) des Interaktionsevents. Im Fallbeispiel SoKNOS ist dies geschehen, wie in Abbildung 10.8 zu sehen. Die Events auf einem Level mit “Set Default Filter” sind alle anwendungsspezifisch aus dem SoKNOS Framework.

Zur Strukturierung der Darstellung bietet sich eine Baumstruktur auf Grund der leichten Navigierbarkeit an. Das oberste Element hierbei sind die UUIBoxen, dann kann nach Interaktionsobjekt und folgend Interaktionsevent zum Fragment navigiert werden. Alternativ kann nach der UUIBox auch der Inter-

aktionsevent und dann das Interaktionsobjekt stehen. Wobei die erste Version adäquater ist, da man ein Verhalten in erster Linie mit einer Komponente aus der Benutzerschnittstelle assoziiert und die Art des Interaktionsevents dabei zweitrangig ist.

Beobachterfragmente dagegen sind alleine an Änderungskategorien des Zustands gebunden (vgl. Kapitel 8.4). Dementsprechend liegt bei der Strukturierung unterhalb der MBS-Variante nur noch die Änderungskategorie vor dem Fragment. Einer Änderungskategorie können mehrere Beobachterfragmente zugeordnet sein. Ein Beobachterfragment kann darüber hinaus auch mehreren Änderungskategorien zugeordnet sein.

Darstellung Neben der Möglichkeit, zu den Fragmenten zu navigieren, wird an der Visualisierung festgehalten, ob ein Fragment lokal (in der jeweiligen MBS-Variante) neu hinzugefügt, gelöscht, modifiziert oder geerbt wurde (vgl. Einteilung in Abschnitt 8.2). Dabei taucht ein Fragment, welches geerbt wird, ggf. an mehreren Stellen in der Ansicht auf: In Abbildung 10.8 ist das Fragment, welches auf den Knopf "SL" reagiert in der Sichter Variante eingeführt und in der Sichter+Hardware Variante übernommen worden. Dabei muss berücksichtigt werden, dass sich bei Verhaltensfragmenten das Zuordnungstupel (Interaktionsobjekt, Interaktionsevent) von Variante zu Variante ändert (vgl. Kapitel 8.2). Dies ist bei Beobachterfragmenten nicht der Fall, da der Zustand und die Änderungskategorien für alle MBS-Varianten gleich sind.

Für einfache Nutzbarkeit (Anforderung 4) werden Aktionen direkt an die Visualisierung gebunden – analog zur Verfeinerungsansicht (Kapitel 10.2.1). Im Rahmen der prototypischen Umsetzung (Kapitel 14) wurden das Editieren des Codes des jeweiligen Fragmentes und das Editieren der Stelle, an welcher das Fragment registriert wird, angebunden. Dies trägt zur besseren Integration der verschiedenen Werkzeuge bei (Unteranforderung 4.3).

Das Konzept bietet eine *Sicht auf einen wichtigen Punkt der Architektur der MBS*. Es erlaubt, zu analysieren, in welcher Verfeinerung welche Verhaltens- und Beobachterfragmente aktiv sind. Einfügen, Vererbung, Löschen und Überschreiben von Fragmenten werden für den Entwickler transparent.

10.3 Zusammenfassung

Das Kapitel 10 stellte interaktive (Kapitel 10.1) und explorative (Kapitel 10.2) Unterstützungskonzepte vor. Die interaktiven Unterstützungskonzepte erlauben Modifikationen der MBS, wohingegen die explorativen Unterstützungskonzepte die Struktur und die Zusammenhänge der MBS dem Entwickler gegenüber transparent machen.

Als zentrales interaktives Unterstützungskonzept wurde der Modellinterpreter (Kapitel 10.1.1) eingeführt. Er unterstützt die direkte Interpretation der UUI-Modelle und ist leicht erweiterbar (Anforderung 2). Der Modellinterpreter

wird durch ein Editierkonzept (Kapitel 10.1.2.1) zum WYSIWYG artigen Editor (Unteranforderung 4.2) erweitert. Dieser erlaubt es dem Entwickler, die Benutzerschnittstelle manuell im Detail anzupassen (Unteranforderung 3.1) und diese Änderungen an der Benutzerschnittstelle direkt zu evaluieren (Unteranforderung 4.4).

Das Konzept der modularen Adaptionen (Kapitel 10.1.3) baut ebenfalls auf dem Interpreter auf. Es stellt einen Rahmen zur Verfügung, über den einzelne Bausteine zur Unterstützung von Anpassungstätigkeiten für den Entwickler modular zur einfachen Nutzung (Anforderung 4) bereitgestellt werden können. Abschließend wurden Verfeinerungsbaum basierte Konzepte (Anforderung 1, Anforderung 6), u.a. zur Propagation von Modifikationen, beschrieben (Kapitel 10.1.4).

Das zentrale explorative Unterstützungskonzept ist die Verfeinerungsansicht (Kapitel 10.2.1), welche nicht nur den Verfeinerungsbaum darstellt, sondern auch für die Untersuchung der Verfeinerungsbeziehungen einzelner Interaktionsobjekte herangezogen werden kann (Unteranforderung 4.1). Die Schnittstellen in der Architektur zwischen Interaktionsobjekten und Verhaltensfragmenten sowie zwischen Zustand und Beobachterfragmenten (Unteranforderung 5.2) werden durch die Visualisierung des aktiven Sets von Fragmenten (Kapitel 10.2.2) transparent.

Alle Unterstützungskonzepte sind stark integriert und basieren auf dem in Kapitel 8 vorgestellten Architekturmuster. Sie benötigen gemeinsamen Zugriff auf das Modell in welchem die Struktur der MBS-Varianten beschrieben ist. Hierdurch stehen Änderungen in allen Unterstützungskonzepten sofort zur Verfügung (Unteranforderung 4.3). Dies ermöglicht dem Entwickler, die Änderungen direkt umfassend zu evaluieren (Unteranforderung 4.1 und Unteranforderung 4.4).

Ein Abgleich der Unterstützungskonzepte mit den Anforderungen findet (zusammen mit den anderen Konzepten dieser Arbeit) in Kapitel 12 statt. Die Umsetzung und Evaluation sind dagegen in Teil III beschrieben.

Kapitel 11

Architektonische Integration

Dieses Kapitel baut auf dem Lösungskonzept, welches in Kapitel 7 vorgestellt wurde und in den Kapiteln 8 zum Architekturmuster, 9 zur domänenspezifischen Sprache (DSL) sowie 10 zu Unterstützungskonzepten ausgeführt wurde, auf. Es beschreibt eine Architektur, welche die bisher vorgestellten Konzepte integriert und für die Entwicklung, aber auch die Laufzeitunterstützung von Multi-Benutzerschnittstellen (MBS) genutzt werden kann. Die Architektur ist abstrakt gehalten und generisch genug, um für verschiedene Plattformen und Szenarien einsetzbar zu sein.

Die Gliederung des Kapitels folgt der Gliederung der Architektur. Sie ist, wie in Abbildung 11.1 dargestellt, in drei große Blöcke geteilt, welche wiederum aus Unterkomponenten oder ganzen Frameworks bestehen können:

Infrastrukturknoten: Der Infrastrukturknoten (Kapitel 11.1) stellt grundlegende Dienste für die MBS und deren Ausführung bereit. Da im Rahmen der Unterstützungskonzepte nicht streng zwischen Entwicklungs- und Laufzeit unterschieden wird, bildet er auch die Basis für die Entwicklungsumgebung.

Entwicklungsumgebung: Die Entwicklungsumgebung (Kapitel 11.3) setzt auf dem Infrastrukturknoten auf und beinhaltet die interaktiven sowie explorativen Unterstützungskonzepte (aus Kapitel 10)).

MBS: Die Multi-Benutzerschnittstelle (Kapitel 11.2) selbst ist das für Infrastruktur und Entwicklungsumgebung zentrale Artefakt. In diesem Block liegen die verschiedenen Bestandteilen der eigentlichen MBS. Sie basiert auf dem in Kapitel 8 vorgestellten Architekturmuster.

Abschließend wird in Kapitel 11.4 die Erweiterbarkeit im Kontext der Gesamtarchitektur beschrieben, da sie quer zu den verschiedenen anderen Aspekten liegt.

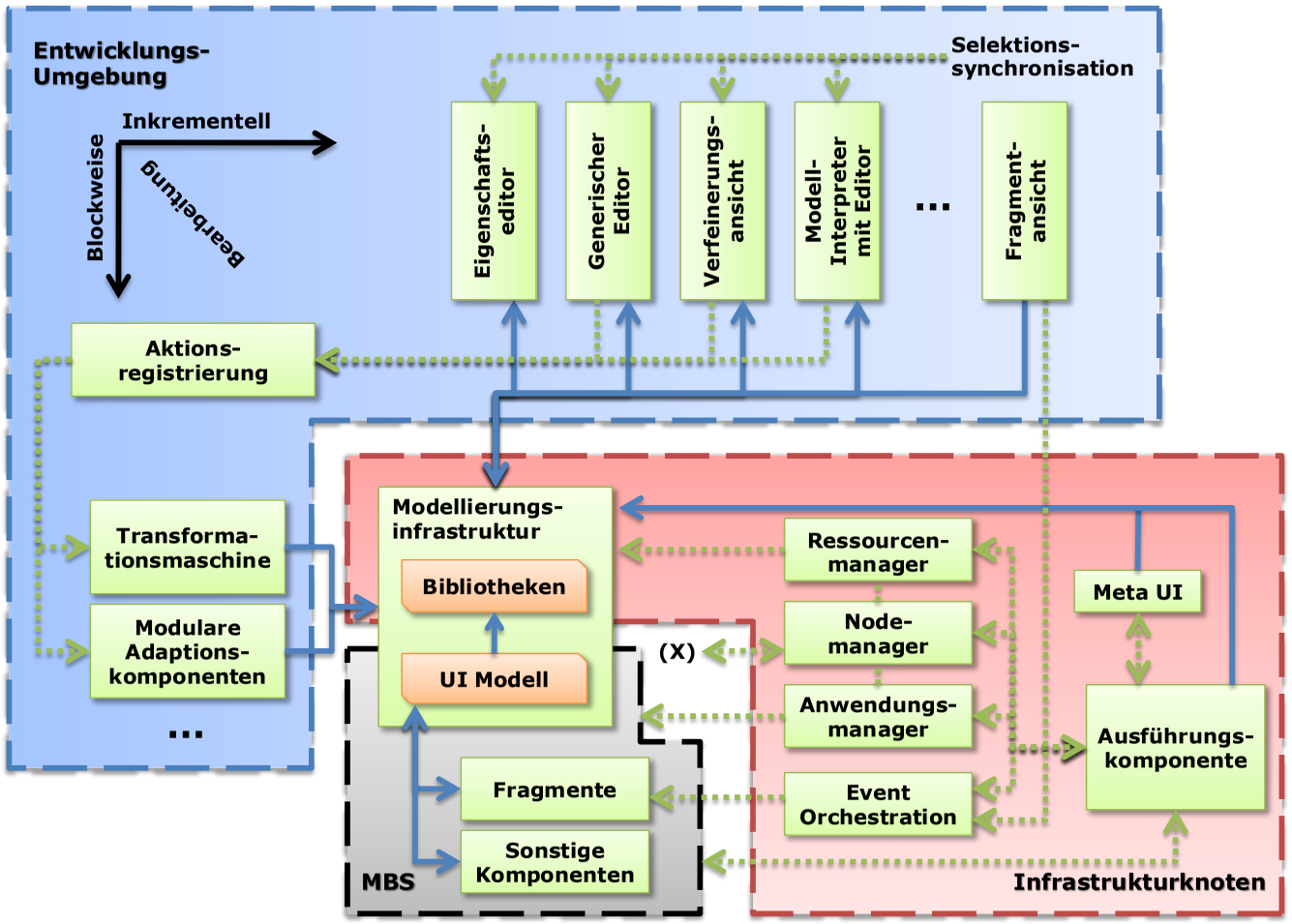


Abbildung 11.1: Überblick über die Architektur zum vorgestellten MBS Konzept. Blöcke, welche durch eine langgestrichelte Linie eingegrenzt sind, bilden größere Einheiten in der Architektur: Die Unterstützung in der Entwicklungsumgebung (links oben, blau), der Infrastrukturknoten (rechts unten, rot) und die MBS (mitte unten, grau). Durchgezogene (blaue) Pfeile sind Modellzugriffe, sind sie bidirektional, auch mit Notifizierungen über Modelländerungen. Gestrichelte Pfeile (grün) sind Zugriffe von Komponenten aufeinander. Das (x) symbolisiert den Zugriff vieler Komponenten auf den Nodemanager, da er der zentrale Einstiegspunkt zur Infrastruktur ist.

11.1 Infrastrukturknoten

Der Infrastrukturknoten stellt grundlegende Dienste bereit, welche sowohl zur Entwicklungs- als auch zur Laufzeit gebraucht werden. In die Infrastruktur werden Funktionen herausfaktorisiert, welche über verschiedene MBS gleich sind und somit ein hoher Wiederverwendungsgrad gegeben ist. Darüber wird die Strategie verfolgt, Funktionen und Daten zu standardisieren, wenn darauf wieder andere Werkzeuge aufsetzen. So profitiert nicht nur die MBS von einer Komponente zur Verwaltung der Modellressourcen, sondern auch Werkzeuge, welche den Entwickler mit diesen arbeiten lassen, was wiederum die Konsistenz und Vorhersagbarkeit (Anforderung 4) erhöht.

11.1.1 Modellierung und Modell

Um Inkonsistenzen zwischen Werkzeugen zu vermeiden und eine einheitliche Sicht auf die MBS zu gewährleisten stellt der Infrastrukturknoten als zentrales Element die `Modellierungsinfrastruktur` bereit. Alle Zugriffe auf das UUI-Modell finden über sie statt. Sie ist der zentrale Speicher für Modelle sowie Bibliotheken und unterstützt beim Laden und Speichern dieser. Einfaches Lesen von und Schreiben in Modelle ist wichtig für das Design der Umsetzung, ebenso wie die Möglichkeit sich auf Änderungen in Modellen zu subskribieren, sodass Änderungen eines Werkzeuges direkt in anderen Werkzeugen sichtbar gemacht werden können und somit ein konsistentes Bild entsteht.

11.1.2 Verwaltungsaufgaben

Die Aufgabe der Verwaltung der verschiedenen Ressourcen und Daten im Umfeld übernehmen Manager. Sie sorgen für Standardisierung und erhöhen somit die Wiederverwendbarkeit.

Der `Nodemanager` bildet den zentralen Einsprungpunkt in die Infrastruktur. Das (x) in Abbildung 11.1 symbolisiert diesen Zugang vieler Komponenten zum Nodemanager. Er startet beim Hochfahren alle anderen benötigten Dienste und bietet eine Schnittstelle, um Plugins zur Erweiterung der Infrastruktur zu registrieren. So werden alle Interpreter als Plugins eingebunden. Dies ist konsistent mit Unteranforderung 2.1, dass keine Sprachen hart einkodiert werden.

Der `Ressourcenmanager` unterstützt die Modellierungsinfrastruktur und verwaltet hierfür die Modelle. MBS müssen sich somit nicht selbst darum kümmern und es können generische Werkzeuge zur Arbeit mit Modellressourcen konzipiert werden. Der Ressourcenmanager initiiert das Laden von Bibliotheken und stellt einheitliche URIs bereit, sowie die Möglichkeit, diese aufzulösen (wichtig z.B. für Bibliothekszugriffe, vgl. Kapitel 11.4).

Der `Anwendungsmanager` schließlich standardisiert und kapselt den Umgang mit Multi-Benutzerschnittstellen. Durch ihn steht eine einheitliche Schnittstelle zum Starten, Unterbrechen und Stoppen, sowie Laden und Entladen von MBS

(inklusive Modell und Fragmenten) zur Verfügung.

11.1.3 Vom Modell zur Interaktion

Eine zentrale Aufgabe der Infrastruktur ist es, mit Hilfe der Interpreter-Plugins die MBS zur Interaktion zu bringen; Sowohl zur Entwicklungs- als auch zur Laufzeit. Hierfür müssen insbesondere die Fragmente einer MBS organisiert und der Prozess der zur-Interaktion-Bringung organisiert werden.

Die `Event Orchestration` kapselt Funktionalität zur Berechnung des aktiven Sets von (Verhaltens- bzw. Beobachter-) Fragmenten in einer gegebenen MBS-Variante. Diese Berechnung muss für verschiedene MBS identisch und reproduzierbar ablaufen, um Vorhersagbarkeit (Anforderung 4) gegenüber dem Entwickler zu gewährleisten. In diesem Rahmen behandelt die Orchestration sowohl Interaktions- als auch Änderungsevents. Dabei ließt sie (kompilierte und lauffähige) Verhaltens- und Beobachterfragmente aus der MBS aus. Diese initialisiert sie und analysiert sie im Bezug auf Löschen, Überschreiben und neu Hinzufügen (vgl. Kapitel 8.2). Letztendlich führt sie auf Nachfrage der Ausführungskomponente die Fragmente auch aus.

Die `Ausführungskomponente` orchestriert die gesamte Ausführung der MBS. Mit Hilfe der Event Orchestration werden die Fragmente der MBS zur Ausführung vorbereitet. Weiter werden passende Interpreter (vgl. Kapitel 10.1.1) gefunden und mit diesen die gewählte MBS-Variante zur Interaktion gebracht. Interaktionsevents vom Interpreter werden von der Ausführungskomponente verarbeitet und mit Hilfe der Event Orchestration die passenden Verhaltensfragmente zur Verarbeitung aktiviert. Ebenso werden die passenden Beobachterfragmente auf Grund von Änderungsevents aktiviert.

Der Infrastrukturknoten stellt auch eine Möglichkeit bereit, die MBS-Variante zu wählen, welche zur Interaktion gebracht werden soll. Dies geschieht mit Hilfe eines `Meta UI`, wie auch in vielen anderen Ansätzen (Coutaz 2006). Die Wahl kann aber auch automatisch erfolgen. Hierfür wird durch die MBS die zum Nutzungskontext passende Variante identifiziert und der Variantenwechsel initiiert. Je nach Schwere der Änderung sollte der Benutzer dabei jedoch befragt werden, ob er wirklich die Variante wechseln möchte. Darüber hinaus sind solche automatischen Wechsel nicht trivial: Eigenschaften wie der soziale Kontext oder der Grad der Unterbrechbarkeit sind schwer maschinell zu bestimmen. Generell ist die Wahl der MBS-Variante stark anwendungsspezifisch getrieben, wie auch das Aufbauen des Verfeinerungsbaumes.

11.2 Multi-Benutzerschnittstelle

Die Bestandteile der Multi-Benutzerschnittstelle (MBS) wurden in Kapitel 8 zum Architekturmuster beschrieben. Wichtig vor allem sind die Fragmente und das Modell mit den verschiedenen modellierten MBS-Varianten. Die Multi-Be-

nutzerschnittstelle ist das Artefakt, welches durch die Tätigkeit der Entwickler entsteht und als Entwicklungsprodukt vorliegt.

11.3 Entwicklungsumgebung

Die Entwicklungsumgebung beherbergt die im Kapiteln 10 vorgestellten Unterstützungskonzepte. Sie bedient sich dabei des Infrastrukturknotens, vorgestellt in Kapitel 11.1, welcher die grundlegenden Funktionen zur Modellverwaltung und Interpretation von Modellen bereitstellt. Darüber hinaus weisen die verschiedenen Konzepte eine hohe Diversität auf. So stellt ein Eigenschaftseditor Eigenschaften und deren Werte in einer Tabelle dar, wohingegen eine Verfeinerungsansicht einen Baum der UUIBoxen in einer Grafik zeigt.

Daher sind die einzelnen Unterstützungskonzepte als Plugins realisiert. Sie können über einen für die jeweilige Entwicklungsumgebung passenden Mechanismus (bei Java Eclipse z.B. über OSGI) bereitgestellt werden. Es ist hier zu unterscheiden zwischen Plugins für den Infrastrukturknoten und Plugins für die Entwicklungsumgebung. Letztere haben eher integrativen Charakter und sind sehr speziell auf die jeweilige Entwicklungsumgebung zugeschnitten (d.h. Visualisierungen, Bereitstellung von Kontextmenü Einträgen für Modifikationen etc.) – sie können also nur dort genutzt werden. Essentiell unterschiedlich ist auch der Ladevorgang: werden Plugins der Infrastruktur beim Laden des Infrastrukturknotens von diesem gestartet, so werden Plugins für die Entwicklungsumgebung meist (so z.B. auch bei Eclipse) von der Entwicklungsumgebung geladen und registrieren sich erst im Nachhinein beim Ressourcenmanager des Infrastrukturknotens.

Dennoch ist es ein wichtiges Ziel der Entwicklungsumgebung, die Entwicklung von Struktur und Verhalten, das Modellieren und Programmieren, eng zu integrieren, wie in Anforderung 5 inkl. Unteranforderung 5.2 gefordert. Die Werkzeugkette ist hochintegriert zu gestalten (Unteranforderung 4.3). Daher wird unter den Editoren das aktuell bearbeitete (selektierte) Modellierungselement synchronisiert, wie in Abbildung 11.1 illustriert. Ferner ist auch das Bereitstellen von Aktionen (z.B. “Umbenennung des Interaktionsobjektes”) in allen Editoren aus zentraler Stelle (`Aktionsregistrierung`) möglich, sodass der Entwickler diese aus der jeweiligen Situation direkt aufrufen kann (Unteranforderung 4.4).

Abschließend ist zu bemerken, dass *zwei Arten der Bearbeitung* von Modellen unterschieden werden können, wie in Abbildung 11.1 illustriert. Dies beeinflusst, in wie weit ein Plugin an der Synchronisation der Änderungen über Observer auf dem Modell (vgl. Abschnitt 11.1) teilnimmt.

Inkrementelle (synchrone) Bearbeitung: Dies sind Modifikationen auf sehr kleiner Granularität, z.B. das Ändern eines einzelnen Eigenschaftswertes in der Eigenschaftsansicht. Daher können die Plugins auch sehr eng

verzahnt eingesetzt werden. Dafür subskribieren sich die jeweiligen Komponenten auf Modelländerungen, und führen diese in ihrer Darstellung sofort nach. Alle Komponenten arbeiten damit ständig synchron und Inkonsistenzen werden vermieden.

Blockweise (dedizierte) Bearbeitung: Blockweise Modifikationen betreffen größere Gruppen von Modellelementen welche zusammen durchgeführt werden. Die Modifikationen können aus mehreren Schritten bestehen, wie beispielsweise beim Anpassen einer UI mit Hilfe des Skalierungswerkzeuges (siehe Kapitel 14.3.3) oder eine Modelltransformation. Dementsprechend werden Plugins zur blockweisen Bearbeitung nicht synchron mit anderen Komponenten betrieben sondern explizit aufgerufen.

11.4 Erweiterung mit neuen Bibliotheken

Dies Kapitel beschreibt die Erweiterung des vorgestellten Ansatzes mit neuen Sprachkonstrukten gemäß Anforderung 2. Es basiert auf der DSL, welche in Kapitel 9 eingeführt wurde und kombiniert diese mit dem Interpreteransatz aus Kapitel 10.1.1. Im Folgenden Abschnitt 11.4.1 wir auf die konzeptionelle Grundlage eingegangen. Darauf folgt eine Beschreibung des gewählten Ansatzes zur Integration in die Architektur.

11.4.1 Erweiterungskonzept

Im Rahmen der DSL wird das Konzept von den Interaktionsobjektklassifizierern unterstützt (vgl. Kapitel 9.2.4), welche die Elemente eines Benutzerschnittstellenmodells (Interaktionsobjekte) klassifizieren. Für eine konsistente Modellierung und um sicherzustellen, dass die Modelle zur Interaktion gebracht werden können, werden Sprachkonstrukte in Sprachen gruppiert. Konstrukte einer Sprache können im Rahmen der Modellierung gemischt werden (wie in Wohlgeformtheitsregel zur Bibliothekskonsistenz in Abschnitt 9.3.3 formuliert). Dies ist am Beispiel von UI Toolkits gut zu verdeutlichen: Java Swing Elemente lassen sich mischen, jedoch nicht beliebig mit HTML Elementen kombinieren.

Es lassen sich somit **zwei Arten der Erweiterung** differenzieren:

Sprache Zum Einen die Erweiterung mit einer komplett neuen Sprache (z.B. einem neuen UI Toolkit), welche nicht auf bereits bestehenden Sprachkonstrukten aufbaut. Dabei wird ggf. auch ein neuer Interpreter für die neue Sprache erstellt.

Die Erweiterung des Ansatzes mit einer neuen Sprache erlaubt es, *beliebige neue Abstraktionsebenen und Formen der Abstraktion* (Anforderung 1) zu integrieren. Die Sprache kann somit auch eine existierendes Toolkit einer generischen Programmiersprache im Rahmen der Modellierung bereitstellen (z.B. Java Swing oder AWT, vgl. Kapitel 7.3.1). Es ist aber

auch möglich, eine neue Sprache einzubinden, ohne einen passenden Interpreter bereitzustellen. In diesem Fall wird die Sprache als reine Zwischenabstraktion auf dem Weg zu einem interpretierbaren UUI-Modell genutzt (z.B. als Dialogmodell).

Bibliothek Zum Anderen die Erweiterung einer bestehenden Sprache um spezialisierte Sprachkonstrukte (z.B. ein spezielles JPanel für die Zuordnung einer Meldung), welche auf vorhandenen Sprachkonstrukten aufbauen. Sie können in schon bestehenden Interpretern der Sprache zur Interaktion gebracht werden.
Die Erweiterung des Ansatzes mit einer neuen Bibliothek für eine bestehende Sprache ermöglicht es, *neue Komponenten und insbesondere Anwendungsfall spezifische Spezialkomponenten in die Modellierung zu integrieren.* Dies ist insbesondere für UI Toolkits wichtig, wenn eigene UI Komponenten (z.B. eine Komponente zur Zuordnung von Meldungskategorien) entwickelt oder gekaufte UI Komponenten mit eingebunden werden sollen.

Zentral bei der Erweiterung ist die **Umsetzung in den Interpretern**. Zur einfachen Erweiterbarkeit ist es wünschenswert, dass bestehende Interpreter mit neuen Sprachkonstrukten genutzt werden können, wenn diese der selben Sprache angehören. So sollte z.B. eine spezielles Java Swing Kombinationsfeld zur Datumseingabe auch durch einen bestehenden Java Swing Interpreter zur Interaktion gebracht werden können. Interpreter nutzen daher idealer Weise *Reflexionsadapter* zur Instanziierung der UI Komponenten (vorgestellt in Kapitel 10.1.1.2). Der hierdurch indirekte Zugriff auf die UI Komponente erlaubt, dass (mit Hilfe von Classloadern, vgl. Kapitel 13.3.1) leicht neue Elemente mit einbezogen werden können.

11.4.1.1 Abwägungen zur Erweiterung und Erweiterbarkeit

Die Erweiterung mit einem neuen Toolkit und Interpreter ist ein deutlich umfangreicheres Unterfangen als die Erweiterung eines bestehenden Toolkits mit einer neuen Bibliothek. Letztere ist besonderes zügig durchführbar, wenn bei der initialen Erstellung des Interpreters für das Toolkit bereits auf Erweiterbarkeit Wert gelegt wurde, indem Reflexionsadapter bereitgestellt werden. Dieser Weg wurde auch im laufenden Beispiel eingeschlagen: der Swing Interpreter wurde mit speziellen Komponenten für die SoKNOS Meldungsverarbeitung erweitert (Kapitel 13.3.1).

Die vorgestellten Architekturkonzepte unterstützen kurze Entwicklungszyklen (Unteranforderung 4.4) – nicht nur für Benutzerschnittstellen, sondern auch für die Erweiterung des Ansatzes. Dazu trägt insbesondere die Möglichkeit bei, Modellinterpretern Komponentenbibliotheken zu übergeben (vgl. Kapitel 10.1.1), sodass parallel zur Modellierung auch Modifikationen an den Bibliotheken möglich sind. Ein solch integriertes Modellieren mit und Erstellen von

neuen Benutzerschnittstellenkomponenten ist auch hilfreich, um neue Elemente zur Erweiterung eines bestehenden Toolkits zu debuggen.

11.4.2 Erweiterbarkeit im Kontext der Gesamtarchitektur

Im Folgenden wird auf die drei wichtigen Aspekte bei der Erweiterbarkeit im Kontext der Gesamtarchitektur eingegangen: Die Referenzierung der Bibliothekselemente bildet die Schnittstelle zwischen Erweiterung und Modellierung. Je nach Form der Erweiterung (mit neuen Sprachkonstrukten oder einer neuen Sprache) findet die Erweiterung dann unterschiedlich statt.

11.4.2.1 Referenzierung von Bibliothekselementen

Bibliotheken von Modellelementen (Bibliotheksmodelle) werden separat von UUI-Modellen gelagert, damit sie in mehreren Modellen genutzt (referenziert) werden können. Daher ist die Referenz auf Interaktionsobjektklassifizierer und Eigenschaften auch unidirektional vom UUI-Modell zur Bibliothek gestaltet, wie in Abschnitt 9.2.4 beschrieben. Das heißt, dass von den Bibliothekselementen aus, keine Referenz in die UUI-Modelle vorhanden ist.

Da die MBS, die Laufzeitumgebung für MBS und die Bibliotheken in verschiedenen Verzeichnissen installiert sein können, aber von allen MBS immer die gleiche Bibliothek für jeweils eine Sprache referenziert werden soll, muss die *Referenzierung von Bibliothekselementen* unabhängig vom Dateisystem sein (z.B. konkreten Dateinamen und Pfaden). Sie kann aber die Verschachtelung der Namensraumstruktur (vgl. Abschnitt 9.2.2) der DSL nutzen. Viele Modellierungswerkzeuge nutzen ein XML basiertes Speicherformat, was die Nutzung von speziellen URIs ermöglicht, welche vom konkreten Pfad abstrahieren. Daher werden Modellelemente in Bibliotheken über das *URI Schema mapachlib://vollqualifizierter Elementname* referenziert. Zur Design- und Laufzeit werden beim Start alle im jeweiligen System verfügbaren Bibliotheken geladen (durch die Infrastruktur, vgl. Kapitel 13.2) und zur Auflösung des mapachelib Referenzschemas zur Verfügung gestellt.

11.4.2.2 Erweiterung durch eine neue Sprache

Zur Erweiterung des Ansatzes mit einer neuen Sprache wird **ein Bibliotheksmodell definiert**. Innerhalb dieses werden die Interaktionsobjektklassifizierer der Sprache sowie deren Eigenschaften modelliert. Zur besseren Strukturierung kann die Verschachtelung über Namensräume genutzt werden.

Anschließend wird, wenn gewünscht, ein Interpreter erstellt, welcher Modelle, die auf der Bibliothek basieren, **zur Interaktion bringen** kann. Der Interpreter (vgl. Kapitel 10.1.1) enthält insbesondere die Instanziierungskomponente, welche die Interpretation ausführt und die Komponenten der zur Interaktion gebrachten Benutzerschnittstelle erzeugt. Auch enthält der Interpreter die passenden Adapter, welche diese Komponenten mit dem Modell synchron

halten. Die Adapter sind dabei idealer Weise vom Typ Reflexionsadapter (vgl. Kapitel 10.1.1.2), damit der Interpreter leicht erweiterbar (mit weiteren Bibliotheken für die Sprache des Interpreters) ist.

Zur Modellierung kann ein generischer Editor genutzt werden, welche agnostisch bzgl. der Sprache ist (wie prototypisch umgesetzt und in Kapitel 14.3.2 beschrieben). Für bessere Nutzbarkeit (Anforderung 4) bietet sich jedoch ein speziell auf die Sprache abgestimmter Editor an, welcher als erweiterter Interpreter umgesetzt ist, wie in Kapitel 10.1.2 beschrieben, und adäquate Editierkonzepte für die Interaktionsform der Sprache bereitstellt.

11.4.2.3 Erweiterung einer bestehenden Sprache mit neuer Bibliothek

Zur Erweiterung einer bestehenden Sprache mit neuen Sprachkonstrukten, wird **ein bestehendes Bibliotheksmodell erweitert oder ein neues Bibliotheksmodell erstellt**. Hierbei sind die Interaktionsobjektklassifizierer sowie deren Eigenschaften zu modellieren. Insbesondere können dabei Vererbungsbeziehungen (vgl. Abschnitt 9.2.3) zwischen Interaktionsobjektklassifizierern (auch zu schon bestehenden und nicht in der Erweiterung enthaltenen) genutzt werden.

Um die **neuen Elemente zur Interaktion zu bringen** wird die Instanziierungskomponenten des bestehenden Interpreters (vgl. Kapitel 10.1.1) mit den notwendigen Klassen erweitert und passende Adapter bereitgestellt. Wurde der Interpreter mit Reflexionsadaptern (vgl. Kapitel 10.1.1.2) umgesetzt, reicht es, die passenden Klassen der Instanziierungskomponente bereitzustellen, sie werden automatisch durch die Reflexionsadapter unterstützt (wie z.B. in Kapitel 13.3.1 beim Swing Interpreter für das Fallbeispiel geschehen).

Am Ende ist zu überprüfen, ob die gegebenen **Editiermöglichkeiten** mit den neuen Elementen kompatibel sind. Die entsprechenden Editierkonzepte in Toolkit spezifischen Interpretern müssend dazu überprüft werden. Generische Editoren, die agnostisch bzgl. der Sprache arbeiten, können ohne Veränderung weitergenutzt werden.

11.5 Zusammenfassung

Dies Kapitel stellte die architektonische Integration der Konzepte vor. Diese wurde in die drei Aspekte Infrastrukturknoten (Kapitel 11.1), Multi-Benutzerschnittstelle (Kapitel 11.2) und Entwicklungsumgebung (Kapitel 11.3) geteilt. Darüber hinaus wurde der quer liegende Aspekt der Erweiterbarkeit beleuchtet (Kapitel 11.4).

Die Architektur faktorisiert durch den **Infrastrukturknoten** die wichtigen Aspekte heraus, welche von verschiedenen MBS wiederverwendet werden bzw. sowohl von Entwicklungs- als auch von Laufzeit genutzt werden. Hierdurch wird auch eine *Standardisierung von Daten und Prozessen* erreicht, welche

für verbesserte Konsistenz und Vorhersagbarkeit (Anforderung 4) gegenüber dem Entwickler sorgt (im Gegensatz zur separaten Implementierung in jeder MBS). Auch erlaubt diese Standardisierung die Konzipierung weiterer, darauf aufbauender Unterstützungen, wie z.B. einem einheitlichen Zugriff für den Entwickler auf die Modellierungsressourcen oder das Starten und Stoppen von Multi-Benutzerschnittstellen. Auch die Unterstützungskonzepte, welche Transparenz (Unteranforderung 4.1) im Bezug auf die Einbindung der Fragmente herstellen (Kapitel 10.2.2) basieren auf dieser Standardisierung. Mit Letzteren kann beispielsweise der bekannten Kritik an MVC begegnet werden, dass die Nutzung des Beobachter-Musters (engl. Observer Pattern) Programme schwer lesbar macht.

Zentral bei allen Bestandteilen der Architektur ist, dass sie *das* ***synchrone Arbeiten*** *auf einem Modell von allen Komponenten aus* ermöglicht und somit für eine konsistentere Darstellung gegenüber dem Entwickler sorgt. Darüber hinaus erlaubt der synchrone Zugriff auf das Modell auch ein flexibleres Arbeiten (Unteranforderung 4.4), da zeitaufwändige Umschalt- und Aktualisierungsprozesse entfallen. Konzeptionell wird dies durch die Aufteilung von Bearbeitungszugriffen in blockweises Editieren und inkrementelles Editieren unterstützt. Somit können größere Änderungen dennoch effizient durchgeführt werden bzw. externe Werkzeuge (wie z.B. eine Modelltransformationsmaschine), welche nur blockweise Modifikationen durchführen können, erst eingebunden werden.

Im Bezug auf die Erweiterbarkeit (Anforderung 2) ist die Architektur konsistent zur DSL (Kapitel 9) gestaltet. Es ist kein Interpreter fest integriert, vielmehr werden alle Interpreter als Plugins eingebunden. Erst hierdurch ist die Erweiterbarkeit im Sinne der Anforderung adressiert.

Ein Abgleich der Architektur mit den Anforderungen findet (zusammen mit den anderen Konzepten) im folgenden Kapitel 12 statt. Die Umsetzung und Evaluation sind dagegen in Teil III beschrieben.

Kapitel 12

Zusammenfassung, Implikationen und Abgleich mit den Anforderungen

Teil II der vorliegenden Arbeit, welcher das entwickelte Lösungskonzept für Multi-Benutzerschnittstellen und deren Entwicklung vorstellte, wird mit diesem Kapitel abgeschlossen. Der folgende Teil III geht konkret auf die prototypische Umsetzung der Konzepte sowie deren Evaluation im Rahmen einer Fallstudie und einer Nutzerstudie ein.

Dies Kapitel besteht aus drei Teilen. Nach einer kurzen inhaltlichen Zusammenfassung des Teils wird das entwickelte Lösungskonzept in Kapitel 12.1 mit den Anforderungen aus Kapitel 5 abgeglichen. Schließlich werden in Kapitel 12.2 methodische Implikationen des Lösungskonzeptes diskutiert.

Begonnen wurde Teil II mit einem **Überblick über das Lösungskonzept** in Kapitel 7. Anschließend wurde das grundlegende **Architekturmuster** für MBS in Kapitel 8 vorgestellt. Es basiert auf MVC, führt jedoch eine stärkere Modularisierung ein, sowie die Möglichkeit zur Vererbung von Verhalten. Das Architekturmuster besteht im Wesentlichen aus einem Modell, welches die verschiedenen Varianten der MBS beinhaltet, sowie Verhaltens- und Beobachterfragmente, welche das Verhalten der Benutzerschnittstelle programmatisch beschreiben und dem Zustand der MBS, welcher für alle MBS-Varianten gleich bleibt.

Die im Anschluss an das Muster eingeführte **DSL** (Kapitel 9) erlaubt das Modellieren des im Architekturmuster referenzierten Benutzerschnittstellenmodells (UUI-Modell). Dafür wurden drei Aspekte einer DSL eingeführt und festgelegt: Semantik, abstrakte Syntax und Wohlgeformtheitsregeln.

Der für eine vollständige DSL fehlende vierte Teil, die konkrete Syntax, wird durch **interaktive und explorative Unterstützungskonzepte** bestimmt. Diese Unterstützungskonzepte wurden im Kapitel 10 eingeführt und beschrieben. Hierbei wurde das zentrale Konzept des Interpreters mit dem dar-

auf aufbauenden Editor vorgestellt. Weitere interaktive Konzepte ermöglichen die Modifikation ein oder mehrerer MBS-Varianten. Dagegen machen explorative Konzepte, wie die Verfeinerungsansicht oder die Visualisierung aktiver Fragmente, den Entwicklungsstand der MBS dem Entwickler transparent.

Im letzten Kapitel 11 wurde die **architektonische Integration** der entwickelten Konzepte in eine Gesamtarchitektur beschrieben. Basierend auf dem Architekturmuster wurden drei Architekturblöcke identifiziert und konzipiert: der Infrastrukturknoten, die Entwicklungsumgebung und die Multi-Benutzerschnittstelle. Die Erweiterbarkeit des Ansatz, durch welche auch Anwendungsfall spezifische Komponenten genutzt werden können, wurde als quer liegender Aspekt abschließend behandelt.

12.1 Abgleich mit den Anforderungen

Die in Kapitel 5 erhobenen Anforderungen bildeten die Grundlage für die Entwicklung der in diesem Teil der Arbeit vorgestellten Konzepte. Im Folgenden wird überprüft, inwieweit die entwickelten Konzepte die erhobenen Anforderungen auch umsetzen. Diese Überprüfung basiert rein auf qualitativen Betrachtungen der Konzepte. Ein Abgleich näher an der Praxis wird im Rahmen der Fallstudie in Kapitel 16.4 besprochen.

In den Tabellen zum Abgleich werden die folgenden **Abkürzungen und Symbole** genutzt: Ges. – Gesamtbewertung, AM – Architekturmuster (Kapitel 8), DSL – Domänenspezifische Sprache (Kapitel 9), USK – Unterstützungskonzepte (Kapitel 10) und AI – Architektonische Integration (Kapitel 11). Dabei bedeutet "✓", dass eine Unteranforderung erfüllt wird bzw. ein Konzept (AM, DSL, USK, AI) die Anforderung bzw. Unteranforderung umsetzt. Eine teilweise Umsetzung wird durch "(✓)" gekennzeichnet, das Fehlen einer Unterstützung, wo sie eigentlich notwendig wäre, wird durch ein "✗" gezeigt. Die Gesamtbewertung der Anforderung (nicht Unteranforderung) schließlich nutzt die gleiche Nomenklatur, wie die Bewertung der verwandten Arbeiten in Kapitel 6, insbesondere Tabelle 6.1: Eine Skala von G bis A beschreibt die möglichen Erfüllungsgrade, welche sich je nach Anforderung unterscheiden können, wobei A den höchsten und G den geringsten Erfüllungsgrad darstellen.

12.1.1 Anforderung 1

Das Architekturmuster bietet mit den Fragmenten (Kapitel 8.2 und 8.4) die Möglichkeit, beliebige Abstraktionsebenen und eine Fall spezifische Abstraktionsnatur zu verwirklichen. Die DSL verwirklicht dabei beides mit Hilfe der UUIBoxen und den Verfeinerungsbeziehungen (Kapitel 9.1.3). Daneben wurden Unterstützungskonzepte vorgestellt, welche den Verfeinerungsbaum unabhängig von Anzahl der Abstraktionsebenen und Natur der Abstraktion darstellen. Mit ihnen können auch die Beziehungen zwischen Elementen untersucht wer-

den (Kapitel 10.2.1), sowie die Fragmente mit ihrer Zuordnung zu beliebigen UUIBoxen (Nutzungskontexten) dargestellt werden (Kapitel 10.2.2).

Anforderung	**Bewertung**				
	Ges.	**AM**	**DSL**	**USK**	**AI**
Anforderung 1: Abstraktion	*A*	✓	✓	(✓)	
Unteranforderung 1.1: Fall spezifische Ebenen	✓	✓	✓	(✓)	
Unteranforderung 1.2: Fall spezifische Abstraktionsnatur	✓	✓	✓	(✓)	

12.1.2 Anforderung 2

Die Erweiterbarkeit wird in der DSL maßgeblich durch das Klassifizierungskonzept für Interaktionsobjekte (Kapitel 9.1.2) umgesetzt. Die Unterstützungskonzepte – Interpreter (Kapitel 10.1.1) und Interpreter-basierte Editoren (Kapitel 10.1.2) für neue Sprachen sowie Erweiterungen von bereits bestehenden Sprachen – sind auf Erweiterbarkeit ausgelegt. Hierfür stellt die architektonische Integration ein Erweiterungskonzept bereit (Kapitel 11.4), welches den Einsatz von Reflexionsadaptern in Interpretern beinhaltet. Weiter wird in keinem der entwickelten Konzepte eine Sprache fest einkodiert.

Anforderung	**Bewertung**				
	Ges.	**AM**	**DSL**	**USK**	**AI**
Anforderung 2: Erweiterbarkeit	*A*		✓		✓
Unteranforderung 2.1: Keine Sprachenhardkodierung	✓		✓		✓

12.1.3 Anforderung 3

Das Architekturmuster lässt explizit die manuelle Bearbeitung jeder MBS-Variante im Detail zu: Fragmente (das Verhalten der MBS) können per MBS-Variante hinzugefügt, modifiziert oder entfernt (CRUD-Operationen) werden (Kapitel 8.2 und Kapitel 8.4). Dafür werden mit Hilfe der DSL (für die Struktur der MBS) alle möglichen Varianten ausmodelliert (Kapitel 7.1). Die DSL erlaubt auch die Einbindung beliebiger Eigenschaften der Interaktionsobjekte – je nach Klassifizierer, sodass der Ansatz nicht auf einen reduzierten, gemeinsamen Nenner von Sprachen beschränkt ist (Kapitel 9.1.2). Dies wiederum wird durch das Interpreter-basierte Editorkonzept (Kapitel 10.1.2) genutzt, um die detaillierte Anpassung via WYSIWYG zu ermöglichen.

Anforderung	Bewertung				
	Ges.	AM	DSL	USK	AI
Anforderung 3: Modellierungsdetailgrad	*A*	✓	✓	(✓)	
Unteranforderung 3.1: Varianten Modifizierbar	✓	✓	✓	(✓)	
Unteranforderung 3.2: Keine Detailbeschränkung	✓		✓	✓	

12.1.4 Anforderung 4

Der vorliegende Ansatz setzt explizit einen Fokus auf die einfache Nutzbarkeit durch den Entwickler. Die einfache Nutzbarkeit hängt dabei stark von den Unterstützungskonzepten ab, wird aber im Falle der Transparenz für den Entwickler und der Flexibilität durch das klare und strikte Architekturmuster sowie die architektonische Integration erst ermöglicht. Ohne die strikten Festlegungen des Architekturmusters und die Standardisierung von Daten und Prozessen im Rahmen der architektonischen Integration wäre die Konzeption einer so weitgehenden Unterstützung nicht möglich.

Explorative **Unterstützungskonzepte**, wie die Verfeinerungsansicht (Kapitel 10.2.1) und die Darstellung der Fragmente (Kapitel 10.2.2), nutzen die standardisierte Information zur Darstellung des Entwicklungszustandes der MBS. Daneben nutzen interaktive Unterstützungskonzepte die Standardisierung, um Modifikationen der MBS zu ermöglichen. So erlauben die Interpreter-basierten Editoren (Kapitel 10.1.2) z.B. das direkte (WYSIWYG) Editieren von MBS-Varianten. Durch das Konzept der architektonischen Integration können die Interpreter hierbei zur Lauf- und zur Entwicklungszeit eingesetzt werden (Kapitel 11.1, vgl. Abschnitt 12.2.2). Auch basieren die modularen Adaptionskonzepte auf ihnen, welche komplexere Adaptionsaufgaben nutzungsgerecht kapseln (Kapitel 10.1.3). Des Weiteren ermöglichen Verfeinerungsbaum basierte Konzepte (Kapitel 10.1.4) die einfache Anwendung von Modifikationen auf mehrere MBS-Varianten.

Die verschiedenen Unterstützungen sind in einer **Werkzeugkette**, welche im Rahmen der architektonischen Integration (Kapitel 11.3) konzipiert wurde, eng integriert. Diese Integration erlaubt auch das Bereitstellen von Funktionalitäten über Unterstützungskonzepte hinweg (über die “Aktionsregistrierung”). Flexibles Arbeiten wird hierdurch, und durch die Möglichkeit, eigene Komponenten leicht einzubinden und weiter zu entwickeln (Kapitel 11.4), unterstützt. Schließlich stellt die DSL Wohlgeformtheitsregeln zur Prüfung der statischen Semantik der Modellierungsartefakte bereit (Kapitel 9.3), wobei ein konkretes Überprüfungswerkzeug und seine Einbindung in die Architektur fehlt.

Anforderung	Bewertung				
	Ges.	AM	DSL	USK	AI
Anforderung 4: Einfache Nutzbarkeit	*A*	✓		✓	
Unteranforderung 4.1: Transparenz MBS Zustand	✓	✓		✓	✓
Unteranforderung 4.2: Direktes Editieren	✓			✓	✓
Unteranforderung 4.3: Integrierte Werkzeugkette	✓				✓
Unteranforderung 4.4: Flexibilität	✓	✓		✓	✓
Unteranforderung 4.5: Semantikprüfung	(✓)		✓	✗	✗

12.1.5 Anforderung 5

Die Integration von Struktur und Verhalten wird speziell vom Architekturmuster unterstützt. Es erlaubt in Kombination mit der DSL die gemeinsame Anpassung beider Aspekte in jeder MBS-Variante. Die Entwicklungsumgebung (Kapitel 11.3) erleichtert hierbei die Synchronisation und Verzahnung beider Aspekte durch die integrierte Werkzeugkette für Modellieren und Programmieren. Auch helfen die Code Vervollständigung (Kapitel 14) sowie explorative Sichten auf gemeinsame Aspekte und die Verknüpfung der Sicht mit Artefakten beider Arten (z.B. den Sprung aus der Verfeinerungsansicht in den Code oder das entsprechende Modell im Editor) dabei. Dies wird durch die in der architektonischen Integration konzipierte Vernetzung der Unterstützungskonzepte erreicht und baut auf dem Architekturmuster und der DSL auf.

Das Verhalten der Benutzerschnittstelle lässt sich im vorliegenden Ansatz komplett programmatisch spezifizieren. Dies wird durch die im Architekturmuster definierten Fragmente erreicht (Kapitel 8.2 und Kapitel 8.4, Beispiel in Kapitel 16.2). Schließlich stellen Architekturmuster und architektonische Integration ein klares Architekturmuster zur Verfügung, welches Standards im Bezug auf Daten und Abläufe definiert und wiederverwendbare Komponenten beschreibt, die diese bereitstellen.

Anforderung	Bewertung				
	Ges.	AM	DSL	USK	AI
Anforderung 5: Integration Struktur und Verhalten	*A*	✓	✓	(✓)	
Unteranforderung 5.1: Gemeinsame Anpassung	✓	✓	✓		
Unteranforderung 5.2: Synchronisation	✓	✓	✓	✓	✓
Unteranforderung 5.3: Programmierbares Verhalten	✓	✓			(✓)
Unteranforderung 5.4: Klares Architekturmuster	✓				✓

12.1.6 Anforderung 6

Die systematische Unterstützung von Modifikationen baut auf den Verfeinerungsbeziehungen der DSL (Kapitel 9.1.3) sowie dem Architekturmuster, welches das Hinzufügen, Entfernen und Modifizieren von Fragmenten in Verfeinerungen zulässt (Kapitel 7), auf. So erlauben Unterstützungskonzepte wie der Interpreter-basierte Editor (Kapitel 10.1.2) das Anbringen von Verfeinerungen, welche passiv propagiert werden. Die aktive Propagation dagegen wird durch Verfeinerungsbaum basierte Unterstützungskonzepte (Kapitel 10.1.4) umgesetzt. Die Kontrolle, wie die Propagation konkret aussieht wird vom Entwickler durch Modifikation der Verfeinerungsbeziehungen aus der DSL ausgeübt (Kapitel 9.1.5). Die passive Propagation unterliegt diesen inhärent, bei der aktiven Propagation werden sie von den Unterstützungskonzepten beachtet. Die architektonische Integration schließlich sorgt dafür, dass Änderungen allen Klienten der Modellierungsinfrastruktur bekannt gemacht und damit dem Entwickler transparent gemacht werden (Kapitel 11.1.1).

Anforderung	Bewertung				
	Ges.	AM	DSL	USK	AI
Anforderung 6: Modifikation	*A*	✓	✓	✓	(✓)
Unteranforderung 6.1: Modifikationskonzept	✓	✓	✓	✓	(✓)
Unteranforderung 6.2: Kontrolle Modifikationsverteilung	✓		✓	✓	(✓)

12.2 Methodische Implikationen

Die im Rahmen dieser Arbeit vorgestellten Konzepte haben auch Einfluss auf die Art, wie Benutzerschnittstellen entwickelt werden können. Die im Folgenden diskutierten Punkte sind keine wissenschaftlichen Ergebnisse der vorliegenden Arbeit, sondern vielmehr Implikationen aus ihr in Kombination mit anderen Arbeiten.

12.2.1 Automatische Erstellung von MBS-Varianten

Unter anderem Gummy (Meskens u. a. 2008) und das Cameleon Referenz Framework (Calvary u. a. 2003) nutzen die Möglichkeit, MBS-Varianten mit Hilfe von Automatismen vorab zu generieren. Designer verändern diese im Nachhinein und passen sie an den jeweiligen Nutzungskontext im Detail an.

Einfache Transformationen können z.B. nach einem festen Muster Elemente aufeinander abbilden und Mehrdeutigkeiten durch Rückfragen lösen. Dies wurde in Zusammenarbeit mit Zadmajid (2010) für die Abbildung von Swing auf eine Aufgabenklassifikation untersucht. Das Ergebnis wurde anschließend in die prototypische Umsetzung der vorliegenden Arbeit integriert. Mit Hilfe einer solchen Abbildung ist die Nutzung des vorliegenden Ansatzes in einem Aufgaben basierten Vorgehen möglich. Beispielsweise im Useware Prozess (Meixner 2010, Meixner und Görlich 2009) oder in den Arbeiten von Feuerstack (2008). Dabei setzt die vorliegende Arbeit nach dem Schritt der Lösung des Abbildungsproblems von Aufgaben- zu Dialogmodell an (vgl. Kapitel 4.2.1).

Komplexe Transformationen gehen weit über einfache Abbildungen hinaus. Die zeitgleich zur vorliegenden Arbeit von Petter durchgeführten Untersuchungen zu Constraint lösenden, optimierenden Modell zu Modell Transformationen basieren auf einer Spracherweiterung von QVT Relations (Petter, Behring und Mühlhäuser 2009, Petter, Zlatkov und Behring 2010, Petter u. a. 2008, Petter u. a. 2009, Thiel, Petter und Behring 2007). Der Ansatz von Petter macht es möglich, Constraints (Eigenschaften, welche erzeugte Modelle erfüllen *müssen*) und eine Zielfunktion zur Optimierung (durch Minimierung des Funktionswertes, auch genannte "Strategie") innerhalb der Modell zu Modell Transformation zu formulieren. Im Rahmen einer Transformation können somit quantitative Metriken der Nutzbarkeit für die Optimierung der Benutzerschnittstelle eingesetzt werden. In den Prototyp der vorliegenden Arbeit (Teil III) wurde hierzu der Prototyp der Arbeiten von Petter integriert, sodass Automatismen zur Erstellung von MBS-Varianten mit manueller Bearbeitung kombiniert werden können.

Diese **Kombination von automatischer und manueller Bearbeitung** erlaubt es, den Aufwand, welcher in eine MBS-Variante investiert wird, fein zu justieren. Hochqualitative MBS-Varianten für wichtige Nutzungskontexte werden von Hand erstellt bzw. angepasst. Nutzungskontexte mit geringeren Qualitätsanforderungen werden mit Hilfe von automatisch erstellten MBS-Varianten

abgedeckt. Die Wichtigkeit eines Nutzungskontexts kann dabei z.B. über betriebswirtschaftliche Kriterien festgelegt werden, beispielsweise auf Grund der Anzahl der Benutzer eines Nutzungskontextes oder ob ein Nutzungskontext für Schlüsselkunden relevant ist.

12.2.2 Fließender Übergang von Design- zu Laufzeit

Wie in Kapitel 7.3 beschrieben, war ein Punkt zur Wahl des Interpreteransatzes, die Möglichkeit, Design- und Laufzeit einer MBS stärker zu integrieren. Diese Integration halten unter anderem Calvary und Pinna (2008) für eine aktuelle Herausforderung.

Das Modellierungskonzept und die starke Modularisierung unterstützen einen fließenden Übergang zwischen Design- und Laufzeit. Durch die starke Modularisierung in Fragmenten (vgl. Abschnitt 8.5.1) wird die Austauschbarkeit der Fragmente – auch zur Laufzeit – vereinfacht. Die Modelle selbst sind auch zur Laufzeit verfügbar, und es werden nicht Code-Artefakte daraus generiert (vgl. Kapitel 3 zur Modellierung und 10.1.1 zu Interpretern). Somit *kann ein Modell mit dem vorliegenden Ansatz auch zur Laufzeit editiert werden.*

Die Modifikationen zur Laufzeit können zum Einen vom Benutzer selbst, durch Einsatz eines Editors basierend auf einem Modellinterpreter (vgl. Kapitel 10.1.1) vorgenommen werden. Zum Anderen aber, da alle Teile des vorliegenden Ansatzes auf das gleiche Modell zugreifen (vgl. dazu Architekturschaubild 11.1 im folgenden Kapitel) durch den Entwickler aus der Entwicklungsumgebung heraus. Auf Grund der Struktur der Modellinterpreter, werden Änderungen am Modell auch dem Benutzer der Laufzeit direkt angezeigt. Dies wurde testweise schon durchgeführt, aber da dies nicht im Fokus der Arbeit liegt, nicht weiter vertieft.

III

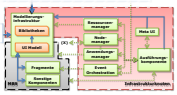
Realisierung und Evaluation

Überblick über Teil III

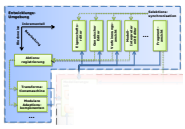
Dieser Teil stellt zuerst die prototypische Realisierung der Konzepte aus Teil II vor, gefolgt von der Evaluation mit einer zusammenfassenden Bewertung. Die Beschreibung des Anwendungsbeispiels SoKNOS und der dabei notwendigen Integration des Ansatzes in das Projekt bildet das Bindeglied.

Realisierung: Die *prototypische Realisierung* macht die entwickelten Konzepte (Teil II) erst nutzbar, sodass sie im Rahmen einer Fallstudie (Kapitel 16) oder einer Nutzerstudie (Kapitel 17) evaluiert werden können. Die Konzepte wurden hierfür in der Programmiersprache Java Version 1.6[1] umgesetzt, wobei die Entwicklungsumgebung Eclipse als Basis genutzt wurde. Das Eclipse Modeling Framework (EMF) kommt für die Verwaltung der UUI-Modelle zum Einsatz. Das Umsetzungsprojekt wurde "`Mapache`" (spanisch für Waschbär[2]) getauft.

Das Konzept der Architektur für Mapache wurde in Kapitel 11 entwickelt und ist in Abbildung 11.1, Seite 138 zu sehen. Gemäß dieser Architektur fand die Umsetzung in den folgenden zwei Teilen statt:



Umsetzung der Konzepte: Die Umsetzung der zentralen Konzepte wird in Kapitel 13 vorgestellt. Dies beinhaltet die Basisklassen auf denen eine Implementierung einer Multi-Benutzerschnittstelle aufsetzt sowie die grundlegenden Komponenten, welche sowohl zur Entwicklungs-, als auch zur Laufzeit benötigt werden und somit die Grundlage für das Framework bilden.



Entwicklungsumgebung: Aufbauend auf den grundlegenden Komponenten wird die Entwicklungsumgebung mit der Umsetzung der Unterstützungskonzepte (Kapitel 10) in Kapitel 14 beschrieben.

Anwendungsbeispiel: Für die Fallstudie in Kapitel 16 und die Nutzerstudie in Kapitel 17 wurde das SoKNOS Projekt als Szenario gewählt. Das Projekt wird in Kapitel 15 genauer beschrieben und dabei die Integration das Ansatzes in das Gesamtprojekt näher ausgeführt.

[1] Oracle Java ist erhältlich unter http://www.oracle.com/technetwork/java/index.html

[2] Der nicht existierende Zusammenhang zwischen Mapache und Apache, sowie zwischen Waschbären und modellgetriebener Entwicklung von Benutzerschnittstellen ist beabsichtigt.

Evaluation: Aufbauend auf dem SoKNOS-Szenario und unter Nutzung der prototypischen Realisierung, wurden die vorgestellten Konzepte (Teil II) in zwei Schritten überprüft: Sie wurden in einem Fallbeispiel angewendet, sowie im Rahmen einer Nutzerstudie evaluiert. Das Ziel beider Untersuchungen war es, den Nutzen des Ansatzes zu untersuchen und wichtige Faktoren bei der Übertragung des Ansatzes auf praxisnahe Anwendungskontexte zu identifizieren.

Die erste Untersuchung wird beschrieben in Kapitel 16 und ist die durchgängige Anwendung des Ansatzes im Rahmen einer **Fallstudie**. Die Umsetzung im Rahmen des SoKNOS Projektes wird diskutiert und bewertet. Damit soll gezeigt werden, dass der Ansatz durchgängig anwendbar ist und nutzbare Ergebnisse produziert. Darüber hinaus wird die Umsetzung der Anforderungen anhand des Fallbeispiels bewertet und weitere gemachte Erkenntnisse geschildert.

Im Rahmen der in Kapitel 17 vorgestellten **Nutzerstudie** wird *i)* die Umsetzung der Anforderungen und ihre Relevanz geprüft und *ii)* der Ansatz aus Sicht von Experten aus der Praxis bewertet. Die Studie wurde als "Kooperative Evaluation" zusammen mit fünf Personen, welche Anwendungen und Benutzerschnittstellen berufsmäßig entwickeln, durchgeführt, um die Praxisrelevanz der Ergebnisse sicherzustellen. Einzig Anforderung 4 (Einfache Nutzbarkeit) konnte nur eingeschränkt evaluiert werden, denn diese muss für jedes Produkt einzeln im Detail überprüft werden. Darüber hinaus ist es nicht Ziel dieser Arbeit, Nutzungsprobleme zu reduzieren, sondern einen Beitrag auf konzeptioneller Ebene zu liefern. Die detailliertere Auswertung der Nutzerstudie ist in Anhang B zu finden.

Insbesondere der zweite Schritt liefert *einen Beitrag über die verwandten Arbeiten hinaus*: Denn die Nutzerstudie wird mit berufsmäßigen Entwicklern durchgeführt und insbesondere auch das Vorgehen zur Evaluation beleuchtet. Dies ist eine Ausnahme im Kontext der verwandten Arbeiten. Im Allgemeinen werden dort nur eine oder mehrere Fallstudien vorgestellt und auf konzeptioneller Ebene diskutiert.

Kapitel 18 schließlich beendet diesen Teil der vorliegenden Arbeit mit einer zusammenfassenden Bewertung der Evaluation.

Kapitel 13

Umsetzung der Konzepte

Dieses Kapitel beschreibt die Umsetzung der grundlegenden Komponenten für Mapache. Sie wurden in Kapitel 11 zur architektonischen Integration erarbeitet und werden zur Entwicklungs- sowie zur Laufzeit benötigt.

Das Kapitel ist wie folgt strukturiert. Die implementierten Basisklassen der Multi-Benutzerschnittstelle werden in Kapitel 13.1 besprochen. Sie müssen zur Erstellung einer MBS implementiert werden, wodurch diese zusammen mit der weiteren vorgestellten Umsetzung lauffähig ist. Die Umsetzung des Infrastrukturknotens wird darauf in Kapitel 13.2 beleuchtet. Hierbei wird auch auf das Modellierungsframework (Abschnitt 13.2.1) als Umsetzung der DSL (aus Kapitel 9) eingegangen. Daneben werden die umgesetzten Interpreter vorgestellt (Kapitel 13.3), welche Swing und Hardware basierte MBS-Varianten zur Interaktion bringen können, aber auch als Grundlage für andere Unterstützungen dienen. Abschließend wird in Kapitel 13.3 näher auf die Umsetzung der Interpreter eingegangen, welche für die Unterstützungskonzepte zentrale Bedeutung haben.

13.1 Multi-Benutzerschnittstelle

Die erstellte MBS selbst beinhaltet verschiedene Artefakte, welche unterschiedliche Informationsarten beherbergen. Das Modell (folgender Abschnitt 13.1.1) beinhaltet die Struktur aller MBS-Varianten. Die Beitragsklassen (Abschnitt 13.1.2) stellen hingegen die Fragmente (Beobachter und Verhalten) für die MBS bereit. Schließlich wird die Infrastruktur durch die MBS-Informationsklasse (Abschnitt 13.1.3) konfiguriert sowie die Fragmente und das Modell miteinander assoziiert.

13.1.1 Modell

Das UUI-Modell der MBS mit den UUIBoxen (vgl. Abschnitt 9.1.3) für die verschiedenen Nutzungskontexte wird in einer oder mehreren Modelldateien abge-

legt. Diese werden in der MBS-Informationsklasse (Kapitel 13.1.3) registriert und damit beim Start der MBS in die Modellierungsinfrastruktur geladen.

13.1.2 Beitragsklassen

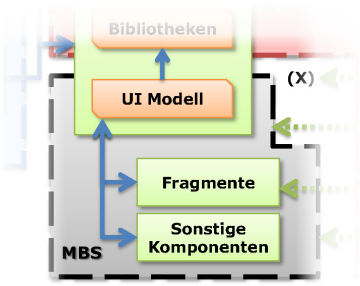


Abbildung 13.1: Komponenten der Multi-Benutzerschnittstelle.

Verhaltensfragmente und Beobachterfragmente (siehe Kapitel 8) werden in `Beitragsklassen` organisiert und zur Verfügung gestellt, wie in Abbildung 13.2 illustriert. Dabei kann für jede Variante (Verfeinerung) der Benutzerschnittstelle je eine Beitragsklasse mit Beobachter- respektive Verhaltensfragmenten über die MBS-Informationsklasse registriert werden. Zur Ermittlung des aktiven Sets von Fragmenten werden die Beitragsklassen über den Beitragsprozessor (siehe Abschnitt 13.2.7) initialisiert und nach Fragmenten abgefragt.

Es wurde eine *Basisklasse* für Beitragsklassen erstellt. Diese stellt Methoden zur Erstellung von neuen Fragmenten und Registrierung von vom Entwickler erstellten Fragmenten zur Verfügung. Insbesondere können Fragmente als sogenannte "*Lokale Methoden*" ausgeführt werden. Das heißt, man registriert eine Methode in der Beitragsklasse als Verhalten. Darüber hinaus werden auch Methoden für das Überschreiben und Löschen von Fragmenten in der Basisklasse bereitgestellt.

13.1.3 MBS-Informationsklasse

Die MBS-Informationsklasse ist der zentrale Konfigurationspunkt einer MBS. In ihr werden alle Modelldateien registriert, welche für die MBS geladen werden müssen. Die Beitragsklassen mit Fragmenten für die einzelnen Verfeinerungen (siehe Abschnitt 13.1.2) werden ebenfalls über sie registriert.

Die MBS-Informationsklasse muss von einer definierten Basisklasse ableiten, welche die Felder für Modelldateien, Beitragsklassen etc. bereitstellt. Sie wird durch die Infrastruktur gefunden, indem das Archiv der MBS nach Klassen untersucht wird, welche von der Basisklasse ableiten. Jede MBS darf folglich nur eine MBS-Informationsklasse enthalten.

13.2 Infrastrukturknoten

Der Infrastrukturknoten stellt das zentrale Element der Architektur dar. Wichtige Bestandteile und Bearbeitungsschritte, wie z.B. das Modellierungsframework oder das Abarbeiten der Beitragsklassen durch den Beitragsprozessor (Kapitel 13.2.7), werden hier durchgeführt. Der passende Teil aus der Archi-

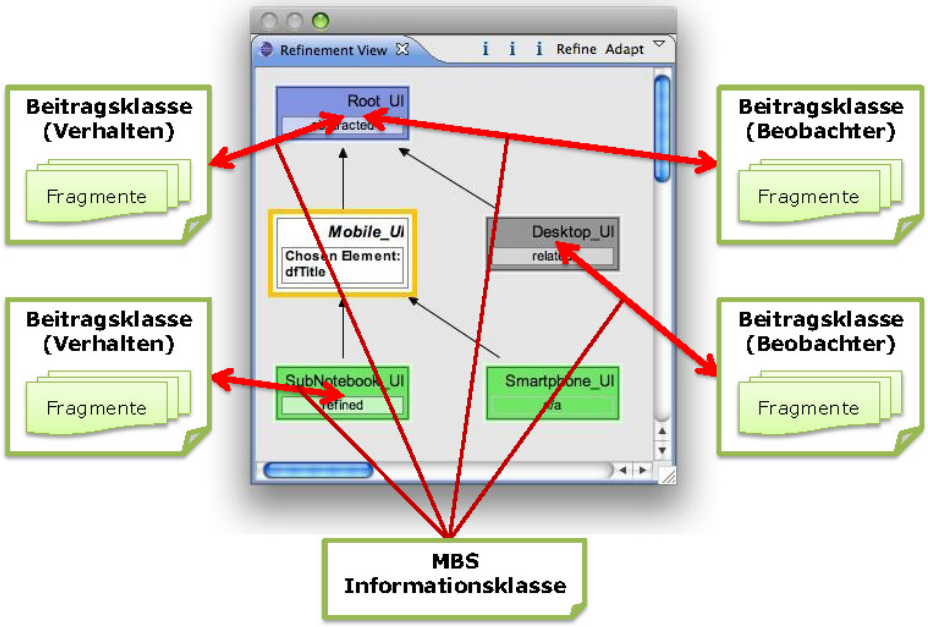


Abbildung 13.2: Zu jeder Verfeinerung können Beitragsklassen für Beobachter und Verhalten assoziiert werden. Die MBS-Informationsklasse dient als zentraler Informationspunkt.

tekturskizze in Kapitel 11 ist in Abbildung 13.3 gezeigt und setzt die einzelnen Komponenten in Zusammenhang.

Das Framework steht erst nach **Start des Infrastrukturknotens** zur Verfügung. Dieser läuft wie folgt ab:

1. Der Nodemanager (Abschnitt 13.2.2) wird instanziiert und seine Methode zur Initialisierung wird aufgerufen.

2. Daraufhin startet und initialisiert der Nodemanager das Modellierungsframework (Abschnitt 13.2.1), den Ressourcenmanager (Abschnitt 13.2.3) und den Anwendungsmanager (Abschnitt 13.2.4).

3. Das Modellierungsframework richtet bei Initialisierung das URI Schema für Bibliotheken ein (vgl. Kapitel 11.4).

4. Dann sucht es das Verzeichnis, in welchem Bibliotheken abgelegt werden, nach Modelldateien ab und lädt die gefundenen.

5. Schließlich wird vom Ressourcenmanager das Verzeichnis für Plugins untersucht, und die dort vorhandenen Plugins werden geladen (wie z.B. die Interpreter in Kapitel 13.3).

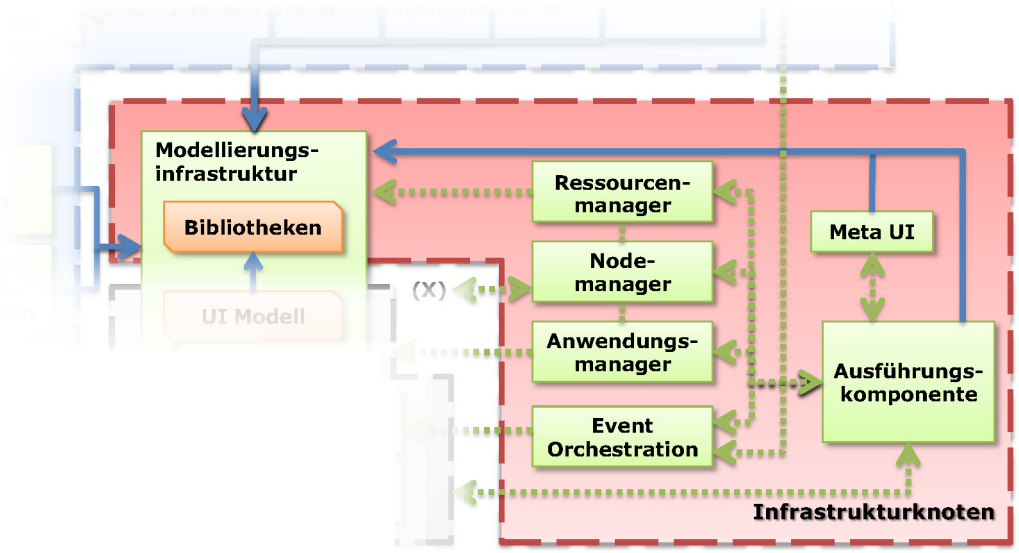


Abbildung 13.3: Umgesetzte Infrastrukturkomponenten im Überblick (Ausschnitt aus Abbildung 11.1). Durchgezogene (blaue) Pfeile sind Modellzugriffe, sind sie bidirektional, auch mit Notifizierungen über Modelländerungen. Gestrichelte Pfeile (grün) sind Zugriffe von Komponenten aufeinander. Das (x) symbolisiert den Zugriff vieler Komponenten auf den Nodemanager, da er der zentrale Einstiegspunkt zur Infrastruktur ist.

Soll dann eine **MBS ausgeführt** werden, stehen die folgenden Schritte an:

1. Der Nodemanager wird zur Ausführung einer MBS benachrichtigt.
2. Er ruft den Anwendungsmanager zum Laden der MBS (Kapitel 13.1) mit MBS-Informationsklasse, Beitragsklassen und Modell auf.
3. Die geladene MBS wird der Ausführungskomponente (Abschnitt 13.2.5) zur Verfügung gestellt, welche das passende Dialog Set (Abschnitt 13.2.6) zur Ausführung erstellt und die Komponenten untereinander verknüpft.
4. Das Dialog Set wählt zur aktiven MBS-Variante die passenden Interpreter mit Hilfe des Ressourcenmanagers und selektiert über den Beitragsprozessor (Abschnitt 13.2.7) die Fragmente, welche in der gewählten Variante aktiv sind.
5. Interaktions- und Änderungsevents (vgl. Kapitel 8) werden vom Interpreter respektive Zustand an das laufende Dialog Set weitergegeben, welches die passenden Fragmente ausführt.

Teile der Kommunikation wurden über die MundoCore (Aitenbichler 2006) Middleware realisiert, was eine sehr schlanke Implementierung ermöglichte.

13.2.1 Modellierungsframework

Die in Kapitel 9 vorgestellte Sprache wurde im Eclipse Modeling Framework (EMF) implementiert. Dazu wurde die in Kapitel 9.2 vorgestellte abstrakte Syntax mit Hilfe von Enterprise Architect als UML Modell beschrieben. Dieses UML Modell wurde durch das EMF Framework in ein Java Projekt überführt, welches Java Klassen enthält, die zu den Klassen in der abstrakten Syntaxbeschreibung korrespondieren. Zur Laufzeit können diese als Modellelemente instanziiert werden.

Diese Klassen bilden somit die abstrakte Syntax nach (so gibt es z.B. eine Wertspezifikationsklasse in diesem Projekt), mussten aber dennoch stark angepasst werden. Getter und Setter erstellt EMF selbst korrekt, aber komplexere Methoden (z.B. Getter, welche einen Subset einer Eigenschaft zurückliefern oder abgeleitete Eigenschaften) wurden manuell implementiert.

Für die *Referenzierung von Bibliotheken* (vgl. Kapitel 11.4) wurde ein Eclipse URIHandler erstellt. Dieser bildet URIs[1] aus Modellen auf eine EMF eigene Referenzart zu Modelldateien ab. Dafür müssen die referenzierten Bibliotheken geladen sein, wofür sie in einem per Installation definierbaren Verzeichnis abgelegt sind.

Alle Komponenten greifen auf das gleiche *Modell* zu, was in EMF über eine identische Transactional Editing Domain für alle zugreifenden Komponenten realisiert ist. Diese Domain kann über den Nodemanager in Erfahrung gebracht werden. So können einzelne Komponenten Listener im Modell installieren und werden bei Änderungen durch andere Komponenten direkt benachrichtigt. Damit ist ein komplett synchrones Arbeiten über mehrere Editoren möglich.

13.2.2 Nodemanager

Der Nodemanager ist der zentrale Zugriffspunkt auf die Infrastruktur und sorgt auch für deren Start. Hierzu ist er als Singleton umgesetzt, sodass man ihn von überall her ohne Objektreferenz nutzen kann. Er stellt auch allgemeine Konfigurationen (z.B. Pfad zu Bibliotheken) zur Verfügung.

Seine Umsetzung erlaubt, dass er *für spezielle Anwendungsfälle spezialisiert* werden kann. Der Nodemanager wurde für den Einsatz in der Entwicklungsumgebung (Kapitel 14) und für die Integration in SoKNOS (Kapitel 15.2) spezialisiert. Hierdurch werden z.B. die Pfade, welche Bibliotheken enthalten, aus den SoKNOS bzw. Eclipse Einstellungen gelesen.

13.2.3 Ressourcenmanager

Am Ressourcenmanager können sich Plugins für die Infrastruktur (z.B. ein Interpreter) registrieren. Er unterstützt mit diesem Wissen die Ausführungskomponente (Abschnitt 13.2.5) bei der Suche nach für eine gegebene UUIBox

[1] URI – Uniform Resource Identifier

adäquaten Interpretern. Dazu werden die obersten Interaktionsobjekte in der UUIBox an alle verfügbaren Interpreter übergeben. Diese liefern zurück, ob sie das übergebene Element interpretieren können oder nicht. Hierzu testet z.B. der Swing basierte Interpreter (Abschnitt 13.3.1), ob die Toolkit ID des übergebenen Modellierungselementes passt.

13.2.4 Anwendungsmanager

Der Anwendungsmanager lädt die MBS (Kapitel 13.1). Dabei stößt er die Modellierungsinfrastruktur zum Laden der Modelldateien an. Er lädt die Beitragsklassen (Kapitel 13.1.2) in einen für die MBS separaten Classloader, wobei er hierbei die MBS-Informationsklasse (Kapitel 13.1.3) der MBS heraussucht.

13.2.5 Ausführungskomponente

Die Ausführung der MBS wird von der Ausführungskomponente organisiert. Sie stellt die Verbindung von der laufenden MBS zur Infrastruktur her. Dabei ist sie so umgesetzt, dass sie für einen gegebenen Anwendungsfall spezialisiert werden kann. Für die Integration in SoKNOS wurde z.B. eine solche angelegt, um die Integration des Canvas des Interpreters in das des SoKNOS Basisplugins zu bewerkstelligen.

13.2.6 Dialog Set

Zur Lauf- und Entwicklungszeit ist das `Dialog Set` für die Organisation der Interaktion zuständig. Es stellt die Verbindung zwischen Beitragsklassen (Kapitel 13.1.2), dem MBS Zustand (Kapitel 8.3) und der interagierten UUIBox her. Zur Identifikation, welche Fragmente in einer MBS-Variante aktiv sind, wird (auch zur Entwicklungszeit) der Beitragsprozessor (Abschnitt 13.2.7) genutzt.

Zur Laufzeit startet und stoppt das Dialog Set die Interpretation der UUIBox mit Hilfe des Ressourcenmanagers (Abschnitt 13.2.3). Es empfängt Interaktionsevents vom Interpreter sowie Änderungsevents vom Zustand und startet die passenden Fragmente zur Abarbeitung der Events.

13.2.7 Event Orchestration

Über den `Beitragsprozessor` wird die Verarbeitung von Fragmenten zur Verfügung gestellt. Diese findet getrennt für die beiden Fragmentarten statt. Bei Wahl einer MBS-Variante werden alle relevanten Beitragsklassen (Kapitel 13.1.2) verarbeitet, um den aktiven Set von Fragmenten zu bestimmen (diese kann über die Verhaltensansicht in Kapitel 10.8 exploriert werden).

Zum Berechnen des aktiven Sets von Fragmenten werden Beitragsklassen aus verschiedenen MBS-Varianten herangezogen. Beginnend von der Wurzel (im laufenden Beispiel die Variante “Meldungen”, siehe Kapitel 16.1) werden alle MBS-Varianten auf *dem direkten Verfeinerungspfad* bis zur aktuell

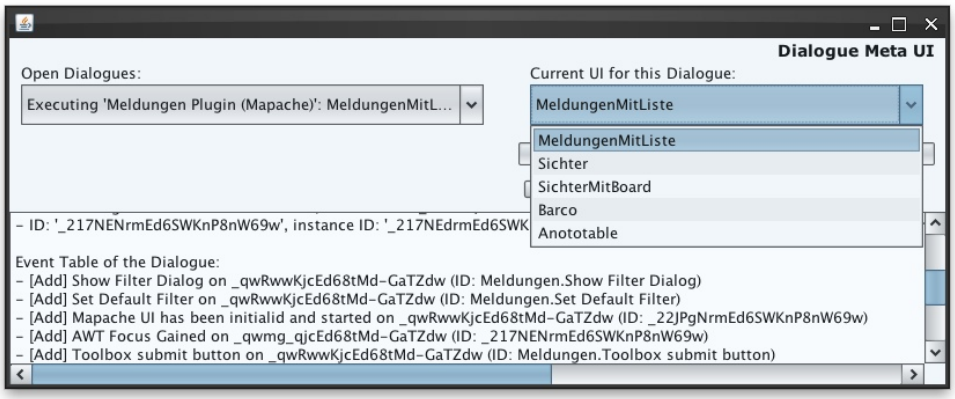


Abbildung 13.4: Das implementierte Meta UI. Es erlaubt die Wahl der zu nutzenden MBS-Variante, sowie die Anzeige rudimentärer Debug Information.

aktiven MBS-Variante genutzt. Ist im laufenden Beispiel die Variante “Sichter+Hardware” aktiv, so werden die Varianten “Meldungen”, “Sichter” und “Sichter+Hardware” mit einbezogen. Die Varianten werden dann von der Wurzel an in Richtung aktiver Variante verarbeitet:

1. Der Prozessor initialisiert die Beitragsklasse.
2. Er liest die Fragmente aus der Beitragsklasse aus und
3. führt diese mit den bereits für vorherige MBS-Varianten ausgelesenen Fragmenten zusammen.

Wird ein Fragment von einer Variante in eine Verfeinerung dieser vererbt, so passt der Beitragsprozessor in dessen Zuordnungstupel an, auf welches Interaktionsobjekt das geerbte Fragment registriert ist (vgl. Kapitel 8.2 und 9.1.6). Ggf. wird auch der Interaktionsevent angepasst, wenn das Toolkit beim Erben gewechselt wurde.

Beim Zusammenführen von Beiträgen für unterschiedliche MBS-Varianten führt er das Überschreiben und Löschen von Fragmenten (vgl. Abschnitt 9.1.1) aus und stellt abschließend die so konsolidierte Information, welche Fragmente in einer bestimmten Variante aktiv sind, zur Verfügung. Dies geschieht auch zur Entwicklungszeit (vgl. Kapitel 14.4), damit z.B. die Verhaltensansicht (vgl. Kapitel 14) die Liste mit aktuell aktiven Fragmenten erhält.

13.2.8 Meta UI

Beim Start der MBS und zur Laufzeit kann die zu nutzende MBS-Variante gewählt respektive gewechselt werden. Diese Auswahl der zu nutzenden MBS-Variante ist durch die MBS selbst möglich, z.B. als Reaktion auf eine beobachtete

Kontextänderung. So führt in SoKNOS das Einstecken der Hardware (nach Rückfrage beim Benutzer) zum Umschalten der MBS-Variante (vgl. Kapitel 15.1).

Alternativ zur Wahl durch die MBS ist die Nutzung eines Meta UI möglich, welches, wie in Abbildung 13.4 zu sehen, implementiert wurde. Neben der Wahlmöglichkeit liefert die Umsetzung rudimentäre Debug Information, bzgl. welche Fragmente aktiv sind oder gerade ausgeführt werden. Auch ist das Protokollieren (loggen) von Interaktions- und Änderungsevents mit der Umsetzung möglich.

In Abbildung 13.4 ist oben links die Wahl der MBS selbst zu sehen, rechts daneben ist die anschließende Wahl der Variante möglich. Mit einem Knopf kann auch zur Laufzeit eine neue Variante abgeleitet werden, welche dann über einen der Editoren (vgl. Kapitel 14.3.1) modifiziert werden kann.

13.3 Interpreter

Interpreter, in Abschnitt 10.1.1 konzipiert, bringen UUI-Modelle zur Interaktion, wie in Kapitel 7.3 beschrieben. Die Instanzen (im objektorientierten Sinne), welche die interagierte UI ausmachen (z.B. Instanzen von JLabel oder JPanel), werden, wie dort eingeführt, "*UIKomponenten*" genannt.

Interpreter werden in der Entwicklungsumgebung und zur Laufzeit eingesetzt. Sie werden als Plugins in die Infrastruktur geladen, daher stehen sie über den Ressourcenmanager (Abschnitt 13.2.3) zur Verfügung.

Es wurden verschiedene Interpreter konzipiert und prototypisch umgesetzt, welche unterschiedliche Toolkits unterstützen:

Swing basiert: Unter anderem die verschiedenen Varianten der Benutzerschnittstelle von SoKNOS in den Abbildungen in Kapitel 16.1 basieren auf dem Swing Toolkit.

Hardware basiert: Der Arduino Hardware basierte Interpreter wurde für das Fallbeispiel (Kapitel 15.1) entwickelt, um eine föderierte UI bestehend aus einem grafischen Teil und einem Teil Spezialhardware zu realisieren.

Text basiert: Der Text basierte Interpreter wurde hauptsächlich zum Testen der Generizität des Konzeptes und der Transformation von Interaktionsevents (vgl. Beitragsprozessor in Abschnitt 13.2.7) genutzt. Er ähnelt in der Nutzung früheren Bulletin Board Systemen.

Die ersten beiden Interpreter werden in den beiden Folgeabschnitten näher beschrieben.

13.3.1 Swing Interpreter

Der Swing Interpreter ermöglicht die Interpretation von AWT und Swing basierten UUI-Modellen. Dabei setzt er einen großen Teil der Swing Komponenten

aus Java 1.6 um, so z.B. Textfelder, Panele, Komponentenränder und Knöpfe. Dazu wurden die passenden Modellierungsbibliotheken erstellt:

- Eine AWT Bibliothek, welche die grundlegenden AWT Komponenten enthält (hiervon erbt Swing) und
- eine darauf aufbauende Swing Bibliothek.

Des Weiteren kann er leicht um Anwendungsfall spezifische Komponenten erweitert werden (wie in Kapitel 11.4 beschrieben).

Die Identifikation, welche Java Klasse für ein Modellelement instanziiert werden muss, damit die passende UIKomponente herauskommt, findet über den Modellelementnamen statt. Die Bibliotheken wurden so modelliert, dass der vollqualifizierte Modellelementname abzüglich eines fixen Präfix dem Klassennamen der gesuchten UIKomponente entspricht. Diese, zum Interaktionsobjektklassifizier korrespondierende UIKomponente wird, wie in Abbildung 7.2 in Kapitel 7.3 gezeigt, bei der Interpretation instanziiert.

Bei der **Durchführung einer Interpretation** werden rekursiv (bzgl. der Verschachtelungsbeziehung, vgl. Abschnitt 9.2.2) die UIKomponenten und Adapter für die gefundenen Modellelemente erzeugt. Die Instanziierung läuft über den Standard Classloader, oder, wenn ein Classloader an den Interpreter übergeben wurde, über diesen. Ist der Interpretationsschritt vorüber, kann die Benutzerschnittstelle sichtbar geschaltet werden.

Passend zur UIKomponente werden *Adapter* mit Reflexionseigenschaft erstellt, welche (wenn benötigt) auch Container- und Komponentencharakter haben können (vgl. Abschnitt 10.1.1.2). Zusätzlich zu den Reflexionsadaptern ist es über eine vorgeschaltete Factory möglich, für eine Klasse von UIKomponenten spezialisierte Adapter zu hinterlegen, welche an Stelle der Standardadapter genutzt werden.

Die Adapter installieren bei ihrer Instanziierung Listener am interpretierten Modellelement, so dass auf Modelländerungen reagiert werden kann. Ebenso subskribieren sie sich auf Events der UIKomponenten, um diese bei der Interaktion einzufangen und an das verbundene Dialog Set (Abschnitt 13.2.6) weiterzugeben.

Die Möglichkeit, dem Interpreter einen Classloader zu übergeben, welchen er zum Instanziieren von UIKomponenten nutzt, erlaubt die *Nutzung von speziell für den Anwendungsfall geschriebenen UIKomponenten.* Auch hierbei wird der Klassenname für UIKomponenten über die Namenshierarchie (der Elemente in der Bibliothek) abgeleitet. Die reflektiven Adapter erlauben die Nutzung der erweiternden Komponenten ohne speziell für sie neue Adapter erstellen zu müssen. Dafür arbeiten sie rein über Namen, welche der Interpreter aus dem Modell beziehen kann. Somit ist die Erweiterbarkeit gewährleistet, wie in Kapitel 11.4 beschrieben.

Mehrere Swing Interpreter Instanzen können über das Swing Interpreter Plugin verwaltet werden. Dadurch ist die parallele Interpretation mehrerer

UUI-Modelle möglich. Auf dies Plugin kann über den Ressourcenmanager zugegriffen werden und mit Hilfe des Plugins eine neue Interpreterinstanz erzeugt werden.

Es gab Befürchtungen[2], dass der Ansatz des Modellinterpreters unter Geschwindigkeitsproblemen leidet. Dem ist nicht so – die Interaktion mit der MBS in SoKNOS (Kapitel 16) lief mit gleicher wahrgenommener Geschwindigkeit ab, wie die Interaktion mit Plugins, welche nicht auf dem MBS Konzept basierten. Auch bei anderen Tests (außerhalb von SoKNOS) konnten keine Geschwindigkeitsabstriche gegenüber programmierten Benutzerschnittstellen wahrgenommen werden.

13.3.2 Arduino basiertes Hardware Toolkit

Abbildung 13.5: Die für SoKNOS gebaute Hardware basiert auf der Arduino Plattform.

Die im Anwendungsfall genutzte Spezialhardware ist in Abbildung 13.5 zu sehen (vgl. Kapitel 15.1 für den Anwendungsfall). Sie basiert auf der frei verfügbaren Arduino Plattform[3] zur Erstellung von Prototypen. In der Hardware ist ein Arduino eingebaut, welcher über seine Ausgänge und zusätzliche elektronische Bauteile mehrere Farb LEDs ansteuert. Die Taster sind mit den Eingängen des Arduino verbunden.

Zum Modellinterpreter wurde eine passende Modellierungsbibliothek mit den Komponenten Device (für die gesamte Hardware), Pushbutton (für Taster) und LED (für Farb LEDs) geschrieben. Mit Hilfe der Bibliothek wird das zu nutzende Hardware Gerät nachmodelliert, wobei die einzelnen Modellelemente über IDs zu den einzelnen Hardware Bauteilen (Taster, LEDs) zuordenbar sind. Diese IDs wurden zuvor fest im Programm auf dem Arduino Baustein einkodiert.

Die **Kommunikation** mit der Hardware findet über die Serielle (via USB zu Seriell) Schnittstelle statt. Dafür wurde ein fehlererkennendes Protokoll zur Kommunikation zwischen dem Arduino Baustein und dem Interpreter entwickelt. Der Interpreter selbst ist wiederum mit einem *Boardmanager* verbunden. Dieser sucht regelmäßig nach Arduino basierter Hardware und stellt eine Liste derer zur Verfügung. Wird die Interpretation von der Ausführungskomponente angestoßen, sucht der Interpreter zum interpretierten UUI-Modell das passende Arduino basierte Gerät im Boardmanager (über eine ID) heraus. Eine weitere Klasse kapselt schließlich das Protokoll und stellt dem Interpreter einfache Getter und Setter zur Verfügung.

[2] die ein Gutachter des Papiers Behring, Petter und Mühlhäuser 2010 äußerte
[3] http://www.arduino.cc

Kapitel 14

Entwicklungsumgebung

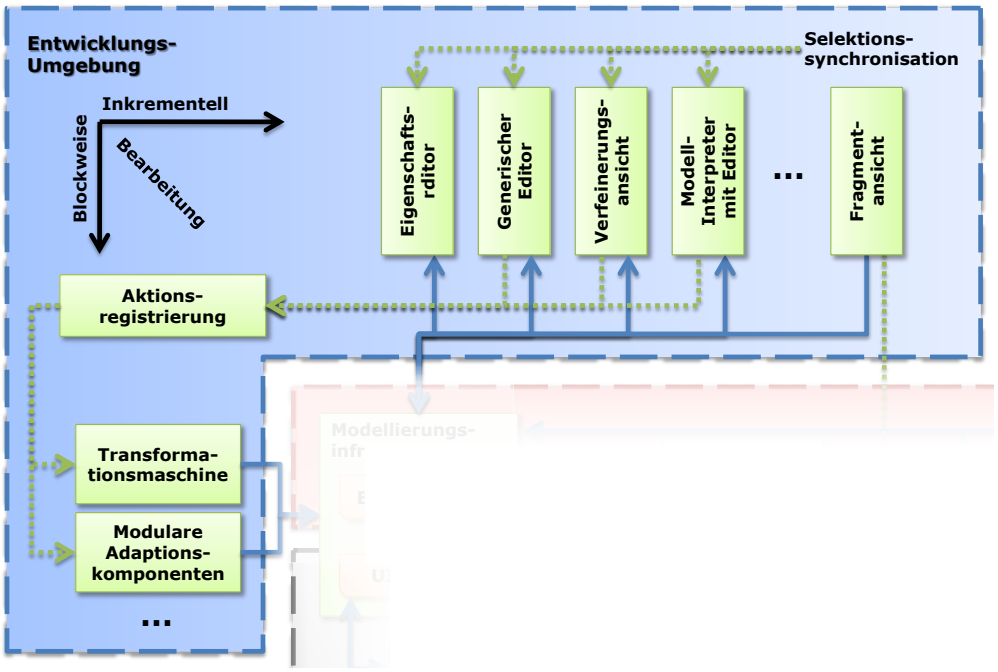


Abbildung 14.1: Umgesetzte Komponenten der Entwicklungsumgebung im Überblick (Ausschnitt aus Abbildung 11.1). Durchgezogene (blaue) Pfeile symbolisieren Modellzugriffe, bidirektional auch mit Notifizierungen über Modelländerungen. Gestrichelte Pfeile (grün) sind Zugriffe von Komponenten aufeinander.

In Kapitel 10 wurden Unterstützungskonzepte entwickelt, welche im Rahmen von Mapache in Eclipse umgesetzt wurden. Die Umsetzung der Entwicklungsumgebung baut auf dem Architekturmuster (Kapitel 8), der architektonischen Integration (Kapitel 11) sowie der Umsetzung der grundlegenden Komponenten (Kapitel 13) auf. Sie wurde im Rahmen der Fallstudie (Kapitel 16)

und der Nutzerstudie (Kapitel 17) eingesetzt.

Im Folgenden wird die Umsetzung der Unterstützungskonzepte beschrieben (Kapitel 14.1), worauf auf einzelne interessante Aspekte der Umsetzung in Eclipse näher eingegangen wird (Kapitel 14.2). Im Folgenden Kapitel 14.3 wird auf die Umsetzung der Editoren (basierend auf den Interpretern, Kapitel 13.3) eingegangen und abschließend die Verknüpfung von Lauf- und Entwicklungszeit diskutiert (Kapitel 14.4).

14.1 Umsetzung der Unterstützungskonzepte

Ziel bei der Umsetzung war insbesondere die enge Integration der verschiedenen Unterstützungskonzepte, sodass die Bearbeitung der UUI-Modelle integriert (Unteranforderung 4.3) mit der Entwicklung des Codes erfolgen kann. Der passende Teil aus der Architekturskizze in Kapitel 11 ist in Abbildung 14.1 gezeigt. Das Ergebnis der Umsetzung – die Mapache Entwicklungsumgebung – ist in Abbildung 14.2 zu sehen.

Die in Abbildung 14.2 zu sehenden Bestandteile sind im Einzelnen:

Die **Verfeinerungsansicht** stellt, wie in Abschnitt 10.2.1 konzipiert, den Verfeinerungsbaum der MBS dar. Auf den Knoten des Baumes (UUIBoxen) sind direkt Aktionen über ein Kontextmenü möglich. Dies sind unter anderem das Abstrahieren oder Verfeinern einer UUIBox (Konzept in Abschnitt 10.1.4.2) für einen Nutzungskontext, die Ausführung einer Modell zu Modell Transformation (vgl. Abschnitt 12.2.1) oder der Start einer Adaption (Konzept in Abschnitt 10.1.3, prototypische Umsetzung in Abschnitt 14.3.3).

Über die **Verhaltensansicht** können, wie in Abschnitt 10.2.2 konzipiert, die aktiven Verhaltensfragmente per UUIBox untersucht werden. Per Doppelklick kann ein Fragment, welches auf einer lokalen Methode beruht (vgl. Abschnitt 13.2.7), zum Editieren im Java Editor geöffnet werden. Existiert kein Fragment für ein Zuordnungstupel (vgl. Kapitel 8.2), kann mit Hilfe eines Wizards ein Fragment basierend auf einer lokalen Methode erstellt werden. Dabei wird, falls erforderlich, auch die nötige Beitragsklasse (vgl. Kapitel 13.1.2) erzeugt und in der MBS-Informationsklasse (Kapitel 13.1.3) registriert.

Der Quellcode von Fragmenten kann über einen **Java Editor** bearbeitet werden. Dieser wurde um eine Code Vervollständigung erweitert, wie in Abschnitt 14.3.4 gezeigt. Hierdurch können vollqualifizierten Modellelementnamen (genutzt zur Referenz) vervollständigt werden.

Die weiteren, in Abbildung 14.2 zu sehenden Komponenten (Editoren) werden im Kapitel 14.3 beschrieben.

14.2 Aspekte der Eclipse Umsetzung

Ein zentrales Thema der Umsetzung war die **Kommunikation zwischen den verschiedenen Komponenten**. Insbesondere die Anbindung der Komponen-

ten, welche nicht direkt in Eclipse eingebettet sind (wie z.B. die Interpreter) musste gelöst werden. Hierzu wird die Middleware **MundoCore** (Aitenbichler 2006) eingesetzt. Mit Hilfe von MundoCore können so genannte Mundo Knoten (z.B. der Interpreter) miteinander kommunizieren und Methoden an entfernten Objekten aufrufen. Die Knoten können dabei auf einer Vielzahl von Plattformen (Windows, Apple, Mobiltelefon, Smartphone, ...) laufen. Einzig die Bindung der Knoten aneinander muss speziell konfektioniert werden, hierfür wurde im Rahmen der vorliegenden Arbeit ein fest definierter Kommunikationskanal genutzt.

Für die Entwicklungsumgebung wurde eine **über alle Editierkomponenten einheitliche Selektion** konzipiert. Die Umsetzung dieser gestaltete sich in Eclipse nicht trivial. Eclipse lässt nur Eclipse eigene Editoren mit grafischer Oberfläche eine Selektion bereitstellen (SelectionProvider). Somit konnte der Swing Interpreter basierte Editor (er ist kein Eclipse basierter Editor, vgl. Abschnitt 14.3.1) nicht einfach zum Selektieren von UIKomponenten genutzt werden.

Daher musste der generische Editor (Abschnitt 14.3.2), welcher ein Eclipse Editor ist, mit einem MundoCore basierten Selektionsdienst erweitert werden. An diesen Dienst wurden nicht Eclipse eigene Komponenten zur Selektionssynchronisation angeschlossen. Der generische Editor dient somit als Proxy zwischen Eclipse und externen Komponenten.

Die verschiedenen Erweiterungen an Eclipse Komponenten mussten teilweise etwas umständlich umgesetzt werden. Die sehr **intensive Nutzung von Patterns** (an sich ein Merkmal für gutes Design) im Code von Eclipse ließen die umzusetzenden Konzepte nicht immer direkt zu. Insbesondere die übermäßige Verwendung des Adapter Patterns[1] muss hier angeführt werden. Somit musste bei Erweiterungen, welche nicht mit dem "Standard Ablauf" oder der "Standard Präsentation" konform waren, "um die Patterns herum"[2] gearbeitet werden.

Konkret waren die Patterns bei der Umsetzung des Eigenschaftseditors (Abschnitt 14.3.2.1) hinderlich, da bestimmte Modifikationen durch die Patternstruktur sehr umständlich wurden. Auch die Darstellung der Verhaltensansicht musste um Patterns herum erstellt werden. Das vom Builder erzeugte Strukturmodell (vgl. Kapitel 14.4) für Fragmente benötigte ein zusätzliches Element per UUIBox als Knoten, um externe Events in der Darstellung zu gruppieren (vgl. Abbildung 10.8 auf Seite 133). Dies musste letztendlich nicht in der Darstellung sondern im Strukturmodell hinzugefügt werden.

[1] Überspitzt lässt sich formulieren, dass in Eclipse *alles* ein Adapter ist.

[2] Vergleich dazu auch der Kommentar von Studienteilnehmer drei in Anhang B.1.4: "*[...] suchen die Entwickler nach Wegen, die Abstraktion wieder einzufangen.*"

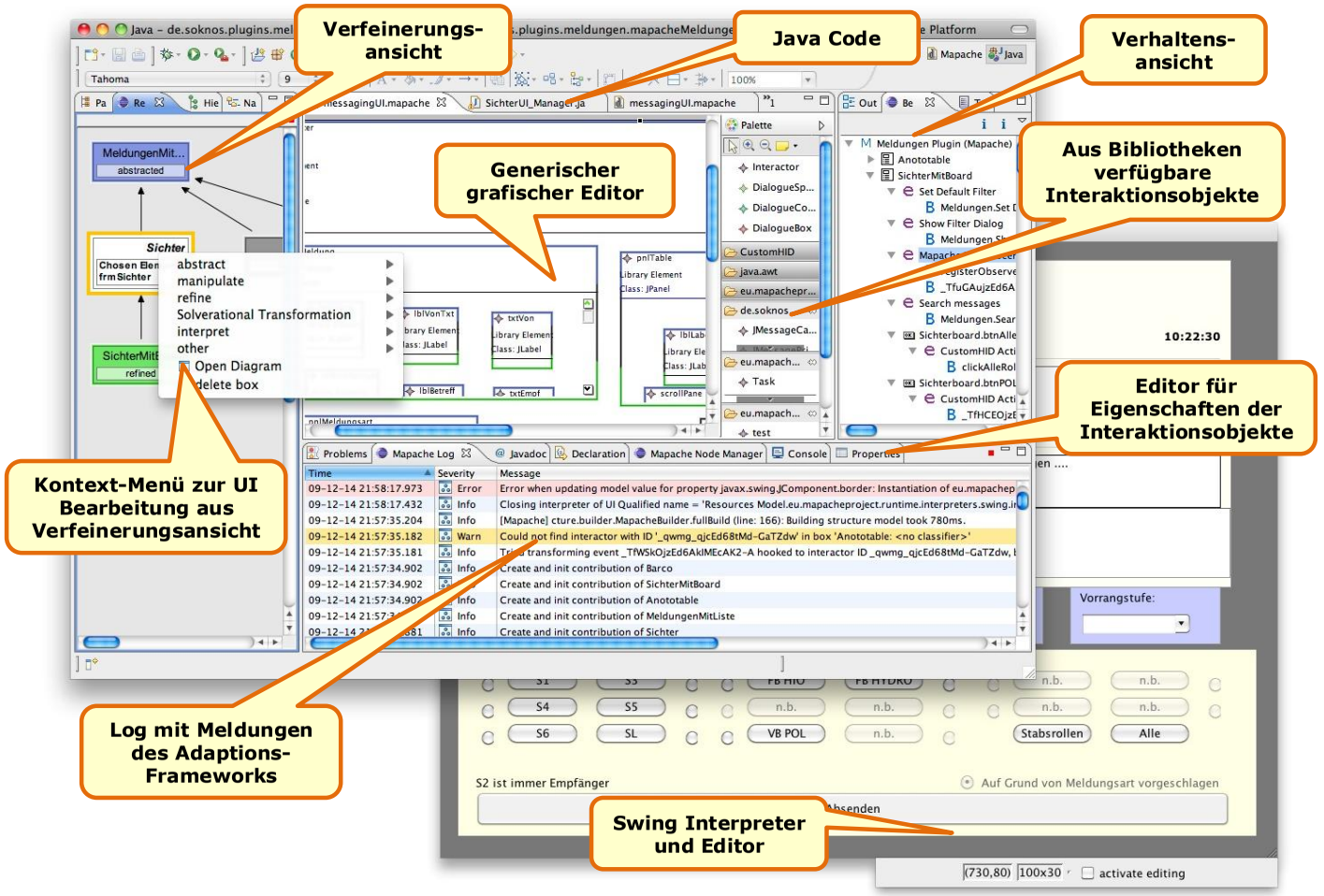


Abbildung 14.2: Überblick über die prototypische Umsetzung der Entwicklungsumgebung in Eclipse. Die verschiedenen Editoren und Ansichten sind miteinander gekoppelt, sodass die aktuelle Selektion in allen Teilen gleich genutzt wird.

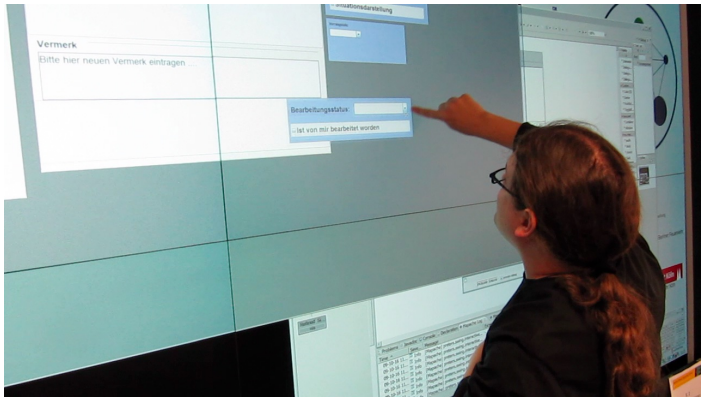


Abbildung 14.3: Das Editieren der Benutzerschnittstelle mit Hilfe des Swing Interpreter basierten Editors direkt an der Großbildwand.

14.3 Editoren

Im Rahmen der Umsetzung wurden verschiedene Editoren erstellt. Diese werden im Folgenden detaillierter beschrieben.

14.3.1 Swing Interpreter basierter Editor

Ein Editor, basierend auf dem Swing Interpreter, (vgl. Kapitel 13.3) wurde umgesetzt (zu sehen in Abbildungen 14.3 und 14.4). Dazu wurde das generische Editorkonzept aus Abschnitt 10.1.2 herangezogen.

Als *Editierkonzept* (vgl. o.g. Konzeptkapitel) wurde ein klassischer Rahmen um die selektierte UIKomponente zur Verfügung gestellt, wie in Abbildung 14.4 gezeigt. An den Ecken hat der Rahmen "Griffe", an denen man die Größe der UIKomponente ändern kann. Durch Klicken auf die Komponenten und Ziehen kann sie repositioniert werden, was auch mit Hilfe der Cursortasten möglich ist. So konnten die Benutzerschnittstellen des Fallbeispiels (Kapitel 16.1) bequem finalisiert werden, wie in Abbildung 14.3 zu sehen.

Der Klick auf eine UIKomponente selektiert diese. Dabei ist die Selektion, wie in Kapitel 14.2 beschrieben, unter allen Editoren synchronisiert. So auch mit dem Eigenschaftseditor von Eclipse (siehe Abbildung 14.4).

14.3.2 Generische Editoren

Neben dem Modellcode (vgl. Abschnitt 13.2.1) wurde mit Hilfe von EMF ein **baumbasierter Editor** automatisch generiert. Dieser wird zum Editieren der Bibliotheken eingesetzt, da er alle Elemente der Sprache unterstützt. Er kann folglich auch zum Editieren von UUI-Modellen eingesetzt werden, was aber sehr umständlich ist, da für das Setzen eines Eigenschaftswertes auch die entsprechenden Slot- und Wertspezfikationselemente modelliert werden müssen.

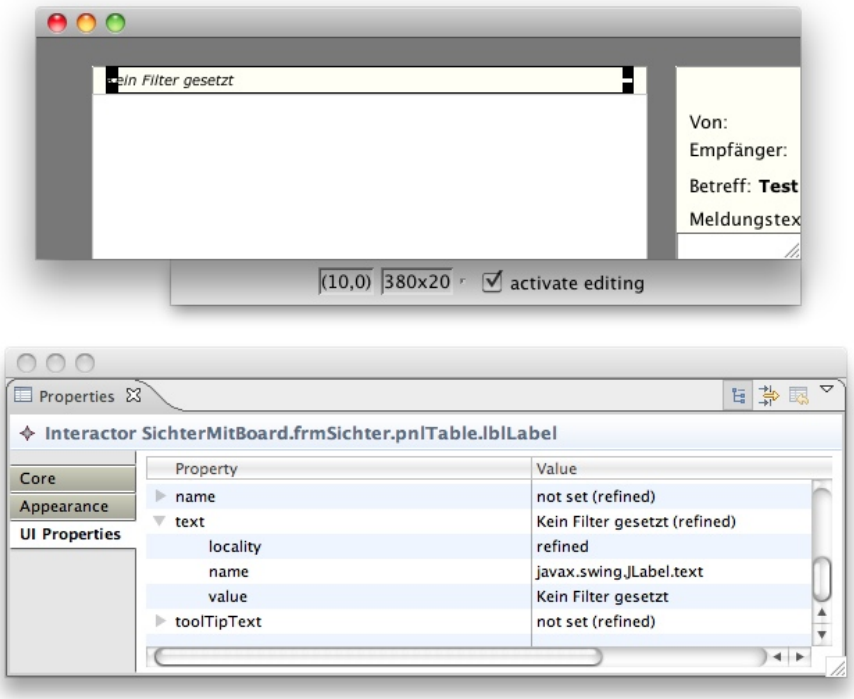


Abbildung 14.4: Eine Benutzerschnittstelle wird interpretiert und editiert. Der Eigenschaftseditor von Eclipse (unten im Bild) ist mit der Selektion des Editors synchronisiert. Er zeigt zu jeder Eigenschaft deren *i)* Verfeinerungszustand, *ii)* vollqualifizierten Namen und *iii)* Wert an.

Auch ist, da alle Elemente eines Modells angezeigt werden, die Orientierung im Editor schwierig.

Mit Hilfe des Graphical Modeling Frameworks (GMF)[3], welches auf EMF aufsetzt, wurde ein **grafischer Editor** generiert und stark angepasst. Er ist in der Mitte von Abbildung 14.2 zu sehen. Seine Nutzbarkeit ist nicht sehr hoch, weil die Darstellung der Elemente generisch und Toolkit unabhängig ist. Dafür kann er aber auch mit beliebigen Bibliotheken, unabhängig vom Toolkit, arbeiten. Der grafische Editor erlaubt das einfache Setzen von Eigenschaftswerten über den Eigenschaftseditor (nächster Abschnitt 14.3.2.1). Dieser wurde im Rahmen der Anpassung des grafischen Editors mit generiert und modifiziert.

14.3.2.1 Eigenschaftseditor

Als Teil des generischen grafischen Editors aus dem letzten Abschnitt, wurde durch GMF auch ein Eigenschaftseditor generiert. Dieser wurde anschlie-

[3] http://www.eclipse.org/gmf

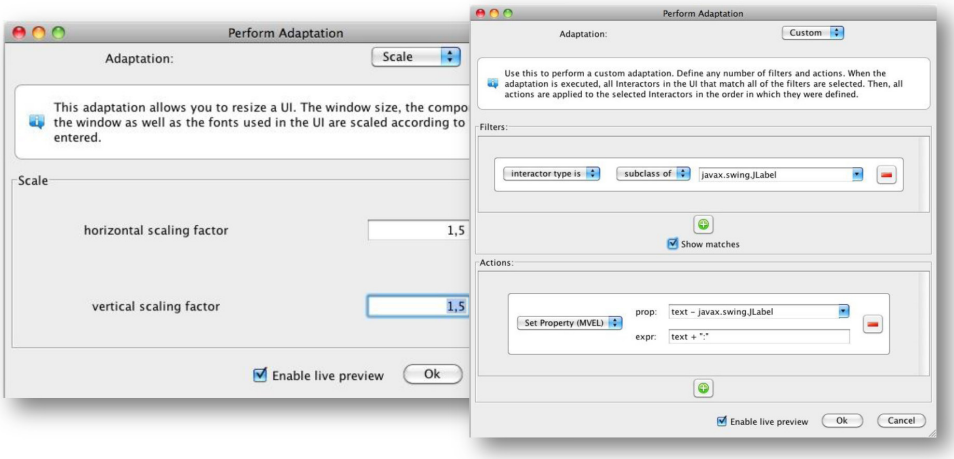


Abbildung 14.5: Verschiedene Adaptionskonzepte: Größenanpassung (links) und eine generische, Makro basierte Anpassung (rechts).

ßend stark angepasst. Er unterstützt die passive Propagation, wie in Abschnitt 9.1.5, Definition 5 eingeführt, und zeigt den jeweils abgeleiteten Wert einer Eigenschaft an. Darüber hinaus kann man erkennen, wie in Abbildung 14.4 zu sehen, ob eine Eigenschaft geerbt oder lokal gesetzt ist.

Das *Setzen von Eigenschaftswerten* wurde passend zur in Kapitel 9 formulierten DSL umgesetzt. Dabei werden automatisch die benötigten Slot- und Wertspezifikationselemente überprüft und ggf. neu erzeugt. Die über den Interaktionsobjektklassifizierer definierten Eigenschaften wurden auf einer neuen Eigenschaftsseite zusammengefasst ("UI Properties" in Abbildung 14.4). Wie auch andere Editoren, ist der Eigenschaftseditor bzgl. der *Selektion synchronisiert*. So können, egal wo ein Element selektiert wurde, seine Eigenschaften immer im Eigenschaftseditor bearbeitet werden.

14.3.3 Modulare Adaptionskonzepte

Das in Abschnitt 10.1.3 diskutierte Adaptionskonzept wurde, basierend auf dem Swing Interpreter, verwirklicht. Zwei der Anpassungen sind in Abbildung 14.5 zu sehen. Aus der Klassifikation von Problemen bei der UI Migration von Arbeitsplatzrechnern auf Großbildwände (Glaubitt 2009), wurde die Skalierung zur Umsetzung ausgewählt. Des Weiteren wurde eine spezielle Adaptionskomponente und -schnittstelle für die direkte Nutzung der Makrosprache durch Entwickler konzipiert.

Alle Anpassungen können live über eine *Voransicht* via Interpreter verfolgt werden. Dafür legt der generische Teil der Adaptionskomponente vorweg eine Kopie der zu adaptierenden UUIBox an. Diese wird dann interpretiert und durch die Umsetzung des jeweiligen Konzeptes modifiziert.

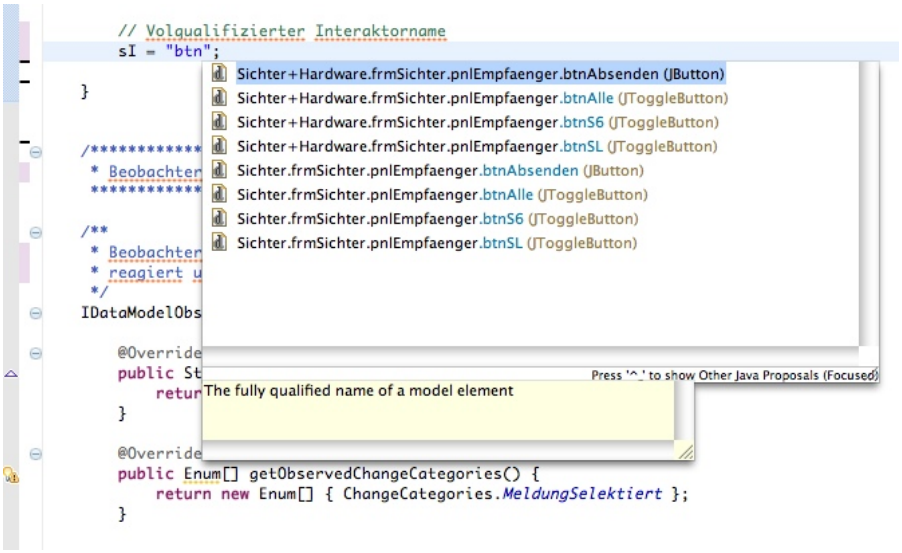


Abbildung 14.6: Aktivierte Codevervollständigung im Java Editor.

Umgesetzt wurde zum Einen ein Konzept zur **Skalierung** von Benutzerschnittstellen (vgl. Abbildung 14.5 linke Seite). Hierbei kann der Benutzer bequem einen Skalierungsfaktor in X und Y Richtung angeben. Neben den UI Elementen werden auch deren Schriften passend skaliert.

Daneben wurde auch ein Konzept zur **Makro basierten** Anpassung umgesetzt (Abbildung 14.5 rechte Seite). Zur Präsentation in der Adaptionsschnittstelle wurde das Makro in zwei Teile gespalten. Der *Filterausdruck* (anzugeben im oberen Teil der UI) selektiert, welche Elemente von der Anpassung betroffen sind. Dies ist z.B. auf Grund von Namen oder der Klassenhierarchie möglich. Der *Modifikationsausdruck* im unteren Teil der Adaptionsschnittstelle beschreibt, was mit betroffenen Komponenten passiert. Der Ausdruck wird in der Makrosprache MVEL[4] formuliert. Die Anpassung wird dabei interaktiv gestaltet: Die vom Filter betroffenen Komponenten werden in der Vorschau hervorgehoben und die Modifikation direkt in der Vorschau angezeigt.

Bei der Umsetzung der Adaptionskonzepte waren **Standardwerte des Toolkits** mit die größte Herausforderung. Diese sind nicht als Werte im Modell abgelegt, sondern müssen während der Adaption vom Toolkit erfragt werden. Alternativ kann man eine passende UIKomponente instanziieren und deren Eigenschaften auslesen.

14.3.4 Zugriff auf Modellelemente aus Fragmenten

Das Verhalten ist im Rahmen des vorliegenden Konzeptes komplett programmatisch spezifizierbar, wie in Unteranforderung 5.3 formuliert. Auf der anderen

[4] http://mvel.codehaus.org

Seite liegen die Benutzerschnittstellen komplett als Modelle vor. Muss ein Fragment eine UI Modifikation vornehmen, z.B. im Falle eines Beobachterfragmentes neue Werte für Eigenschaften setzen, so muss das passende Modellelement vorliegen. Es bietet sich also an, den Entwickler beim Referenzieren vom Code in das Modell zu unterstützen (Anforderung 4). Ein gängiges Konzept hierzu ist es, Codevervollständigung zu bieten (gezeigt in Abbildung 14.6).

Im Rahmen der prototypischen Umsetzung können vollqualifizierte Modellelementnamen, mit denen sich das passende Modellierungsobjekt abrufen lässt, vervollständigt werden. Zur Vervollständigung werden alle Modellelemente der MBS angeboten.

14.3.4.1 Typsicherheit

Ein interessanter Aspekt beim Zugriff auf Modellelemente ist die Typsicherheit. Diese kann nicht einfach garantiert werden, da Modellelemente aus dem Programmcode heraus "nur" Interaktionsobjekte sind. Das heißt, das vorliegende Java Objekt aus dem Modellcode (vgl. Abschnitt 13.2.1) ist immer das gleiche, unabhängig von seiner Klassifizierung durch den Interaktionsobjektklassifizierer (JLabel, JPanel, ...). Der Zugriff auf Eigenschaften ist also nur über generische Getter und Setter, z.B. `Element.setProperty("text", "Neuer Text")`, möglich.

Dies ist ein *inhärenter Trade Off*: wenn Erweiterbarkeit (Anforderung 2) gewünscht ist, muss ein generisches Zugriffskonzept genutzt werden. Da ein generisches Konzept aber die Elemente (bzw. deren Typen) nicht kennt, wird weniger Typsicherheit geboten.

Als Möglichkeit zur Verbesserung bieten sich *Wrapper* an. Zu einer Modellierungsbibliothek werden sie spezifisch für die angebotenen Interaktionsobjektklassifizierer ausgeliefert (z.B. ein JLabelWrapper passend zum JLabel Interaktionsobjektklassifizierer). Ein Wrapper stellt Methoden für den typsicheren Zugriff auf die Eigenschaften eines klassifizierten Interaktionsobjektes bereit (z.B. `Element.text = "Besserer Text"`). Er wird dafür (wie beim Adapter-Pattern) um ein Interaktionsobjekt herum gelegt. Eine genauere Betrachtung und Umsetzung eines Wrappers ist auf Grund des geringen zu erwartenden wissenschaftlichen Gehalts nicht Teil dieser Arbeit.

14.4 Verknüpfung Lauf- und Entwicklungszeit

Verschiedene Komponenten benötigen zur Entwicklungszeit **eine Analyse der vom Entwickler erstellten Klassen**. So muss z.B. der Beitragsprozessor (Abschnitt 13.2.7) laufen, um festzustellen, welche Fragmente für welche Verfeinerung aktiv sind. Dies wiederum benötigt die Verhaltensansicht für ihre Visualisierung. Um diese Analyse zur Entwicklungszeit in Eclipse durchzuführen, wurde ein Builder implementiert.

Der Builder führt den Beitragsprozessor aus und stellt als Ergebnis ein `Strukturmodell` der Fragmente zur Verfügung. Die Abstraktion auf ein Strukturmodell ist nötig, da die Nutzung echter Klassen zu rechen- und speicherintensiv wäre. Damit der Prozessor arbeiten kann, wird zur Entwicklungszeit ebenfalls ein Dialog Set (Abschnitt 13.2.6) instanziiert.

Ein weiterer Punkt zur Verzahnung der Lauf- und Entwicklungszeit sind die Interpreter. Diesen kann **ein Classloader zu Instanziierung der UI-Komponenten** übergeben werden (vgl. Kapitel 13.3). Der übergebene Classloader beinhaltet auch die Klassen des aktuellen Java Projektes. Somit können *Anwendungsfall spezifische Komponenten parallel zur MBS Modellierung entwickelt werden*: Der Entwickler kann sie direkt in einer MBS-Variante einsetzen und layouten (die Struktur ändern), aber parallel auch ihren Quellcode modifizieren.

Kapitel 15

Anwendungsbeispiel und Folgerung

Dies Kapitel stellt das Anwendungsbeispiel vor. Es bildet die Grundlage und das Szenario für die Fallstudie (Kapitel 16) und die Nutzerstudie (Kapitel 17).

Das Anwendungsbeispiel ist aus dem SoKNOS Projekt entnommen, welches in Kapitel 15.1 beschrieben wird. Hierfür wurde die prototypische Realisierung Mapache (Kapitel 13 und 14) in das SoKNOS System integriert, wie in Kapitel 15.2 ausgeführt. Zur Integration wurde insbesondere das SoKNOS Portal und seine Plugins erweitert. Abschließend werden in Kapitel 15.3 Folgerungen zur prototypischen Realisierung gezogen.

15.1 Das Anwendungsszenario SoKNOS

Die Arbeit wurde im Rahmen des BMBF[1] Projektes SoKNOS[2] in einer Fallstudie angewandt. Dabei wurden die prototypische Umsetzung des Lösungskonzeptes in die SoKNOS Umgebung integriert (beschrieben in Kapitel 15.2) und mit Hilfe der Entwicklungsumgebung (Kapitel 14) verschiedene Varianten einer SoKNOS Benutzerschnittstelle entwickelt.

Zusammenfassend ist *SoKNOS ein IT System, welches Krisenstäbe unterstützt* und z.B. für die Koordination von Behörden und Organisationen mit Sicherheitsaufgaben (BOS) eingesetzt werden kann. Die drei dabei verfolgten Ziele sind[3] die Verkürzung der Strukturierungsphase nach dem Katastrophenfall, die kontinuierliche und umfassende Informationsbeschaffung aus unterschiedlichen Quellen und die Unterstützung für die Zusammenarbeit unterschiedlicher Akteure auch über Organisationsgrenzen hinweg. Dabei wurde auf einen Ser-

[1] BMBF – Bundesministerium für Bildung und Forschung

[2] SoKNOS – **S**ervice-**o**rientierte Archite**K**turen zur Unterstützung von **N**etzwerken im Rahmen **O**effentlicher **S**icherheit, http://www.soknos.de

[3] http://www.bmbf.de/pubRD/SoKNOS_Ziegert_Auftakt_IPF_SuRvM.pdf gibt eine Übersicht über das Projekt

vice orientierten Ansatz (SOA) gesetzt, um ein möglichst modulares System zu schaffen (Braune u. a. 2011, Döweling u. a. 2009). Semantische Technologien werden genutzt, um die einfache Integration von Datenquellen und neuen Funktionalitäten zu ermöglichen. Dementsprechend ist das SoKNOS Portal – die Benutzerschnittstelle von SoKNOS – modular aus Plugins aufgebaut.

Das Projekt selbst stellt seine Relevanz und seinen Praxisbezug durch die Integration von potenziellen Anwendern sicher. Es fand mit folgenden Partnern statt: B2M Software AG, Berliner Feuerwehr, Berufsfeuerwehr Köln, Bundesanstalt Technisches Hilfswerk, Deutsche Hochschule der Polizei (DHPol), Deutsches Forschungszentrum für künstliche Intelligenz (DFKI), DHI-Wasy, ESRI, Fraunhofer Institute für Experimentelles Software Engineering (IESE) und Graphische Datenverarbeitung (IGD), itelligence, ontoprise, SAP, TU Darmstadt, Technische Universität Dresden und Westfälische Wilhelms-Universität Münster.

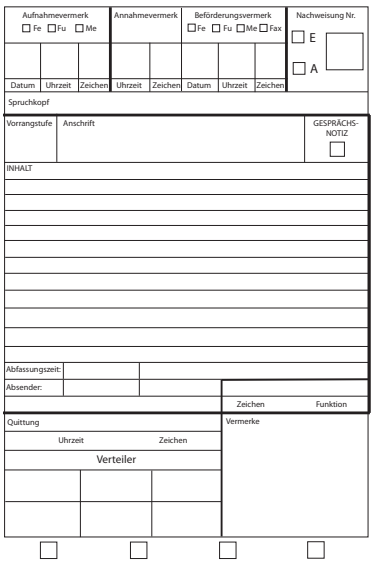


Abbildung 15.1: Schematische Darstellung des traditionell genutzten Vierfachvordrucks.

In enger Zusammenarbeit mit den Anwendern wurde innerhalb des Projektes ein für Deutschland mögliches **Katastrophenszenario** entwickelt. Örtlich ist dieses im Bereich Köln und Leverkusen angesiedelt. Dabei kommt es auf Grund von Sturm und Starkregen zu Überflutungen und Hochwasserlagen an mehreren Stellen. Dammbrüche führen zu Stromabschaltungen und der Überflutung eines Chemiewerks. Da ein kontrolliertes Herunterfahren des Chemiewerks nicht mehr möglich ist, kommt es zu mehreren Explosionen im Werk mit Austritt von Schadstoffen und Beschädigungen an Werk und Wohneinheiten.

In Zusammenarbeit mehrerer Organisationen werden Menschen gerettet, Ortsteile evakuiert, Brände bekämpft und kontinuierlich die Schadstoffe in der Luft gemessen. Die Organisationen sind dabei in Stäben aufgestellt, wie in *Führung und Leitung im Einsatz* (1999) beschrieben. Typische Rollen der Feuerwehr (welche einen starken Fokus im Projekt erhielt) sind im Folgenden kurz aufgeführt.

S1 ist verantwortlich für Personal und den inneren Dienst,

S2 erstellt ein gemeinsames Lagebild des Stabes und konsolidiert die Situationsinformation,

S3 plant den Einsatz vor Ort,

S4 organisiert die Versorgung,

S5 ist verantwortlich für die Pressearbeit und

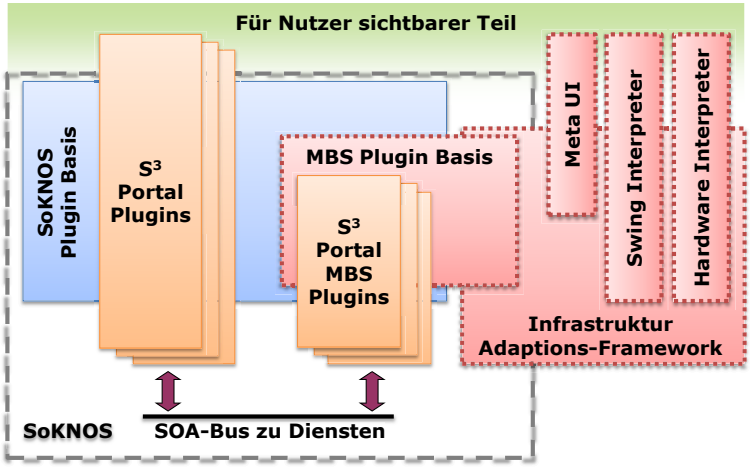


Abbildung 15.2: Die Architektur des SoKNOS Portals mit der Möglichkeit, Plugins zu nutzen, welche auf den Konzepten dieser Arbeit basieren. Gepunktete (rote) Komponenten sind Teil des Adaptionsframeworks dieser Arbeit.

S6 sorgt für ein reibungsloses Informations- und Kommunikationswesen.

Fachberater und Verbindungspersonen haben spezielle Fachkenntnisse (z.B. zu Chemiewerken oder Hochwasserlagen) oder repräsentieren andere Organisationen (Polizei, Militär, ...) und die spezielle

Sichter sorgt dafür, dass jedes Stabsmitglied die für sie relevanten Meldungen weitergeleitet bekommt.

Ein zentrales Element bei der Koordination der Maßnahmen im Katastrophenfall ist die Kommunikation zwischen den Krisenstäben untereinander und mit den Einheiten vor Ort. Diese wird oft (noch) mit so genannten `Vierfachvordrucken` durchgeführt (dargestellt in Abbildung 15.1), welche von Hand ausgefüllt und im Stab verteilt werden. Dafür werden alle (über Telefon, Fax, Funk etc.) eingehenden Meldungen auf je einem Papier erfasst und an den Sichter gegeben. Dieser stellt dann fest, wer im Stab (S1 bis S6, Fachberater, Verbindungspersonen, ...) die Meldung benötigt und gibt die Durchschläge des Vordrucks an diese Rollen weiter. Ausgehende Meldungen werden auch auf einen Vordruck geschrieben und dann an die Fernmeldestelle zur Übermittlung gegeben.

15.2 Erweiterung eines Portals und seiner Plugins

Die IT Lösung SoKNOS besteht aus einem Frontend, Portal genannt, sowie mehreren Backend Diensten mit unterschiedlichen Funktionen. Das Frontend

hat eine Plugin Architektur, welche es einfach erweiterbar macht. Der vorliegende Ansatz wurde im Frontend integriert, sodass MBS basierte SoKNOS Plugins entwickelt werden können. Hierzu wurden ein Swing basierter Interpreter und die Infrastruktur in das Portal integriert, wie in Abbildung 15.2 illustriert.

Die Entwicklung eines MBS basierten SoKNOS Plugins wird durch die in Kapitel 14 vorgestellte Entwicklungsumgebung unterstützt. Vorteilhaft ist dabei, dass auch normale SoKNOS Plugins in der Eclipse Umgebung entwickelt werden.

Alle SoKNOS Plugins erben von einer *Basisklasse*. Diese wurde für MBS basierte SoKNOS Plugins erweitert, so dass das Plugin mit dem Infrastrukturknoten (vgl. Kapitel 13.2) verbunden ist. Darüber hinaus konvertiert und leitet die Basisklasse spezielle SoKNOS Events weiter an das Dialog Set (vgl. Abschnitt 13.2.6). Dabei wird als emittierendes Interaktionsobjekt die aktuelle UUIBox angegebene. Somit kann in den MBS basierten Plugins *mit den Mitteln des hier vorgestellten Ansatzes auf die SoKNOS spezifischen Events reagiert* werden.

Daneben wurden einzelne *Komponenten der Infrastruktur auf das SoKNOS System spezialisiert*. Die Ausführungskomponente (Abschnitt 13.2.5) und der Nodemanager (Abschnitt 13.2.2) wurden an den entsprechenden Stellen im SoKNOS Portal (z.B. Start eines Plugins respektive Start des Portals) integriert. Weiter wurde der Swing basierte Interpreter (Abschnitt 13.3.1) in das Portal eingebaut, indem der Canvas der Plugin Basisklasse auf den des Interpreters umgestellt wurde.

Bei der Integration von Mapache in SoKNOS musste der *Ladevorgang für Klassen in SoKNOS* signifikant geändert werden, um verschiedene Versionen ein und derselben Bibliothek gleichzeitig nutzen zu können. Das Modellierungsframework (Abschnitt 13.2.1) nutzt das Eclipse Modeling Framework (EMF), welches aber auch von der in SoKNOS genutzten SOA (Service Oriented Architecture) Umgebung Apache Tuscany[4] einsetzt wird. Dabei basiert Tuscany auf einer älteren EMF Version, welche inkompatibel mit der Version der prototypischen Umsetzung dieses Ansatzes ist.

Hierfür wurde ein `Separationsclassloader` mit invertierter Ladesemantik konzipiert und umgesetzt. Normaler Weise prüfen Classloader beim Laden einer Klasse, ob ihr Superclassloader diese schon geladen hat. Der konzipierte Separationsclassloader lädt Klassen eines gegebenen Sets von Namensräumen statt dessen direkt, so dass er und seine Unterclassloader ein zum Superclassloader alternatives Set von Klassen im gleichen Namensraum bereitstellen können. Dabei muss jedoch sichergestellt werden, dass Instanzen der separierten Klassen aus dem Separationsclassloader nicht an Instanzen von Klassen aus dem Superclassloader übergeben werden, da diese nicht auf die Klassendefinitionen im Separationsclassloader zugreifen können.

[4] http://tuscany.apache.org

15.3 Folgerung

Die prototypische Umsetzung, genannt Mapache, der in Teil II vorgestellten Konzepte bedient sich der in Kapitel 11 entwickelten Architektur und dem in Kapitel 8 vorgestellten Architekturmuster. Es kann erwartet werden, dass die Nutzer von Mapache gezwungen sein werden, sich eng an das Architekturmuster zu halten. Ebenso ist zu erwarten, dass Teile der Benutzerschnittstelle, ihrer Bindung an Events und ihrer Funktionen im Rahmen einer Verfeinerung weiterverwendet werden können. Maßgeblich hierfür ist die Umsetzung des Beitragsprozessors (Abschnitt 13.2.7) und des Modellierungsframeworks (Kapitel 13.2.1).

Mit Hilfe der in Kapitel 15.2 vorgestellten Integration in SoKNOS können die Konzepte dieser Arbeit in der Fallstudie eingesetzt werden. Durch die enge Anbindung des Interpreter-basierten WYSIWYG Editors an die Entwicklungsumgebung (Kapitel 14.3.1) deutet sich an, dass die UIs mit dem umgesetzten Prototypen gut zu entwickeln sind. Hierbei verspricht die Verknüpfung von Lauf und Entwicklungszeit (Kapitel 14.4) auch Vereinfachungen, da existierende SoKNOS UIKomponenten weiterverwendet werden können und die MBS-Struktur schon zur Entwicklungszeit transparent wird.

Die Umsetzung wurde sehr sorgfältig durchgeführt, was jedoch keine Fehlerfreiheit garantiert, zumal es sich um einen Forschungsprototypen und kein Produkt handelt. Es ist jedoch zu erwarten, dass der erreichte Qualitätsgrad für die Durchführung der Studien gut ausreicht.

Kapitel 16

Fallstudie

Die in Teil II vorgestellten Konzepte werden in diesem Kapitel einer Fallstudie unterzogen. Den Rahmen setzt das in Kapitel 15.1 vorgestellte Anwendungsszenario SoKNOS. Dabei wird mit der Fallstudie der Zweck verfolgt, die durchgängige Anwendung des Ansatzes zu zeigen. Auch soll in einer ersten Stufe die Umsetzung der Anforderungen durch die prototypische Implementierung und das Lösungskonzept begutachtet werden. Dies kann als Vorstufe für die in Kapitel 17 folgende Nutzerstudie verstanden werden, in welcher die Praxisrelevanz zusammen mit Anwendern untersucht wird.

Wie in Kapitel 15.2 beschrieben, wurden die grundlegenden Komponenten (Kapitel 13) der prototypischen Realisierung in SoKNOS integriert. Mit Hilfe der prototypisch umgesetzten Entwicklungsumgebung (Kapitel 14) wurden darauf verschiedene MBS-Varianten für SoKNOS erstellt.

Für die Fallstudie wird in Kapitel 16.1 zunächst auf die für SoKNOS erstellten MBS-Varianten eingegangen. Die Varianten sind voll funktionstüchtig und wurden zum Bearbeiten von Meldungen genutzt. Die Durchführung einer solche Verfeinerung wird im nächsten Kapitel 16.2 am konkreten Beispiel vertieft. Daraufhin werden allgemeine Schlussfolgerungen aus der Umsetzung des Fallbeispiels in Kapitel 16.3 gezogen. Kapitel 16.4 schließlich geht im Speziellen auf den Abgleich mit den Anforderungen aus Kapitel 5 im Kontext der Fallstudie ein (im Gegensatz zu Kapitel 12.1, welches diesen Abgleich rein konzeptuell durchführte).

Die Videodokumentation der Ableitung einer neuen Variante für SoKNOS mit Mapache ist im Internet verfügbar[1].

16.1 Für SoKNOS erstellte MBS-Varianten

Die verschiedenen umgesetzten Varianten der Benutzerschnittstelle für die Meldungsverarbeitung in SoKNOS wurden in dieser Arbeit als laufendes Beispiel

[1] http://www.youtube.com/watch?v=b8h35W903vM

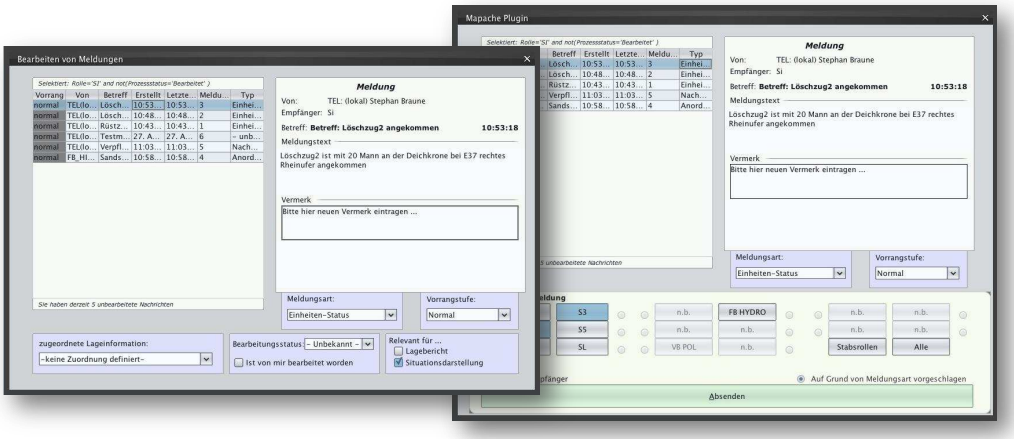


Abbildung 16.2: Die MBS-Varianten *Meldungen* (links) und *Sichter* (rechts).

(Kapitel 7.4) genutzt. Dafür wurden, wie in Abbildung 16.1 zu sehen, 5 Varianten für verschiedene Nutzungskontexte gestaltet. Die verschiedenen Varianten nutzen alle das grafische Swing Toolkit, da dies auch die Basis für das gesamte SoKNOS Portal ist. Dementsprechend werden sie mit dem Swing basierten Interpreter (Abschnitt 13.3.1) zur Interaktion gebracht. Daneben setzt die föderierte Benutzerschnittstelle der Variante "Sichter+Hardware" zusätzlich das Hardware Toolkit ein, dessen Interpreter und Toolkit in Abschnitt 13.3.2 vorgestellt wurden.

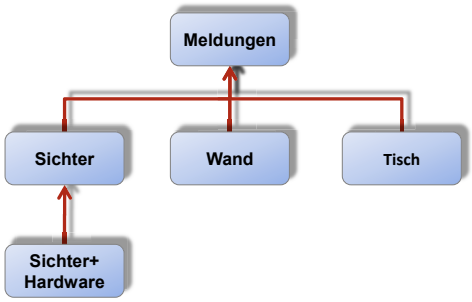


Abbildung 16.1: Verfeinerungsbaum der verschiedenen MBS-Varianten des laufenden Beispiels aus dem Anwendungsfall SoKNOS.

Die umgesetzten MBS-Varianten sind voll funktionsfähig und wurden zur Bearbeitung von Meldungen mit SoKNOS eingesetzt. Die jeweils genutzte Variante kann zur Laufzeit mit Hilfe des Meta UI (vgl. Abschnitt 13.2.8) gewechselt werden. Auf Rückfrage wechselt das System auch nach Einstecken einer Spezialhardware die Variante automatisch. Die Varianten werden im Folgenden kurz vorgestellt.

Meldungen Die Variante Meldungen steht allen Stabsmitgliedern zur Verfügung und beinhaltet generische Funktionalität zur Arbeit mit Meldungen. Sie stellt die Liste mit Meldungen zur Verfügung sowie die Anzeige der aktuellen Meldung im rechten Teil der UI (vgl. Abbildung 16.2). Diese Variante definiert das grundlegende Layout (Struktur) und Verhalten auch für alle anderen Varianten: Sie ist die Wurzel des Verfeinerungsbaumes.

Sichter Die Sichter Rolle wird durch eine spezielle Sichter Variante unterstützt, welche in Abbildung 16.2 zu sehen ist. Die UI zielt darauf ab, den Sichter bei der Bearbeitung seiner primären Aufgabe, dem Weiterleiten von Meldungen, zu unterstützten. Hierfür wurde die Auswahl der Adressaten sehr einfach und schnell durch Knöpfe im unteren Bereich der Schnittstelle gestaltet. Unnötige Funktionalität (im Gegensatz zur abstrakteren Version Meldungen) wurde entfernt. Der Knopf zum Absenden wurde prominent in das Plugin verlagert, da er oft bedient werden muss. Die Verfeinerung beinhaltet somit sowohl Anpassungen der Struktur, als auch des Verhaltens.

Sichter+Hardware Die Verfeinerung der Sichter Variante genannt Sichter+Hardware ist eine föderierte Benutzerschnittstelle. Wie in Abbildung 16.4 zu sehen, hat sie einen grafischen und einen Spezialhardware Teil. Sie lagert die zusätzliche Adressierungsfunktionalität und den Absenden Knopf der Sichter Variante auf die Spezialhardware aus und passt den grafischen Teil an den erweiterten Bildschirmplatz an. Die Verfeinerung beinhaltet somit Anpassungen der Struktur und teilweise Migration auf ein zweites Toolkit.

Wand Zur Unterstützung einer in SoKNOS genutzten großen und hochauflösenden Bildschirmwand wurde die MBS-Variante Wand realisiert. Sie basiert auf der Variante Meldungen, welche aber den Rahmenbedingungen der Wand angepasst wurde. So wurde sie deutlich vergrößert und außerdem in der Struktur verändert, sodass sie breiter, und damit an der Wand einfacher zu bedienen ist. Die Verfeinerung beinhaltet damit nur Anpassungen der Struktur.

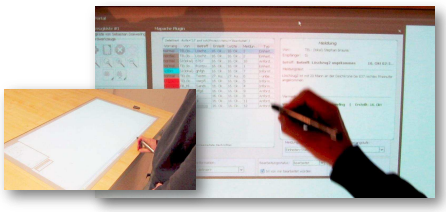


Abbildung 16.3: Die Variante Tisch mit Stiftbedienung.

Tisch Zur Nutzung eines Tisches mit Stiftbedienung wurde die Variante Tisch entwickelt. Neben einer Anpassung der UI an die größere Auflösung des Tisches wurden Felder zur Eingabe ersetzt bzw. ergänzt durch die (simulierte) Möglichkeit der Schrifterkennung auf dem Tisch. Die UI selbst lässt sich mit Hilfe des Stiftes bedienen. Die Verfeinerung beinhaltet sowohl Anpassungen der Struktur, als auch des Verhaltens. Die zusätzlichen Elemente für die Stiftbedienung wurden mit Hilfe einer Modelltransformation (entwickelt von Petter, vgl. Kapitel 12.2.1), welche in der Entwicklungsumgebung integriert ist, eingefügt.

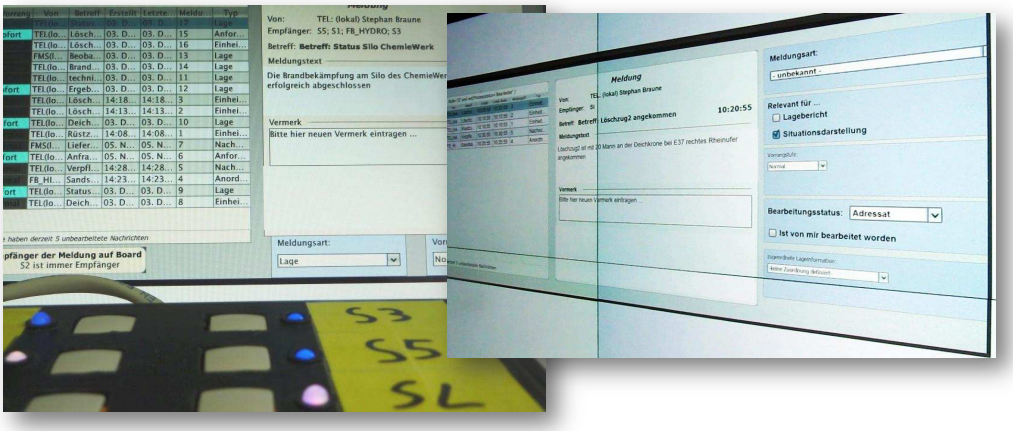


Abbildung 16.4: Die MBS-Varianten *Sichter mit Spezialhardware* (links) und *Wand* (rechts).

16.2 Durchführung einer Verfeinerung am konkreten Beispiel

Abschnitt 16.1 gab eine Übersicht über die verschiedenen erstellten MBS-Varianten mit den dabei notwendigen Anpassungen. Im Folgenden wird auf die Anpassung der Sichter Variante auf die Sichter+Hardware Variante genauer eingegangen, um zu verdeutlichen, was eine solche Anpassung für den Entwickler bedeutet. Hierbei wird auch auf entwickelte Standardvorgehen bei der Umsetzung einer konkreten MBS hingewiesen, Referenzen in die jeweiligen Realisierungskapitel gegeben und der Zusammenhang zu den durch das Beispiel abgedeckten Anforderungen hergestellt (die Abdeckung der Anforderungen wird im Detail in Kapitel 16.4 diskutiert). Die im Folgenden beschriebene Anpassung wurde auch von den Teilnehmern der Studie durchgeführt, wie in Kapitel 17.2.1 beschrieben.

Die Anpassung bedarf dreier Schritte: *i)* dem Erstellen der Verfeinerung, *ii)* dem Überführen der grafischen Knöpfe in Hardwareknöpfe im Modell, sowie *iii)* dem Modellieren der Hardware LEDs und Erstellen einer passenden Beobachterklasse. Die vorgestellte Anpassung stellt nur eine Möglichkeit dar – andere Pfade führen zum gleichen Resultat.

Die im Folgenden genutzten Basisklassen und Entwicklungswerkzeuge sind in den Kapiteln 13 respektive 14 beschrieben. Ihre enge Integration (Unteranforderung 4.3) in die Entwicklungsumgebung Eclipse ist in Abbildung 14.2 auf Seite 172 zu sehen. Die entwickelte MBS kann jederzeit (d.h. sobald sich die entsprechenden Klassen kompilieren lassen) gestartet werden. Es sind keine weiteren Generierungsschritte notwendig, was kurze Iterationszyklen ermöglicht (Unteranforderung 4.4).

i) Verfeinerung Erstellen:

- Im Verfeinerungsbaum (Kapitel 14) wird mit Hilfe eines Rechtsklicks auf die Sichter-Variante eine neue Verfeinerung erstellt. Hierbei wird eine weitere Verfeinerungsebene eingefügt (Unteranforderung 1.1), bei welcher die Geräteföderation konkretisiert wird (Unteranforderung 1.2).

ii) Bestehende grafische Knöpfe auf die Hardware umhängen:

- Es wird im UUIModell der neuen Verfeinerung eine neue UUIBox für die Hardware erstellt (Elementsemantik in Kapitel 9.1), welche die ID bekommt, die das ArduinoBoard (Kapitel 13.3.2) identifiziert.

- Die grafischen Knöpfe werden in die neue UUIBox gezogen und ihr Typ auf Hardwareknopf geändert. Dies ist ein entwickeltes Standardvorgehen, denn hierdurch werden die Verfeinerungsbeziehungen erhalten und die Knöpfe behalten geerbtes Verhalten bei (vgl. Event Orchestration, Kapitel 13.2.7). Dann wird in den Eigenschaften der Knöpfe die jeweilige ID zur Identifizierung des Hardware Pendants angegeben.
  Die Anbindung der Hardwareknöpfe und LEDs folgt streng dem entwickelten Architekturmuster, welches klare Vorgaben zur Umsetzung der Benutzerschnittstelle macht (Unteranforderung 5.4). Hierbei wird die Eingabe (die Hardwareknöpfe) getrennt von der Ausgabe (die LEDs) behandelt. Dieses wichtige Merkmal des vorliegenden Ansatzes ermöglicht im konkreten Beispiel die einfache Wiederverwendung des Knopfverhaltens.

- Auf Grund der nun fehlenden Knöpfe in der grafischen Benutzerschnittstelle, bedarf sie einer Anpassung. Dies geschieht mit Hilfe des Swing Interpreter basierten Editors (Kapitel 14.3.1) durch WYSIWYG Bearbeitung (Unteranforderung 4.2). Die neue MBS-Variante wurde somit manuell im Detail angepasst (Unteranforderung 3.1). Dabei wurden abstrakte (über die verschiedenen Teile der UI Föderation übergreifend) und konkrete Modellierung (Anpassung des konkreten grafischen oder Arduino basierten Teils der UI) eng miteinander verknüpft; ein herausragendes Merkmal des präsentierten Ansatzes.

- Der veränderte Zustand der MBS kann jederzeit in der Verhaltensansicht und mit Hilfe des Verfeinerungsbaumes untersucht werden (Unteranforderung 4.1) – eine Unterstützung, welche den vorgestellten Ansatz von verwandten Ansätzen abhebt. In der Verhaltensansicht erkennt man nun, dass das Verhalten, welches die grafischen Knöpfe geerbt hatten, auch von den Hardwareknöpfen geerbt wird. Es war kein Programmieren notwendig und die Hardwareknöpfe können verwendet werden.

```
/**
 * Führe die Identifikation von Modellelemente im Rahmen
 * von {@link #prepare()} aus.
 */
private void identifyInteractors() {
    // LEDs Modellelemente zusammen mit den ihnen zugewiesenen Adressaten erfassen
    iLedS1 = idAndRegister( "SichterMitBoard.Sichterboard.ledS1", MeldungsKonstanten.ROLE_ID_S1 );
    iLedS3 = idAndRegister( "SichterMitBoard.Sichterboard.ledS3", MeldungsKonstanten.ROLE_ID_S3 );
    iLedS4 = idAndRegister( "SichterMitBoard.Sichterboard.ledS4", MeldungsKonstanten.ROLE_ID_S4 );
```

Abbildung 16.5: Code-Ausschnitt, welcher die LEDs registriert und Adressaten zuordnet.

iii) Hardware LEDs einbinden:

- In der neuen UUIBox werden die LEDs modelliert, wobei in ihren Eigenschaften die jeweilige ID zur Identifizierung des Hardware Pendants angegeben wird.
- Zur Ausgabe über die LEDs wird eine neue Beobachter Beitragsklasse (Kapitel 13.1.2) erstellt. Sie wird von der bestehenden Beobachter Basisklasse abgeleitet und in der MBS-Informationsklasse (Kapitel 13.1.3) registriert. Die Beobachterklasse wird im Folgenden mit Inhalt gefüllt.
- Gemäß entwickeltem Standardvorgehen werden Felder für genutzte Modellelemente definiert und bei Aufruf von `prepare` durch das Framework die Modellelemente diesen zugewiesen. Im konkreten Falle bietet es sich an, gleichzeitig auch die Zuordnung von Adressaten zu LEDs mit zu erfassen, wie in Abbildung 16.5 zu sehen. Unterstützt wird der Entwickler hierbei durch Codevervollständigung mit vollqualifizierten Modellelementnamen (Kapitel 14.3.4). Zu Beachten ist, dass solche Abfragen auch mit dem Fall umgehen müssen, wenn Modellelemente nicht gefunden werden - z.B. wenn eine Variante weiter verfeinert wurde und dabei Elemente gelöscht wurden.
- Ebenfalls als Standardvorgehen entwickelt, werden bei Aufruf von `preShowInterface` durch das Framework eventuell benötigte letzte UI Modifikationen angestoßen. Im konkreten Beispiel ist dies der Fenstertitel, welcher der SoKNOS Pluginklasse zugewiesen wird (da er je nach Variante differiert). Dabei muss überprüft werden, welche Variante interagiert wird, denn ggf. werden Beitragsklassen nicht interagierter Varianten ebenfalls initialisiert.
- Schließlich wird die eigentliche Beobachterklasse innerhalb der Beitragsklasse geschrieben und eingehängt. Diese wird im konkreten Fall von der Basisklasse `SwingUIChangingObserver` abgeleitet, wobei nur noch 3 Methoden implementiert werden müssen: `getIdentifier()` liefert eine ID für den Beobachter zurück. `getObservedChangeCategories()` liefert

(wie in Abbildung 16.6 zu sehen) eine Liste der Änderungskategorien (vgl. Kapitel 8.3) zurück, bei welchen der Beobachter aktiv wird. Und schließlich wird in `executeUIChanges( DataModelChangeEvent arg0 )` die Applikation der UI Veränderung implementiert. Im konkreten Fall wird die `color` Eigenschaft der LEDs gesetzt, je nachdem, ob der ihr zugeordnete Adressat für die Meldung ausgewählt ist und ob der Benutzer den Adressat auswählen kann.

```
public Enum[] getObservedChangeCategories() {
    return new Enum[]{ ChangeCategories.Empfaenger };
}


@Override
protected void executeUIChanges(DataModelChangeEvent arg0) {
    MessageFacade msgFac = arg0.getReference() == null ? null : (M
    Message msg = msgFac == null ? null : msgFac.theMessage;
```

Abbildung 16.6: Code-Ausschnitt eines Beobachter-Fragments: Die Änderungskategorien, auf welche das Fragment reagiert, sowie die Methode zur Aktualisierung der UI sind zu sehen.

16.3 Allgemeine Ergebnisse

Dieses Kapitel behandelt allgemeine Ergebnisse aus der Umsetzung der Fallstudie mit dem Konzept dieser Arbeit. Die Umsetzung wurde zum größten Teil vom Autor selbst durchgeführt. Daneben waren Kollegen und weitere wissenschaftliche Hilfskräfte daran beteiligt.

Herausforderungen bei der Integration in SoKNOS gelöst: Bei der Integration der prototypischen Realisierung in das SoKNOS Portal (geschildert in Kapitel 15.2), wurden zwei zentrale Herausforderungen gelöst. *i)* Die Einbindung des Canvas des Interpreters in das Frontend wurde durch Anpassung der SoKNOS Plugin Basisklasse für MBS Plugins erreicht. Es wird der Canvas des Plugins (die Plugin Klasse leitet von JInternalFrame ab) beim Start des Plugins durch den Canvas des genutzten Interpreters ersetzt. Daneben löst *ii)* ein speziell konzipierter Separationsclassloader die Inkompatibilitäten zwischen verschiedenen EMF Versionen. Der entwickelte Classloader hat hierfür eine inverse Ladesemantik, wie in Kapitel 15.2. beschrieben. Die Nutzung und Entwicklung der MBS in SoKNOS lief daraufhin problemlos.

Integration in SoKNOS ohne Beschränkungen: Bei der Entwicklung der MBS basierten SoKNOS Plugins ist es möglich, *jede beliebige Funktion*

zu nutzen, welche auch von den nicht MBS fähigen Versionen der Meldungsverarbeitung genutzt werden können. Auch *spezielle SoKNOS Events* können verarbeitet werden, wie in Abschnitt 15.2 beschrieben.

Reduzierter Aufwand durch Wiederverwendung: Wie vom Architekturmuster (Kapitel 8) dieser Arbeit intendiert, *konnten Verhaltens- und Strukturspezifikationen von einer auf die andere MBS-Variante vererbt werden.* Dies ermöglichte eine sehr schlanke Anpassung, insbesondere für die Variante, welche nur Strukturänderungen beinhaltet (Variante Wand). Ebenso konnte das Erben von Verhalten von grafischen (Variante Sichter) auf Hardware Interaktionsobjekte (Variante Sichter + Hardware) wie geplant durchgeführt werden, ohne weitere Anpassungen vorzunehmen. *Die Trennung von Ein- und Ausgabe im Architekturmuster hat somit wirklich den gewünschten Effekt* (vgl. Kapitel 8.5).

Verbesserungspotenzial bei Modell-Code-Integration: Es wurde auch klar, dass – insbesondere beim Erstellen von Beobachterfragmenten – man *darauf achten muss, zu prüfen, ob referenzierte Interaktionsobjekte in Verfeinerungen weiter existieren.* Will man z.B. den Inhalt eines Textfeldes aktualisieren, aber dies wurde in einer Verfeinerung entfernt, so sollte das Beobachterfragment darauf vorbereitet sein und vorab prüfen, ob das zu modifizierende Interaktionsobjekt noch vorhanden ist. Da Verhaltensfragmente gegen den Zustand der MBS arbeiten, welcher über alle Varianten konstant ist, trifft sie dies Problem nicht.

Gute Unterstützung der Entwicklung von MBS: Die Entwicklungsumgebung und die in ihr verwirklichten *Unterstützungskonzepte haben die Entwicklung vereinfacht.* Es konnten Aufgaben, welche viel Detailwissen über die prototypische Umsetzung erfordern, von Wizards übernommen werden, so z.B. die Erstellung von Fragmenten und MBS-Informationsklassen (vgl. Kapitel 14). Bei der Referenzierung von Modellelementen brachte die Codevervollständigung enorme Geschwindigkeitsvorteile (Abschnitt 14.3.4), weil man die Namen zum Einen nicht mehr im Modell nachschauen musste und zum Anderen die Namen fehlerfrei eingegeben werden konnten. Die Verfeinerungsansicht wurde im Rahmen der Umsetzung der Fallstudie intensiv für die Navigation und Orientierung eingesetzt und an Hand der Verhaltensansicht überprüft, ob alle notwendigen Änderungen durchgeführt wurden (vgl. Kapitel 14). *Viele dieser Unterstützungen wurden erst durch die klare Definition des Architekturmusters (Kapitel 8) ermöglicht.*

16.4 Abgleich mit den Anforderungen

Dieses Kapitel diskutiert die Erfüllung der in Kapitel 5 erarbeiteten Anforderungen anhand der Umsetzung des Fallbeispiels. Ein konzeptueller Abgleich, welcher die beiträge der verschiedenen Konzepte beleuchtete wurde hierfür wurde bereits in Kapitel 12.1 vorgenommen. Zur Referenz sind alle Anforderungen in kompakter Form in Kapitel 5.7 auf Seite 49 gelistet. Die Nomenklatur zu den Erfüllungsgraden der Anforderungen ist die gleiche, wie die in Kapitel 6.2 zur Bewertung der verwandten Arbeiten genutzte.

16.4.1 Anforderung 1

Anforderung 1 (Abstraktionsebenen) wurde in Kapitel 5.1 erarbeitet.

Sie behandelt die Angabe von Information auf einem für den jeweiligen Anwendungsfall passenden Abstraktionsniveau. Die Umsetzung der Anforderung durch das Lösungskonzept wurde erfolgreich genutzt. So wurde ein Verfeinerungsbaum mit fünf Verfeinerungen passend zum SoKNOS Anwendungsfall erstellt (Unteranforderung 1.1), wie in Abschnitt 16.1 beschrieben. Dabei wurden die Arten der Abstraktion passend gewählt (Unteranforderung 1.2):

- Es wurde von Rollen abstrahiert (Variante Sichter zu Meldungen),
- es wurde von Hardware abstrahiert (Tisch und Wand zu Meldungen) und
- es wurde vom Toolkit abstrahiert (Sichter+Hardware zu Sichter).

Das Verhalten und die Struktur konnten somit auf dem jeweils passenden Abstraktionsniveau gegeben werden. Bei der Verfeinerung wurden Varianten erstellt, welche reiner Strukturänderungen bedurften (Variante Meldungen zu Wand) und Änderungen des Verhaltens (Meldungen zu Sichter). Es wurde aber auch bei einer Verfeinerung das Toolkit geändert, und es wurden neue Beobachter zur Ansteuerung der Leuchtdioden geschrieben, wobei die Verhaltensfragmente komplett unberührt blieben (Variante Sichter zu Sichter + Hardware).

Der Erfüllungsgrad von Anforderung 1 ist somit A 1.2.

16.4.2 Anforderung 2

Anforderung 2 (Erweiterbarkeit) wurde in Kapitel 5.2 erarbeitet.

Die Erweiterbarkeit des Ansatzes, beschrieben in Kapitel 11.4, wurde erfolgreich getestet und genutzt: Die SoKNOS MBS-Varianten basieren u.a. auf speziellen SoKNOS Interaktionsobjekten, welche die Standard Swing Bibliothek erweitern. So wurde auch ein spezielles Swing Panel zur Klassifikation von Meldungen integriert. Diese Spezialkomponenten sind also Erweiterungen eines bestehenden Toolkits (Swing). Die reflektiven Adapter vereinfachten die Erweiterung signifikant und die Spezialkomponenten konnten einfach durch den Swing basierten Interpreter (Abschnitt 13.3.1) zur Interaktion gebracht werden.

Der zweite Fall der Erweiterung, die Erstellung eines komplett neuen Toolkits, wurde mit der Integration der Arduino basierten Hardware gezeigt. Dazu wurde auch ein komplett neuer Interpreter geschrieben, wie in Abschnitt 13.3.2 beschrieben.

Der Erfüllungsgrad von Anforderung 2 ist somit *A*.

16.4.3 Anforderung 3

Anforderung 3 (Modellierungsdetailgrad) wurde in Kapitel 5.3 erarbeitet.

Die verschiedenen MBS-Varianten, welche in Abschnitt 16.1 zu sehen sind, wurden alle im Detail angepasst. Beispielsweise ist die Variante "Wand" *manuell* auf die speziellen Bedingungen einer hochauflösenden Bildschirmwand angepasst worden (Unteranforderung 3.1). Auch konnten beliebige, Toolkit spezifische Details modifiziert werden (z.B. die Rahmen der Elemente oder eine genaue Platzierung) wie in Unteranforderung 3.2 gefordert.

Der Erfüllungsgrad von Anforderung 3 ist somit *A* 3.1.

16.4.4 Anforderung 4

Anforderung 4 (Einfache Nutzbarkeit) wurde in Kapitel 5.4 erarbeitet.

Die in der Entwicklungsumgebung umgesetzten Unterstützungskonzepte sollen durch den Entwickler einfach zu nutzen sein. Eine wichtige Dimension dabei ist die Übereinstimmung von editiertem Artefakt und resultierender Benutzerschnittstelle, d.h. die Nutzung von WYSIWYG Konzepten beim Editieren (Unteranforderung 4.2). Mit Hilfe des Swing Interpreter basierten Editors (vgl. Kapitel 14.3.1) konnte dies erreicht werden.

Mit Hilfe des Interpreters konnten die Änderungen sofort evaluiert werden, was wichtig zur *Unterstützung kürzerer Iterationszyklen* (Unteranforderung 4.4) ist. Auch Änderungen an den speziell für SoKNOS erstellten UI Komponenten konnten direkt evaluiert werden, da die Entwicklungsumgebung dem Modellinterpreter diese per Classloader regelmäßig in aktueller Version übergibt. Ebenso lief das Ausliefern und Installieren neuer Plugin Versionen in das SoKNOS Portal problemlos über ein knappes Ant Script (Kopieren der Modelldatei und des Java Archivs des Plugins), sodass dies sehr zügig durchgeführt werden konnte.

Durch die Einbindung von *Editoren auf verschiedenen Abstraktionsniveaus* (Swing basiert bis generischer grafischer Editor, vgl. Kapitel 14.3) erlaubt der Ansatz es, den Spagat zwischen Arbeiten auf abstraktem und konkretem Niveau zu machen. Dies unterstützt insbesondere die Arbeiten an föderierten Benutzerschnittstellen, wie auch in Anhang B.2.2.1 im Rahmen der Nutzerstudie diskutiert. Dies wurde durch eine enge Integration der Werkzeuge innerhalb der Eclipse Umgebung erreicht (Unteranforderung 4.3), wie in Kapitel 14 beschrieben. So ist das bearbeitete Modell zwischen verschiedenen Editoren synchronisiert, aber auch das aktuell selektierte Element.

Beim Entwickeln konnte über die verschiedenen Unterstützungskonzepte jederzeit der Stand der Multi-Benutzerschnittstelle untersucht werden, sodass dieser transparent wurde (Unteranforderung 4.1). Insbesondere ließ sich mit Hilfe der Verhaltensansicht die *architektonische Schnittstelle* zwischen UI und Verhalten explorieren und feststellen, ob gemachte Änderungen wirksam sind. Lediglich die Prüfung der statischen Semantik konnte nicht getestet werden (Unteranforderung 4.5).

Natürlich kann die prototypische Umsetzung im Bezug auf die Nutzbarkeit in keiner Weise mit einem kommerziellen Werkzeug mithalten. Dies war auch nicht Ziel der Arbeit, es sollten vielmehr die Konzepte getestet werden.

Der Erfüllungsgrad von Anforderung 4 ist somit *A* 4.1, 4.2, 4.3, 4.4, (4.5).

16.4.5 Anforderung 5

Anforderung 5 (Integration Struktur und Verhalten) wurde in Kapitel 5.5 erarbeitet.

Die gemeinsame Formulierung von Struktur und Verhalten wurde in den verschiedenen MBS-Varianten durchgeführt. Beide Aspekte konnten auf jeder Ebene, auch konkreteren (z.B. in der Sichter oder Tisch Variante), angepasst werden (Unteranforderung 5.1). Da sowohl Struktur (Layout), als auch Verhalten innerhalb der eng integrierten Eclipse Entwicklungsumgebung formuliert wurden, war ein schneller Wechsel zwischen den beiden Aspekten möglich (Unteranforderung 5.2). Das Verhalten wurde dabei komplett programmatisch in den Fragmenten spezifiziert (Unteranforderung 5.3) und in Beitragsklassen organisiert, wie in Kapitel 13.1.2 beschrieben. Die einzelnen Bestandteile konnten in das vorgestellte Architekturmuster gut eingeordnet werden (Unteranforderung 5.4). Das umgesetzte Architekturmuster erlaubt auch, wie in Abschnitt 16.3 beschrieben, die intendierte schlanke Anpassung mit der Möglichkeit, Verhalten zu erben.

Die Codevervollständigung nimmt viel Arbeit bei der Referenzierung von Modellelementen aus den Fragmenten ab. Jedoch wäre es wünschenswert, weitergehende Unterstützung zu erhalten: Zum Beispiel, dass die Entwicklungsumgebung *i)* prüft, ob die in einem Fragment referenzierten Modellelemente in der genutzten Verfeinerung (noch) existieren, *ii)* dass eine Möglichkeit zum gemeinsamen Refactoring von Modell und Code bereitgestellt wird und dass *iii)* ein einfacherer Zugriff (z.B. über Wrapper, vgl. Abschnitt 14.3.4.1) auf die Modellelemente und ihre UIKomponenten möglich wäre.

Der Erfüllungsgrad von Anforderung 5 ist somit *A* 5.2, 5.3, 5.4.

16.4.6 Anforderung 6

Anforderung 6 (Modifikation) wurde in Kapitel 5.6 erarbeitet.

Modifikationen werden durch den vorliegenden Ansatz mit seiner prototypischen Umsetzung explizit unterstützt. Neben der Möglichkeit, alle Benutzer-

schnittstellen im Nachhinein zu modifizieren (Unteranforderung 6.1), erlaubt die passive Propagation (Abschnitt 9.1.5), Änderungen direkt auf verfeinernde Varianten zu übertragen. Ebenfalls wirkt sich die Modifikation von Verhaltensfragmenten (passiv) auf erbende Verfeinerungen aus.

Neue Interaktionsobjekte konnten mit Hilfe des passenden Unterstützungskonzeptes (Abschnitt 10.1.4.1) in verfeinernde Varianten aktiv propagiert werden. Dabei erlaubt das Trennen oder Setzen der Verfeinerungsbeziehungen die Kontrolle über die Propagation von Änderungen (Unteranforderung 6.2). Kritisch ist hier zu sehen, dass die Modifikation von Verfeinerungsbeziehungen nur eine indirekte Manipulation darstellt. Wie später in der Nutzerstudie eruiert wurde, ist das direkte Manipulieren von Vorteil (vgl. Anhang B.2.5.1).

Der Erfüllungsgrad von Anforderung 6 ist somit *A*.

16.5 Zusammenfassung

Dies Kapitel stellte die durchgeführte Fallstudie im Rahmen des SoKNOS Projektes vor. Dabei konnte gezeigt werden, dass die prototypische Realisierung des Lösungskonzeptes (Teil II) eine durchgängige Anwendung des Ansatzes ermöglicht. Dazu stellte das Kapitel die für SoKNOS entwickelten MBS-Variante vor (Abschnitt 16.1), gefolgt von einer detaillierten Beschreibung der Entwicklungstätigkeiten im Rahmen einer Verfeinerung am konkreten Beispiel in Abschnitt 16.2. Schließlich wurden auf Basis der Fallstudie zwei Diskussionen zur Bewertung geführt.

Allgemeine Erkenntnisse: In Abschnitt 16.3 wurden allgemeine Erkenntnisse über den Ansatz aus der Umsetzung der Fallstudie gezogen. So konnte gezeigt werden, dass die Integration des Ansatzes in das SoKNOS Projekt erfolgreich war und eine MBS mit mehreren Varianten für SoKNOS mit Hilfe von Mapache erstellt werden konnte. Dabei wurde der Aufwand zur Erstellung der Varianten durch die Wiederverwendung von Verhaltens- und Strukturspezifikationen reduziert. Des Weiteren wurde Verbesserungspotenzial bei der Modell-Code-Integration identifiziert – Beobachterfragmente müssen mit der prototypischen Realisierung abfangen, ob ein referenziertes Strukturelement in einer Verfeinerung entfernt wurde. Schließlich lässt sich resümieren, dass die umgesetzten Unterstützungskonzepte sehr hilfreich waren, jedoch nur durch das strikte Architekturmuster und die strikte architektonische Integration ermöglicht wurden.

Abgleich mit den Anforderungen: In Abschnitt 16.4 schließlich, wurde ein Abgleich mit den Anforderungen auf Basis der Fallstudie durchgeführt. Dieser ist in Tabelle 16.1 kurz zusammengefasst und bestätigt die Ergebnisse aus dem konzeptuellen Abgleich in Kapitel 12.1.

Es kann abschließend festgestellt werden, dass die Fallstudie erfolgreich durchgeführt wurde. Der Ansatz ermöglicht ein hohen Grad an Wiederverwendung (vgl. dazu auch Aussagen der Nutzerstudie in Abschnitt 17.3.4) und unterstützt den Entwickler gezielt bei der Erstellung von MBS, wobei die weitgehende Unterstützung erst durch das strikte Architekturmuster und die strikte architektonische Integration möglich wird. Eine noch stärkere Integration von Modell und Code (z.B. für Refactoring) in der Entwicklungsumgebung birgt weiteres Potenzial. Darüber hinaus wurden alle Anforderungen bis auf die Umsetzung der Prüfung der statischen Semantik adressiert und konnten mit der Fallstudie getestet werden.

Eine abschließende Bewertung des Ansatzes folgt auf die Nutzerstudie in Kapitel 18.

Anforderung	**Erfüllung**	**Anwendung in Fallstudie**
Anforderung 1	*A* 1.2	Abstraktion von Rolle, Hardware und Toolkit in Fall spezifischem Baum
Anforderung 2	*A*	Erweiterung bestehender Sprache (Swing) mit SoKNOS Spezialkomponenten und mit neuer Sprache, dem Arduino Toolkit
Anforderung 3	*A* 3.1	Manuelle Anpassung der MBS-Varianten im Detail
Anforderung 4	*A* 4.1, 4.2, 4.3, 4.4, (4.5)	WYSIWYG Editieren für Swing mit Interpreter-basiertem Editor und Editieren föderierter UI auf multiple Abstraktionsebenen, sowie enge Integration in Eclipse mit Unterstützungskonzepten zur Exploration der Schnittstelle zwischen Struktur und Verhalten
Anforderung 5	*A* 5.2, 5.3, 5.4	Gemeinsame Verfeinerung mit Anpassung von Struktur und Verhalten (z.B. Varianten Sichter, Sichter+Hardware), dabei Unterstützung bei Integration von Modell und Code (z.B. Codevervollständigung) – jedoch mit weiterem Potenzial
Anforderung 6	*A*	Nutzung Propagation für Modifikation Struktur und Verhalten, Änderungen durch Manipulation via Verfeinerungsbeziehungen jedoch indirekt

Tabelle 16.1: Tabellarische Zusammenfassung des Abgleichs mit den Anforderungen auf Basis der Fallstudie.

Kapitel 17

Nutzerstudie

Neben der Fallstudie in Kapitel 16 wurden die Konzepte aus Teil II durch eine Nutzerstudie evaluiert, welche in diesem Kapitel vorgestellt wird. Den Rahmen der Nutzerstudie setzt (wie schon für die Fallstudie) das Anwendungsbeispiel SoKNOS, welches in Kapitel 15 vorgestellt wurde.

Der Aufbau des Kapitels ist wie folgt. Zuerst wird explizit auf die Wahl einer adäquaten Evaluationsmethode in Kapitel 17.1 eingegangen, da dies ein essentieller Schritt der Evaluierung ist. Nach der Beschreibung der eingesetzten Methode in Kapitel 17.2 werden in Kapitel 17.3 die wesentlichen Ergebnisse vorgestellt. Die detailliertere Auswertung hierzu ist im Anhang B zu finden, eine Zusammenfassung der Ergebnisse erfolgt mit der abschließenden Bewertung des Ansatzes in Kapitel 18.

17.1 Wahl einer geeigneten Evaluationsmethode

Im Gegensatz zu anderen Forschungsgebieten, wie z.B. der Mathematik, der Physik oder der Psychologie und den Sozialwissenschaften, ist die Informatik im Bereich *Mensch Maschine Interaktion* (*MMI*, *Human Computer Interaction*, *HCI*) sehr jung und Ihre genaue Zielsetzung noch immer umstritten (Carroll 2009, Dix 2009). Sie hat seit ihrer Entstehung viele Methoden auf dem Bereich der Evaluierung aus anderen Gebieten übernommen, aber bisher keinen auf sich angepassten Korpus von Vorgehensweisen entwickelt (Dix 2009).

Dementsprechend werden auch sehr grundsätzliche Debatten geführt. So diskutierten Liebermann (2003) und Zhai (2003) darüber, wie Evaluationsmethoden angewendet werden. Dies ist verständlich vor dem Hintergrund, dass kein fester Methodenkorpus etabliert ist. So warnen unter anderem Carroll (2009), Dix (2009), Ellis und Dix (2006) daher vor einer “blinden” Übernahme von Methoden, denn dies kann kontraproduktiv sein. Selbst dann, wenn eine Methode der de facto Standard der Community ist, sollte dies überprüft werden (Greenberg und Buxton 2008). *Es sollte somit transparent gemacht werden, warum eine bestimmte Methode gewählt wurde.*

Die Wahl der Methode hat unter anderem auch starken Einfluss auf die Validität der Evaluierung, welche in zwei Aspekte unterteilt werden kann:

Interne Validität: bewertet, ob aus den Beobachtungen wirklich der präsentierte Schluss gezogen werden kann.

Externe Validität: beschreibt, ob man die Schlussfolgerungen auf die gewünschte Zielgruppe wirklich generalisieren bzw. übertragen kann.

Conte, Dunsmore und Shen (1986) stellen dazu fest, dass es *extrem schwierig ist, Experimente zu gestalten, welche hohe interne und externe Validität haben.* Man kann also von einer Art Trade Off sprechen, bei welchem im Rahmen dieser Arbeit versucht wird, einen Mittelweg zu gehen.

Unabdingbar ist es, vorweg klarzustellen, was evaluiert wird und was das *Ziel der Evaluation ist.* So ist beispielsweise bei der Entwicklung neuer Interaktionskonzepte oder für die Entwicklung kommerzieller Anwendungen die Bewertung der Gebrauchstauglichkeit ("Usability") wichtig. Daher gibt es in diesem Bereich auch Normen, wobei die wichtigste wohl ISO 9241 (2012) mit ihren verschiedenen Teilen (bspws. ISO 9241-110 (2006) zur Dialoggestaltung) ist[1].

Das Ziel des Lösungskonzeptes dieser Arbeit (Teil II) ist die Unterstützung bei Entwicklung und Modifikation mehrerer Benutzerschnittstellen einer Anwendung. Das Ziel der vorliegenden Evaluierung ist es, den Nutzen des Konzeptes zu untersuchen und wichtige Faktoren bei der Übertragung des Ansatzes auf praxisnahe Anwendungskontexte zu identifizieren und zu untersuchen. Also sind die Nutzer des Lösungskonzeptes, und damit die relevanten Anwender, berufsmäßige Entwickler von (Multi-)Benutzerschnittstellen. Das Ziel der Evaluierung dieser Arbeit steht somit in Kontrast zu Arbeiten, welche auf die Evaluierung der Gebrauchstauglichkeit abzielen.

Bei der Evaluation ist es auch wichtig, zwischen diesen beiden Zielen – Gebrauchstauglichkeitsproblemen ("Usability") und konzeptionellen Problemen – zu trennen. Dix beschreibt dies in (Dix 2009) mit "Konfusion zwischen Usability Evaluierung und Evaluation der Forschung". Diese Trennung ist nicht scharf, weil offensichtlich Gebrauchstauglichkeitsprobleme auch aus konzeptionellen Problemen entstehen können und ein gutes Konzept mit einem schlecht nutzbaren Werkzeug unbrauchbar gemacht werden kann. Dennoch sollte der Gedanke der Unterscheidung zwischen Gebrauchstauglichkeitsproblemen und konzeptionellen Problemen die Evaluierung leiten. Denn: Das Forschungsergebnis ist ein gutes Konzept, welches evaluiert wurde und kein Produkt, welches eine gute Nutzbarkeit ausweisen muss.

[1] Auch hier wird gerade an einer Norm zu Wahl von Methoden (allerdings für User Centered Design) gearbeitet (Bevan 2009).

17.1.1 Mögliche Vorgehensweisen

Bei grober Betrachtung kann zwischen verschiedenen Vorgehensweisen gewählt werden, welche nun kurz umrissen und in den folgenden Abschnitten genauer diskutiert werden. Neben den genannten gibt es natürlich noch weitere Methoden, z.B. Expertendurchläufe oder Modell basierte Evaluation, welche in einschlägigen Textbüchern auch angeführt werden. Mit den hier vorgestellten werden jedoch die prominentesten und vielversprechendsten Vertreter abgedeckt.

Empirische Vorgehensweisen basieren auf der Sammlung von Beobachtungen (im Labor, im Feld, ...). Dabei können diese *quantitativer Art* sein, z.B. Zeiten oder Fehler. Solche quantitativen Messungen laufen stark kontrolliert ab[2] und können sehr stark belastbare Ergebnisse liefern. Verschiedene Vorgehensweisen sind z.B. in den Arbeiten von Conte, Dunsmore und Shen (1986) sowie Juristo und Moreno (2001) zu finden. Abschnitt 17.1.2 geht näher auf diese Form der Beobachtung ein.

Qualitative Beobachtungen dagegen produzieren keine Zahlen, sondern textuelle Beschreibungen, z.B. des Verhaltens oder eines Diskurses. Diese Form der Beobachtung ist sehr flexibel und ermöglicht die genaue Untersuchung von Gründen. Flick (2009) sowie Monk u. a. (1993) beschreiben verschiedene Vorgehensweisen aus diesem Bereich. In Abschnitt 17.1.3 wird untersucht, ob diese Form der Evaluation für die vorliegende Arbeit in Frage kommt. Ansätze aus beiden Richtungen der Empirie werden auch in Überblicksbüchern behandelt, z.B. in den Büchern von Sharp, Rogers und Preece (2007) oder Dix u. a. (2003). Jedoch bleibt eine tiefergehende Diskussion in Überblicksbüchern i.A. leider aus.

Konzeptionelle Methoden untersuchen den Gegenstand ohne empirische Hinweise, rein auf konzeptioneller Ebene. Dabei spielen Diskussion und Deduktion eine Rolle. Die Autoren Glass, Ramesh und Vessey (2004) sowie Dan R. Olsen (2007) beschreiben solche Vorgehensweisen, welche in Abschnitt 17.1.4 näher behandelt werden.

17.1.2 Quantitative Beobachtungen

Lazar, Feng und Hochheiser (2010) stellen fest, dass wichtige, generalisierbare Resultate durch kontrollierte Experimente (Quantitative Beobachtungen) ermittelt wurden, was mit anderen Methoden nicht möglich gewesen wäre.

Interne Validität: *Quantitative Methoden haben im Allgemeinen eine höhere interne Validität* als qualitative Studien – auf Grund ihrer strengeren mathe-

[2] daher werden diese Untersuchungen auch oft “Kontrollierte Experimente” oder “Laborexperimente” genannt

matischen Auswertung, die weniger Variationsmöglichkeiten zulässt. Das heißt, es lässt sich sehr gut nachvollziehen, wie aus den Experimentdaten das Ergebnis zu Stande kommt. Bei der Auswertung sind normalerweise keine subjektiven Bewertungen (z.B. von Aussagen oder der Relevanz von Faktoren) nötig. Dabei werden Unsicherheiten klar und kompakt durch z.B. Varianzen mit im Ergebnis erfasst. Das Ergebnis gibt eine *klare und knappe (quantitative) Auskunft über das Experiment und kann oft auch von Fachgebietsfremden leicht nachvollzogen werden.*

Der Ansatz eignet sich also *hervorragend für objektive Messung und Quantifizierung von Sachverhalten, Testen von klar abzugrenzenden Hypothesen und Überprüfen statistischer Zusammenhänge.* Wird eine Methode gesucht, bei der systematisch auch die Konfidenz mit ermittelt wird, so sind quantitative Studien die einzige Möglichkeit (Lazar, Feng und Hochheiser 2010).

Beherrschbarkeit der Einflussfaktoren: In Analogie zu Dix Aussagen bzgl. der Evaluierung von Visualisierungstechniken (Ellis und Dix 2006) wird argumentiert, dass eine quantitativ empirisch unterlegte, verlässliche Aussage für den vorliegenden Ansatz jedoch schwer ist. Die *Komplexität der Einflussfaktoren ist so gut wie nicht beherrschbar.* Eine solche Studie im Bereich der Benutzerschnittstellenentwicklung muss viele Variable kontrollieren (z.B. Projekthintergründe der Teilnehmer, Programmiererfahrung, Werkzeugerfahrung, Vorbildung, ...). Auch muss man auf die Frage eingehen, welchen Einfluss die prototypische Umsetzung auf das Ergebnis hat (im Gegensatz zu dem Einfluss des Konzeptes auf die Messung).

Nutzbarkeitsprobleme müssen eliminiert werden, da diese das Resultat verwässern. So kann insbesondere im Rahmen einer Studie nicht davon ausgegangen werden, dass die "Walk up and use" Annahme (Dan R. Olsen 2007) gilt; also dass der Proband das Konzept und die Umsetzung sofort erlernt hat und hierdurch keine Auswirkungen auf das Messergebnis zu erwarten sind. Daher muss die Messung von *Lerneffekten* ausgeschlossen werden, wie auch Dan R. Olsen (2007), Nylander, Bylund und Waern (2003) schreiben. Solche Schwierigkeit wurden auch schon in einer eigenen vorläufigen Studie festgestellt (Behring und Petter 2009).

Das *Argument der Studienkomplexität ist ein übliches gegen quantitative Methoden* (Lazar, Feng und Hochheiser 2010). Juristo und Moreno (2001) lassen dies nicht gelten: "*If we were to be put off by complexity and did not use experiments to try to combat and control it, we would never get a thorough understanding of software development, or, alternatively, SE would never mature.*" Weniger radikal dagegen argumentieren Conte, Dunsmore und Shen (1986), welche ebenfalls Metriken für Software Engineering untersuchen: "*The failure of these [A.d.A.: laboratory] experiments to establish cause-and-effect-relationsships implies that programming is actually more complex than anyone has imagined.*"

Vergleichbarkeit und Generalisierbarkeit des Ergebnisses: Die starke intrinsische Validität bedeutet auf der anderen Seite, dass *Generalisierungen (externe Validität) schwierig sind*, wie in Abschnitt 17.1 beschrieben. Conte, Dunsmore und Shen (1986) schreiben, dass *die Anwendung (einer Metrik) auf neue Situationen ein verzerrtes Bild geben könnte.* So sollten z.B. auf Grund von Metriken erhaltene Ergebnisse nicht blind von Managern als Entscheidungsgrundlage genutzt werden. Es sollte immer auch die Möglichkeit in Betracht gezogen werden, die "mathematische Aussage als irreführend" zu verwerfen.

Die deutlich *größere Objektivität von quantitativen Studien* muss diskutiert werden. Die Auswertung selbst lässt durchaus weniger Toleranz zu und ist sehr objektiv (hohe intrinsische Validität). Jedoch sind detaillierte Kenntnisse über die Rahmenbedingungen der jeweiligen Studie unverzichtbar. Beim Vergleich von Ergebnissen und deren Übertragung auf eine andere Nutzungssituation müssen die Rahmenbedingungen kompatibel sein, sonst kann keine quantitative Aussage abgeleitet werden.

Auch diese starke Abhängigkeit der Ergebnisse von den Rahmenbedingungen kann gemeint sein, wenn Metrik affine Autoren einräumen, dass *quantitative Messungen der Interpretation unterliegen* (Conte, Dunsmore und Shen 1986). Darüber hinaus tritt die Frage auf, ob alle relevanten Einflussfaktoren bedacht wurden und kontrolliert sind. Dies kann nicht mit Zahlen oder in den seltensten Fällen deduktiv behandelt werden. Ebenso wird das Ergebnis (also die gemessenen Zahlen) am Ende auch interpretiert.

17.1.3 Qualitative Beobachtungen

Im Gegensatz zu quantitativen liefern qualitative Beobachtungen keine Zahlen, sondern untersuchen vielmehr Zusammenhänge "warum etwas geht". Hierfür werden Einzelfälle detaillierter untersucht, ggf. auch durch Diskussion mit den Teilnehmern. So nutzen z.B. Plaisant u. a. (1996) informelles Feedback zur Evaluation des LifeLines Ansatzes (eine Visualisierungstechnik). Dennoch dominiert in der Mehrheit der Textbücher der quantitative über den qualitativen Ansatz (Flick 2009).

Interne Validität: *Qualitative Beobachtungen haben eine stark subjektive Prägung.* Auf der Seite des Teilnehmers ist diese sogar erwünscht, denn es werden hierbei einzelne Teilnehmer im Detail untersucht, um Genaueres über die Zusammenhänge und Gründe zu erfahren. Dagegen sollte die Subjektivität auf der Seite des Beobachters auf ein Minimum reduziert sein, was aber nicht einfach möglich ist, da von Anfang an mehr Interpretationsschritte als bei quantitativen Beobachtungen nötig sind.

Dem kann *Nachvollziehbarkeit und Transparenz entgegengesetzt* werden. Im Rahmen der Auswertung muss daher klar nachverfolgbar sein, auf Grund welcher Beobachtungen die Schlussfolgerungen gezogen wurden und warum eine konkrete Methode genutzt wurde.

Beherrschbarkeit der Einflussfaktoren: *Qualitative Beobachtungen können komplexe Ansätze untersuchen.* So schlägt Nylander (2005) auf Grund der Komplexität des Studienziels vor, statt quantitativen Laborstudien qualitative Fallstudien durchzuführen. Dabei ist geplant, Studenten für längere Zeit mit dem Konzept arbeiten zu lassen und sie anschließend zu befragen. Schließlich wird in (Nylander, Bylund und Waern 2003) über die Ergebnisse der Pilotstudie resümiert und festgestellt, dass die Teilnehmer eine gute Leitung bei der Studie benötigen. Diese *benötigte Leitung kann als generelles Problem bei Evaluation komplexer Ansätze bezeichnet werden.*

Vergleichbarkeit und Generalisierbarkeit des Ergebnisses: Ist die Wahl des Ansatzes von Nylander noch nachvollziehbar, so kann über ihr Sampling diskutiert werden. Dix (2009) schreibt, dass er *Studien mit Studenten für nicht zielführend* erachtet, da diese nicht die notwendigen Kenntnisse und Erfahrungen haben. Dazu ist anzumerken, dass dies natürlich vom jeweiligen Themengebiet und den Studenten abhängt, aber für den Bereich des Software Engineering weitestgehend so formuliert werden kann.

Kosara u. a. (2003) stellen fest, dass "*[...] better information can be obtained by, e.g., using a small number of domain experts involved in more qualitative studies.*" Laut den Autoren erlaubt ein (komplexes) anwendungsnahes Setting (i.G. zu stark eingeschränkten, gut kontrollierbaren Experimenten), die Ergebnisse besser zu generalisieren. Dabei müssen die Teilnehmer komplexe Aufgaben unter mehr Freiheiten (als bei kontrollierten Experimenten) lösen. Die Autoren fassen zusammen, dass *die Diskussionsbeiträge der Teilnehmer oft wichtiger als andere Messungen sind.*

Eine noch weitergehende, komparative Studie hält Nylander für schwierig (Nylander, Bylund und Waern 2003). Sie argumentiert, dass diese schwierig durchzuführen sei und es unwahrscheinlich ist, dass sie klare Resultate erzeugt. Die Differenzen der Teilnehmer bei z.B. Projekthintergrund und Erfahrung mit Entwicklungswerkzeugen sowie weitere Probleme mit den Prototypen sowie unterschiedliche Reifegrade der zu vergleichenden Prototypen verzerren die Ergebnisse zu stark.

17.1.4 Konzeptionelle Evaluation

Neben den empirischen Beobachtungen, ist auch eine konzeptionelle Evaluation möglich. Ein klarer Vorteil von ihr ist, dass sie ohne Beobachtungen auskommt. Ist sie angebracht und wird sie sauber durchgeführt, kann das Ergebnis sehr klar eine Hypothese bestätigen oder widerlegen.

Die konzeptionelle Evaluation wurde in einem Survey von Glass, Ramesh und Vessey (2004) als die vorherrschende Methode im Bereich der Software Engineering Forschung identifiziert. Der Studie nach arbeiten 72%[3] der untersuchten Papiere auf diese Weise – viele der verwandten Arbeiten nutzen also

[3] Das im renommierten Communications of the ACM (CACM) veröffentlichte Papier hat

diese Methode. Die Durchführung der Evaluation läuft im Allgemeinen als Diskussion der Vor- und Nachteile ab.

In seinem Papier "Evaluating User Interface Systems Research" (Dan R. Olsen 2007) diskutiert Olsen mögliche Kriterien für die (konzeptionelle) Bewertung von Forschung zu UI Systemen. Jedoch ist für die Generalisierung bzw. Übertragung eines Ansatzes auf andere Nutzungskontexte dennoch ein großes Expertenwissen um die relevanten, zu betrachtenden Parameter nötig.

Dix (2009) schreibt, dass insbesondere die Technik der Deduktion zu wenig genutzt wird. Man muss aber wissen, wo man sie einsetzt. Dix schlägt vor, an weniger eindeutigen Schritten der Deduktion daher gezielt Experimente (empirische Beobachtungen) einzusetzen. Es ist somit schwer, eine aussagekräftige Evaluation allein durch konzeptionelle Techniken zu erhalten, insbesondere, wenn die Schritte nicht im mathematischen Sinne und in ihrer Strenge als korrekt nachvollzogen werden können.

17.1.5 Resümee

Alle betrachteten Techniken haben Vor- und Nachteile. So bergen *empirische Beobachtungen* immer den Aufwand einer Nutzerstudie, wohingegen eine *konzeptionelle Evaluation* (Abschnitt 17.1.4) ohne diese auskommt. Dafür muss sie aber, um gut belastbare Ergebnisse zu produzieren, in einer größeren mathematischen Strenge durchgeführt werden, was oft nicht möglich ist.

Zu empirischen Beobachtungen wurden zwei Unterarten vorgestellt. Die *quantitativen Techniken* (Abschnitt 17.1.2) nutzen Messungen mit anschließender statistischer Auswertung, um ein zahlenmäßiges Ergebnis, welches sehr klare Aussagen zulässt, zu produzieren. Um dies richtig auswerten und vergleichen zu können, ist allerdings eine große Kontrolle über die Parameter des Experiments nötig. *Qualitative Techniken* (Abschnitt 17.1.3) dagegen betrachten Einzelfälle und ermöglichen damit Gründe und Zusammenhänge zu entdecken. Auch ist ihr Einsatz flexibler, da sie eine weniger rigide Kontrolle der Parameter des Experiments benötigen.

Der vorliegende Ansatz ist als sehr komplex zu bewerten, daher benötigen die Teilnehmer eine *Leitung während der Studie*, wie in Abschnitt 17.1.3 beschrieben, und es besteht die *Gefahr von Lerneffekten*. Darüber hinaus ergeben sich auf Grund der sehr geringen Verfügbarkeit des Teilnehmerkreises (Berufsmäßige Entwickler von Benutzerschnittstellen) und der stark divergierenden relevanten Hintergründe (Projekte, Erfahrungen, genutzte Werkzeuge) *Probleme beim statistischen Sampling*. Daher ist für die vorliegende Arbeit eine *qualitative Methode klar zu bevorzugen*. Da sie auch die interaktive Leitung der Teilnehmer ermöglicht, wird für die Studie die Methode der "*Kooperativen Evaluation*" genutzt (vgl. dazu folgendes Kapitel 17.2).

einen rechnerischen Fehler (Grundgesamtheit sind nur ca. 88% an Stelle von 100%). Nach Rücksprache mit dem Autor bleibt die Aussage aber die gleiche.

Neben den genannten Faktoren können die Ergebnisse einer qualitativen Evaluation auch leichter *auf andere Nutzungsszenarien und in die Praxis übertragen werden*, da Gründe und Zusammenhänge untersucht werden. Hierbei ist die externe Validität durch den Expertenstatus der Teilnehmer gesichert. Die interne Validität wird durch die Schaffung maximaler Transparenz der Auswertung weitestgehend hergestellt.

17.2 Methode der Studie

Wie am Ende des vorigen Kapitels resümiert, wird für die Evaluierung der vorliegenden Arbeit die Methode der `kooperativen Evaluation` (Monk u. a. 1993) eingesetzt – eine qualitative Methode (vgl. 17.1.3) zur Evaluation. Hierbei werden explizit die Nutzer nicht von außen (und ohne Eingriff) beobachtet, sondern der Proband und der Beobachter untersuchen gemeinsame "kooperativ" die Software. Während der Proband die geplanten Aufgaben durchführt, halten die Beobachter nach besonderen Vorkommnissen Ausschau, um diese zu notieren und später zu bewerten. Besondere Vorkommnisse sind lt. Monk u. a. insbesondere *i)* Kommentare bzgl. des zu evaluierenden Systems und *ii)* unerwartetes Verhalten des Nutzers. Förderlich ist hierfür eine lockere, kooperative Atmosphäre, welche während der gesamten Studie gewahrt werden sollte, allerdings müssen die Beobachter immer in Kontrolle sein. Ein zu starkes Abdriften ist zu vermeiden.

Dabei muss auch klar kommuniziert werden, dass nach Fehlern im Konzept und nicht in der Bedienung ("des Nutzers") gesucht wird. Weiter wird der Nutzer vor (und ggf. auch im Laufe) der Studie aufgefordert, "laut zu denken" während er mit der Entwicklungsumgebung interagiert. Für auftretende Probleme stehen die Beobachter für Rückfragen zur Verfügung. Dies ist für die vorliegende Arbeit wesentlich, da aufgrund der Komplexität des Themenfeldes nicht davon ausgegangen werden kann, dass die Probanden ohne Anleitung alle Funktionen der prototypischen Implementierung der Konzepte nutzen können.

Nach dieser kurzen Zusammenfassung der grundlegenden Studienmethode wird im Folgenden die Methode zur Evaluation der vorliegenden Arbeit im Detail geschildert. Das heißt, wie die Studie ablief und welche Aufgaben von den Teilnehmern durchgeführt wurden (Abschnitt 17.2.1), wer die Teilnehmer waren (Sampling, Abschnitt 17.2.2), wie die Studie aufgebaut war (Abschnitt 17.2.3) und wie die Auswertung vonstatten ging (Abschnitt 17.2.4).

17.2.1 Aufgaben und Ablauf

Die Teilnehmer mussten verschiedene Aufgaben aus dem Szenarios SoKNOS Projekt (Abschnitt 15.1) durchführen. Dafür wurden Aufgaben entwickelt, welche zentrale Aspekte des Ansatzes einbeziehen; das Aufgabenblatt ist in Anhang A.1 abgedruckt. Die Teilnehmer sollten während der Bearbeitung der Aufgaben ihre Aktionen und Denkvorgänge laut kommentieren.

Die Aufgaben lassen sich wie folgt zusammenfassen:

Probeaufgabe: Ein Knopf in alle Verfeinerungen einfügen, sein Text überall ändern und mit Verhalten hinterlegen.

Aufgabe 1: Ableitung einer Variante für eine Großbildwand.

Aufgabe 2: Hinzufügen eines Drucken Knopfes in allen Varianten, welcher bei Benutzung die aktuelle Meldungsnummer ausgibt.

Aufgabe 3: Ersetzen der grafischen Knöpfe der Sichter Variante durch die Knöpfe auf der Spezialhardware.

Aufgabe 4: Erstellung eines Beobachters, welcher die Leuchtdioden auf der Spezialhardware ansteuert.

Aufgabe 5: Abspeichern der Zeit der Sichtung in der Meldung bei Betätigung des Absenden Knopfes in der Sichter Variante.

Ablauf der Studie Der Ablauf der Studie war wie folgt: Nach einer kurzen Einführung mit Probeaufgabe führten die Experten die Aufgaben aus. Dabei wurde zur *Einführung* das Szenarioblatt (Anhang A.2) und die zusätzlichen Grafiken in Anhang A.3 genutzt. Es wurde insbesondere das Verfeinerungskonzept (Kapitel 9.1.3) und die verschiedenen prototypischen Entwicklungswerkzeuge (Kapitel 14) vorgestellt. Eine *Einführung zur Modellierung von Hardware* fand direkt nach der Pause vor Aufgabe 3 statt

Auf Grund der Studiendauer wurde zwischen Aufgabe 2 und 3 eine Pause gemacht. Aufgaben 4 und 5 waren zu komplex, als dass sie die Teilnehmer 2 Stunden nach Kennenlernen des Ansatzes selbst lösen konnten. Sie dienten vielmehr dazu, den Teilnehmern einen Überblick über das Gesamtkonzept zu vermitteln. Diese Aufgaben wurden daher nicht vollständig ausgeführt, sondern es wurde diskutiert, wie sie mit dem vorliegenden Ansatz umgesetzt werden würden.

Zwischen den Aufgaben wurde mit den Teilnehmern diskutiert, inwiefern der Ansatz sie bei der Bewältigung der Aufgabe unterstützt hat. Vor Aufgabe 1 wurde darüber hinaus explizit gefragt, was die Teilnehmer für die Ideallösung dieses Problems (Anpassung an größeren Bildschirm) halten würden. Am Ende der Studie schließlich wurden die Teilnehmer zur Bewertung der Anforderungen befragt, wobei sie auf die Dimensionen Wichtigkeit und Umsetzung eingehen sollten.

Bei Problemen (speziell Nutzbarkeitsproblemen) wurde den Teilnehmern weitergeholfen. Das gesamte Vorgehen der Evaluatoren wurde in einem Beobachterleitfaden festgehalten, welcher in Anhang A.4 hinterlegt ist.

17.2.2 Sampling der Teilnehmer

Passend zur Methode der Kooperativen Evaluierung mit Ziel das maximale Wissen von den Teilnehmern zu erhalten, wurde das Sampling gewählt. Wie eingangs erwähnt, wurde die Studie mit berufsmäßigen Entwicklern von Benutzerschnittstellen durchgeführt. Diese sind extrem schwer zu rekrutieren, da aber keine statistischen Auswertungen gemacht wurden, wurde keine große Teilnehmeranzahl benötigt.

Ausgewählt wurden insbesondere Entwickler mit Bezug zu Anwendungen auf Mobiltelefonen oder extravaganten Interaktionsgeräten und -techniken. Schlussendlich konnten fünf Experten für die 3 bis 6 Stunden (je nach Teilnehmer) dauernde Studie engagiert werden. Dabei bekamen die Experten keine Kompensation, um Befangenheit zu verhindern. Ihnen wurde versprochen, dass sie die Forschungskonzepte ausgiebig kennenlernen können und Ergebnisse der Studie bekommen.

Die Teilnehmer (5 Personen in 4 Gruppen) lassen sich wie folgt charakterisieren (die Nummerierung startet bei 3, weil Nr. 1 und 2 Vorstudien zugeordnet waren):

Teilnehmer 3: Im Projektgeschäft tätig, vornehmlich Beratung und Konzeption. Starker Web Hintergrund für meist firmeninterne Anwendungen unter Nutzung von meist Visual Studio C#. Teilnehmer 3 sind 2 Personen, einer mit der Erfahrung eines technischen Projektleiters. Beide haben mehr als 5 Jahre Berufserfahrung. Der Schwerpunkt beider Personen liegt im Bereich der Programmierung.

Teilnehmer 4: Reiner Design Hintergrund mit $2^1/_2$ Jahren Berufserfahrung in verschiedenen Projekten auf einer Vielzahl von Endgeräten. Als Werkzeuge kommen hauptsächlich Adobe Produkte zum Einsatz. Programmiererfahrung liegt nur in sehr geringem Umfang vor, ebenso ist das OOP Vererbungskonzept nicht geläufig.

Teilnehmer 5: Im Projektgeschäft tätig, dabei Fokus meist auf Java basierten Desktop und Mobile Anwendungen. Der Teilnehmer gestaltet und setzt die Benutzerschnittstellen gleichermaßen um. Sie werden teilweise unter Visual Studio oder mit Netbeans grafisch erstellt oder in Java unter Eclipse von Hand programmiert.

Teilnehmer 6: Im Projektgeschäft tätigt, dabei eher Leitungsfunktion. Er gestaltet und setzt Benutzerschnittstellen gleichermaßen um. Benutzt werden dabei eine Vielzahl unterschiedlichster Interaktionsgeräte – wobei auch einige MBS Projekte schon umgesetzt wurden. Zum Einsatz kommen sehr diverse Produkte, wie z.B. Visual Studio, Adobe Illustrator, Aptana, Netbiscuit, XCode oder Axure.

Die Ausnahme bildete Teilnehmer drei, welcher aus zwei Personen bestand. Nach dem Design der Studie und den Vorstudien wurde es im Rahmen des Samplings als opportun betrachtet, die Diskussion des Ansatzes durch eine Studie mit mehr als einer Person zu intensivieren. Dies glückte auch, es konnten in der abschließenden Diskussion Themen aus unterschiedlichen Blickwinkeln beleuchtet werden, mehr, als wenn mit nur einem Teilnehmer diskutiert wurde. Auf der anderen Seite wurde während der Studie festgestellt, dass bei diesem Team weniger interessante Vorkommnisse beobachten werden konnten. Es wird davon ausgegangen, dass sich immer derjenige der beiden zurückgehalten hat, der gerade etwas nicht wusste. Somit wurde "das Beste von beiden" getestet.

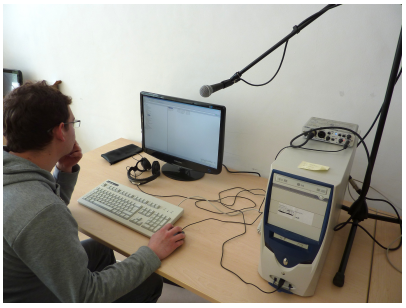


Abbildung 17.1: Der Studienaufbau. Die Evaluatoren saßen links und rechts neben dem Teilnehmer.

17.2.3 Aufbau der Studie

Für die Studie wurde ein PC eingesetzt, auf welchem ein Eclipse in der Version Ganymede mit den Plugins dieses Ansatzes (vgl. Kapitel 14) ausgestattet war. Der Computer wurde bei allen Studien außer für Teilnehmer drei in den Räumen der TU Darmstadt aufgestellt, wie in Abbildung 17.1 zu sehen. Für Teilnehmer drei wurde der Aufbau zur Firma des Teilnehmers transportiert und dort gleichermaßen aufgestellt.

Bei der Studie wurden der *Bildschirm und die Gespräche aufgenommen.* Die Tonaufnahme erfolgte dabei über ein Edirol USB Audiogerät, an welchem ein Mikrofon angeschlossen war, wie in Abbildung 17.1 gezeigt. Das Mikrofon wurde so ausgerichtet, dass der Teilnehmer möglichst zentral hineinsprechen konnte. Die beiden Evaluatoren saßen links und rechts vom Teilnehmer. Ihre Äußerungen wurden somit auch auf der Tonspur festgehalten.

Die Aufnahme erfolgte mit der Software CamStudio[4]. Auf Grund von Limitierungen der Aufnahmegröße musste diese nach jeder Aufgabe kurz angehalten werden, um die Aufnahme zu komprimieren und abzuspeichern. Danach wurde die Aufnahme schnellstmöglich, vor Beginn der Folgeaufgabe, wieder fortgesetzt. Während dieser Pause wurden Gespräche über die Studie möglichst vermieden.

17.2.4 Auswertung der Studie

Abgesehen von zwei Vorstudien, welche zum Testen der Installation und der Methode dienten, wurden die Studien alle mit zwei Evaluatoren durchgeführt. Dabei machten sich beide Evaluatoren Notizen, stellten Fragen an den Teilnehmer und beantworteten die Fragen des Teilnehmers.

[4] http://camstudio.org

Abbildung 17.2: Die zusammengefassten Notizen wurden nach Themen gruppiert.

Interessante Kommentare der Teilnehmer und unerwartetes Verhalten wurden dokumentiert und ggf. direkt hinterfragt: Immer mit Ziel, den Nutzen der Konzepte und ihre Übertragbarkeit auf praxisnahe Nutzungsszenarien zu untersuchen. Dabei wurde von den Evaluatoren versucht, Probleme in der Umsetzung (der prototypischen Implementierung) des Ansatzes von konzeptionellen Problemen des Ansatzes zu trennen.

Als Ergebnis der Auswertung sollte eine Diskussion der als relevant identifizierten Themen stehen (Anhang B) sowie eine Auswertung der Anforderungen in den Dimensionen Wichtigkeit und Umsetzung (Anhang B.2). Bei der Auswertung wurde auf Transparenz Wert gelegt und immer die Quellen angegeben, aus denen ein Schluss gezogen wurde.

Die *Auswertung nach relevanten Themen* im Anschluss an eine Sitzung erfolgte in zwei Schritten:

1. Zu zweit wurden die **relevanten Themen**, vgl. auch "Grounded Theorie Coding", z.B. in Flick 2009, extrahiert.

   (a) Zuerst wurden dafür die Beobachtungen jedes einzelnen Teilnehmers thematisch zusammengefasst. Dies geschah direkt nach der jeweiligen Studiensitzung, damit alle Information noch frisch waren. Interessante Ergebnisse wurden bei der nächsten Studie ggf. wieder hinterfragt bzw. überprüft.

   (b) Dann wurden, nach Durchführung aller Studien, die zusammengefassten Beobachtungen aller Teilnehmer miteinander verglichen und die Blöcke relevanter Themen gebildet.

2. Die so identifizierten Themenblöcke wurden dann wiederum aus allen Zusammenfassungen mit Beobachtungen und Aussagen der Teilnehmer unterlegt und unter den Evaluatoren diskutiert. Dieser Schritt fand nicht gemeinsam statt, aber die Ergebnisse wurden im Evaluatorenteam diskutiert und auf Grund der Diskussion wieder überarbeitet.

Die *Auswertung der Anforderungen* wurde von einer Person durchgeführt, das Ergebnis im Team diskutiert und überarbeitet. Dabei wurden die Notizen zur Diskussion der Anforderungen als Grundlage genutzt und mit Beobachtungen aus dem Rest der Studie angereichert.

17.3 Wesentliche Ergebnisse aus der Studie

Dieses Kapitel fasst die Ergebnisse der Auswertung der Anforderungen und der Auswertung der als relevant identifizierten Themen zusammen. Die zwei

Auswertungen ergänzen sich, da sie unterschiedlichen Strukturen folgten. Die Auswertung der Anforderungen (in Anhang B.1 im Detail nachzulesen) basiert hauptsächlich auf der Diskussion der Anforderungen mit den Teilnehmern nachdem sie alle Aufgaben erledigt hatten (vgl. Ablauf in Abschnitt 17.2.4). Dabei wurde zu jeder Anforderung ihre Wichtigkeit und Umsetzung durch die prototypische Implementierung diskutiert.

Dagegen basiert die thematische Auswertung (in Anhang B.2 im Detail nachzulesen) auf den durch die Studie und durch die Auswertung als relevant identifizierten Themen. Die Belegung der verschiedenen Aussagen wurden als Referenzen auf die zusammenfassenden Notizen angegeben. Diese wiederum sind belegt mit Referenzen auf konkrete Stellen in der Ton- und Bildschirmaufnahme der Studie.

17.3.1 Anforderungen

Es kann resümiert werden, dass die 6 in Kapitel 5 *erhobenen Anforderungen wichtig sind* (vgl. Anhang B.1). Dies wurde insbesondere bei Anforderung 2, Anforderung 3, Anforderung 4 und Anforderung 6 sehr deutlich.

Anforderung 1 wurde bzgl. Wichtigkeit eher gemischt bewertet, was erstaunlich ist, da ihr Korrespondent Anforderung 6 für wichtig befunden wurde. Es ist zu vermuten, dass die Teilnehmer den praktischen Aspekt des Konzeptes (das Propagieren von Änderungen etc.) eher Anforderung 6 zugeschrieben haben, wohingegen Anforderung 1 für das abstrakte Konzept stand.

Anforderung 5 wurde ebenfalls gemischt bewertet. Ein möglicher Grund waren Verständnisprobleme, welche Aspekte der prototypischen Umsetzung und der Studie die Anforderung einbeziehen. Denn prinzipiell wurde von Teilnehmer 4 konstatiert, dass die Anforderung zu Zeitersparnis (Z4.3.4, Z4.2.6) und besserer Konsistenz führen kann (Z4.1.4).

Zusammenfassend wurde die prototypische Umsetzung der Anforderungen gut bewertet. Allerdings hinkt sie kommerziellen Produkten, vor allem im Bereich Erweiterbarkeit und Nutzbarkeit, hinterher. Die Bemängelung der Nutzbarkeit ist verständlich und vollkommen akzeptabel, da es sich um einen reinen Forschungsprototypen handelt. Dagegen wurde insbesondere die Umsetzung von Anforderung 6 sehr positiv bewertet.

> Die Wichtigkeit der Anforderungen wird von den Teilnehmern als mittel bis hoch eingestuft, die Umsetzung als gut bewertet.

17.3.2 Unterstützung der Entwickler

Die in diesem Abschnitt vorgestellten Resultate beziehen sich auf die Unterstützung der Entwickler.

Nutzbarkeit Schon die Fülle der Kommentare zur Nutzbarkeit (nicht nur bei der Diskussion von Anforderung 4, Anhang B.1.4) für Entwickler belegt deren Wichtigkeit. Darüber hinaus haben die Teilnehmer selbst die Nutzbarkeit eines Ansatzes als das Kernkriterium bezeichnet, wie in der Bewertung von Anforderung 4 in Abschnitt B.1.4 beschrieben.

Der vorgestellte Ansatz zeigt einige gute Ideen zur Nutzbarkeit. So wurde die Verfeinerungsansicht von Teilnehmer 4 bei der Bewertung von Anforderung 4 genannt, aber auch die zielgerichtete und komfortable Nutzung der modularen Adaptionskonzepte (Z5.1.5). In Summe steht die Nutzbarkeit aber natürlich kommerziellen Produkten nach.

Speziell Eclipse als Basis der Umsetzung der prototypischen Entwicklungsumgebung ist gewöhnungsbedürftig in der Nutzung. Der Eigenschaftseditor und der grafische, generische Editor (beide sind sehr stark von Eclipse Bedienparadigmen geprägt) stechen hier negativ hervor (Z4.4.3), (Z5.1.1) und Teilnehmer 6 bei Bewertung von Anforderung 3. Teilnehmer 4 resümierte bei der Bewertung von Anforderung 4, dass oft das Konzept selbst nicht schwer ist, nur die Nutzung des Konzeptes. Jedoch ist die Trennung zwischen Nutzbarkeitsprobleme und konzeptionellen Problemen nicht scharf, wie später in diesem Kapitel ausgeführt wird.

In den allermeisten Fällen ist die direkte Manipulation zu bevorzugen (vgl. Anhang B.2.5.1). Sonst "*[...]suchen die Entwickler nach Wegen, die Abstraktion wieder einzufangen*", so Teilnehmer 3 bei Bewertung von Anforderung 4.

> Die Nutzbarkeit ist das Kernkriterium für Teilnehmer. Der Ansatz zeigt laut Teilnehmern gute Ideen für Unterstützungskonzepte. Der umgesetzte Forschungsprototyp steht allerdings (wie erwartet) kommerziellen Produkten deutlich nach.

Nutzbarkeit und Pragmatismus Teilnehmer 6 forderte, dass Entwickler nicht durch ein starres Framework oder ein zu stringentes Architekturmuster an einem pragmatischen Vorgehen gehindert werden (Z6.3.9). Das pragmatisch vorgegangen wird, wurde durch die Studie bestätigt, wie in Anhang B.2.1.1 zu allgemeinen Architekturthemen dargelegt.

In diesem Kontext ist auch bei Bewertung von Anforderung 4 (Einfache Nutzbarkeit, Anhang B.1.4) zu verstehen: Das Editieren soll WYSIWYG artig (Unteranforderung 4.2) geschehen und weniger mit abstrakten Modellen. Dabei ist das *direkte Manipulieren im Gegensatz zur indirekten Manipulation wichtig*, wie in Anhang B.2.5.1 resümiert. So kann indirektes Manipulieren zu Verständnisproblemen führen. Zum Beispiel versuchten Teilnehmer 4,5 und 6 Verhalten direkt per Drag und Drop zu erben. Das Konzept, die Verfeinerungsbeziehungen dafür zu verwenden wurde nicht intuitiv klar (Z51-7) – vermutlich, weil es eine indirekte Interaktion ist.

Der Wunsch nach stärker situativer und mehr WYSIWYG Unterstützung (Anhang B.2.4.1) *soll dafür sorgen, dass in jeder Situation das passende Werk-*

zeug direkt erreichbar ist. So empfand es Teilnehmer 4 als hilfreich, dass viele der Standardoperationen für Interaktionsobjekte über einen Rechtsklick auf das Element im Editor direkt auswählbar sind (Z42-1). Dies *impliziert eine stärkere Integration der Entwicklungswerkzeuge (Unteranforderung 4.3) und -konzepte*, wie in Bewertung zu Anforderung 4 resümiert (Anhang B.1.4).

> Pragmatismus ist für die Teilnehmer wichtig. Die direkte und situativ eingebundene Manipulation ist abstrakten Modellen vorzuziehen.

Visualisierungen Zur besseren Nutzbarkeit wurde von den Teilnehmern auch nach mehr Visualisierungen verlangt, wie in Anhang B.2.4.2 zur Bewertung des Einsatzes von Visualisierungen ausgeführt.

Teilnehmer 3 befand insbesondere die Visualisierung der View Controller Schnittstelle in der Verhaltensansicht für gut (Z33-7). Teilnehmer 5 dagegen bemerkte, dass die *Exposition der Verfeinerungsbeziehungen* in der Verhaltensansicht zum Zweck der Nachverfolgbarkeit gut ist (Z54-1). Weiter bieten, laut Teilnehmer 4 bei Bewertung Anforderung 4, Konzepte wie die Verfeinerungsansicht eine gute Übersicht und Orientierung bei Aktionen wie dem Propagieren von Änderungen. Es lässt sich folgern, dass die Teilnehmer insbesondere die durch Visualisierungen erlangte Transparenz der MBS Struktur (Unteranforderung 4.1) für gut befanden.

Noch *umfangreichere Visualisierungen* regte Teilnehmer 5 an, der den MBS Zustand inkl. Änderungskategorien interaktiv untersuchen wollte (Z52-2). Darüber hinaus schlug er vor, Beobachterfragmente in einer Tabelle zu visualisieren und zu manipulieren (Z52-3). Er gab an, dass es hilfreich wäre, alle Teile einer MBS grafisch darzustellen (Z52-2). Dies lässt auch erkennen, dass *Visualisierungen der Architektur als Grundlage für direkte Manipulationen gut geeignet sind.*

> Die Visualisierungen sind für Teilnehmer hilfreich und bieten eine gute Grundlage zur Umsetzung direkter Manipulation.

Deklarative Unterstützungskonzepte Es wurde an verschiedenen Stellen nach weiterführenden, deklarativen Konzepten verlangt. Dies trat insbesondere in den Auswertungen zu Modellierungspräferenzen (Anhang B.2.1.3), Toolkit übergreifenden Arbeiten (Anhang B.2.2.1) und den weitergehenden Forderungen nach WYSIWYG (Anhang B.2.4.1) auf. Des Weiteren wurde es bei der Auswertung der modularen Adaptionskonzepte (Anhang B.2.6.1) sowie bei der Auswertung von Anforderung 3 (Anhang B.1.3) und Anforderung 4 (Anhang B.1.4) eingebracht.

Die geforderten deklarativen Konzepte sollten aber *dennoch eine direkte (im Gegensatz zu indirekter) Manipulation der Benutzerschnittstelle* ermöglichen, wie in Anhang B.2.5.1 beschrieben. So schlugen Teilnehmer 3, 5 und 6 z.B. in Bezug auf die Skalierung eines UIs mit dem Prototyp der modularen

Adaptionskonzepte vor, ein eher interaktives "Großziehen" der UI zu nutzen, als Skalierungsfaktoren einzugeben (Z32-4), (Z51-3), (Z61-6).

Weiter wurde ein Ausbau der Unterstützungskonzepte gewünscht, welcher z.B. bei der Migration einer MBS-Variante von einem auf ein anderes Toolkit unterstützt. Es sollten *semantisch zusammengefasste Gruppen von Interaktionsobjekten* unterstützt und gemeinsam behandelt werden (Z6.3.4). Interessant ist hierbei, wie sich dieser Vorschlag mit z.B. der Forderung der Teilnehmer nach fein granularer Kontrolle des Layouts (der Struktur) (vgl. Bewertung Anforderung 3, Anhang B.1.3) verbinden lässt.

> Mehr deklarative Unterstützungskonzepte sind von den Teilnehmern gewünscht, welche dennoch direkt manipulierbar sein sollten.

Erweiterbarkeit und Modifikation im Detail Anforderung 2 (Erweiterbarkeit, Anhang B.1.2) und Anforderung 3 (Modellierungsdetailgraf, Anhang B.1.3) wurden von den Teilnehmern einhellig als wichtig bewertet. Kommerzielle Werkzeuge sind bzgl. Erweiterbarkeit sehr weit und sehr brauchbar, erklärte Teilnehmer 6 zur Bewertung von Anforderung 2. Sehr klar war auch seine Aussage zu Anforderung 3: "*Was will ich mit einem Tool, das nichts kann*".

Im Fallbeispiel (Kapitel 16) konnte beides erfolgreich genutzt werden, bei der Studie stand dies jedoch nicht im Fokus der Aufgaben.

> Erweiterbarkeit und Modifikation im Detail ist für die Teilnehmer ein Muss.

Anbindung Verhalten Wie in Anhang B.2.5.2 beschrieben, versuchten alle Teilnehmer bis auf Teilnehmer 4 *Verhaltensfragmente im Eigenschaftseditor aufzufinden* bzw. zu bearbeiten (Z34-5), (Z51-9), (Z62-7). Die Teilnehmer suchten somit im Eigenschaftseditor den Eintrag zur Event-Verarbeitung (Event Handler) des jeweiligen Interaktionsobjektes. Dies kann auf den Hintergrund der Teilnehmer zurückgeführt werden: Alle bis auf Teilnehmer 4 waren mit gängigen Entwicklungswerkzeugen (wie VisualStudio oder Eclipse) vertraut, in denen Event Handler als Eigenschaft von Komponenten dargestellt werden.

> Die Zuordnung von Verhalten wird von den Teilnehmer als Eigenschaft der Interaktionsobjekte gesehen.

Architekturmuster und architektonische Integration Die Nutzung und Akzeptanz des Architekturmuster und des Konzeptes zur architektonische Integration (Unteranforderung 5.4) wurde als relevantes Thema der Studie identifiziert und in Anhang B.2.1 diskutiert. Das Architekturmuster wurde von den Teilnehmern *als weitgehend konsistent* bezeichnet (Z3.5.4), (Z4.3.2), wobei ein Teilnehmer leichte Abstriche auf Grund einer Mehrdeutigkeit machte (Z6.3.8).

Die *Bewertung und Akzeptanz* (Anhang B.2.7.2) *gehen in Bezug auf die Strenge der Konzepte auseinander*. So fühlte sich Teilnehmer 6 durch die Vorgaben und Standardisierungen zu eingeengt und machte einen Vorschlag für ein weniger spezialisiertes Eventkonzept (Z6.1.1). Dagegen bewertete Teilnehmer 3 den (starken) Zwang, sich an ein Framework oder Muster zu halten, positiv (Z3.5.3).

Aus der Literatur ist bekannt, dass mittelfristige Effizienzgewinne durch ein strenges Muster möglich sind (Stahl u. a. 2007). Im Zusammenhang mit dem Lösungskonzept ist es aber auch nützlich für Unterstützungskonzepte. Diese können umso spezifischer werden, je klarer (und strenger) das Architekturmuster bzw. die Architektur ist, da mehr Information standardisiert vorhanden und auswertbar vorliegt. Dies erlaubt wiederum die geforderte, weitergehende Unterstützung mit Visualisierungskonzepten (Anhang B.2.4.2) und die geforderte, bessere situative Integration der Werkzeuge für pragmatisches Arbeiten (Anhang B.2.4.1).

> Die Teilnehmer sehen das Architekturmuster als weitgehend konsistent an, die Akzeptanz der Strenge variiert jedoch.

17.3.3 Verfeinerung und Abstraktionen

Dieser Abschnitt befasst sich mit Ergebnissen zu Verfeinerung und Abstraktion.

Erben Das Vorgehen, eine Benutzerschnittstelle zu verfeinern, wurde sehr positiv aufgenommen. Das Ableiten und darauffolgende Anpassen einer UI ist ein typisches Vorgehen, wie in der Auswertung von Anforderung 6 in Anhang B.1.6 durch Teilnehmer 3, Teilnehmer 5 und Teilnehmer 6 geäußert.

Die Vererbung ist die zentrale Eigenschaft des Ansatzes, so Teilnehmer 5. Und Teilnehmer 3 und 6 fügen hinzu, dass sie "sauber nutzbar" ist und das gewünschte Ergebnis liefert. So kann in der Auswertung von Anforderung 6 (Anhang B.1.6) und der Bewertung des Erbens von Verhalten (Anhang B.2.3.3) geschlussfolgert werden, dass das Vererben den Entwickler unterstützt. Es muss aber bedacht werden, dass der von Teilnehmer 4 und 6 als wichtig erachtete Konzipierungsschritt nicht unmöglich gemacht wird, wie in Anhang B.2.6.2 ausgeführt.

> Die Teilnehmer sehen das Konzept der Verfeinerung sehr positiv und die Umsetzung als "sauber nutzbar".

Multiple Abstraktionsebenen Die Bewertungen der Teilnehmer zu Anforderung 6 in Anhang B.1.6 und zu Anforderung 1 in Anhang B.1.1 stehen bisweilen in Konflikt. Bei Anforderung 6 ist der Mehrebenenansatz und das

Propagiermodell von Teilnehmer 3 und Teilnehmer 6 gut bewertet worden. Dagegen ist laut Teilnehmer 3 bei Auswertung von Anforderung 1 die Nutzung verschiedener Abstraktionsniveaus nicht praxisrelevant.

Es ist zu vermuten, dass die wahrgenommene Funktionalität (das Propagieren und Erben), welche positiv auffiel, eher Anforderung 6 attributiert wurde. Das komplexere Konzept an sich und die Verfeinerungsbeziehungen dagegen könnten Anforderung 1 zugeschlagen worden sein, was darauf hindeutet, dass der gesamte Zusammenhang für die Teilnehmer vielleicht nicht ersichtlich war.

Die Zweifel an der Praxisrelevanz von Anforderung 1 allerdings müssen auch diskutiert werden. So wurde geäußert, dass die Implementierung der UIs, für welche der vorliegende Ansatz Effizienzgewinne verspricht (vgl. Anhang B.2.7.1), nicht der Hauptkostentreiber ist (Teilnehmer 6), da ein vorgeschalteter Konzipierungsschritt deutlich umfangreicher ist. Dies hängt vom konkreten Entwicklungsprozess ab. Auch wird von Teilnehmer 3 nur die Abstraktion vom Toolkit als relevant gesehen, was aber schon durch die Variantenstruktur des Fallbeispieles an Gewicht verliert. Auch zu bemerken ist dabei, dass Anforderung 1 eigentlich auch Kern der (Industrie getriebenen) MDA[5] Initiative ist.

> Die Teilnehmer sehen Konzept der multiplen Abstraktionsebenen nicht als praxisrelevant an, wobei Vererbung und Verfeinerung begrüßt wird.

Denken in Nutzungskontexten Wie in Anhang B.2.3.2 diskutiert wurde, hatten die Teilnehmer Schwierigkeiten, die Aufgabe einer UUIBox zu verstehen. Insbesondere, dass eine UUIBox genau einem Nutzungskontext entspricht. Diese Probleme traten vor allem bei Aufgabe 3, der Modellierung einer föderierten Benutzerschnittstelle, auf (bewertet in Anhang B.2.2.2).

So versuchte Teilnehmer 3 die Aufgabe durch Kombination zweier UUIBoxen zu lösen (Z3.4.1). Dagegen versuchte Teilnehmer 4 bei Aufgabe 1 zwei Varianten in die gleiche UUIBox zu legen (Z41-8). Teilnehmer 4 sagte aus, dass ihm nicht klar sei, dass eine "Box in der Verfeinerungsansicht" tatsächlich dem Modellierungselement UUIBox entspricht (J43-9). Er bezeichnete das UUIBox Konzept explizit als "schwierig" und schlug vor, gegebenenfalls dessen Benennung zu überdenken (A48-12:07).

Es lässt sich resümieren, dass *stark differenzierte Nutzungskontexte für die Teilnehmer aus der Praxis ein nicht geläufiges Konzept* sind. Sie werden bisher nur regelmäßig in der wissenschaftlichen Literatur verwendet. Am naheliegendsten ist dabei die Klassifikation des Kontextes nach Geräten bzw. Toolkits. Eine abstraktere Definition, welche z.B. auch Umgebung, Rolle oder primäre Aufgabe als Parameter mit einschließt, ist dagegen schwerer nachvollziehbar.

> Das Denken in Nutzungskontexten ist für Teilnehmer aus der Praxis gewöhnungsbedürftig.

[5] MDA – Model Driven Architecture, http://www.omg.org/mda

Kombination abstrakter und konkreter Modellierung Der Ansatz schafft den Spagat zwischen abstrakter und konkreter Modellierung der Benutzerschnittstelle. Mit Hilfe des grafischen, generischen Editors wurden erfolgreich Toolkit übergreifende Arbeiten durchgeführt (diskutiert in Anhang B.2.2.1). Teilnehmer 6 fasste zusammen, dass er diese Arbeiten "*angenehm*" fand, sie "*favorisiert*" und bezeichnete sie als "*effizient*" (Z6.3.5).

Auch wurde die Möglichkeit, Interaktionsobjekte aus einem Swing basierten Teil UI in das Hardware basierte Teil UI zu ziehen, als "*genial*" bezeichnet (Z4.2.9) und war für Teilnehmer 5 "*schlüssig*" (Z5.3.2).

Der Swing Interpreter basierte Editor ist mit dem grafischen, generischen Editor synchronisiert, und ermöglicht das direkte Manipulieren des grafischen Teils der Benutzerschnittstelle. Anforderung 3 erhebt dazu, dass Editieren im Detail ein Muss ist, was die Studiendiskussion in Anhang B.1.3 belegt. Auch ist das Arbeiten nach dem WYSIWYG Paradigma laut Anforderung 4 wichtig. Die Diskussion der Anforderung in Anhang B.1.4 belegt dies und wird durch das Thema "mehr Einsatz von WYSIWYG", diskutiert in Anhang B.2.4.1, untermauert. Daher können *abstrakter und konkreter Editor sich nicht gegenseitig ersetzen – aber ergänzen.*

> Der Ansatz ermöglicht durch die Kombination von abstrakter und konkreter Modellierung, Toolkit-übergreifende Arbeiten "angenehm" und "effizient" durchzuführen.

Föderierte Benutzerschnittstellen Insbesondere föderierte Benutzerschnittstellen konnten mit dem Ansatz gut modelliert werden (diskutiert in Anhang B.2.2.2). Auf dem gemeinsamen Abstraktionsniveau wurde die Modellierung von Hardware UIs (das Sichterboard aus Aufgabe 3 und 4) identisch zu der von grafischen UIs (Z3.4.9), wobei die grafischen UIs noch im Detail über den Swing Interpreter basierte WYSIWYG Editor angepasst werden konnten.

Jedoch muss bzgl. der Intuitivität des UUIBox Konzeptes nachgedacht werden, da dies, wie in Anhang B.2.3.2 diskutiert, zu Verständnisproblemen führen kann. Auch scheinen bestimmte Nutzergruppen Rekombinationsansätze an Stelle des gewählten Verfeinerungsansatzes zu bevorzugen, was jedoch nutzerspezifisch ist, wie in Anhang B.2.1.3 eruiert. Andere Teilnehmer hielten dagegen die vom Ansatz vorgegebene Anordnung der UIs zur Modellierung föderierter Benutzerschnittstellen für sinnvoll (Z5.2.5).

> Der Ansatz bietet Vorteile bei der Modellierung föderierter Schnittstellen, wobei einige Teilnehmer auch Rekombinationsansätze bevorzugen würden.

17.3.4 Arbeitsablauf

Die hier vorgestellten Ergebnisse beziehen sich auf die Auswirkungen auf den Arbeitsablauf bei der Entwicklung.

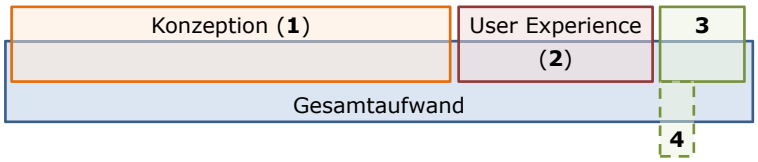


Abbildung 17.3: Schaubild erstellt von einer Zeichnung des Teilnehmers 6 unter Erhaltung der relativen Größen der Einzelteile. Es zeigt das Verhältnis von Konzipierungsaufwand (Bereiche 1 und 2) zu Erstellungsaufwand (traditionell Bereich 3 bzw. mit Verwendung des vorliegenden Ansatzes Bereich 4) bei der Entwicklung von MBS.

Implementierungsaufwand Wie von Teilnehmer 6 angemerkt wird der Konzipierungsschritt vom Ansatz nicht betrachtet (vgl. Anhang B.2.6.2). Aber laut Teilnehmer 6 *fällt bei diesem Konzipierungsschritt der Hauptteil des Aufwandes an, und nur ein geringer Anteil wird für das tatsächliche Erstellen der Benutzerschnittstelle gebraucht* (Z61-3). Zur Veranschaulichung der Verhältnisse der einzelnen Aufwände malte er Abbildung 17.3. Dabei illustriert Bereich 4 den *Erstellungsaufwand mit dem vorliegenden Ansatz, welcher nach Meinung von Teilnehmer 6 tatsächlich im Gegensatz zum herkömmlichen Vorgehen (Bereich 3) reduziert* wird.

Da aber nur Teilnehmer 4 und 6 auf den Konzipierungsschritt eingingen bzw. ihn durchführen wollten, lässt sich schließen, dass *der Effizienzgewinn für den Gesamtprozess stark von der jeweiligen Arbeitsweise abhängt. Der Umsetzungsaufwand an sich kann aber signifikant reduziert werden.*

> Der Implementierungsaufwand für Multi-Benutzerschnittstellen kann laut Teilnehmern signifikant reduziert werden, wobei der umfangreiche Konzipierungsschritt vom Ansatz nicht betrachtet wird.

Teamorganisation bei der Entwicklung Der typische Projektarbeitsfluss in der Firma von Teilnehmer 6 bedingt eine starke Trennung zwischen Designer und Entwickler, welche auch bestehen bleiben soll (vgl. Bewertung von Anforderung 4, Anhang B.1.4). Der vorliegende Ansatz unterstützt die Trennung nicht in ausreichendem Maße und so müssen nach Einschätzung von Teilnehmer 6 Designer und Entwickler im Tandem arbeiten, was in der Praxis nicht möglich ist.

Im Rahmen des Lösungskonzeptes wurde auch der Konzipierungsschritt für neue MBS-Varianten nicht explizit mit integriert (vgl. Anhang B.2.6.2). Daher sollte für eine weiterführende Arbeit diese Frage explizit adressiert werden.

> Die Aufhebung der Trennung zwischen Entwicklung und Designer durch den Ansatz wird von einem Teilnehmer kritisiert.

17.3.5 Reflektion zur Auswertung

Im Folgenden werden zwei Aspekte der Auswertung reflektiert.

Nutzbarkeitsprobleme und konzeptionellen Problemen Es wurde bei der Durchführung der Studie festgestellt, dass Probleme der Nutzbarkeit und Probleme mit dem Konzept stark verschränkt sind. Teilnehmer 4 formulierte hierzu, "*[...] oft ist ein Konzept an sich nicht schwer, aber die Nutzung des Konzeptes birgt Hürden*" (Z4.3.8). Mit einer guten Benutzerführung können konzeptionelle Probleme kaschiert bzw. behoben werden. Mit einer schlechten Benutzerführung kann aber ein gutes Konzept auch unbenutzbar gemacht werden.

Bei der Auswertung wurde das Problem der Entscheidung auf die Abschätzung des Änderungsaufwandes zurückgeführt: Erhöhter Änderungsaufwand bedeutet, dass mehr konzeptionelle Anpassungen nötig sind. Somit wurde versucht, das Gros der Nutzbarkeitsprobleme mit sehr geringem konzeptionellem Anteil vorab schon zu filtern.

> Auf Grund ihrer starken Verschränkung sind Nutzbarkeitsprobleme und konzeptionelle Probleme schwer zu trennen.

Wahl der Methode Abschließend kann resümiert werden, dass die *Wahl der Methode* "Kooperative Evaluation" sinnvoll gewesen ist. Die Methode erlaubt es, während der Studie Nachfragen zu dem sehr komplexen Thema zu beantworten, ohne die die Teilnehmer nie durch alle Aufgaben gekommen wären. Es wäre somit ein deutlich kleineres Spektrum abgedeckt worden. Darüber hinaus ermöglichte die Beantwortung der Rückfragen, dass die Teilnehmer fast den gesamten Ansatz kennengelernt hatten. Dies erst machte die sehr wertvolle Diskussion im Anschluss an die Aufgaben möglich.

Während der gesamten Studie wurde von den Evaluatoren Wert darauf gelegt, ihre Objektivität zu wahren. Dazu wurden *i)* die Notizen frühzeitig überarbeitet und zusammengefasst, sodass die konkreten Situationen fast immer noch direkt vor Augen waren, und *ii)* möglichst so notiert und zusammengefasst, dass keine Mehrdeutigkeit auftreten konnte und die Interpretation möglichst eindeutig war. Schließlich wurden *iii)* die Studien per Bildschirm- und Tonaufzeichnung festgehalten und archiviert.

> Die Wahl der Methode "Kooperative Evaluation" war für die gegebene Aufgabenstellung sinnvoll.

17.3.6 Übertragen der Ergebnisse auf anderen Kontexte

Die gewählte Methode war nicht auf eine statistische Aussage mit einer repräsentativen Gruppe ausgelegt. Somit wurde auch das Sampling eher breit

angelegt. Eine allgemeingültige Aussage für eine solche Gruppe wäre auch auf Grund der Komplexität nicht möglich gewesen (vgl. Kapitel 17.1).

Stattdessen konnten relevante Faktoren identifiziert werden, die die Anwendbarkeit des Ansatzes auf andere Nutzungsszenarien beeinflussen. Die als relevant identifizierten Themen wurden im vorangegangenen Abschnitt 17.3 vorgestellt. Die detaillierten Betrachtungen in Bezug auf diese Themengebiete sind in Anhang B.2 zu finden. Die Betrachtungen in Bezug auf die Diskussion von Anforderungen sind in Anhang B.1 nachzulesen.

Zur Übertragung des Ansatzes auf ein anderes Nutzungsszenario müssen die vorgestellten Themen im Einzelfall für das jeweilige Nutzungsszenario überprüft und bewertet werden. Insofern ist *das Ergebnis der Studie nicht die Generalisierung per se (nicht möglich), sondern die Identifikation von wichtigen Faktoren bei der Anwendung des Ansatzes auf einen neuen Anwendungskontext.*

Die Zusammenfassung der Ergebnisse erfolgt zusammen mit einer abschließenden Bewertung in Kapitel 18.

Kapitel 18

Zusammenfassende Bewertung

Der in dieser Arbeit entwickelte Lösungsansatz (Teil II) wurde in einer prototypischen Realisierung (genannt Mapache) umgesetzt und evaluiert (Teil III). Die Evaluation beinhaltete einen Abgleich mit den Anforderungen auf konzeptueller Ebene und im Rahmen einer Fallstudie. Beide bestätigten, dass der Ansatz die erhobenen Anforderungen erfüllt. Darüber hinaus konnte mit der Fallstudie gezeigt werden, dass der Ansatz die *notwendige durchgängige Unterstützung, um eine Multi-Benutzerschnittstelle zu entwickeln und zur Laufzeit auszuführen* bietet. Eine qualitative Nutzerstudie ergab schließlich eine positive Bewertung des Ansatzes durch berufsmäßige Entwickler von Benutzerschnittstellen sowie weiterführende Ergebnisse.

Abgleich mit den Anforderungen

Es wurden zwei unterschiedliche **systematische Abgleiche** des vorliegenden Lösungsansatzes mit den anfangs erhobenen Anforderungen (Kapitel 5) durchgeführt. In Kapitel 12.1 wurde konzeptuell überprüft, ob der Ansatz die Anforderungen erfüllt. Dabei stellte sich heraus, dass alle Anforderungen gemäß den in Kapitel 6.2 beschriebenen Erfüllungsgraden voll erfüllt sind. Einzige Ausnahme bildet hierbei Unteranforderung 4.5, die Prüfung der statischen Semantik, zu der zwar Regeln formuliert wurden, jedoch die Umsetzung ausgelassen wurde.

Dieses Ergebnis wurde durch den praxisnäheren Abgleich mit den Anforderungen im Rahmen der Fallstudie in Kapitel 16.4 bestätigt. Hierbei konnte der Ansatz erfolgreich eingesetzt werden, um eine Multi-Benutzerschnittstelle für das SoKNOS System zu erstellen.

Im Rahmen der Nutzerstudie mit berufsmäßigen Entwicklern von Benutzerschnittstellen wurde schließlich die **Wichtigkeit** der erhobenen Anforderungen systematisch hinterfragt. Die Teilnehmer der Studie stuften diese als (in unterschiedlichem Maße) mittel bis hoch ein. Eine Diskrepanz in der Bewertung von Anforderung 1 (Abstraktionsebenen) und Anforderung 6 (Modifikationen) lässt jedoch darauf schließen, dass die in den Diskussionen mit den Teilneh-

mern identifizierten Vorteile in der Praxis für den Entwickler nicht direkt aus dem theoretischen Konzept ersichtlich sind. Die stark variierende Bewertung von Anforderung 5 (Integration Struktur und Verhalten) untermauert die These, dass das Denken in Nutzungskontexten für Anwender aus der Praxis nicht gängig ist.

Die **Umsetzung** der Anforderungen im Prototyp wurde von den Teilnehmern der Studie weitgehend positiv bewertet. Insbesondere Anforderung 6 (Modifikation) wurde voll erfüllt. Dagegen konnte der Prototyp bei Anforderungen, welche starken Bezug zu Nutzbarkeit hatten, nicht mit professionellen Entwicklungswerkzeugen mithalten, was jedoch zu erwarten war.

Durchgängige Unterstützung und Lauffähigkeit

Für die Fallstudie wurde eine Multi-Benutzerschnittstelle entwickelt, welche fünf verschiedene MBS-Varianten für unterschiedliche Nutzungskontexte bereitstellt (Kapitel 16.1). Hierfür wurde die prototypische Umsetzung in das SoKNOS System erfolgreich integriert und die MBS konnte im Rahmen von SoKNOS zum Einsatz gebracht werden. Wie in Kapitel 16.3 ausgeführt, wurde die Erstellung durch die entwickelten Konzepte gut unterstützt und die im Ansatz konzipierte schlanke Anpassung an neue Nutzungskontexte wurde erfolgreich eingesetzt.

Das Fallbeispiel wurde hierbei in einem großen anwendungsorientierten Projekt (SoKNOS, Kapitel 15.1) mit mehreren Anwendungspartnern entwickelt, was die Qualität und wirkliche Lauffähigkeit des Prototyps sicherstellt. So konnte mit Hilfe der Fallstudie die durchgängige Anwendbarkeit und Lauffähigkeit des Ansatzes gezeigt werden.

Qualitative Ergebnisse der Nutzerstudie

Schließlich wurde im Rahmen einer Nutzerstudie (Kapitel 17) der Ansatz von *berufsmäßigen Entwicklern von Benutzerschnittstellen* bewertet. Dabei wurden wichtige Punkte identifiziert, die beim Transfer des Ansatzes auf andere praxisrelevante Nutzungsszenarien zu beachten sind. Diese Punkte wurden mit Hilfe der Methode "Kooperative Evaluation" in Zusammenarbeit mit den Experten identifiziert und diskutiert. Somit ist die notwendige *Praxisrelevanz* gegeben. Dabei wurde bei der Anwendung der Methode auf *Transparenz und Nachvollziehbarkeit* Wert gelegt, sodass subjektive Bewertungen möglichst ausgeschlossen sind (vgl. Diskussion dazu in Kapitel 17.1.1). So wurde auch die *Wahl der Evaluationsmethode* in Kapitel 17.1 transparent gemacht, was, zusammen mit dem Fakt, dass professionelle Anwender zur Bewertung des Ansatzes herangezogen wurden, deutlich über die Vorgehensweise anderer verwandter Arbeiten hinausgeht.

Die **wesentlichen Ergebnisse** der Nutzerstudie sind in Kapitel 17.3 dargelegt und stehen im Detail in Anhang B zur Verfügung. Zusammenfassend wurde

festgestellt, dass der vorliegende Ansatz es insbesondere sehr gut erlaubt, das *Arbeiten auf unterschiedlichen Abstraktionsebenen* zu kombinieren. Dies ist insbesondere für die Arbeit an föderierten Benutzerschnittstellen von Vorteil. Die *Verfeinerung und Vererbung* wurde von den Anwendern positiv begrüßt. Ebenso wurden die auf dem strengen Architekturmuster (Kapitel 8) basierenden *Visualisierungen, welche Schnittstellen in der Architektur transparent machen* (vgl. Kapitel 10.2), sehr positiv bewertet, was die Einschränkungen durch die *Strenge des Musters* nivelliert. Dabei führt der Ansatz voraussichtlich zu *Effizienzgewinnen* bei der Umsetzung von MBS im Gegensatz zu bestehenden Vorgehensweisen. Jedoch ist das *Denken in Nutzungskontexten* in der Praxis noch nicht so weit verankert, dass mit dem Konzept einfach gearbeitet werden kann. Da das Konzept des Nutzungskontextes aber nur bedingt komplex ist, ist zu vermuten, dass eine mittelfristige Beschäftigung mit dem Thema dieses Problem behebt.

IV

Fazit

Überblick über Teil IV

Techniken und Werkzeuge zur Entwicklung von Benutzerschnittstellen sind wichtig, da Benutzerschnittstellen einen großen Teil der Anwendungsentwicklung ausmachen. Insbesondere mit der zunehmenden Anzahl an mobilen Endgeräten und deren Diversität stehen Entwickler jetzt schon vor dem Problem, dass *eine Anwendung mehrere Varianten ihrer Benutzerschnittstelle bereitstellen muss* – je nach Nutzungskontext. Wie in Kapitel 1 eingeführt, stellt sich somit die in dieser Arbeit adressierte Frage nach der Unterstützung bei der Entwicklung solcher Multi-Benutzerschnittstellen (MBS).

In der vorliegenden Arbeit wurde aufbauend auf eine fundierte Begriffsdiskussion und Diskussion der Anforderungen ein Lösungskonzept für Multi-Benutzerschnittstellen und zur Unterstützung deren Erstellung entwickelt. Somit können *multiple Benutzerschnittstellen für eine Anwendung entwickelt und bereitgestellt* werden. Dieses Kapitel rekapituliert die Ergebnisse der vorliegenden Arbeit und gibt einen Ausblick auf mögliche Anschlussforschung.

Kapitel 19

Ergebnisse der Arbeit

In Kapitel 1.1 wurde ein einführender Überblick über die Beiträge dieser Arbeit gegeben. Im Folgenden werden die einzelnen Ergebnisse der Arbeit zusammengefasst.

In Teil I (Analyse des Standes der Wissenschaft)

Begriffe und Anforderungen Im Rahmen der vorliegenden Arbeit wurde eine tiefgehendere Diskussion der Begriffe und Anforderungen gegeben als in verwandten Arbeiten üblich. Bei der **Begriffsdiskussion** (Kapitel 4) wurde dargelegt, dass die Begriffe der abstrakten und konkreten Benutzerschnittstelle in der Literatur inkonsistent gebraucht werden. Insbesondere stellt sich die Frage, von was bzw. wie weit eine abstrakte Benutzerschnittstelle abstrahiert. Diese Inkonsistenzen wurden im Rahmen der Arbeit behoben, indem die Dichotomie zwischen AUI und CUI durch die Definition einer abstraktionsunabhängigen Benutzerschnittstelle (UUI) aufgelöst wurde. Diese legt den Abstraktionsgrad nicht fest, sodass dieser je nach Anwendungsfall gewählt werden kann.

Die sechs erhobenen **Anforderungen** (Kapitel 5) basieren auf umfangreichen Literaturstudien und Studien von verwandten Ansätzen. Sie adressieren die Bereiche Abstraktion, Erweiterbarkeit, Modellierungsdetailgrad, Einfache Nutzbarkeit (der Entwicklungswerkzeuge), Integration Struktur und Verhalten, sowie Modifikation. Dabei nimmt der Bereich Abstraktion das Thema der UUI wieder auf und der Bereich zur einfachen Nutzbarkeit fasst Anliegen des Entwicklers im Bezug auf seine Werkzeuge zusammen. Die Anforderungen wurden durch Unteranforderungen detailliert und erheben Gültigkeit über die vorliegende Arbeit hinaus.

In Teil II (Konzept)

Architekturmuster Das vorgestellte Architekturmuster (Kapitel 8) basiert auf MVC, wurde aber im Hinblick auf Multi-Benutzerschnittstellen angepasst.

Eine stärkere Modularisierung mit Hilfen von Beobachter- und Verhaltensfragmenten ermöglicht Teile der Verhaltensbeschreibung wieder zu verwenden bzw. gezielt auszutauschen. Darüber hinaus wird das Konzept der Verfeinerung definiert, welches als Grundlage für die weiteren Teile der Arbeit diente. Es definiert die Schnittstelle zwischen Struktur und Verhalten einer Benutzerschnittstelle, also wie die modellierte Benutzerschnittstelle mit dem programmierten Verhalten zusammenarbeitet. Das Muster beschreibt auch, wie sich die Verfeinerung von Struktur und Verhalten in Artefakten niederschlägt.

Die Evaluation bestätigte, dass das Architekturmuster klar und sehr strikt ist, wodurch Unterstützungskonzepte speziell darauf zugeschnitten werden können. Es sorgt also für deren reibungslose Integration (Anforderung 5). Insofern ist das Architekturmuster zentral und auch die Grundlage für das weitere Lösungskonzept. Trotz der Wichtigkeit eines Architekturmusters, wird es in verwandten Arbeiten selten ausgeführt, wodurch sich die vorliegende Arbeit abhebt.

Domänenspezifische Sprache (DSL) Die vorgestellte DSL (Kapitel 9) zur Beschreibung der Struktur von Benutzerschnittstellen (UUI-Modelle) ist auf das Architekturmuster abgestimmt und besteht aus den Teilen Semantik, abstrakte Syntax und Wohlgeformtheitsregeln. Die konkrete Syntax wird durch die Unterstützungskonzepte bestimmt.

Die DSL integriert beliebig viele UUI Modelle von MBS-Varianten in einem Verfeinerungsbaum, wobei beliebige Abstraktionen je nach Anwendungsfall unterstützt werden (Anforderung 1). Mit Hilfe der in der Sprache vorhandenen Verfeinerungsbeziehungen werden die Varianten und ihre Konstituenten verbunden. So ist der Entwickler in der Lage zu kontrollieren, wie eine Modifikation auf mehrere MBS-Varianten angewendet wird. Durch die damit verbundene passive Propagation wird der Entwickler explizit beim Modifizieren von MBS-Varianten unterstützt (Anforderung 6). Darüber hinaus können mit der Sprache beliebige Eigenschaften von Elementen abgebildet werden, und somit kann sie frei mit neuen Modellierungselementen erweitert werden (Anforderung 2).

Unterstützungskonzepte Basierend auf den erhobenen Anforderungen wurden verschiedene Unterstützungskonzepte mit Fokus auf der einfachen Nutzbarkeit (Anforderung 4) für den Entwickler konzipiert (Kapitel 10). Dafür wird die Standardisierung der Artefakte durch das Architekturmuster und die architektonische Integration ausgenutzt, um möglichst konkrete Hilfen zu bieten. Es wurde stark auf Visualisierung von Zusammenhängen gesetzt, z.B. die Struktur des Verfeinerungsbaumes oder die Vererbungsbeziehungen zwischen Verhalten der einzelnen MBS-Varianten, was die Studienteilnehmer sehr positiv bewerteten.

Interaktives Unterstützungskonzepte erlauben die Modifikation einer MBS-Variante. Zentral hierbei ist der Modellinterpreter, welcher UUI-Modelle direkt

zur Interaktion bringt. Er wird durch ein Editierkonzept zum WYSIWYG-artigen Editor und erlaubt es dem Entwickler somit, die Benutzerschnittstelle manuell im Detail anzupassen (Anforderung 3). Dieser Ansatz ermöglicht auch das Editieren der Benutzerschnittstelle zur Laufzeit. Auch die modularen Adaptionskonzepte nutzen ihn und stellen einzelne Werkzeuge für Anpassungstätigkeiten modular zur einfachen Nutzung bereit. Weitere Verfeinerungsbaum basierte Konzepte erlauben u.a. die aktive Propagation von Modifikationen zu mehreren MBS-Varianten.

Explorative Unterstützungskonzepte hingegen machen den Entwicklungsstand der MBS für den Entwickler transparent (Unteranforderung 4.1). So stellt die Verfeinerungsansicht den Verfeinerungsbaum dar, kann aber auch zur Untersuchung der Verfeinerungsbeziehungen einzelner Interaktionsobjekte herangezogen werden. Die Verhaltensansicht hingegen zeigt die architektonische Schnittstelle zwischen Interaktionsobjekten und Verhaltensfragmenten. Mit ihrer Hilfe kann der Entwickler die Vererbung von Verhalten über MBS-Varianten hinweg untersuchen.

Architektonische Integration Um die Umsetzbarkeit des Lösungskonzeptes zu gewährleisten, wurde eine Architektur für Lauf- und Entwicklungszeit entwickelt (Kapitel 11). Durch ihre Festlegungen entsteht ein Standard im Bezug auf Daten und Prozesse der MBS, welcher erst die Konzipierung von Unterstützungskonzepten ermöglicht, da diese auf den Standards aufbauen.

Die Architektur besteht aus drei Teilen: Der *i)* konzipierte Infrastrukturknoten bildet die Basis und wird sowohl zur Lauf- als auch zur Entwicklungszeit eingesetzt. Führt er zur Laufzeit die MBS aus, so stellt er zur Entwicklungszeit der Entwicklungsumgebung Verwaltungsfunktionalität und Information über die MBS zur Verfügung. Er ermöglicht den gemeinsamen Zugriff auf das UUI-Modell, wodurch Änderungen eines Werkzeugs direkt in allen anderen Werkzeugen und Sichten zur Verfügung stehen. Es wird dem Entwickler daher möglich, direkt eine umfassende Evaluation einer Änderung durchzuführen. Die *ii)* Entwicklungsumgebung setzt auf der Infrastruktur auf und integriert die verschiedenen Unterstützungskonzepte in ein Gesamtbild. Dies ermöglicht u.a. die Erstellung föderierter Benutzerschnittstellen mit parallelem Modellieren auf unterschiedlichen Abstraktionsebenen. Schließlich setzt die Architektur *iii)* einen Rahmen dafür, wie die MBS-Artefakte konkret beschrieben werden müssen, damit sie mit der Infrastruktur ausgeführt werden können.

Als querliegenden Aspekt beschreibt die architektonische Integration, wie der Ansatz mit Hilfe eines Bibliothekskonzeptes um einzelne Komponenten aber auch ganze Sprachen erweitert werden (Anforderung 2). Durch den Einsatz von Classloadern können neue UIKomponenten flexibel dem Interpreter zur Verfügung gestellt, in der entwickelten MBS genutzt und selbst im Rapid Prototyping Verfahren weiterentwickelt werden. Der Ansatz ist somit nicht auf eine vordefinierte Menge von Toolkits und UIKomponenten festgelegt. Daher konnten im

SoKNOS Projekt Anwendungsfall spezifische UIKomponenten leicht hinzugefügt und ein komplett neues Hardware Toolkit integriert werden.

In Teil III (Realisierung und Evaluation)

Prototypische Realisierung Der Ansatz wurde prototypisch unter dem Namen "Mapache" umgesetzt (Teil III). Für die Umsetzung der MBS wurden unter anderem die Beitragsklassen eingeführt, welche das Verhalten und die Ausgabe in Fragmenten kapseln. Der für den Infrastrukturknoten umgesetzte Beitragsprozessor ermittelt aus ihnen für eine gegebene MBS-Variante den Satz aktiver Fragmente. Das vorgestellte Meta UI schließlich erlaubt die Wahl der anzuzeigenden MBS-Variante, welche durch die umgesetzten Interpreter für Swing basierte und Hardware basierte MBS-Varianten zur Interaktion gebracht werden kann.

Die Unterstützungskonzepte wurden in der Entwicklungsumgebung Eclipse umgesetzt, sodass eine hoch integrierte Entwicklung der MBS in Java möglich wird. Darunter war auch der Swing basierte Modellinterpreter, welcher zu einem WYSIWYG-Editor ausgebaut wurde. Auf ihm basieren die umgesetzten modularen Adaptionskonzepte. Weitere umgesetzte Editoren ermöglichen auch das Toolkit unabhängige Modellieren (auf einer abstrakteren Ebene).

Als Übergang zur Evaluation wurde das Anwendungsbeispiel des Projektes SoKNOS beschrieben. Die Realisierung wurde in das SoKNOS System integriert, wodurch im SoKNOS Projekt Multi-Benutzerschnittstellen erstellt werden konnten.

Evaluation Die Evaluation bestand aus einer Fallstudie aus dem Anwendungsbeispiel SoKNOS und einer Nutzerstudie mit berufsmäßigen Entwicklern von Benutzerschnittsetllen. Aus Perspektive der **Fallstudie** wurden die erhobenen Anforderungen erfolgreich umgesetzt. Die konzipierte schlanke Anpassung an neue Nutzungskontexte wurden erfolgreich genutzt und eine lauffähige MBS für SoKNOS mit fünf Varianten erstellt. Die Varianten sind auf verschiedene Rollen und unterstützte Hardware zugeschnitten. Sie erlauben die Interaktion über den Desktop, aber auch über eine Großbildwand sowie eine föderierte Benutzerschnittstelle.

Die Wahl der Methode der **Nutzerstudie** ist ein sehr wichtiger Aspekt der vorliegenden Arbeit, welcher in verwandten Arbeiten nicht in diesem Detailgrad besprochen wird. Insbesondere wurde hierbei auf die Vor- und Nachteile von qualitativen gegenüber quantivativen Methoden eingegangen. Auch die Wahl der Teilnehmer, berufsmäßige Entwickler von Benutzerschnittstellen (Experten), geht über die der meisten verwandten Arbeiten hinaus. Mit Hilfe der qualitativen Methode "Kooperative Evaluation" wurden durch die Studie wichtige Punkte identifiziert, welche beim Transfer des vorliegenden Ansatzes auf andere praxisrelevante Nutzungsszenarien zu beachten sind. Darüber hinaus wurde der entwickelte Ansatz von den Teilnehmer überwiegend positiv

bewertet, wobei insbesondere die Arbeit auf mehreren Abstraktionsebenen sehr positiv aufgenommen wurde.

Schließlich wurden die Wichtigkeit und Umsetzung der erhobenen Anforderungen bewertet. Die Experten schätzten sie (in unterschiedlichem Maße) als mittel bis sehr wichtig ein. Die Auswertung lässt dennoch auf Probleme mit dem Konzept "Nutzungskontext" und seiner Nutzung schließen, was aber in der Praxis durch einfache Trainings lösbar ist. Die Umsetzung der Anforderungen wurde weitgehend positiv bewertet. Jedoch konnte der Prototyp bei Anforderungen, welche starken Bezug zu Nutzbarkeit hatten, nicht mit professionellen Entwicklungswerkzeugen mithalten, was jedoch zu erwarten war. Dagegen hob sich die Umsetzung von Anforderung 6 (Modifikation) sehr positiv ab.

Kapitel 20

Weitergehende Forschung

Aufbauend auf der vorliegende Arbeit wurden weitergehende Forschungsfragen identifiziert, welche in diesem Kapitel skizziert werden. So wurde die Forderung nach deklarativeren Unterstützungskonzepten im Rahmen der Nutzerstudie identifiziert. Die zweite Fragestellung nach der tiefergehenden Integration von Modellierung und Programmierung ergab sich aus der Kombination von verwandten Arbeiten und Hürden, welche bei der Umsetzung des Ansatzes zu bewältigen waren. Ähnlich ergab sich die dritte Fragestellung nach der tiefergehenden Integration der Struktur- und Verhaltensbeschreibung bei der Umsetzung der MBS für die Fallstudie. Der vierte Aspekt, das Zusammenspiel von Klassifizierung und Verfeinerung ist das Resultat aus der theoretischen Betrachtung in Abschnitt 9.1.6. Konträr dazu stellt sich die fünfte Fragestellung nach der Methodik auf Grund von Kommentaren der Studienteilnehmer. Die sechste Fragestellung nach den Aufgabenmodellen drängt sich schließlich beim Vergleich mit den verwandten Arbeiten auf.

Die sechs Fragestellungen werden im Folgenden detailliert vorgestellt.

Deklarativere Unterstützungskonzepte Wie in der Zusammenfassung der Ergebnisse der Nutzerstudie in Kapitel 17.3 beschrieben, werden deklarativere Unterstützungskonzepte von den Entwicklern verlangt. Dies beinhaltet z.B. die Zusammenfassung von UI Elementen zu semantischen Gruppen oder die Skalierung einer UI durch Ziehen an der Fensterecke.

Wichtig ist hierbei die Verbindung mit konkretem Editieren: Das Lösungskonzept sollte den Entwickler nicht zwingen, auf abstrakter Ebene zu arbeiten (Anforderung 4), sondern die Deklarativität mit dem Konkreten verbinden. Bei der Unterstützung muss der Pragmatismus der Lösung gewahrt bleiben.

Es liegt also nahe, diese weitergehenden Konzepte spezifisch für unterschiedliche Anwendungsfälle (“Adaptionsfälle”) zu gestalten, ähnlich zu den modularen Adaptionskonzepten (Kapitel 10.1.3). Da diese durch die Studienteilnehmer einfach zu benutzen waren und z.B. das Skalierungskonzept leicht an die geforderte, deklarativere Interaktion angepasst werden kann, scheinen diese ein

Schritt in die richtige Richtung zu sein.

Die Umsetzung solcher Konzepte könnte sich bestehender Techniken bedienen, beispielsweise Ontologien sowie Design Advisors und Design Critics. Die beiden letzteren wurden von Szekely (1996) identifiziert und unterstützen den Entwickler aktiv bei der Gestaltung der UI respektive analysieren und kritisieren eine erstellte UI. Solche Techniken wurden bereits untersucht, z.B. von Demeure u. a. (2006) oder Furtado u. a. (2004), Vanderdonckt und Bodart (1993). So könnten zum Einen Zusammenhänge zwischen Elementen identifiziert, auf der anderen Seite aber auch bestehendes Wissen für die Unterstützung genutzt werden.

Auch die Ausnutzung von Entwurfsmustern (**Patterns**) ist in diesem Zusammenhang erwähnenswert. Von verschiedenen Autoren wurden bereits UI Patterns vorgeschlagen und genutzt, z.B. (Borchers 2000b, Lin und Landay 2008, Luyten u. a. 2005, Rathsack, Wolff und Forbrig 2006, Tidwell 2007). Ebenso beschäftigen sich aktuelle Arbeiten und Konferenzen mit dem Thema (Breiner u. a. 2010, Seissler, Meixner und Breiner 2010).

Der Ansatz der Modellierung ist für die Konzipierung deklarativerer Konzepte sehr passend, da er, wie in Kapitel 3.4 dargelegt, zusätzliche Information mit einbeziehen kann. So könnten Informationen aus Modellen zur Wichtigkeit von UI Elementen oder Aufgaben von Benutzern in Patterns oder anderen Konzepten genutzt werden.

Tiefergehende Integration Modellierung und Programmierung Die Konzepte der Modellierung und Programmierung (wie in Kapitel 3 beschrieben) konvergieren. Domänenspezifische Sprachen sind inzwischen inhärenter Teil solch konvergierender Ansätze (z.B. MPS, Intentional Language Workbench, vgl. Abschnitt 3.3.1). Dies rührt letztlich daher, dass die neuen Modellierungswerkzeuge es schaffen, das "Programmiergefühl" und seine Techniken mit Modellierungsparadigmen zu kombinieren. Hierbei wird an Stelle der üblichen Textdateien (konkrete Syntax) beim Programmieren gleich die abstrakte Syntax erzeugt und auch abgespeichert (an Stelle der Textdatei). Somit tritt die abstrakte an Stelle der konkreten Syntax als zentrales Element (vgl. Abschnitt 3.2).

Die Vorteile beider Ansätze können somit kombiniert werden – auch für die Entwicklung von Benutzerschnittstellen. Lägen UUI-Modelle und Programmcode in einer einheitlichen abstrakten Syntax vor, wäre die gegenseitige Referenzierung deutlich einfacher. Auch Probleme mit Inkonsistenzen, weil z.B. ein aus dem Programmcode referenziertes Modellelement gelöscht wurde, wären drastisch reduziert.

Tiefergehende Integration der Struktur- und Verhaltensbeschreibung Der präsentierte Ansatz zeigt, dass eine enge Integration von Verhaltensbeschreibungen (hier in Code) und UI Struktur (hier als Modelle) Vorteile für

die UI Entwicklung bietet. Die Integration kann noch weiter intensiviert werden. So kann das in Abschnitt 14.3.4.1 angesprochene Wrapper Konzept für den Zugriff auf Modellelemente aus Code heraus weiterverfolgt werden. Alleine wissenschaftlich weniger interessant, gewinnt es in Kombination mit dem im folgenden Abschnitt beschriebenen Zusammenspiel zwischen Klassifizierung und Verfeinerung an Gehalt.

Auch wäre statt eines einfachen, statischen *Wrappers ein dynamisches Konzept* interessant. Ein statischer Wrapper um ein Interaktionsobjekt würde erlauben, die Methoden und Eigenschaften, welche vom Interaktionsobjekt referenzierten Klassifizierer definiert werden, zu nutzen. Dagegen würde ein dynamisches Konzept mehr als nur einen referenzierten Interaktionsobjektklassifizierer unterstützen. So könnten Operationen nicht nur auf einem Interaktionsobjekt, sondern auf einer Menge von Interaktionsobjekten (die unterschiedlich klassifiziert sind) angewendet werden, z.B. wenn mehrere Verfeinerungen eines Elementes in Betracht gezogen werden sollen. Auch könnte der Entwickler bei der Verfeinerung und Änderung einer Klassifizierung gewarnt werden, wenn bestimmte Operationen nicht mehr durchführbar sind und somit Verhalten oder Beobachter angepasst werden müssen.

Diese detaillierte Untersuchung für eine *semantisch tiefergehende Unterstützung der architektonischen Schnittstelle zwischen UI Struktur und Fragmenten* bietet sich auch an. Fragen bezüglich in welchen Verfeinerungen die Gefahr besteht, dass ein Fragment nicht mehr funktioniert, weil die UI zu stark geändert wurde oder wie der Entwickler bei der Anpassung der Fragmente geschickt unterstützt werden kann, können hierbei adressiert werden.

Konzepte für das Zusammenspiel zwischen Klassifizierung und Verfeinerung In Abschnitt 9.1.6 wurde das Zusammenspiel zwischen Klassifizierung und Verfeinerung beleuchtet. Hierbei wurden (vgl. Tabelle 9.2) verschiedene Klassen des Zusammenspiels identifiziert. Auf Grund des Umfangs wurden aber nicht alle identifizierten Problemklassen behandelt. Insbesondere das Aufspalten bzw. Zusammenführen von Interaktionsobjekten im Rahmen der Verfeinerung (z.B. ein Datumsfeld in getrennte Felder für Jahr, Monat, Tag) stellt eine größere Herausforderung dar, bietet aber Potenzial den Entwickler bei seiner Tätigkeit zu entlasten.

Die kann insbesondere als *Übersetzungsproblem* aufgefasst werden. Welche Methode kann wie für die Verfeinerung bzw. Abstraktion übersetzt werden? Welche Eigenschaften der verschiedenen Verfeinerungen lassen sich wie aufeinander abbilden? Es liegt nahe, dass das Problem nicht allgemein lösbar ist, da viel pragmatisches Wissen für die Übersetzung benötigt wird (z.B. könnte ein Teil eines Datums aus anderen Elementen abgeleitet werden). Insofern sollte eine Anwendungsfall spezifische Lösung untersucht werden.

Zum Einsatz könnten hierbei z.B. Techniken aus dem Ontologiebereich kommen, welche Konzepte miteinander in Beziehung setzen können. Diese scheinen

geeignet, weil die Zuordnung dynamisch erfolgen kann. Wird die Datenstruktur (das Datenmodell) als Modell (im Sinne der Modellierung) hinterlegt, könnte diese zusätzliche Information (vgl. Abschnitt 3.4) mit ausgenutzt werden.

Methodische Fragestellungen Aus der Nutzerstudie ging hervor (vgl. Kapitel 17.3), dass der notwendige Konzipierungsschritt vor der Erstellung einer Benutzerschnittstelle vom vorliegenden Ansatz nicht explizit betrachtet wird. Die teilweise unterschiedliche Einschätzung der Probanden in der Studie lässt vermuten, dass es mehr als eine Entwicklungsmethode für MBS zu unterstützen gilt.

Somit bietet es sich an, eine empirische Erhebung dieser verschiedenen Herangehensweisen in der Praxis durchzuführen. Die Notwendigkeit, Untersuchungen im Bereich Methodik anzustellen, wurde auch von Calvary und Pinna 2008 erkannt. Entscheidend ist hier aber der Praxisbezug. Einige Arbeiten, z.B. von Feuerstack (2008), Meixner und Görlich (2009), Paternò und Santoro (2003) befassen sich mit dem Thema, ebenso wie die in Arbeit befindliche Norm ISO 9241-230 (Bevan 2009).

Fragestellungen in diesem Gebiet sind unter anderem: Nach welchen Grundsätzen sollten MBS entwickelt werden? Welche Rollen kommen im Entwicklungsprozess vor? Wie unterscheidet sich die Entwicklung der MBS von "normalen" Benutzerschnittstellen in der Praxis?

Ein großes Fragezeichen besteht generell bei der *Integration von Software und Usability Engineering* – der Integration des Konzipierungsschrittes der Benutzerinteraktion mit dem Software Entwicklungsprozess. Hierzu wurden bereits Workshops und Konferenzen veranstaltet. Unter www.se-hci.org sind (ältere) Aktivitäten zu finden. Inzwischen ist die deutsche Gemeinschaft aber aktiver geworden und der vorliegende Ansatz wurde in diesem Kontext auch schon diskutiert (Behring, Petter und Mühlhäuser 2009).

Aufgabenmodelle Im Korpus der verwandten Arbeiten sind Aufgabenmodelle ein zentrales Element. Wie in Abschnitt 4.2.1 formuliert, handelt es sich um Beschreibungen der Tätigkeit mit restriktivem Charakter. Das heißt, Aufgabenmodelle sind bezüglich der endgültigen Interaktion nicht eindeutig und somit sind zu einem Aufgabenmodell immer mehrere Möglichkeiten vorhanden, die Interaktion zu durchlaufen. Sie ermöglichen vor allem die Strukturierung der Aufgaben im Rahmen von benutzerzentrierten Entwicklungsprozessen (Meixner und Görlich 2008).

Aufgabenmodelle wurden im vorliegenden Ansatz nicht detailliert betrachtet. Mit dem vorgestellten Ansatz können keine temporalen Beziehungen modelliert werden. Es ist aber möglich, die Aufgaben(typen) selbst und ihre Verschachtelungsstruktur mit Interaktionsobjekten und einem adäquaten "Aufgaben Toolkit" zu modellieren. Die Einbindung der temporalen Beziehungen wäre dennoch sinnvoll, da *sie eine weitere Informationsquelle für Transformationen*

und Unterstützungskonzepte sind. Die Herausforderung bestünde in der Integration der Aufgabenmodelle in das Verfeinerungskonzept mit aktiver und passiver Propagation.

Es stellt sich darüber hinaus die Frage, wie diese konkret in der Praxis eingesetzt werden. Denn Erfahrungen aus EMODE (Domene u. a. 2007) legen nahe, dass Entwickler sie teilweise als umständlich und unnütz betrachten. Ob dies eine rein organisatorische Frage ist, oder andere Gründe hat, bleibt zu klären. Dennoch könnten sie einen Teil der *Verknüpfung zwischen Software und Usability Engineering* darstellen, wie im vorigen Absatz zu methodischen Fragestellungen angesprochen.

V

Anhang

Anhang A

Materialien zur Nutzerstudie

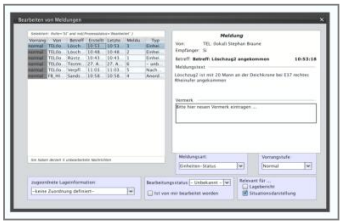
Dieses Kapitel beinhaltet die Materialien zur Nutzerstudie. Ab Seite 244 findet sich das Aufgabenblatt, Seite 246 zeigt die Szenariobeschreibung, ab Seite 247 finden sich weitere Grafiken, welche zur Erläuterung des Ansatzes benutzt wurden und ab Seite 251 ist der Beobachterleitfaden zu finden.

Aufgaben zur Mapache-Studie

1.1. UI auf großen Bildschirm anpassen

Einige Rollen im Stab haben die Möglichkeit, auf größeren Bildschirmen zu arbeiten (Barco-Wand, Tisch, …). Für die Arbeit an diesen großen, hochauflösenden Geräten ist eine neue UI Version nötig, die auch bei etwas größerem Abstand vom Bildschirm noch lesbar ist. Sie soll wie die „normale“ Nachrichten UI aussehen und ca. ¾ des Bildschirmes füllen.

1.2. Drucken-Funktionalität

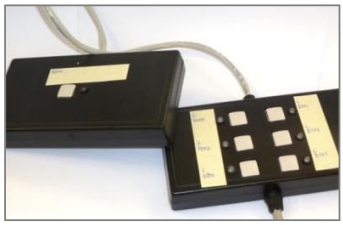


Da Meldungen bisweilen auch ausgedruckt werden sollen, muss diese Funktion in der Anwendung und allen ihren UIs nachgerüstet werden. Es soll ein Drucken-Knopf in alle UIs eingefügt werden. Das Verhalten des Drucken Knopfes soll durch eine simple Ausgabe von „Gedruckt: Meldung Nummer XYZ“ ersetzt werden.

Pause

1.3. Spezialhardware für Sichter einbinden

Die primäre Aufgabe des Sichters ist das Weiterleiten der Meldungen an die betroffenen Rollen im Stab. Das existierende Sichter UI ist auf diese Tätigkeit zugeschnitten. Nun wurde eine Spezialhardware entwickelt, welche die Knöpfe zur Empfängerauswahl aus dem UI in Hardware realisiert. Bringen Sie die Knöpfe „zum Laufen“.



1.4. Leuchtdioden für Sichter-Board anbinden

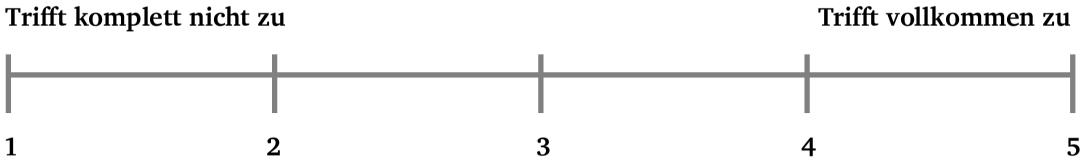
Nun sollen die Leuchtdioden des Boards angebunden werden. Die Farbgebung ist nicht wichtig.

1.5. Zeit der Sichtung einfügen

Das Sichten der Meldung ist ein wichtiger Zeitpunkt. Es soll festgehalten werden, wann der Sichter eine Meldung weiterleitet. Ein entsprechendes Feld ist im Datenobjekt „Message“ schon vorgesehen. Füllen sie es beim Absenden aus der UI Version für den Sichter mit der aktuellen Uhrzeit aus.

Wir testen gemeinsam die Tauglichkeit von Mapache:
Bitte sprechen Sie während der gesamten Zeit lauf über Ihre Gedanken.
Stellen Sie bitte Fragen – wir fragen Sie ja auch.

Skala



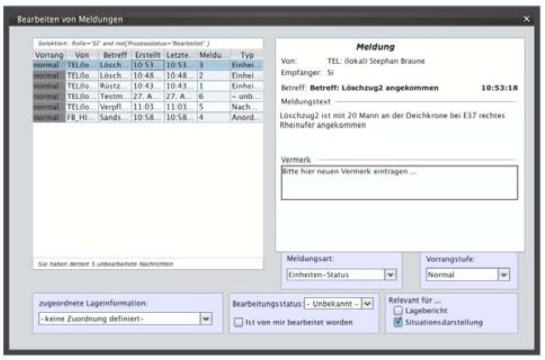
Dimensionen

1. Beliebige Abstraktionsebenen und –Modelle zur Beschreibung (Erstellung und Modifikation) von UIs
2. Leicht auf neue UI Toolkits erweiterbar
3. Details des Look and Feel der UIs modifizierbar
4. Einfache Nutzbarkeit für den Entwickler/Designer
5. UI Layout und Verhalten („Look and Feel“) werden zusammen formuliert
6. Modifikationen an bestehenden UIs werden unterstützt (Werkzeuge, Beeinflussen der Verfeinerungsbeziehungen)
    - Anwendungen auf eine UI und mehrere UIs

1. Szenario zur Kooperativen Evaluation der Mapache Konzepte

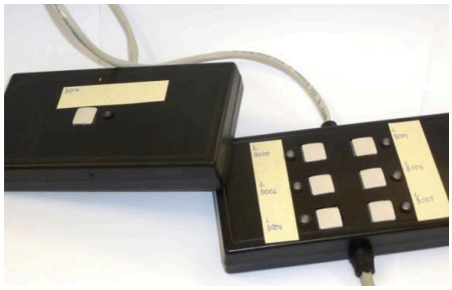
Die Evaluation der Konzepte von Mapache wird am Beispiel von SoKNOS durchgeführt. **SoKNOS** ist ein Projekt, welches IT-Unterstützung für Stabsarbeit (Feuerwehr-, Polizei-, THW-, … Stäbe) untersucht. Eine wichtige Motivation für die UI Adaption im Rahmen von SoKNOS ist die Vielfältigkeit (vgl. Bild rechts) der *i)* Nutzerrollen, *ii)* beteiligten Organisationen, und *iii)* der zur Verfügung stehenden Interaktionsgeräte.

Im vorliegenden Fall wird die **Nachrichtenanwendung** von SoKNOS untersucht. Diese wird von **unterschiedlichen Rollen** in unterschiedlicher Weise genutzt. Der **normale Stabsmitarbeiter** betrachtet hiermit eingegangene Nachrichten, annotiert diese und markiert sie als bearbeitet. Außerdem versendet er Nachrichten an andere Stabsmitarbeiter und Personen außerhalb des Stabs. Der **Sichters** übernimmt dahingegen die Verteilung von eingegangenen Nachrichten. Er muss für eine Nachricht feststellen, für welche anderen Stabsrollen sie relevant ist, und sie an diese weiterleiten. Darüberhinaus wird die Nachrichtenanwendung auch auf **unterschiedlichen Geräten** eingesetzt. Neben normalen PCs kommt eine Spezialhardware für den Sichter und ein hochauflösendes, Tisch-basiertes Display zum Einsatz.

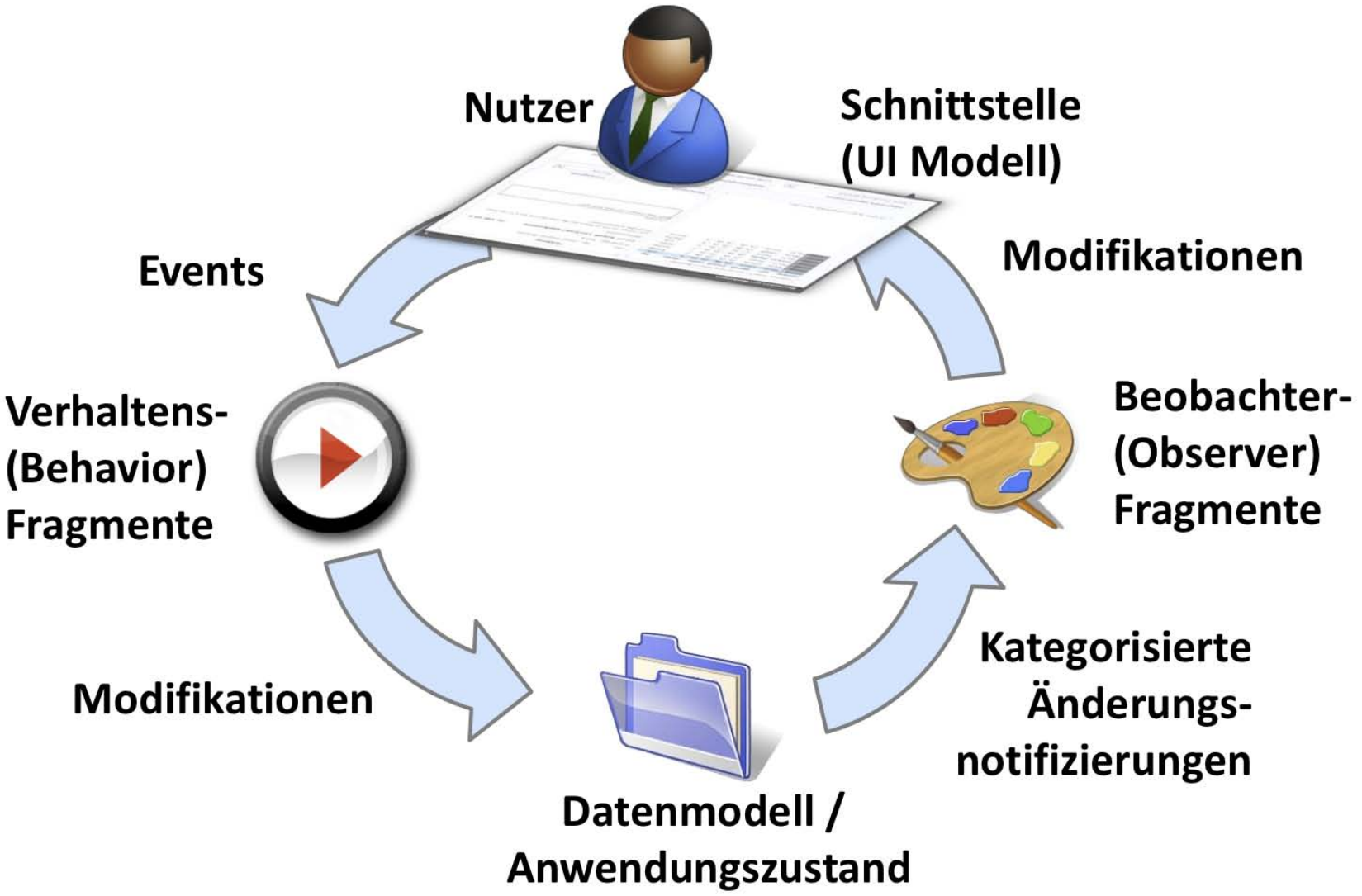


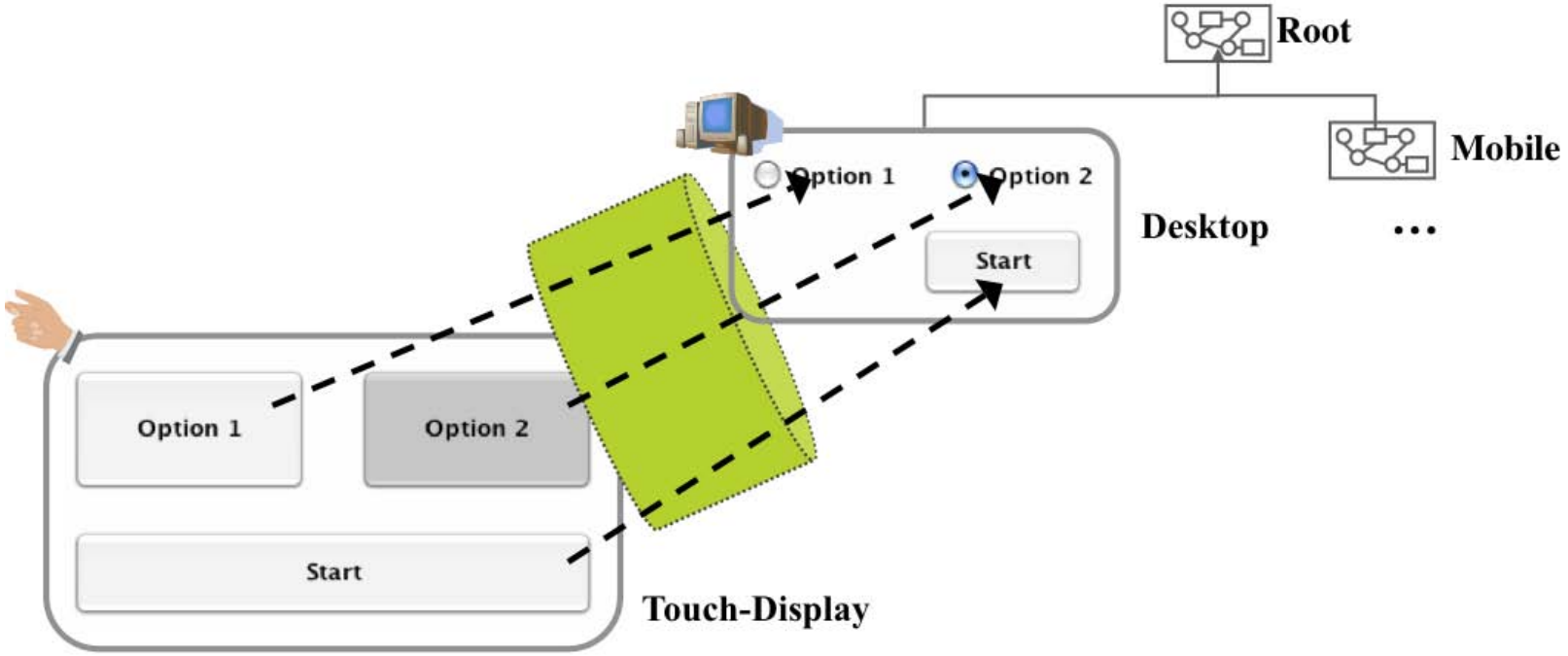
Den **Ausgangspunkt für Verfeinerungen** der UI bietet die **Version für normale Stabsmitarbeiter**. Sie ist links dargestellt. Im linken Bereich werden die (gefilterten) Nachrichten angezeigt. Durch Klick kann eine Nachricht ausgewählt, und damit im rechten Bereich angezeigt werden. Der untere Bereich zeigt spezifische Eigenschaften der Nachricht, wie z.B. deren Bearbeitungsstatus im Bezug auf den Nutzer an und lässt deren Bearbeitung zu.

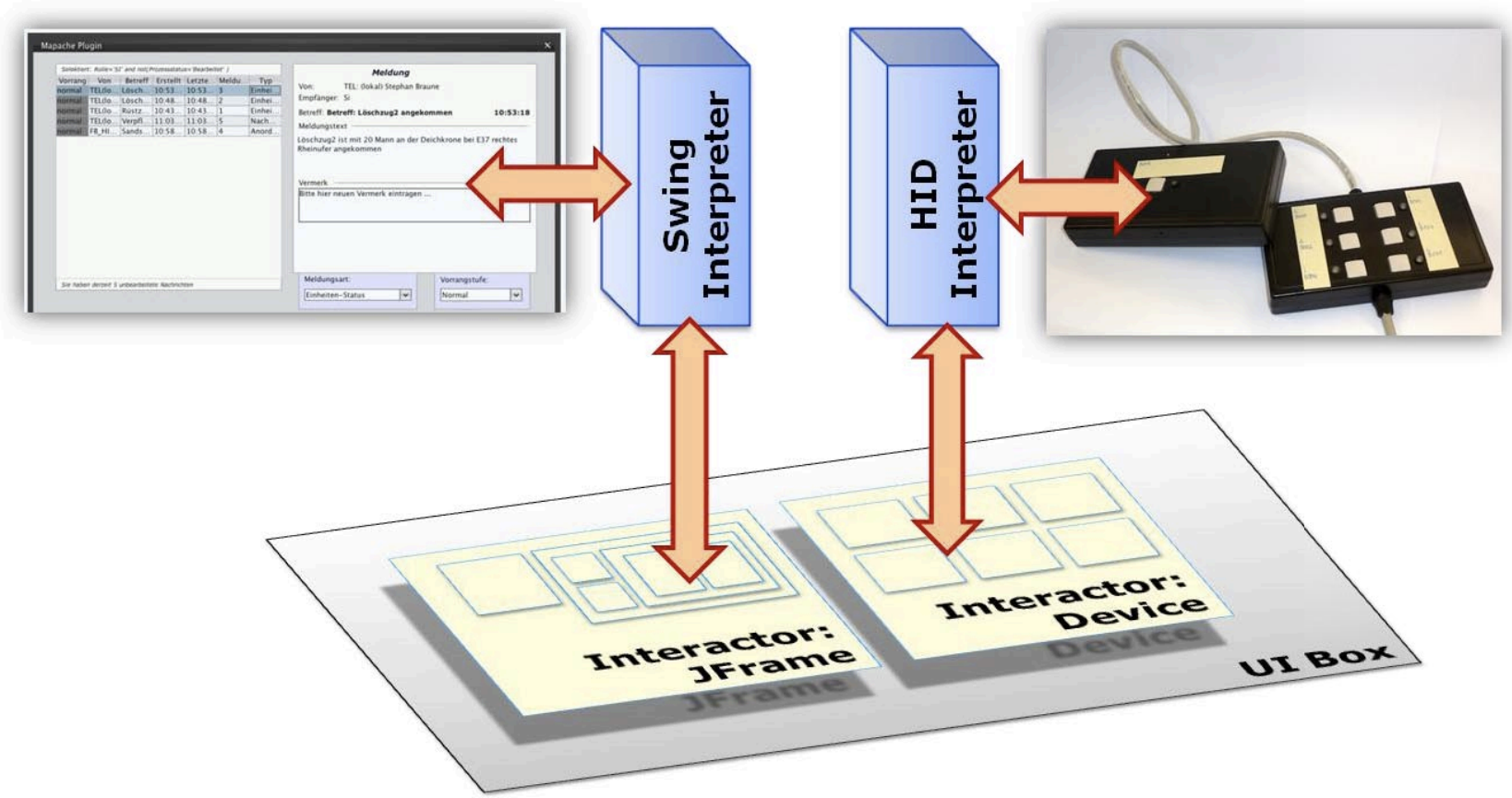
Das sogenannte **Sichterboard**, welches rechts dargestellt ist, ermöglicht eine direkte Adressierung mit Hilfe von Tastern. Jeder Taster stellt einen Adressaten dar. Die Leuchtdiode neben dem Taster gibt Auskunft über seinen Adressierungsstatus. Daneben gibt es einen Taster zum Absenden sowie zwei Taster, welche ganze Gruppen von Adressaten auswählen können.

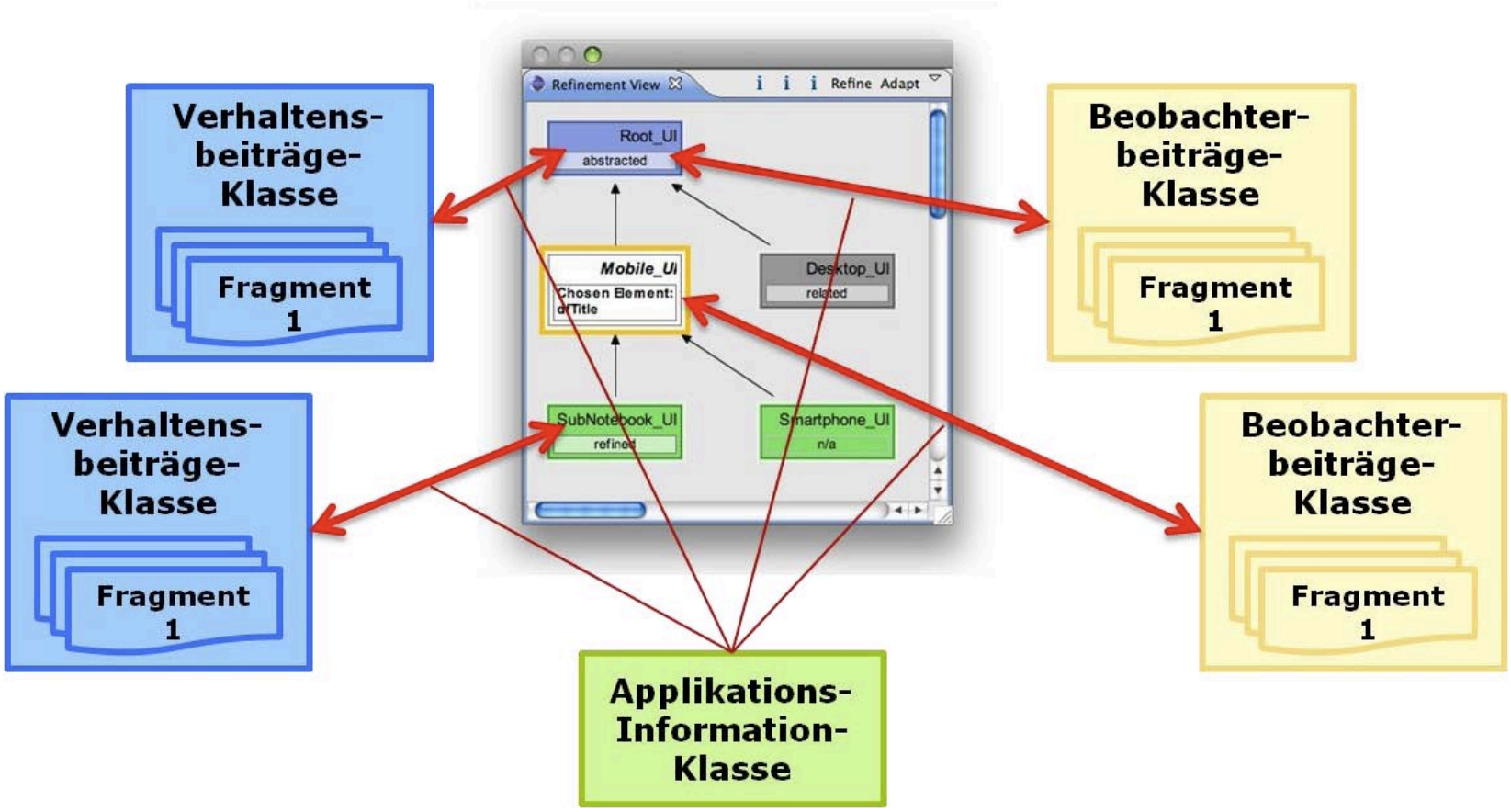


Der **Tisch** verfügt über ein **hochauflösendes Display.** Er kann von den Stabsmitarbeitern für Besprechungen oder zur Schichtübergabe benutzt werden. Er ist mit Hilfe eines Stiftes bedienbar.


Nutzer
Schnittstelle
(UI Modell)
Events
Modifikationen
Verhaltens-
(Behavior)
Fragmente
Beobachter-
(Observer)
Fragmente
Modifikationen
Kategorisierte
Änderungs-
notifizierungen
Datenmodell /
Anwendungszustand


Root
Mobile
Option 1
Option 2
Desktop
Start
Option 1
Option 2
Start
Touch-Display


Mapache Plugin
Meldung
Swing Interpreter
HID Interpreter
Interactor: JFrame
Interactor: Device
UI Box


Verhaltens-
beiträge-
Klasse
Fragment
1
Verhaltens-
beiträge-
Klasse
Fragment
1
Refinement View
Refine
Adapt
Root_UI
abstracted
Mobile_UI
Chosen Element:
dTitle
Desktop_UI
related
SubNotebook_UI
refined
Smartphone_UI
n/a
Beobachter-
beiträge-
Klasse
Fragment
1
Beobachter-
beiträge-
Klasse
Fragment
1
Applikations-
Information-
Klasse

1. Beobachter-Leitfaden

Die Erklärungen und Einführungen sollten möglichst mündlich („frei“) gegeben werden, um die Atmosphäre nicht zu stören-

1.1. Start

Vorstellen. Bedanken. Ziel der Studie nennen.

Ablauf:
- Einführung
  - Allgemeines
  - Zur Software
- Kurzes Training
- Lösen von Aufgaben
- Nachbesprechung

Privatsphärenerklärung.

Verhalten:
- Nachfragen ist erlaubt!
- Wir messen keine Zeiten!
- Wir lieben Feedback!
- Laut denken!
- Man kann keine Fehler machen!

1.2. Trainingssitzung

Aufnahme starten. Startzeit eintragen.

Szenario mit SoKNOS einführen. Adaption zeigen.

Konzepte einführen:
- Refinements (mit Bild)
- UIBox, Interaktor, Interaktorklasse, Refinement zwischen Interakoren
- Programmiermodell (mit Bild)

Kurze **Eclipse**-Übersicht:
- Refinement View
  - Kontext-Menü
- Swing Interactive Interpreter
- Generischer Editor mit Libs
  - Elemente löschen, erstellen
  - Propagieren v. Elementen
- Behavior View

Kurze **Einführung** in die Werkzeuge mit exerzieren einfacher Beispiele:
- Ablegen-Knopf für Meldung in alle UIs einfügen + Verhalten
- Label für Ablegen-Knopf in allen UIs auf „Ablage“ ändern und in Sichter UI, Knopf verschieden

1.3. Observation

Kooperative Evaluation vorstellen.

Hinweise erneuern:
- Nachfragen ist erlaubt! Wir werden Fragen stellen!
- Wir messen keine Zeiten!
- Man kann hier keine Fehler machen!
- Möglichst natürlich bleiben und entspannen!
- Bitte ganze Zeit reden!

Den Probanden am Reden halten – wissen, was gerade abläuft.

Nützlichen Fragen:
- Wie machen wir das?
- Was möchten Sie gerade tun?
- Was wird passieren wenn…?
- Was hat das System jetzt getan?
- Was versucht das System durch diese Meldung zu vermitteln?
- Warum hat das System das gemacht?
- Was haben Sie erwartet, das passiert?
- Was machen Sie gerade?

Dann Vorstellung Aufgaben-Bogen… Beginn.

Nach Aufgabe 2 und am Ende Pause für Teilnehmer - selbst Punkte für Besprechung identifizieren.
Vor Aufgabe 3 Einführung in Hardware.

1.4. Zwischenfragen nach jeder Aufgabe

- Wie hat Mapache Sie bei der Bewältigung unterstützt?
- Passt das Konzept auf die Bewältigung der erledigten Aufgabe?

1.5. Zwischenfragen vor Aufgabe 2

- Sie müssen die Größe anpassen – welche nicht-Mapache Mittel würden Ihnen einfallen bzw., welche würden Sie einsetzen?
- Wie sollte es idealer Weise laufen?

1.6. Ende der Sitzung, Nachbereitung

Nachbesprechung auf sep. Bogen.

Bedanken. Verabschieden.

Notizen aufarbeiten:
- Weitere interessante Dinge notieren
- Durchdiskutieren und zusammenfassen
    - Danach hauptsächlich mit der Zusammenfassung gearbeitet!
- Interessante Zitate mit in die Zusammenfassung schreiben oder mit gelbem Marker markieren.

2. Hintergrund

2.1. Start

Am Anfang den Nutzer „beruhigen“, Erklärung zur **Privatsphäre** abgeben (separates Blatt) und Rapport herstellen (kooperative Atmosphäre).

Wichtig: **Ziel der Studie** ist es nicht, den Probanden zu testen, sondern wir testen gemeinsam den Forschungsprototyp zu evaluieren, und insbesondere das Konzept hinter der Software:

- Evaluation des Mapache- Konzeptes (Vor- und Nachteile, Generalisierbarkeit, Anwendbarkeit)
- Evaluation der Adaptionstools (Konzept und Nutzbarkeit)

Regeln zur Studie:

- **Nachfragen ist erlaubt**! Das ist ein komplett neues Stück Software, man kann sich also nicht sofort damit auskennen.
- **Wir messen keine Zeiten**, d.h. Bio-Breaks, Trinken, Kekse sind bei Bedarf möglich.
- **Wir lieben Feedback**. Also Forscher sind wir dankbar für Rückmeldungen und auch auf diese angewiesen. Positives sowie Negatives hilft uns beides weiter, denn unser einziges Ziel ist es, Dinge zu verbessern! Also bitte nicht zurückhalten!
- Auch während des Arbeitens ruhig **alles sagen** was einem gerade durch den Kopf geht („Thinking aloud“), auch wenn es kein konkretes Feedback ist.
- **Man kann hier keine Fehler machen**! Wir wollen vom Probanden lernen, wie Dinge einfach zu machen sind.

2.2. Durchführung

Die Studie wird in Form einer kooperativen Evaluierung (Monk, 93) durchgeführt. Es werden der Ton, sowie der Bildschirm des Probanden aufgenommen. Der Beobachter achtet auf besondere Vorkommnisse bei der Durchführung der Aufgaben und steht dem Probanden bei Rückfragen zur Verfügung. Rückfragen sollten im Hinblick auf die Informationsausbeute beantwortet werden – eventuell sollte man den Probanden über seine Motive und Vorstellungen, was die Werkzeuge bewerkstelligen ausfragen und mehrmals versuchen lassen, bevor man „die richtige Antwort“ gibt.

Besondere Vorkommnisse sind lt. Monk vor allem

1. Kommentare (bzgl. des zu testenden Systems), und
2. unerwartetes Verhalten des Nutzers.

Weitere interessante Beobachtungen sind:

- Löschen von Dingen → hat Konzept etwas nicht hergegeben?
- Längeres Grübeln → Konzept bietet keine gute Lösung an?
- Grunzlaute, freudige Aussprüche und andere unterbewusste Äußerungen sind auch Kommentare
- Nervosität, Stimmungsänderungen → Werkzeuge tun nicht was erwartet wurde? Grund für Frustration?

Diese werden (mit Stift! PC ist störend) in die Notizseiten eingetragen, zusammen mit der aktuellen Uhrzeit.

Beides weißt auf eine Misskonzeption bzw. ein Missverständnis hin. Wird ein Kommentare oder unerwartetes Verhalten erkannt, muss bei Verdacht auf konzeptionelle Probleme direkt (unvoreingenommen, offen und wertfrei!) nachgefragt werden:

- Was für ein Verhalten wurde erwartet?

- Mit welchem Ziel wurde es genutzt / sollte es genutzt werden?
    - Inwiefern hätte das erwartete Verhalten zur MUI-Entwicklung beigetragen?
- Mapache macht das so..., wie ist das im Hinblick auf Ziel zu bewerten?

Eine lockere, kooperative Atmosphäre sollte gewahrt werden, allerdings muss der Beobachter immer in Kontrolle sein. Ein zu starkes Abdriften ist zu vermeiden.

Am Wichtigsten: Den Probanden am Reden halten – wissen, was gerade abläuft.
Am Zweitwichtigsten: Notizen machen.

Nutzen von offenen, wertneutralen Fragen. Eher „Wie hilft XYZ“ an Stelle von „ist XYZ gut“.

2.3. Vorwort zum eigentlichen Hauptteil

Kurz Vorstellen, was **kooperative Evaluation** ist:

- Man untersucht gemeinsam „kooperativ“ eine Software:
    - Proband führt Aufgaben durch und setzt dadurch die Konzepte ein,
    - Beobachter untersuchen währenddessen, wie die Konzepte zu der Arbeitsweise des Probanden passen (und nicht umgekehrt!).
    - Proband fragt Beobachter ggf. zur Bedienung / Einsatz.
- Aber wie kommen wir an „die Gedanken des Probanden“? Er muss laut denken!
    - Ein „laufender Kommentar über das was man denkt und tut.“

Vor dem Start nochmals auf folgende Punkte hinweisen:

- **Nachfragen ist erlaubt!**
- **Wir messen keine Zeiten,** d.h. Bio-Breaks, Trinken, Kekse sind bei Bedarf möglich.
- Nicht wundern, wenn **wir Fragen stellen** – wir möchten Herausfinden, wie nützlich die Konzepte sind, daher bohren wir evtl. auch öfter mal nach, oder sind ausweichend bei Antworten. Dann bitte einfach weiterprobieren. Danke!!
    - Wir **unterbrechen gewollt an bestimmten Stellen**, um Dinge nachzufragen.
- **Wir lieben Feedback** - also bitte nicht zurückhalten!
- **Man kann hier keine Fehler machen!**
- Aber bitte **möglichst natürlich bleiben und entspannen**, wir sorgen schon für alles. Wenn z.B. nicht laut gedacht wird, erinnern wir freundlich daran ;)

2.4. Nachbesprechung

Ziel ist es, etwas abstraktere Rückmeldung vom Probanden einzuholen.

Fragen auf sep. Bogen.

Anhang B

Ergebnisse der Nutzerstudie

Dieser Anhang beinhaltet die detaillierte Auswertung der Nutzerstudie, welche in Kapitel 17 vorgestellt wurde. Der Anhang teilt sich, wie in Kapitel 17.3 beschrieben, in zwei Teile: die Auswertung der Anforderungen in Kapitel B.1 und die Auswertung der als relevant identifizierten Themen in Kapitel B.2.

B.1 Diskussion der Anforderungen

Nach Durchführung der Aufgaben wurden die Teilnehmer bzgl. der 6 Anforderungen in den Dimensionen Wichtigkeit und Umsetzung durch die vorliegende prototypische Implementierung befragt. Die Anforderungen wurden in Kapitel 5 erarbeitet und sind in der Zusammenfassung (Kapitel 5.7 auf Seite 49) zusammengefasst.

Die Bewertung geschah, indem die Teilnehmer die Wichtigkeit und die Umsetzung der Anforderung jeweils auf einer Skala von 1 bis 5 bewerten mussten. Wichtig war dabei, dass sie anschließend die Bewertung mit Kommentaren begründeten, welche von den Evaluatoren notiert und diskutiert wurden. Diese Diskussion und Bewertung der Anforderungen wird in diesem Kapitel zusammengefasst.

B.1.1 Anforderung 1: Abstraktionen

Anforderung 1 wurde entwickelt in Kapitel 5.1.

In *Summe* ergab sich ein Trade Off zwischen Barrierefreiheit gegenüber Praxisrelevanz. Die Umsetzung wurde durchweg positiv bewertet und sorgt für eine gute Übersicht.

B.1.1.1 Wichtigkeit

Die Anforderung wurde als “sehr wichtig” oder als “nicht wichtig” eingestuft, aber von keinem Teilnehmer als “mittel”. Insbesondere wurde erwähnt, dass

die Anforderung wichtig sei, da somit dem Entwickler “keine Grenzen gesetzt” werden (Z4.3.6).

Die Anforderung ermöglicht, dass *auf konkretem und abstraktem Niveau* gearbeitet werden kann, was positiv aufgenommen wurde (Z4.2.9), wie auch in Abschnitt B.2.2.1 zur Toolkit übergreifenden Arbeiten besprochen. Des Weiteren kann die Vererbung die *Konsistenz zwischen den verschiedenen Varianten* verbessern (Z4.1.4).

Die *Strukturierung der Verfeinerung je nach Anwendungsfall* wurde von Teilnehmer 4 und Teilnehmer 5 für gut befunden. Dabei ist insbesondere die Baumstruktur geeignet, da sie einen besseren *Überblick* als andere Strukturen verschafft (Teilnehmer 5). Diese Übersicht ist laut seiner Aussage wichtig. Dagegen merkte Teilnehmer 3 an, dass zu große *Beliebigkeit bei der Wahl der Ebenen* herrscht. Er vermutet, dass Toolkit Abstraktionen als Ebenen genutzt werden sollten.

Darüber hinaus sei es unklar, wann Verfeinerungen erstellt werden, was die Vermutung aus Abschnitt B.2.3.2 untermauert, dass das Denken in Nutzungskontexten nicht geläufig ist. Teilnehmer 3 konstatierte dann, dass die Varianten vermutlich von Anfang an feststehen würden. Dem gegenüber bevorzugte Teilnehmer 4 das hierdurch unterstützte iterative Vorgehen.

Daneben wurde die *Praxisrelevanz der Anforderung* von Teilnehmer 6 angezweifelt. So muss lt. Teilnehmer 6 bei Anpassung an ein neues Gerät bzw. Nutzungskontext zuerst das Grundkonzept überarbeitet werden, weil starke Unterschiede zwischen Geräten existieren. In diesem Zusammenhang merkte Teilnehmer 6 an, dass der Ansatz diesen Konzipierungsschritt nicht betrachtet (vgl, Abschnitt B.2.6.2).

B.1.1.2 Umsetzung

Die Umsetzung der Anforderung durch das Konzept und die prototypische Implementierung wurde allgemein, bis auf Nutzbarkeitsmängel (vgl. auch Bewertung von Anforderung 4 in Abschnitt B.1.1) *sehr positiv bewertet*, insbesondere die Darstellung des Verfeinerungsbaumes (Teilnehmer 5). Dabei wurde die direkte Interaktion (Aktion auf UUIBoxen) im Verfeinerungsbaum positiv aufgenommen (Teilnehmer 3, zu direkter Manipulation vgl. Abschnitt B.2.5.1).

Die *Verfeinerungsansicht* (als zentrales Element der Umsetzung und zur Interaktion mit dem Entwickler) ist gut für die *Übersicht* (Teilnehmer 4 und Teilnehmer 5) und einfach zu bedienen (Teilnehmer 6). Das Erstellen von neuen Verfeinerungen und Vererbung ist intuitiv (Teilnehmer 3).

B.1.2 Anforderung 2: Erweiterbarkeit

Anforderung 2 wurde entwickelt in Kapitel 5.2.

In *Summe* waren die Teilnehmer der einhelligen Meinung, dass die Anforderung sehr wichtig ist. Jedoch wurde sie wenig in der Studie behandelt und die

Teilnehmer konnten sich kein detailliertes Bild machen. Generell beherrschen kommerzielle Tools das Thema Erweiterbarkeit gut.

B.1.2.1 Wichtigkeit

Alle Teilnehmer waren sich einig, dass dies eine *wichtige Anforderung* ist. Insbesondere für Werkzeuge mit grundlegender Funktionalität (Teilnehmer 5). Auch sollte der Entwickler nicht eingegrenzt werden, weil die UI Sprache sich nicht erweitern lässt (Z4.3.6). Diese Anforderung ist somit auch verwandt mit der gewünschten Flexibilität (vgl. Abschnitt B.2.1.1 zu allgemeinen Architekturthemen).

Eine leichte Erweiterbarkeit gewährt die Nutzung des Ansatzes, wenn *neue Entwicklungen* (Toolkits) unterstützt werden sollen oder alternative Toolkits genutzt werden (Teilnehmer 5 und Teilnehmer 6). Insbesondere der generische Editor (vgl. Abschnitt B.2.2.1) erlaubt das schnelle Einbinden neuer Toolkits. Diese Flexibilität ist wichtig, weil die *Anforderungen sich in diesem Bereich schnell ändern* (Teilnehmer 3).

B.1.2.2 Umsetzung

Weniger eindeutige *(sehr gemischte) Meinungen hatten die Teilnehmer bzgl. der Umsetzung* von Anforderung 2. Wohl auch, weil die Studie die Erweiterbarkeit aus Zeitgründen nicht im Detail adressieren konnte. Prinzipiell sieht Teilnehmer 5 die *Erweiterbarkeit aber als nachvollziehbar* an, da SoKNOS Spezialkomponenten genutzt werden konnten.

Kommerzielle Werkzeuge sind bei der Umsetzung dieser Anforderung im Allgemeinen sehr weit und sehr brauchbar (Teilnehmer 6). Es steht aber zu vermuten, dass es für die Teilnehmer nach der Durchführung der Aufgaben nicht deutlich ersichtlich war, dass die *Erweiterbarkeit auch das Hinzufügen ganzer Toolkits* betrifft (und nicht nur die Erweiterung des einen unterstützten Toolkits), was i.A. bei kommerziellen Werkzeugen nicht der Fall ist. Eventuell wurde die präsentierte Hardware nicht direkt als “Toolkit” wahrgenommen, da sie keine Ausprägung in einem grafischen Interpreter oder UI hat.

B.1.3 Anforderung 3: Modellierungsdetailgrad

Anforderung 3 wurde entwickelt in Kapitel 5.3.

In *Summe* haben die Teilnehmer die einhellige Meinung, dass die Anforderung sehr wichtig ist. Der Prototyp steht aber kommerziellen Tools nach, insbesondere auf Grund von schlechterer Nutzbarkeit beim Setzen von Eigenschaftswerten.

B.1.3.1 Wichtigkeit

Wie auch bei Anforderung 2, sind die Teilnehmer bei Anforderung 3 einhellig der Meinung, dass sie *sehr wichtig* ist. Die *Entwickler sollten nicht eingegrenzt werden* (Z4.3.6), sondern die volle Leistungsfähigkeit des Toolkits zur Verfügung haben (Teilnehmer 5). Bei Einschränkung wird die Gefahr gesehen, dass Anforderungen an die UI nicht erfüllt werden können (Teilnehmer 4). Teilnehmer 6 ging noch weiter und formulierte zur Unnötigkeit der Einschränkung: "*was will ich mit einem Tool, das nichts kann*".

Neben der Freiheit ist auch die Abstraktion ein wichtiges Kriterium. Werden *nur abstrakte Eigenschaften angeboten, so wird der Entwickler versuchen, diese "wieder einzufangen"* (darum herumzukommen, um das gewünscht Konkrete zu erhalten) (Teilnehmer 3).

B.1.3.2 Umsetzung

Auch hier die Parallele zu Anforderung 2 – eine *gemischte Beurteilung* durch die Teilnehmer. Die prinzipielle Abdeckung des notwendigen Detailgrades wurde durch Teilnehmer 5 gesehen, der bestätigte, dass alle Eigenschaften über den Eigenschaftseditor von Eclipse bearbeitet werden können. Auch hat seiner Meinung nach das getestete Adaptionskonzept zur Skalierung alle relevanten Eigenschaften modifiziert.

Dagegen bemängelten Teilnehmer 4 und Teilnehmer 6 *Nutzbarkeitsprobleme. Insbesondere der durch Eclipse geprägte Eigenschaftseditor* wurde als veraltet bezeichnet und stach durch umständliche Interaktion heraus. Grundlegend wurden mehr WYSIWYG Möglichkeiten beim Bearbeiten von Eigenschaften wie Komponentenbeschriftungen gewünscht (vgl. Abschnitt B.2.4.1)

Teilnehmer 3 resümierte, dass generell *alle Anpassungen möglich* sind und vor allem *ohne den UI typischen "Boilerplate" Code.*

B.1.4 Anforderung 4: Einfache Nutzbarkeit

Anforderung 4 wurde entwickelt in Kapitel 5.4.

In *Summe* ist es das wohl wichtigste aller Kriterien. Rein Code basierte Werkzeuge sind nicht mehr adäquat, stattdessen ist direktes Manipulieren im Gegensatz zu abstrakten Modellen wichtig – die pragmatische Herangehensweise bestimmt maßgeblich, was nutzbar ist. Der Ansatz zeigt gute Ideen für die Umsetzung, aber ist kommerziellen Werkzeugen nachstehend.

B.1.4.1 Wichtigkeit

Von allen Teilnehmern wurde diese Anforderung als sehr wichtig bewertet – die wohl *Wichtigste aller Anforderungen.* Teilnehmer 6 fasste zusammen, dass sich Werkzeuge ohne gute Nutzbarkeit nicht mehr verkaufen lassen.

Die Nutzbarkeit *bestimmt maßgeblich die Effizienz* sowie den Arbeitsalltag durch Spaß oder Frust, so Teilnehmer 4. *Oft ist ein Konzept an sich nicht schwer, aber die Nutzung des Konzeptes birgt Hürden* (Z4.3.8). Dabei *bestimmt die Zielgruppe, was für sie "nutzbar" bedeutet*: So wurden in Abschnitt B.2.4.3 der Unterschied von Designern und Programmierern diskutiert, Abschnitt B.2.3.1 beschreibt unterschiedliche Nutzerintentionen beim Propagieren und Abschnitt B.2.1.3 Nutzer (Entwickler) spezifische Modellierungspräferenzen.

Die *direkte Manipulation ist zu bevorzugen*, wie in Abschnitt B.2.5.1 diskutiert, sonst "*suchen die Entwickler nach Wegen, die Abstraktion wieder einzufangen*" (Teilnehmer 3). Generell wird einfache Nutzbarkeit stark mit WYSIWYG Themen in Verbindung gebracht. So wurden von den Teilnehmern *noch weitergehende WYSIWYG Konzepte* gewünscht, wie in Abschnitt B.2.4.1 ausgeführt (Z5.2.2), (ZZ6.1.6). Die *Konzepte sollten deklarativ(er) aber gleichzeitig sehr konkret sein*, wie z.B. die Skalierung eines Fensters inkl. Inhalts durch interaktives Ziehen an der rechten unteren Ecke (Z4.1.6), (Z3.2.4).

Aber neben WYSIWYG sind andere Komponenten auch wichtig. Die *(visuelle) Darstellung nicht visueller Bestandteile* der Schnittstelle (z.B. über die Verhaltensansicht), wie in Abschnitt B.2.4.2 zu Visualisierungen erörtert. Auch sollten die verschiedenen *Werkzeuge für eine einfache Benutzung sehr gut integriert* sein, um zu jeder Situation (und in jeder Ansicht) die passenden Aktionen "situativ" anbieten zu können (Z6.1.6, Z6.1.8).

Dies hängt stark mit dem beobachteten *Pragmatismus* zusammen, der in Abschnitt B.2.1.1 zu allgemeinen Architekturthemen B.2.4.1 beschrieben ist: Man will in jeder Situation "alles einfach machen" können.

B.1.4.2 Umsetzung

Die Umsetzung der Anforderung, so die Bewertung der Teilnehmer, birgt einige gute Punkte, ist in Summe aber kommerziellen Produkten nachstehend.

Konzepte, wie die Verfeinerungsansicht bieten eine gute Übersicht und Orientierung bei Aktionen wie dem Propagieren von Änderungen (Teilnehmer 4). Auch die Kapselung und einfach nutzbare Bereitstellung der modularen Adaptionskonzepte wurde gut aufgenommen – sie ist zielgerichteter und komfortabler (Z5.1.5).

Dagegen ist die Basis der prototypischen Umsetzung (Eclipse), insbesondere mit dem Eclipse basierten Eigenschaftseditor, sowie der generierte generische Editor gewöhnungsbedürftig in der Nutzung (Z4.4.3), (Z5.1.1) (Teilnehmer 6 bei Diskussion der Umsetzung von Anforderung 3). Grundsätzlich bildet sich aber nach einiger Gewöhnung ein intuitives Gefühl für das Vorgehen mit dem Ansatz heraus (Teilnehmer 5).

B.1.5 Anforderung 5: Integration Struktur und Verhalten

Anforderung 5 wurde entwickelt in Kapitel 5.5.

In *Summe* wurde die Wichtigkeit gemischt aufgenommen. Das Erben von Verhalten hat sehr positive Effekte, ist aber, wohl auf Grund seiner Indirektheit, nicht intuitiv. Die Verhaltensansicht zeigt eine interessante Sicht auf die MBS, sollte aber mit mehr direkter Manipulation ausgebaut werden.

B.1.5.1 Wichtigkeit

Die Wichtigkeit von Anforderung 5 wurde sehr gemischt aufgenommen. Es wird aber gesehen, dass sie wichtig für die Anpassung an unterschiedliche Plattformen ist (Teilnehmer 3). Bei dieser Anpassung sollte man nicht "von Null" anfangen müssen (Teilnehmer 4, Teilnehmer 5). Durch die Erfüllung der Anforderung somit kann Arbeit gespart werden (Z4.3.4, Z4.2.6), und die Konsistenz unterstützt werden (Z4.1.4) Das implizite Erben von Verhalten über die Verfeinerungsbeziehung ermöglicht dies alles, steht aber teilweise in Konflikt mit herkömmlichen Ansätzen und Produkten. Dort werden Event Handler als Eigenschaften des Interaktionsobjektes zum Anknüpfen von Verhalten genutzt und die Manipulation findet i.A. explizit und nicht implizit statt, wie in Abschnitt B.2.3.3 beschrieben.

Auch ist eine klare Architektur wichtig, wie aus Abschnitt B.2.1.1 hervorgeht. Wie streng die Architektur jedoch sein muss, wird von verschiedenen Teilnehmern unterschiedlich gesehen (vgl. Abschnitt B.2.1.2).

B.1.5.2 Umsetzung

Die Umsetzung wurde von den Teilnehmern positiv aufgenommen, wohl auch (so Teilnehmer 4), weil die Aufgaben den Aspekt gut beleuchteten.

Positiv bewertete Teilnehmer 3, dass die Verhaltensansicht die Schnittstelle zwischen View und Controller zeigt (Z3.3.7). Dabei wurden Verbesserungen an der Darstellung gewünscht (Teilnehmer 6), sowie weitere Wizards zur einfachen Erstellung von Beobachter- und Verhaltensfragmenten. Intuitiv jedoch wurde von einigen Teilnehmern die Verhaltensansicht als Pool von zuordenbaren Verhaltensfragmenten interpretiert, welche über direkte Manipulation bearbeitet werden können (vgl. Abschnitt B.2.5.1).

Letztlich lässt sich auch diskutieren, ob ein generischeres Event Modell Sinn macht, wie in Abschnitt B.2.1.2 behandelt.

B.1.6 Anforderung 6: Modifikation

Anforderung 6 wurde entwickelt in Kapitel 5.6.

In *Summe* wurde die Anforderung einhellig als wichtig bewertet. sie ist eine zentrale Eigenschaft des Ansatzes und ist im Einklang mit dem Standard "Kopieren und Anpassen" Vorgehen der Praxis. Die Kombination der Anforderung mit Generierungstechniken (z.B. Transformationen) ist gut möglich. Schließlich ist das Anbringen einer Änderung an mehreren Stellen (Propagation) gut und der Ansatz Konsistenz fördernd. Die Umsetzung ist sauber und nutzbar.

B.1.6.1 Wichtigkeit

Die Anforderung wurde von allen Teilnehmern als wichtig bewertet. Teilnehmer 3, Teilnehmer 5 und Teilnehmer 6 bezeichneten den unterstützten Ansatz mit Ableiten und Anpassen einer MBS-Variante als ein typisches Vorgehen. Es kann ein Grundlayout verfeinert werden. Teilnehmer 6 hob hervor, dass die Verfeinerung über mehrere Ebenen gut ist. Dies steht interessanter Weise im Konflikt mit der Aussage zu Anforderung 1, wie im Hauptteil der vorliegenden Arbeit, Abschnitt 17.3.3 diskutiert.

Der verfolgte Ansatz kann auch gut mit Generierungsmechanismen verbunden werden (Teilnehmer 6). Auch ist es vorteilhaft, dass Modifikationen auch in konkreteren Verfeinerungen einfach möglich sind (Teilnehmer 4). So resümiert Teilnehmer 4, dass eventuell die Konsistenz verbessert werden kann und eine Zeitersparnis möglich ist, wie in Abschnitt B.2.7.1 zur Effizienz des Ansatzes zusammengefasst.

Positiv äußerten sich Teilnehmer 3 und Teilnehmer 6 über das dabei genutzte Propagationskonzept, welches sie "gut" fanden. Insbesondere findet Teilnehmer 4 gut, dass man die Vererbungen in der Verhaltensansicht nachvollziehen kann. Allerdings ist Verständnis von Vererbungshierarchien notwendig, so Teilnehmer 3. Letzteres wurde aber durch Teilnehmer 4 in Frage gestellt, da der Teilnehmer (Design Hintergrund) keine tiefer gehenden Kenntnisse zu Vererbungskonzepten hatte, das Konzept aber dennoch nutzte.

B.1.6.2 Umsetzung

Von allen Teilnehmer wurde die Umsetzung einhellig als gut bewertet. Teilnehmer 5 bezeichnete dies als die zentrale Eigenschaft des Ansatzes. Sie ist "sauber nutzbar" und liefert das gewünschte Ergebnis (Teilnehmer 3, Teilnehmer 6).

B.2 Diskussion der identifizierten Themen

Die folgenden Unterkapitel sind das Ergebnis des dritten Schrittes der im Abschnitt 17.2.4 vorgestellten Auswertung: Die Diskussion der in der Studie als relevant identifizierten Themen. Die Unterkapitel sind in thematische Blöcke aufgeteilt, welche während der Analyse identifiziert wurden.

B.2.1 Architektur

Dieser Abschnitt behandelt Ergebnisse bzgl. der Architektur von MBS (Unteranforderung 5.4). Zusammenfassend sollte ein Architekturmuster vorliegen, welches ein klare Zuordnung von Funktionalitäten zu Architekturteilen ermöglicht. Des Weiteren soll es flexibel sein; Architekturmuster (und Framework) sollten eine pragmatische Herangehensweisen nicht blockieren. Der vorliegende Ansatz bietet laut Teilnehmer ein weitgehend konsistentes Konzept an, welches den Entwickler zur Einhaltung des Architekturmusters zwingt.

Die Wahrnehmung, welche Eigenschaften für eine Architektur relevant sind, ist jedoch stark Teilnehmer spezifisch, wie auch die Präferenz zur Modellierung von Hintergrund und Rolle der Teilnehmer abhängt. Einen erfolgsversprechenden alternativen Ansatz zur Modellierung bieten Rekombinationskonzepte, da diese bekannt und intuitiv von Entwicklern nutzbar sind. Von den Teilnehmer wurden noch weitergehende und deklarativere Unterstützungskonzepte gewünscht und die tiefere Integration aller Teile der Entwicklungsumgebung.

B.2.1.1 Allgemeine Architekturüberlegungen

Im Rahmen der Studie wurde auch das in Kapitel 8 vorgestellte Architekturmuster besprochen. Bei Architekturen sind laut Teilnehmer 6 die typischen Probleme *i)* die *unklare Zuordnung* von konkreten Elementen zu Teilen des Frameworks oder des Musters, *ii)* die *unzureichende Flexibilität* des Frameworks, und *iii)* dass oft *ein pragmatischer Ansatz* gegen das Framework verstößt (Z6.3.9). In den Studien konnte vor allem der pragmatische Umgang bestätigt werden, so wurde z.B. der Systemzustand in der UI gesucht (Z6.2.8).

Allerdings sehen andere Teilnehmer den starken Zwang, sich an ein Framework oder Muster zu halten, auch positiv (Z3.5.3). Autoren wie Stahl u. a. 2007 schreiben einem solchen Zwang, auf Grund der somit sauberen Architektur, auch Kostenersparnisse in der Wartung zu[1]. Die Wahrnehmung dieser von Teilnehmer 6 geäußerten Punkte scheint somit *Teilnehmer spezifisch*. Dies legt auch die Diskussion der unterschiedlichen Modellierungspräferenzen in Abschnitt B.2.1.3 nahe sowie Kommentare spezifisch zum evaluierten Ansatz im Folgeabschnitt B.2.1.2.

B.2.1.2 Architekturüberlegungen spezifisch für Ansatz

Das vorgestellte Architekturmuster aus Kapitel 8 wurde von den Teilnehmern *weitgehend als konsistent* bezeichnet (Z3.5.4), (Z4.3.2). Wobei ein Teilnehmer leichte Abstriche machte: So ergaben sich bei der Zuordnung einer Funktion(sänderung) in das Architekturmuster mehrere Lösungsmöglichkeiten (Z6.3.8). Dass die prototypische Umsetzung der Infrastruktur die Beitragsklassen (vgl. Kapitel 13.1.2) nicht voneinander erben lässt, wurde als inkonsistent identifiziert. So sollten Beitragsklassen analog zur Verfeinerungshierarchie voneinander erben können (Z3.5.6). Generell wurde die Behandlung von Fragmenten mit den Möglichkeiten Create, Update und Delete (vgl. Kapitel 8.2) als passend bewertet (Z4.3.3), vgl. Unteranforderung 5.1 und Unteranforderung 6.1.

Beim vorgestellten Ansatz wird der *Entwickler stärker gezwungen, ein Framework und Architekturmuster einzuhalten* und anzuwenden (Z3.5.3). Dies widerspricht teilweise dem von Nutzern intuitiv eingesetzten Pragmatismus, wie

[1] Eine Eigenschaft, welche die Autoren spezifisch modellgetriebenen Ansätzen zuschreiben, vgl. Kapitel 3.2.

in Abschnitt B.2.1.1 beschrieben, wird aber von den Teilnehmern unterschiedlich bewertet.

Abschnitt B.2.1.1 spricht auch die teilweise gewünschte, größere Flexibilität an. Teilnehmer 6 machte hierbei Vorschläge für ein weniger spezialisiertes Eventkonzept (Z6.1.1) im Gegensatz zur scharfen Trennung von Interaktions- und Änderungsevents im vorliegenden Ansatz (vgl. Kapitel 8). Auch ein leicht anderer Ansatz zur Umsetzung von Beobachtern wurde diskutiert (Z.6.2.9).

Schlussendlich wurde eine *noch weitergehende Integration der einzelnen Unterstützungskonzepte gewünscht*. So sollte das Propagieren eines Interaktionsobjektes die vorher auf die Zielvarianten angewendeten Adaptionen mit berücksichtigen (Z4.2.5), sonst können inkonsistente Ergebnisse entstehen (Z5.2.4). Weitere Ergebnisse zur stärken Integration sind auch in Abschnitten B.2.4.1 und B.2.5.1 zu finden.

B.2.1.3 Konkrete Modellierungspräferenz nutzerabhängig

Es zeigte sich, dass der *Hintergrund der Teilnehmer und ihre Rollen die Erwartungshaltung bzgl. Konzepten und deren Nutzung bestimmen*. Entwickler unterschiedlichen Projekthintergrunds (z.B. Web- gegenüber Rich Clients Entwicklung) haben laut Teilnehmer 3 verschiedene Zugänge zur Problemstellung (Z3.5.1). Auch starten unterschiedliche Rollen (z.B. Usability fokussierte versus technisch orientierte) mit unterschiedlichen Arbeiten bei der Adaption einer Benutzerschnittstelle (Z4.1.1). Darüber hinaus stellen unterschiedliche Rollen auch unterschiedliche Anforderungen an ihre Werkzeuge (Z4.4.4). Insbesondere unterscheiden sich auch die Bedürfnisse von technisch orientierten Rollen versus Design orientierten Rollen, wie in Abschnitt B.2.4.3 diskutiert.

Generell bietet das Konzept der *Rekombination eine gute Alternative zur Verfeinerung* (vorgestellt in Kapitel 9.1.3). So werden insbesondere föderierte Benutzerschnittstellen intuitiv als Rekombination existierender Teilbenutzerschnittstellen wahrgenommen, wie in Abschnitt B.2.2.2 diskutiert. Teilnehmer 6 präferiert diese Regruppierungsmetapher, welche gleich mächtig wie der vorliegende Ansatz wäre. Er weißt aber darauf hin, dass dies nur für ihn persönlich adäquat sei und andere Personen andere Präferenzen haben (Z6.3.7). Bei dem vorgeschlagenen Ansatz wären nicht UUIBoxen primäres Modellierungselement, sondern die Interaktionsobjekte selbst.

Auch wurden *weitergehend deklarative Konzepte zur Unterstützung gewünscht*. So schlägt Teilnehmer 4 vor, semantische Gruppen zu nutzen (Z4.3.1). Eine solche ist eine Zusammenstellung von Interaktionsobjekten. Die Gruppe kann zusammen angelegt oder gelöscht werden und die Entwicklungsumgebung weißt den Entwickler z.B. darauf hin, wenn ein Element der Gruppe nicht eingesetzt wird. Allgemein muss aber beachtet werden, dass der gewünschte Grad der Automatisierung variiert (vgl. Abschnitt B.2.3.1), aber auch der gewünschte Grad der Flexibilität der Architektur und des Frameworks (vgl. Abschnitte B.2.1.1 und B.2.1.2).

B.2.2 Modellierung

Dieser Abschnitt behandelt Studienergebnisse zur Modellierung. Generell wurde die Kombination von abstrakter und konkreter Modellierungsebene sehr positiv aufgenommen (Anforderung 1), wobei keine die andere ersetzen kann. In der abstrakten Ebene können Hardware basierte Benutzerschnittstellen genau so modelliert werden, wie grafische. Toolkit übergreifende Arbeiten können hier einfach erledigt werden, so auch die Beschreibung föderierter Benutzerschnittstellen, wobei diese intuitiv als Kombination existierender Teilbenutzerschnittstellen und nicht als eine UI aufgefasst werden. Generell werden weitergehende und gleichzeitig mehr deklarative Unterstützungskonzepte gewünscht.

B.2.2.1 Toolkit übergreifende Arbeiten mit generischem Editor

Der generische Editor wurde mit Hilfe von GMF[2] generiert und leicht angepasst (vgl. Kapitel 14.3.2). Diese leichten Anpassungen konnten aber die Nutzbarkeitsprobleme des generierten Editors nicht kompensieren (Z5.1.1). Dennoch bietet der generische Editor, vor allem für Toolkit übergreifende Arbeiten auf einer gemeinsamen Abstraktionsebene, Potenzial (Z3.4.7). Insbesondere wird die Migration zwischen Toolkits als Anwendungsfall gesehen (Z4.2.9).

Die Arbeiten auf einer *gemeinsamen Abstraktionsebene (Anforderung 1) wurden von den Teilnehmer sehr positiv aufgenommen*. So wurde die Möglichkeit, Interaktionsobjekte aus einem Swing-basierten Teil UI in das Hardware basierte Teil UI (vgl. Kapitel 13.3.2 für das Hardware Toolkit) zu ziehen als "genial" bezeichnet (Z4.2.9) und war für Teilnehmer 5 "schlüssig" (Z5.3.2). Auch die anderen Teilnehmer befanden die Möglichkeit für gut: Änderungen an der Klasse eines Interaktionsobjektes vorzunehmen wurde als "charmant" (Z6.3.5) bzw. "super" (Z4.2.9) bezeichnet. Teilnehmer 6 fasste zusammen, dass er die Arbeit mit dem generischen Editor an Toolkit übergreifenden Themen "angenehm" fand, sie "favorisiert" und bezeichnete sie als „effizient" (Z6.3.5).

Abschließend wurde ein *eher deklarativer Ausbau der Unterstützungskonzepte gewünscht*, welcher z.B. in Richtung Makros zur Toolkitmigration, sowie Behandlung ganzer Gruppen von Interaktionsobjekten gehen kann (Z6.3.4).

Auf der gemeinsamen Ebene wurden aber nur solch abstrakte Tätigkeiten, wie das Ändern von Klassen vorgenommen. Daher kann davon ausgegangen werden, dass ein solch *generischer Editor in keiner Weise eine WYSIWYGartigen Editor ersetzt* - was auch der Wunsch nach noch stärkerem WYSIWYG der Teilnehmer reflektiert (siehe Abschnitt B.2.4.1).

B.2.2.2 Beschreibung Föderierter Benutzerschnittstellen

Bei der Modellierung föderierter Benutzerschnittstellen *denken Teilnehmer intuitiv an Rekombination existierenden Teilbenutzerschnittstellen* für die einzel-

[2] GMF – Graphical Modeling Framework von Eclipse

nen Geräte. Dies führt zu initialen Verständnisproblemen mit dem vorgestellten Ansatz: So versuchte Teilnehmer 3 Aufgabe 3 durch die Kombination zweier UUIBoxen zu lösen. Jede Box sollte ein Teil UI enthalten (Z3.4.1). Das Problem ist jedoch Teilnehmer spezifisch: Teilnehmer 5 wählte bei der Aufgabe gleich den korrekten Ansatz, beide Teil UIs in einer UUIBox zu modellieren (Z5.2.5, Z5.3.1). Auch äußerte er explizit, dass er die vom Ansatz vorgegebene Anordnung der UIs zur Modellierung föderierter Benutzerschnittstellen für sinnvoll halte (Z5.2.5).

Das Modellieren föderierter Benutzerschnittstellen hängt stark mit der Arbeit auf einem gemeinsamen Abstraktionsniveau für verschiedene Toolkits zusammen, wie in Abschnitt B.2.2.1 diskutiert. Dabei wird auf dem gemeinsamen Abstraktionsniveau die *Modellierung von Hardware UIs (z.B. das Sichterboard aus Aufgabe 3 und 4) identisch zu der von grafischen UIs* (Z3.4.9).

B.2.3 Verfeinerung

Das zentralste Konzept des Ansatzes ist das der Verfeinerung, wie in Kapitel 9.1.3 eingeführt (Anforderung 1 und Anforderung 6). Dabei traten Verständnisprobleme dadurch auf, dass sich nicht alle Objekte beim Verfeinern gleich verhalten: Es gibt einen Unterschied zwischen der Erstellung von Interaktionsobjekten und dem Setzen von Eigenschaften. Daneben stellte sich das Konzept der UUIBox als schwierig heraus. Während viele Teilnehmer grundsätzlich keine Schwierigkeiten mit dem Ausführen der Verfeinerung hatten, entstanden beim Verwenden von UUIBoxen zum Modellieren föderierter UIs konzeptionelle Missverständnisse. Auch die Möglichkeit, Verhalten mittels Verfeinerung zu erben, stellt einerseits eine Erleichterung im Arbeitsablauf dar, führt aber andererseits eine möglicherweise unerwünschte konzeptionelle Indirektion ein.

Insgesamt wurde das Konzept der Verfeinerung positiv aufgenommen. Alle Teilnehmer erkannten das Potential, durch die Abstraktion gemeinsamer Eigenschaften von Benutzerschnittstellen Arbeit zu sparen.

B.2.3.1 Nutzerintention beim Einfügen von Interaktionsobjekten

Die Teilnehmer stellten bei den Studien eine *Inkonsistenz zwischen dem Anlegen eines Interaktionsobjektes und dem Vererben von Eigenschaften* fest. Nachdem die Teilnehmer die passive Propagation für Eigenschaften (vgl. Kapitel 9.2) kennen gelernt hatten, nahmen die Meisten an, dass beim Hinzufügen eines Interaktionsobjektes, dies automatisch auch in die darunter liegenden MBS-Varianten eingefügt wird (Z33-1, Z35-7), (J41-3). Stattdessen muss es mit Hilfe eines Werkzeugs propagiert werden (vorgestellt in Kapitel 10.1.4.1).

Die *automatische Propagation eines neuen Interaktionsobjektes* wird jedoch von Teilnehmern 4 und 5 als intuitiver und besser für den Arbeitsablauf bewertet (Z41-7), (Z51-2). Lediglich bei Teilnehmer 6 ergeben sich Abweichungen. Er sagt zwar, dass der “Standard Usecase” darin bestehe, neu erzeugte In-

teraktionsobjekte zu propagieren (Z63-1), allerdings möchte er die Aktion des Propagierens nach wie vor *selbst anstoßen* (Z63-2). Der Teilnehmer hatte mehrere Interaktionsobjekte in einer abstrakteren Verfeinerung erstellt, weshalb er gerne alle neu erstellten Elemente in einem einzigen Schritt propagieren können würde (Z63-1). Der Großteil der Teilnehmer favorisiert somit das automatische Propagieren des Interaktionsobjektes, allerdings zeigt die Aussage von Teilnehmer 6, dass einzelne Benutzer mehr Kontrolle behalten wollen und diese Lösung somit nicht für jeden Benutzer optimal ist.

B.2.3.2 UUIBox Konzept (Nutzungskontexte)

Es kann resümiert werden, dass die Teilnehmer *Schwierigkeiten* hatten, *die Aufgabe einer UUIBox zu verstehen und dass eine UUIBox genau einem Nutzungskontext* entspricht. Diese Probleme traten vor allem bei Aufgabe 3, der Modellierung einer föderierten Benutzerschnittstelle, auf (vgl. Abschnitt B.2.2.2).

Teilnehmer 3 versuchte, Aufgabe 3 durch die Kombination (vgl. Kommentare dazu in Abschnitt B.2.1) zweier UUIBoxen zu lösen (Z33-1). Dies entspricht jedoch nicht dem intendierten Vorgehen, bei dem beide Teil-UIs in einer UUIBox modelliert werden (vgl. Kapitel 10.1.1). Um festzustellen, ob hier tatsächlich ein konzeptioneller Fehler vorlag, wurde in den darauffolgenden Studien zu Anfang genau erklärt, dass föderierte Benutzerschnittstellen in einer UUIBox abgelegt werden.

Dies löste bei Teilnehmer 4 ein gegenläufig geartetes Problem aus: Er versuchte bei Aufgabe 1 die für den Tisch angepasste MBS-Variante mit der MBS-Variante, von welcher es abgeleitet wurde, in die gleiche UUIBox zu legen (Z41-8). Auch sagte Teilnehmer 4, dass ihm nicht klar sei, dass eine “Box in der Verfeinerungsansicht” (Kapitel 10.2.1) tatsächlich dem Modellierungselement UUIBox entspricht (J43-9). Er bezeichnete das UUIBox-Konzept explizit als “schwierig” und schlug vor, gegebenenfalls dessen Benennung zu überdenken (A48-12:07).

Teilnehmer 5 wählte bei Aufgabe 3 direkt den korrekten Ansatz und platzierte die Swing und Hardware Interaktionsobjekte nebeneinander in einer UUIBox (Z52-5). Allerdings erstellte Teilnehmer 5 nicht als ersten Schritt eine neue UUIBox. Ihm war also nicht intuitiv klar, dass die Aufgabe 3 die Erstellung eines neuen Nutzungskontexts erforderte und daher die Ableitung einer neuen Verfeinerung (wie z.B. in Kapitel 10.1.4.2 beschrieben) der erste Schritt ist (Z52-8,9). Bei Teilnehmer 6 hingegen traten keine derartigen Schwierigkeiten auf.

Zusammenfassend hatte ein Großteil der Teilnehmer *Probleme, das von UUIBoxem modellierte Konzept zu verstehen oder die Verbindung zwischen dem Modellierungselement und dem Konzept eines Nutzungskontextes herzustellen.* Möglicherweise ist einer der Gründe dafür eine inkonsistente oder unintuitive Benennung des Modellierungselements.

B.2.3.3 Erben von Verhalten

Die Möglichkeit, von übergeordneten MBS-Varianten *Verhalten zu erben wurde von den meisten Teilnehmern gut angenommen* (vgl. Unteranforderung 1.1 und Unteranforderung 5.1). Teilnehmer 3 fand sich sehr schnell damit zurecht (Z33-1). Allerdings ging er beim Implementieren der Beitragsklassen davon aus, dass diese analog zu ihren UUIBoxen im objektorientierten Sinn voneinander erben (Z35-6). Dies ist in der prototypischen Umsetzung (vgl. Kapitel 13.1.2) nicht der Fall.

Teilnehmer 4 äußerte, dass das *Verfeinerungskonzept (auch im Bezug auf Verhalten) gut für die Konsistenz* zwischen den verschiedenen MBS-Varianten sei und *im Entwicklungsprozess helfe* (Z41-4, Z43-4). Teilnehmer 6 versuchte sogar innerhalb einer Verfeinerung Verhalten zu erben (Z62-5). Wie in Abschnitt B.2.5.1 besprochen, hatten die Teilnehmer jedoch Probleme mit der Indirektion durch die Verfeinerung.

Generell wurde aber die Behandlung von Fragmenten mit den Möglichkeiten *Create*, *Update* und *Delete* von Teilnehmer 4 als passend bewertet (Z4.3.3), vgl. auch Abschnitt B.2.1.2.

B.2.4 Unterstützung des Arbeitsablaufes

Dieser Abschnitt befasst sich damit, wie die durch das Konzept und die prototypische Umsetzung implizierten Arbeitsabläufe von den Teilnehmern bewertet wurden. Dabei wurde vor allem die konsequente Nutzung des WYSIWYG Paradigmas zum Editieren (vgl. Unteranforderung 2.1) und Visualisieren der MBS (vgl. Unteranforderung 1.1) sehr positiv aufgenommen. Die Teilnehmer waren sich jedoch einig, dass noch Verbesserungspotential besteht. Außerdem wünschten sich einige Teilnehmer, noch stärker mit Visualisierungen zu arbeiten um die Orientierungsmöglichkeit bei der Arbeit weiter zu verbessern.

Schließlich wurde festgestellt, dass sich der Arbeitsablauf von Entwicklern unterscheidet, je nachdem, ob sie sich auf Design oder die Programmierung von Benutzerschnittstellen konzentrieren. Dies deckt sich auch mit unterschiedlichen Modellierungspräferenzen, welche in Abschnitt B.2.1.3 diskutiert wurden. Im Rahmen der Diskussion von Anforderung 4 (Kapitel B.1.4) wurde darüber hinaus resümiert, dass der Ansatz die Rollen von Designer und Entwickler nicht scharf trennt, was je nach Arbeitsablauf aber gewünscht wäre.

B.2.4.1 WYSIWYG ist gut, kann aber noch weiter gehen

Die Teilnehmer hoben mehrfach hervor, dass es positiv sei, Benutzerschnittstellen nach dem WYSIWYG Prinzip erstellen zu können. Teilnehmer 4 bemerkte, dass das *direkte Manipulieren von Benutzerschnittstellen sowie direktes Feedback für den Designer sehr wichtig* sind (Z44-1) – vgl. Unteranforderung 4.1 und Unteranforderung 4.2. Jedoch wurde auch oft bemerkt, dass die Unterstützung noch weiter gehen könnte. Teilnehmer 4 wünschte sich z.B. die Beschriftung von

Komponenten direkt im WYSIWYG Interpreter setzen zu können und nicht in einem separaten Eigenschaftseditor (Z42-1).

Insbesondere würde *eine noch engere Integration der Werkzeuge dafür sorgen, dass in jeder Situation das passende Werkzeug direkt erreichbar ist* (vgl. Unteranforderung 4.3). So empfand es Teilnehmer 4 als hilfreich, dass viele der Standardoperationen für Interaktionsobjekte über einen Rechtsklick auf das Element im Editor direkt auswählbar sind (Z42-1) – vgl. Kapitel 10.2.1 und 14.3. Er sagte, dass der vorliegende Prototyp ein *"Super Tool" wäre, wenn alle Funktionen nach dem WYSIWYG Prinzip verwendbar wären* (Z42-1). Ähnliches zur engeren Integration bemerkte auch Teilnehmer 6 (Z6.1.8, Z6.1.6).

Für die direkte Unterstützung *wurden mehr deklarative Konzepte gewünscht.* Teilnehmer 3, 5 und 6 sagten im Bezug auf die Skalierung eines UIs mit dem Prototyp des Adaptionswerkzeugs (vgl. Kapitel 14.3.3), dass ein interaktives "Großziehen" der UI besser wäre als die Angabe horizontaler und vertikaler Skalierungsfaktoren (Z32-4), (Z51-3), (Z61-6).

Auch wünschten sich Probanden Visualisierungsmöglichkeiten, die über das Darstellen von tatsächlich visuellen Elementen hinausgeht, vgl. hierzu den folgenden Abschnitt B.2.4.2.

B.2.4.2 Visualisierung nicht-visueller Konzepte ist hilfreich

Die *Visualisierung von nicht visuellen Konzepten wird von den Teilnehmern als hilfreich empfunden.* Teilnehmer 3 befand insbesondere die Visualisierung der View Controller Schnittstelle in der Verhaltensansicht (Kapitel 10.2.2) für gut (Z33-7). Teilnehmer 4 bemerkte, dass die *Exposition der Verfeinerungsbeziehungen* in der Verhaltensansicht zum Zweck der Nachverfolgbarkeit gut ist (Z54-1), vgl. Unteranforderung 4.1 und Unteranforderung 5.2.

Noch umfangreichere Visualisierungen regte Teilnehmer 5 an, der den MBS Zustand inkl. Änderungsevents (vgl. Kapitel 8.3 und 8.5.2) interaktiv untersuchen wollte (Z52-2). Darüber hinaus, schlug er vor, Beobachterfragmente in einer Tabelle zu visualisieren und zu manipulieren (Z52-3). Er gab an, dass es hilfreich wäre, alle Teile einer MBS grafisch darzustellen (Z52-2). Teilnehmer 6 äußerte sich nicht zu dieser Thematik.

B.2.4.3 Bedürfnisse Designer versus Programmierer

Abschnitt B.2.1.3 beschrieb bereits unterschiedliche Präferenzen bei der Modellierung – insbesondere bei Designern gegenüber technisch orientierten Personen. Das Konzipierungsverständnis unterscheidet sich schon innerhalb der technisch orientierten Personen, z.B. mit dem Hintergrund in Entwicklung von Webanwendungen, mit Pull Paradigma, gegenüber Desktopanwendungen, mit Push Paradigma (Z51-1). Im Vergleich zu Teilnehmer 4, der als einziger ein reiner Designer ist und im beruflichen Alltag nicht programmiert, wurden *Unterschiede in der Vorgehensweise* noch deutlicher. So begann Teilnehmer 4 bei Aufgabe

1 (Adaption der UI an ein Tischgerät, ähnlich, wie in Kapitel 16.3 ausgeführt) damit, über die genauen Gegebenheiten auf dem Zielgerät und deren Konsequenzen für das Interaktionsdesign nachzudenken, während Teilnehmer 3 direkt anfing die Benutzerschnittstelle anzupassen (Z41-1).

Teilnehmer 4, *dem Designer, erschien die Benennung der Konzepte in der prototypischen Umsetzung oft unintuitiv* (Z41-9, Z42-3). Außerdem bezeichnete Teilnehmer 4 die Eclipse basierten Werkzeuge als schwierig zu bedienen (Z43-8, Z44-3). Daher empfahl Teilnehmer 4, zwei getrennte Arbeitsumgebungen anzubieten, die jeweils entweder den Designer oder den Programmierer als stereotype Zielgruppe haben. Es sollten dementsprechend unterschiedliche Namen für die Konzepte genutzt werden, die die jeweilige Zielgruppe intuitiv versteht (Z44-4).

B.2.5 Events und Verhalten

Dieser Abschnitt befasst sich damit, wie die Teilnehmer der Studie mit der Art, wie Verhaltensfragmente im Ansatz behandelt werden, umgehen. Es wurde insbesondere festgestellt, dass die Teilnehmer versuchten, die Zuordnung von Verhaltensfragmenten interaktiv mit der Verhaltensansicht zu modifizieren. Dies entspricht dem Wunsch der Teilnehmer nach mehr Möglichkeiten zur direkten Manipulation von Verhalten (Anforderung 4). Darüber hinaus wurde beobachtet, dass Teilnehmer, die mit gängigen Entwicklungswerkzeugen vertraut waren, Verhalten mittels einer Eigenschaft zuordnen wollten.

B.2.5.1 Direkte versus indirekte Manipulation

Während die Teilnehmer mit Verhaltensfragmenten arbeiteten, fiel auf dass sie oft versuchten, bereits bestehende Fragmente aus der Verhaltensansicht direkt mit Interaktionsobjekten zu verbinden. Im konkreten Fall (Aufgabe 3) konnten die Teilnehmer das Problem jedoch nur durch Manipulation der Verfeinerungsbeziehung lösen. Diese identifizierte *konzeptionelle Indirektion im Gegensatz zur direkten Manipulation machte den Teilnehmern Schwierigkeiten.*

Stattdessen wurden direkte Interaktionen von den Teilnehmern versucht. So versuchte Teilnehmer 4, ein Verhaltensfragment mit Hilfe der Verhaltensansicht (vgl. Kapitel 10.2.2) an ein anderes Interaktionsobjekt zu kopieren (Z42-7). Teilnehmer 6 hatte erwartet, dass in der Verhaltensansicht auch -Fragmente angezeigt werden, die keinem Interaktionsobjekt zugeordnet sind (Z62-3). Außerdem versuchte er, ein Verhaltensfragment aus der Ansicht heraus auf ein Interaktionsobjekt zu ziehen (Z62-4), was semantisch als dem Versuch des Kopierens von Teilnehmer 4 gleich gewertet werden kann. Teilnehmer 3 hätte das Kopieren von bestehendem Verhalten an neue Interaktionsobjekte als intuitiver empfunden als das Erben von Verhalten (Z34-5). Teilnehmer 5 versuchte an einer Stelle, ein Verhaltensfragment erst zu spezifizieren und dann manuell

an die entsprechenden Interaktionsobjekte zu binden. Das Konzept, die Verfeinerungsbeziehungen dafür zu verwenden wurde nicht intuitiv klar (Z51-7).

Die Indirektion, welche durch die Vererbung von Verhaltensfragmenten und Manipulierbarkeit der Verfeinerungsbeziehung entsteht, führt mitunter zu Verständnisproblemen. Es ist allerdings wichtig hervorzuheben, dass an dieser Stelle nur dann Probleme auftraten, wenn die Vererbung von Verhalten manipuliert werden sollte. Das Erben von Verhalten selbst wurde mehrfach als nützliches Feature bezeichnet (vgl. Abschnitt B.2.3.3).

B.2.5.2 Verhalten als Eigenschaft

Alle Teilnehmer bis auf Teilnehmer 4 versuchten während der Studie, *Verhaltensfragmente im Eigenschaftseditor aufzufinden* bzw. zu bearbeiten (Z34-5), (Z51-9), (Z62-7). Dies kann darauf zurückgeführt werden, dass in gängigen Entwicklungswerkzeugen (wie VisualStudio oder Eclipse) dies dort zu finden sind. Dafür spricht, dass Teilnehmer 4 die Zuordnung von Verhalten nicht im Eigenschaftseditor suchte und er der einzige Teilnehmer ist, der auf Grund seines Hintergrundes (Designer) nicht mit dieser Konvention vermutlich also nicht vertraut ist.

B.2.6 Adaption einer Benutzerschnittstelle

In diesem Abschnitt wird behandelt, wie die Teilnehmer auf Aufgabe 1, der Anpassung der Benutzerschnittstelle an einen interaktiven Tisch mit Hilfe des modularen Adaptionskonzeptes zur Skalierung (Konzept Kapitel 10.1.3, prototypische Umsetzung Kapitel 14.3.3), reagierten. Fast alle Teilnehmer nahmen die Arbeit mit dem Werkzeug sehr gut an, wobei ein deklarativeres Interaktionskonzept bevorzugt werden würde. Insbesondere die Möglichkeit der manuellen Nachbearbeitung war wichtig. Jedoch würde bei der direkten Anwendung ein evtl. vorab notwendiger Konzipierungsschritt übergangen.

B.2.6.1 Konzept geeignet für Aufgabe

Die Umsetzung des Adaptionskonzeptes zur Skalierung von Benutzerschnittstellen (vgl. Kapitel 14.3.3) wurde von fast allen Teilnehmern sehr gut angenommen. Teilnehmer 3 befand das Werkzeug als geeignet für die Aufgabe (Z32-2) und hob positiv hervor, dass man das Ergebnis des Adaptionsschritts noch nachbearbeiten kann (Z32-1). Teilnehmer 4 bemerkte bevor er das Werkzeug kennenlernte, dass er gerne eine skalierte Version des zu vergrößernden UIs als Arbeitsgrundlage hätte (Z41-4) und schlug Mechanismen ähnlich zum Skalieren von Grafiken vor (Z41-2). Nach Vorstellung des Werkzeugs bestätigte er, dass *es die automatisierbaren Schritte des Anpassungsprozesses abnimmt und somit Arbeit spart* (Z41-5, Z41-2). Dennoch sei *auf eine manuelle Nachbearbeitung nicht zu verzichten* (Z41-3, Z41-5), wie in Anforderung 3 formuliert. Teilnehmer 5 bezeichnete die Arbeitsweise mit dem Werkzeug als “komfortabler” und

"zielgerichteter" als den konventionellen Ansatz mit adaptiven Layoutmanagern (Z51-5). Teilnehmer 6 dagegen äußerte sich nach der Benutzung nicht positiv über das Werkzeug. Bei der Frage vor der Durchführung der Aufgabe nach dem optimalen Ansatz für solch eine Aufgabenstellung, schlug er aber auch vor, eine Funktionalität ähnlich dem Skalieren eines Bildes in einem Grafikprogramm zu verwenden (Z61-5).

Auch andere Teilnehmer kritisierten, dass die Parameter für die Skalierung als Faktoren in Dezimalbruchdarstellung eingegeben werden mussten. Teilnehmer 3, 5 und 6 schlugen vor, dass durch das Ziehen an einer Ecke des Interpreterfensters die *Skalierung auf die Zielgröße eher deklarativ* erfolgen sollte (Z32-4), (Z51-3), (Z61-6). Teilnehmer 4 hingegen schlug vor, die Zielgröße entsprechend der Gerätespezifikation direkt in Pixeln angeben zu können (Z41-6).

B.2.6.2 Konzipierungsschritt wird eventuell übergangen

Teilnehmer 4 ging bei der Lösung der Aufgabe 1 nicht direkt die Vergrößerung des gesamten UIs an. Stattdessen machte er sich zuerst Gedanken über die besonderen Eigenschaften des Zielgeräts. Er führte also *vor der eigentlichen Anpassung des UIs erneut einen Interaktionsdesignschritt* durch (Z41-1). Er bemerkte, dass es bei einer solchen Adaption entscheidend sei, wichtige Elemente puzzleartig an die passende Position zu verschieben (Z41-3) und das Layouting des UI von Hand zu "tunen", um vorhandenen Platz optimal auszunutzen (A64-4). Teilnehmer 6 merkte in diesem Zusammenhang ebenfalls an, dass *die Anpassung der Benutzerschnittstelle an die Spezifika des Zielgeräts bei MBS extrem wichtig* ist und kritisierte die nicht ausreichende Beachtung dieses zusätzlichen Schrittes (Z61-4). Dagegen kann argumentiert werden, dass der Ansatz einen vorgeschalteten Design Schritt nicht ausschließt.

B.2.7 Übergreifende Bewertungen des Ansatzes

Dieser Abschnitt fasst die Meinungen der Teilnehmer zum Ansatz insgesamt zusammen. So wurde unter anderem geäußert, dass der Ansatz die Erstellung von MBS zwar signifikant beschleunigen könnte, aber der Beitrag zur Reduktion des Gesamtaufwandes (inklusive Konzeption und UX Design) gering sei. Das strenge Architekturmuster hatte Fürsprecher, die die strikte Einhaltung sehr begrüßten, aber auch Gegner, welche eher generischere und flexiblere Konzepte wünschten. Generell wurde der Ansatz von allen anderen Teilnehmern als durchweg positiv bewertet.

B.2.7.1 Effizienzgewinn für Umsetzung möglich aber für Gesamtprozess gering

In Abschnitt B.2.6.2 wurde angesprochen, dass der Konzipierungsschritt im Rahmen der Erstellung einer neuen MBS-Variante sehr wichtig ist. Laut Teilnehmer 6 *fällt bei diesem Konzipierungsschritt der Hauptteil des Aufwandes*

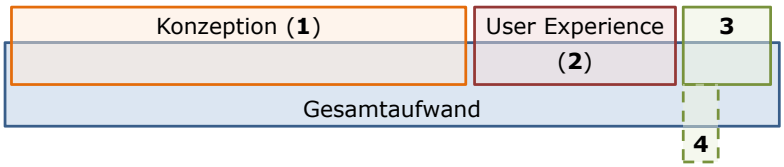


Abbildung B.1: Schaubild erstellt von einer Zeichnung des Teilnehmers 6 unter Erhaltung der relativen Größen der Einzelteile. Es zeigt das Verhältnis von Konzipierungsaufwand (Bereiche 1 und 2) zu Erstellungsaufwand (Bereich 3 bzw. mit Verwendung des Ansatzes Bereich 4) bei der Entwicklung von MBS.

an, und nur ein geringer Anteil wird für das tatsächliche Erstellen der Benutzerschnittstelle gebraucht (Z61-3). Zur Veranschaulichung der Verhältnisse der einzelnen Aufwände malte er Abbildung B.1. Bereich 1 stellt den Konzipierungsaufwand, Bereich 2 den Aufwand für den Entwurf der User Experience und Bereich 3 den Aufwand für die tatsächliche Erstellung der Benutzerschnittstelle dar. Bereich 4 illustriert (grob) den *Erstellungsaufwand durch den vorliegenden Ansatz, welcher nach Meinung von Teilnehmer 6 tatsächlich reduziert* wird. Aufgrund dieser Argumentation kam Teilnehmer 6 zu dem Schluss, dass der mögliche Effizienzgewinn durch den Ansatz gering und *der Ansatz daher "kein Game-Changer"* sei – also die Arbeitsweise bei der Erstellung von MBS nicht signifikant verändern könne (Z61-3).

Die anderen Teilnehmer diskutierten diesen Punkt nicht. Jedoch wurde der Konzipierungsschritt nur von Teilnehmer 4 und 6 erwähnt. Die anderen Teilnehmer gingen sofort zur Anpassung der Benutzerschnittstelle über. Es lässt sich somit resümieren, dass der Effizienzgewinn für den Gesamtprozess stark von der jeweiligen Arbeitsweise abhängt, der Umsetzungsaufwand aber signifikant reduziert werden kann.

B.2.7.2 Akzeptanz des Ansatzes

Trotz (oder gerade wegen) seiner insgesamt kritischen Haltung bestätigte Teilnehmer 6, dass der Ansatz erwartungsgemäß funktioniere (Z63-3). Alle anderen Teilnehmer sahen die Konzepte der vorliegenden Arbeit durchweg positiv. Teilnehmer 3 bezeichnete das Gesamtkonzept als gut (Z35-9) und lobte vor allem das (MVC artige) Architekturmuster (vorgestellt in Kapitel 8) und die strikte Einhaltung dessen (Z35-3, Z35-4). Teilnehmer 4 fand das Architekturmuster stimmig und hatte eher mit der Benutzbarkeit der Werkzeuge Probleme (Z43-2, Z43-8). Teilnehmer 5 bezeichnete die Konzepte immer wieder als "schlüssig" (Z51-4, Z52-7). Insgesamt kann somit resümiert werden, dass *die umgesetzten Konzepte durchaus auf Akzeptanz bei den Teilnehmern stoßen.*

VI

Verzeichnisse und Erklärungen

Literaturverzeichnis

Aitenbichler, Erwin (2006). „System Support for Ubiquitous Computing“. Diss. TU Darmstadt.

Atkinson, C. und T. Kühne (Sep. 2003). „Model-driven development: a metamodeling foundation“. In: *Software, IEEE* 20.5, S. 36 –41. ISSN: 0740-7459. DOI: 10.1109/MS.2003.1231149.

Atkinson, Colin und Thomas Kühne (2005). „Concepts for Comparing Modeling Tool Architectures“. In: *MoDELS*, S. 398–413.

Behring, Alexander und Andreas Petter (Mai 2009). *Mapache Dialogue Refinement Tools: a Preliminary User Study*. Techn. Ber. TR-10. ISSN 1864-0516. Darmstadt: Telecooperation Research Division, TU Darmstadt.

Behring, Alexander, Andreas Petter und Max Mühlhäuser (Sep. 2009). „Towards Integrating Usability and Software Engineering Using the Mapache Approach“. In: *Informatik 2009*. Hrsg. von Stefan Fischer, Erik Maehle und Rüdiger Reischuk (Hrsg.) Bd. P-154. Lecture Notes in Informatics. Lübeck, Germany: GI. ISBN: 978-3-88579-248-2.

— (2010). „A Domain Specific Language for Multi User Interface Development“. In: *Modellierung 2010*, S. 335–350.

Behring, Alexander u. a. (Dez. 2007). „Werkzeugunterstützte Modellierung multimodaler, adaptiver Benutzerschnittstellen“. In: *i-com - Zeitschrift für interaktive und kooperative Medien* 6.3, Peter Forbig AND Gerd Szwillus, S. 31–36. DOI: 10.1524/icom.2007.6.3.31.

Behring, Alexander u. a. (Apr. 2008). „Towards Multi-Level Dialogue Refinement for User Interfaces“. In: *CHI Workshop on User Interface Description Languages*.

Bergh, Jan Van den und Karin Coninx (2004). „Contextual ConcurTaskTrees: Integrating dynamic contexts in task based design“. In: *PERCOMW '04: Proceedings of the Second IEEE Annual Conference on Pervasive Computing and Communications Workshops*. Bd. 00. Los Alamitos, CA, USA: IEEE Computer Society, S. 13. ISBN: 0-7695-2106-1. DOI: 10.1109/PERCOMW.2004.1276897.

Bevan, Nigel (2009). „Criteria for Selecting Methods in User Centred Design“. In: *I-USED*. Hrsg. von Silvia Mara Abrahão u. a. Bd. 490. CEUR Workshop Proceedings. CEUR-WS.org.

Blumendorf, Marco (Juli 2009). „Multimodal Interaction in Smart Environments A Model-based Runtime System for Ubiquitous User Interfaces“. Diss. Berlin, Germany: Technische Universität Berlin.

Blumendorf, Marco, Sebastian Feuerstack und Sahin Albayrak (Okt. 2006). „Event-based Synchronization of Model-Based Multimodal User Interfaces“. In: *LNCS MoDELS'06 workshop proceedings: Workshop on Model Driven Development of Advanced User Interfaces (MDDAUI 2006)*.

Blumendorf, Marco, Grzegorz Lehmann und Sahin Albayrak (2010). „Bridging Models and Systems at Runtime to Build Adaptive User Interfaces“. In: *EICS '10: Proceedings of the 2nd ACM SIGCHI symposium on Engineering interactive computing systems*. ACM.

Blumendorf, Marco u. a. (2008). „Executable Models for Human-Computer Interaction“. In: S. 238–251. DOI: 10.1007/978-3-540-70569-7_22.

Borchers, Jan O. (2000a). „A pattern approach to interaction design“. In: *DIS '00: Proceedings of the 3rd conference on Designing interactive systems*. New York, NY, USA: ACM, S. 369–378. ISBN: 1-58113-219-0. DOI: 10.1145/347642.347795.

— (2000b). „A pattern approach to interaction design“. In: *DIS '00: Proceedings of the 3rd conference on Designing interactive systems*. New York, NY, USA: ACM, S. 369–378. ISBN: 1-58113-219-0. DOI: 10.1145/347642.347795.

Botterweck, Goetz (2006). „A Model-Driven Approach to the Engineering of Multiple User Interfaces“. In: *MoDELS Workshops*, S. 106–115.

Braune, Stephan u. a. (2011). „A Service-oriented Architecture for Emergency Management Systems“. In: *SE2011 Workshop zur IT-Unterstützung von Einsatz- und Rettungskräften*.

Breiner, Kai u. a. (2010). „PEICS: towards HCI patterns into engineering of interactive systems“. In: *PEICS '10: Proceedings of the 1st International Workshop on Pattern-Driven Engineering of Interactive Computing Systems*. New York, NY, USA: ACM, S. 1–3. ISBN: 978-1-4503-0246-3. DOI: 10.1145/1824749.1824750.

Browne, Thomas u. a. (1996). „The MASTERMIND User Interface Generation Project“. In:

Calvary, Gaëlle, Joëlle Coutaz und David Thevenin (2001). „A Unifying Reference Framework for the Development of Plastic User Interfaces“. In: *EHCI '01: Proceedings of the 8th IFIP International Conference on Engineering for Human-Computer Interaction*. London, UK: Springer-Verlag, S. 173–192. ISBN: 3-540-43044-X.

Calvary, Gaëlle und Anne-Marie Pinna (2008). „Lessons of Experience in Model-Driven Engineering of Interactive Systems: Grand challenges for MDE?“ First International Workshop on Challenges in Model-Driven Software Engineering (ChaMDE), MODELS'08, Toulouse, 28 Septembre 2008.

Calvary, Gaëlle u. a. (2002). „Plasticity of User Interfaces: A Revised Reference Framework“. In: *TAMODIA '02: Proceedings of the First Interna-*

tional Workshop on Task Models and Diagrams for User Interface Design. INFOREC Publishing House Bucharest, S. 127–134. ISBN: 973-8360-01-3.

Calvary, Gaëlle u. a. (2004). „Towards a New Generation of Widgets for Supporting Software Plasticity: The "Comet"“. In: *EHCI/DS-VIS*. Hrsg. von Rémi Bastide, Philippe A. Palanque und Jörg Roth. Bd. 3425. Lecture Notes in Computer Science. Springer, S. 306–324. ISBN: 3-540-26097-8.

Calvary, Gaëlle u. a. (Apr. 2008). *The Many Faces of Plastic User Interfaces*. April 5-10, 2008, Florence, Italy.

Calvary, G. u. a. (Juni 2003). „A Unifying Reference Framework for Multi-Target User Interfaces“. In: *Interacting with Computers* 15.3, S. 289–308.

Card, Stuart K., Allen Newell und Thomas P. Moran (2000). *The Psychology of Human-Computer Interaction*. Hillsdale, NJ, USA: L. Erlbaum Associates Inc. ISBN: 0898598591.

CARE Technologies u. a. (Aug. 2008). *Semantics of a Foundational Subset for Executable UML Models*. available online at http://www.omg.org/cgi-bin/doc?ad/08-08-03.

Carroll, John M. (2009). „Conceptualizing a possible discipline of human-computer interaction“. In: *Interacting with Computers* In Press, Corrected Proof, ISSN: 0953-5438. DOI: 10.1016/j.intcom.2009.11.008.

Clerckx, Tim, Kris Luyten und Karin Coninx (2004). „The mapping problem back and forth: customizing dynamic models while preserving consistency“. In: *Proceedings of the 3rd annual conference on Task models and diagrams*. TAMODIA '04. New York, NY, USA: ACM, S. 33–42. ISBN: 1-59593-000-0. DOI: 10.1145/1045446.1045455.

Collignon, Benoit, Jean Vanderdonckt und Gaëlle Calvary (2008). „An Intelligent Editor for Multi-Presentation User Interfaces“. In: *Proc. of 23rd Annual ACM Symposium on Applied Computing SAC 2008*. ACM Press, New York, Fortaleza, 16-20 March 2008, S. 1634–1641.

Collignon, Benoît, Jean Vanderdonckt und Gaëlle Calvary (2008). „Model-Driven Engineering of Multi-target Plastic User Interfaces“. In: *Proceedings of the Fourth International Conference on Autonomic and Autonomous Systems*. Washington, DC, USA: IEEE Computer Society, S. 7–14. ISBN: 978-0-7695-3093-2. DOI: 10.1109/ICAS.2008.37.

Conte, S. D., H. E. Dunsmore und V. Y. Shen (1986). *Software engineering metrics and models*. Redwood City, CA, USA: Benjamin-Cummings Publishing Co., Inc. ISBN: 0-8053-2162-4.

Coutaz, Joëlle (Sep. 1987). „PAC, an Object Oriented Model for Dialog Design“. In: *Interact 87*. Hrsg. von H.-J. Bullinger und B. Shackel. Stuttgart, Germany, S. 431 –436.

— (Okt. 2006). *Meta-User Interfaces for Ambient Spaces*. TAMODIA'06.

Coutaz, Joëlle und Gaëlle Calvary (2008). „HCI and Software Engineering: Designing for User Interface Plasticity“. In: *The Human-Computer Interaction Handbook: Fundamentals, Evolving Technologies, and Emerging Applicati-*

ons. Hrsg. von A. Sears und J. Jacko. 2nd. Human Factor and Ergonomics series. Taylor & Francis CRC Press. Kap. 56, S. 1107–1125.

Crowley, James L. u. a. (2002). „Perceptual Components for Context Aware Computing“. In: UbiComp '02, S. 117–134.

Dan R. Olsen, Jr.Ev (2007). „Evaluating user interface systems research“. In: *UIST '07: Proceedings of the 20th annual ACM symposium on User interface software and technology*. New York, NY, USA: ACM, S. 251–258. ISBN: 978-1-59593-679-2. DOI: 10.1145/1294211.1294256.

Demeure, Alexandre und Gaelle Calvary (Apr. 2008). „Requirements and models for next generation UI languages“. April 5-10, 2008, Florence, Italy.

Demeure, Alexandre u. a. (2006). „The Comets Inspector: Towards Run Time Plasticity Control Based on a Semantic Network“. In: *TAMODIA*, S. 324–338.

Demeure, Alexandre u. a. (2008). „Design by Example of Plastic User Interfaces“. In: *Proceedings of CADUI 2008, the 7th International Conference on Computer-Aided Design of User Interfaces*.

Dix, Alan (2009). „Human-computer interaction: A stable discipline, a nascent science, and the growth of the long tail“. In: *Interacting with Computers* In Press, Corrected Proof, ISSN: 0953-5438. DOI: 10.1016/j.intcom.2009.11.007.

Dix, Alan u. a. (2003). *Human-Computer Interaction (3rd Edition)*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc. ISBN: 0130461091.

Doan, AnHai u. a. (2004). „Ontology Matching: A Machine Learning Approach“. In: *Handbook on Ontologies*. Springer, S. 385–404.

Domene, Alexander u. a. (Dez. 2007). *EMODE Evaluation Report*.

Döweling, Sebastian u. a. (2009). „Soknos — An Interactive Visual Emergency Management Framework“. In: *GeoSpatial Visual Analytics*. Hrsg. von Raffaele De Amicis, Radovan Stojanovic und Giuseppe Conti. NATO Science for Peace and Security Series. 10.1007/978-90-481-2899-$0_2$0. Springer Netherlands, S. 251–262. ISBN: 978-90-481-2899-0.

Ellis, Geoffrey und Alan Dix (2006). „An explorative analysis of user evaluation studies in information visualisation“. In: *BELIV '06: Proceedings of the 2006 AVI workshop on BEyond time and errors*. New York, NY, USA: ACM, S. 1–7. ISBN: 1-59593-562-2. DOI: 10.1145/1168149.1168152.

Farenc, Christelle, Véronique Liberati und Marie-France Barthet (1996). „Automatic Ergonomic Evaluation: What are the Limits?“ In: *CADUI*, S. 159–170.

Feuerstack, Sebastian (Dez. 2008). „A Method for the User-centered and Model-based Development of Interactive Applications“. Diss. Berlin, Germany: Technische Universität Berlin.

Feuerstack, Sebastian, Marco Blumendorf und Sahin Albayrak (Okt. 2006). „Bridging the Gap between Model and Design of User Interfaces“. In: *Proccedings of Informatik 2006 Conference, Dresden, Germany*. Springer.

— (2007). „Prototyping of Multimodal Interactions for Smart Environments based on Task Models“. In: *European Conference on Ambient Intelligence: Workshop on Model Driven Software Engineering for Ambient Intelligence Applications*.

Feuerstack, Sebastian u. a. (2008). „Model-based Layout Generation“. In: *AVI '08: Proceedings of the working conference on Advanced visual interfaces*. Hrsg. von Paolo Bottoni und Stefano Levialdi. Proceedings of the working conference on Advanced visual interfaces 2008. New York, NY, USA: ACM, S. 217–224. ISBN: 0-978-60558-141-5. DOI: 10.1145/1385569.1385605.

Fischer, Stefan, Erik Maehle und Rüdiger Reischuk, Hrsg. (2009). *Informatik 2009: Im Focus das Leben, Beiträge der 39. Jahrestagung der Gesellschaft für Informatik e.V. (GI), 28.9.-2.10.2009, Lübeck, Proceedings*. Bd. 154. LNI. GI. ISBN: 978-3-88579-248-2.

Flick, Uwe (2009). *An Introduction to Qualitative Research*. 4th. London: SAGE Publications Ltd.

Florins, Murielle (2006). „Graceful Degradation: a Method for Designing Multiplatform Graphical User Interfaces“. Diss. Université catholique de Louvain.

Freudenstein, Patrick u. a. (2008). „A domain-specific language for the model-driven construction of advanced web-based dialogs“. In: *WWW*, S. 1069–1070.

Führung und Leitung im Einsatz (Dez. 1999). online. Köln: Ständige Konferenz für Katastrophenvorsorge und Katastrophenschutz.

Furtado, Elizabeth u. a. (2004). „KnowiXML: a knowledge-based system generating multiple abstract user interfaces in USIXML“. In: *TAMODIA '04: Proceedings of the 3rd annual conference on Task models and diagrams*. New York, NY, USA: ACM, S. 121–128. ISBN: 1-59593-000-0. DOI: 10.1145/1045446.1045469.

Gamma, E. u. a. (1995). *Design Patterns: Elements of Reusable Object-Oriented Software*. Hrsg. von Brian W. Kernighan. Addison-Wesley.

Glass, Robert L., V. Ramesh und Iris Vessey (2004). „An analysis of research in computing disciplines“. In: *Commun. ACM* 47.6, S. 89–94. ISSN: 0001-0782. DOI: 10.1145/990680.990686.

Glaubitt, Urs (Apr. 2009). „Wall-size, touch-screen Displays - Analyzing Usability Issues and designing Solutions“. Magisterarb. TU Darmstadt.

Göbel, Steffen u. a. (2006). „A Device-Independent Multimodal Mark-up Language.“ In: *GI Jahrestagung (2)*. Hrsg. von Christian Hochberger und Rüdiger Liskowsky. Bd. 94. LNI. GI, S. 170–177. ISBN: 978-3-88579-188-1.

Greenberg, Saul und Bill Buxton (2008). „Usability evaluation considered harmful (some of the time)“. In: *CHI '08: Proceeding of the twenty-sixth annual SIGCHI conference on Human factors in computing systems*. New York, NY, USA: ACM, S. 111–120. ISBN: 978-1-60558-011-1. DOI: 10.1145/1357054.1357074.

Hailpern, B. und P. Tarr (2006). „Model-driven development: the good, the bad, and the ugly“. In: *IBM Syst. J.* 45.3, S. 451–461. ISSN: 0018-8670.

Helms, James u. a. (Jan. 2008). *User Interface Markup Language (UIML) Version 4.0 Committee Draft*. Available at http://www.oasis-open.org/committees/download.php/28457/uiml-4.0-cd01.pdf. Last Access 19.02.2009.

Helms, James u. a. (2009). „Human-Centered Engineering of Interactive Systems with the User Interface Markup Language". In: Hrsg. von Ahmed Seffah, Jan Gulliksen und Michel C. Desmarais. 1sr. Bd. Human-Centered Software Engineering. Human-Computer Interaction Series. ISBN 978-1-84800-906-6. Springer, London. Kap. 7, S. 139–171. DOI: 10.1007/978-1-84800-907-3_7.

ISO 24752 (Okt. 2011). *Information technology – User interfaces – Universal remote console (6 parts)*. Geneva, Switzerland: ISO.

ISO 9241 (2012). *Ergonomics of human-system interaction (multiple parts)*. Geneva, Switzerland: ISO.

ISO 9241-110 (März 2006). *Ergonomics of human-system interaction – Part 110: Dialogue principles*. Geneva, Switzerland: ISO.

Janssen, C, A Weisbecker und J Ziegler (1993). „Generating user interfaces from data models and dialogue net specifications". In: *Language*, S. 418–423.

Juristo, Natalia und Ana M. Moreno (2001). *Basics of Software Engineering Experimentation*. Boston: Kluwer Academic Publishers.

Kavaldjian, Sevan u. a. (2007). „Transforming a Discourse Model to an Abstract User Interface Model". In: *MDDAUI*.

Knapp, Alexander u. a. (2007). „UWE - An Approach to Model-Driven Development of Web Applications". In: *i-com, Oldenbourg* 6.3. In German, S. 5–12.

Koch, Thomas, Axel Uhl und Dirk Weise (Jan. 2002). *Model Driven Architecture*.

Kosara, Robert u. a. (2003). „User Studies: Why, How, and When?" In: *IEEE Comput. Graph. Appl.* 23.4, S. 20–25. ISSN: 0272-1716. DOI: 10.1109/MCG.2003.1210860.

Laugesen, John und Yufei Yuan (2010). „What Factors Contributed to the Success of Apple's iPhone?" In: *Proceedings of the 2010 Ninth International Conference on Mobile Business / 2010 Ninth Global Mobility Roundtable*. ICMB-GMR '10. Washington, DC, USA: IEEE Computer Society, S. 91–99. ISBN: 978-0-7695-4084-9. DOI: http://dx.doi.org/10.1109/ICMB-GMR.2010.63.

Lazar, Jonathan, Jinjuan Feng und Harry Hochheiser (2010). *Research Methods in Human-Computer Interaction*. Indianapolis, IN: Wiley. ISBN: 978-0-470-72337-1.

Lehmann, Grzegorz, Marco Blumendorf und Sahin Albayrak (2010). „Development of context-adaptive applications on the basis of runtime user interface models". In: *EICS '10: Proceedings of the 2nd ACM SIGCHI symposium*

on Engineering interactive computing systems. New York, NY, USA: ACM, S. 309–314. ISBN: 978-1-4503-0083-4. DOI: 10.1145/1822018.1822068.

Liebermann, Henry (2003). *The Tyranny of Evaluation*. ACM CHI Fringe. web.media.mit.edu/ lieber/Misc/Tyranny-Evaluation.html, last access 26.01.2010.

Limbourg, Quentin (2004). „Multi-Path Development of User Interfaces“. Diss. Universite catholique de Louvain.

Limbourg, Quentin und Jean Vanderdonckt (2003). „Comparing Task Models for User Interface Design“. In: *The Handbook of Task Analysis for Human-Computer Interaction*. Hrsg. von D. Diaper und N. Stanton. Mahwah: Lawrence Erlbaum Associates. Kap. 6, S. 135–154.

— (2009). „Multipath Transformational Development of User Interfaces with Graph Transformations“. In: Hrsg. von Ahmed Seffah, Jan Gulliksen und Michel C. Desmarais. 1st. Bd. Human-Centered Software Engineering. Human-Computer Interaction Series. ISBN 978-1-84800-906-6. Springer London. Kap. 6, S. 107–138. DOI: 10.1007/978-1-84800-907-3_6.

Limbourg, Quentin u. a. (2004). „USIXML: A Language Supporting Multi-path Development of User Interfaces“. In: *EHCI/DS-VIS*, S. 200–220. DOI: 10.1007/11431879_12.

Limbourg, Q. u. a. (2004). „UsiXML: A User Interface Description Language for Context-Sensitive User Interfaces“. In: *Proceedings of the ACM AVI'2004 Workshop "Developing User Interfaces with XML: Advances on User Interface Description Languages"(Gallipoli, May 25, 2004)*. Hrsg. von K. Luyten u. a., S. 55–62.

Lin, James und James A. Landay (2002). „Damask: A tool for early-stage design and prototyping of multi-device user interfaces“. In: *In In Proceedings of The 8th International Conference on Distributed Multimedia Systems (2002 International Workshop on Visual Computing*. ACM Press, S. 573–580.

— (2008). „Employing patterns and layers for early-stage design and prototyping of cross-device user interfaces“. In: *CHI '08: Proceeding of the twenty-sixth annual SIGCHI conference on Human factors in computing systems*. New York, NY, USA: ACM, S. 1313–1322. ISBN: 978-1-60558-011-1. DOI: 10.1145/1357054.1357260.

Ludewig, Jochen (2003). „Models in Software Engineering“. In: *Software and System Modeling* 2.1, S. 5–14. DOI: 10.1007/s10270-003-0020-3.

Luyten, Kris u. a. (2003). „Derivation of a Dialog Model from a Task Model by Activity Chain Extraction“. In: *DSV-IS*, S. 203–217.

Luyten, Kris u. a. (2005). „Integrating UIML, Task and Dialogs with Layout Patterns for Multi-Device User Interface Design“. In: *The 11th International Conference on Human-Computer Interaction, Las Vegas*, S. 22–27.

Luyten, Kris u. a. (2008). „Meta-gui-builders: generating domain-specific interface builders for multi-device user interface creation“. In: *CHI '08: CHI '08 extended abstracts on Human factors in computing systems*. New York,

NY, USA: ACM, S. 3189–3194. ISBN: 978-1-60558-012-X. DOI: 10.1145/1358628.1358829.

Maier, Ingo, Tiark Rompf und Martin Odersky (2010). *Deprecating the Observer Pattern*. Techn. Ber.

Meixner, Gerrit (Feb. 2010). „Entwicklung einer modellbasierten Architektur für multimodale Benutzungsschnittstellen“. Diss. Technische Universität Kaiserslautern.

Meixner, Gerrit und Daniel Görlich (März 2008). „Aufgabenmodellierung als Kernelement eines nutzerzentrierten Entwicklungsprozesses für Bedienoberflächen“. In: *Workshop "Verhaltensmodellierung: Best Practices und neue Erkenntnisse", Fachtagung Modellierung. Berlin*. o.A.

— (Sep. 2009). „Eine modellbasierte Architektur für den Useware-Engineering Prozess“. In: *Informatik 2009*. Hrsg. von Stefan Fischer, Erik Maehle und Rüdiger Reischuk (Hrsg.) Bd. P-154. Lecture Notes in Informatics. ISBN 978-3-88579-248-2. Lübeck, Germany: GI. ISBN: 978-3-88579-248-2.

Mellor, Stephen J. und Marc Balcer (2002). *Executable UML: A Foundation for Model-Driven Architectures*. Foreword By-Ivar Jacoboson. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc. ISBN: 0201748045.

Meskens, Jan, Kris Luyten und Karin Coninx (2009). „Shortening user interface design iterations through realtime visualisation of design actions on the target device“. In: *Proceedings of the 2009 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. Washington, DC, USA: IEEE Computer Society, S. 132–135. ISBN: 978-1-4244-4876-0. DOI: 10.1109/VLHCC.2009.5295281.

— (2010). „Jelly: a multi-device design environment for managing consistency across devices“. In: *AVI*. Hrsg. von Giuseppe Santucci. ACM Press, S. 289–296. ISBN: 978-1-4503-0076-6.

Meskens, Jan u. a. (2008). „Gummy for multi-platform user interface designs: shape me, multiply me, fix me, use me“. In: *AVI '08: Proceedings of the working conference on Advanced visual interfaces*. New York, NY, USA: ACM, S. 233–240. ISBN: 0-978-60558-141-5. DOI: 10.1145/1385569.1385607.

Monk, Andrew u. a. (1993). *Improving your Human-Computer Interface*. BCS practitioner series. Hemel Hempstead: Prentice Hall.

Mori, Giulio, Fabio Paternò und Carmen Santoro (2004). „Design and Development of Multidevice User Interfaces through Multiple Logical Descriptions“. In: *IEEE Trans. Softw. Eng.* 30.8, S. 507–520. ISSN: 0098-5589. DOI: 10.1109/TSE.2004.40.

Mueller, Wolfgang, Robbie Schaefer und Steffen Bleul (2004). „Interactive Multimodal User Interfaces for Mobile Devices“. In: *Proceedings of the Proceedings of the 37th Annual Hawaii International Conference on System Sciences (HICSS'04) - Track 9 - Volume 9*. HICSS '04. Washington, DC, USA: IEEE Computer Society, S. 90286.1–. ISBN: 0-7695-2056-1.

Münchner Kreis e.V. u. a. (Nov. 2009). *Zukunft und Zukunftsfähigkeit der Informations- und Kommunikationstechnologien und Medien*. online at http://www.bmwi.de/.

Myers, Brad A. und Andrew J. Ko (Feb. 2009). „The Past, Present and Future of Programming in HCI“. In: *Human-Computer Interaction Consortium*. published online at http://www.cs.cmu.edu/ NatProg/papers/MyersKoHCIC09.pdf. Winter Park, CO.

Myers, Brad A., John F. Pane und Andy Ko (Sep. 2004). „Natural programming languages and environments“. In: *Commun. ACM* 47 (9), S. 47–52. ISSN: 0001-0782. DOI: 10.1145/1015864.1015888.

Myers, Brad A. und Mary Beth Rosson (1992). „Survey on user interface programming“. In: *Proceedings of the SIGCHI conference on Human factors in computing systems*. CHI '92. New York, NY, USA: ACM, S. 195–202. ISBN: 0-89791-513-5. DOI: 10.1145/142750.142789.

Myers, Brad A. u. a. (Nov. 1990). „Garnet: Comprehensive Support for Graphical, Highly Interactive User Interfaces“. In: *Computer* 23 (11), S. 71–85. ISSN: 0018-9162. DOI: 10.1109/2.60882.

Myers, Brad, Scott E. Hudson und Randy Pausch (März 2000). „Past, Present, and Future of User Interface Software Tools“. In: *ACM Transactions on Computer-Human Interaction (TOCHI)* 7.1, S. 3–28.

Myers, Brad u. a. (2008). „How designers design and program interactive behaviors“. In: *Proceedings of the 2008 IEEE Symposium on Visual Languages and Human-Centric Computing*. Washington, DC, USA: IEEE Computer Society, S. 177–184. ISBN: 978-1-4244-2528-0. DOI: 10.1109/VLHCC.2008.4639081.

Newman, Mark W. u. a. (2003). „DENIM: an informal web site design tool inspired by observations of practice“. In: *Hum.-Comput. Interact.* 18.3, S. 259–324. ISSN: 0737-0024.

Nichols, Jeffrey und Brad A. Myers (Nov. 2009). „Creating a lightweight user interface description language: An overview and analysis of the personal universal controller project“. In: *ACM Trans. Comput.-Hum. Interact.* 16 (4), 17:1–17:37. ISSN: 1073-0516. DOI: 10.1145/1614390.1614392.

Nichols, Jeffrey u. a. (2002). „Generating remote control interfaces for complex appliances“. In: *UIST '02: Proceedings of the 15th annual ACM symposium on User interface software and technology*. New York, NY, USA: ACM, S. 161–170. ISBN: 1-58113-488-6. DOI: 10.1145/571985.572008.

Nóbrega, Leonel, Nuno Jardim Nunes und Helder Coelho (2005). *Mapping ConcurTaskTrees into UML 2.0*.

Nunes, Nuno Jardim und João Falcão e Cunha (2000). „Wisdom - A UML Based Architecture for Interactive Systems“. In: *DSV-IS*. Bd. 1946. Limerick, Ireland, S. 191–205.

Nylander, Stina (2005). „Semi-automatic generation of device-adapted user interfaces“. In: *Proceedings of Pervasive 2005 (doctoral colloquium)*. Munich, Germany, S. 6.

Nylander, Stina, Markus Bylund und Annika Waern (Okt. 2003). *The Ubiquitous Interactor - Mobile Services with Multiple User Interfaces*. SICS Technical Report T2003:19. ISSN 1100-3154 ISRN:SICS-T–2003/19-SE. SICS publications database [http://eprints.sics.se/perl/oai2] (Sweden).

Object Management Group (2005). *Unified Modeling Language: Superstructure 2.0*. Available at: http://www.omg.org/docs/formal/05-07-04.pdf. Authoren: Adaptive Ltd., Alcatel, Borland Software Corporation, Computer Associates International, Inc., Telefonaktiebolaget LM Ericsson, Fujitsu, Hewlett-Packard Company, I-Logix Inc., International Business Machines Corporation, IONA Technologies, Kabira Technologies, Inc., MEGA International, Motorola, Inc., Object Management Group., Oracle Corporation, SOFTEAM, Telelogic AB, Unisys, X-Change Technologies Group, LLC.

— (Jan. 2006a). *Meta Object Facility (MOF) Core Specification*. OMG Available Specification formal/06-01-01. Version 2.0.

— (Mai 2006b). *Object Constraint Language Version 2.0*. 2.0.

— (Feb. 2007). *Unified Modeling Language: Infrastructure*. OMG Available Specification formal/07-02-06. version 2.1.1.

Park, Sun Young, Brad Myers und Andrew J. Ko (2008). „Designers' natural descriptions of interactive behaviors". In: *Proceedings of the 2008 IEEE Symposium on Visual Languages and Human-Centric Computing*. Washington, DC, USA: IEEE Computer Society, S. 185–188. ISBN: 978-1-4244-2528-0. DOI: 10.1109/VLHCC.2008.4639082.

Paternò, Fabio (2001). „Task Models in Interactive Software Systems". In: *Handbook of Software Engineering & Knowledge Engineering*. Hrsg. von S. K. Chang. World Scientific Publishing Co. ISBN: 981-02-4973-X.

Paternò, Fabio, Cristiano Mancini und Silvia Meniconi (1997). „ConcurTaskTrees: A Diagrammatic Notation for Specifying Task Models". In: *INTERACT '97: Proceedings of the IFIP TC13 International Conference on Human-Computer Interaction*. London, UK, UK: Chapman & Hall, Ltd., S. 362–369. ISBN: 0-412-80950-8.

Paternò, Fabio und Carmen Santoro (2002). „One Model, Many Interfaces". In: *CADUI*, S. 143–154.

— (2003). „A unified method for designing interactive systems adaptable to mobile and stationary platforms". In: *Interacting with Computers* 15.3, S. 349–366.

Paterno', Fabio, Carmen Santoro und Lucio Davide Spano (Nov. 2009). „MARIA: A universal, declarative, multiple abstraction-level language for service-oriented applications in ubiquitous environments". In: *ACM Trans. Comput.-Hum. Interact.* 16 (4), 19:1–19:30. ISSN: 1073-0516. DOI: 10.1145/1614390.1614394.

Paternò, Fabio, Carmen Santoro und Lucio Davide Spano (2009). „Model-Based Design of Multi-device Interactive Applications Based on Web Services". In: *Proceedings of the 12th IFIP TC 13 International Conference on Human-*

Computer Interaction: Part I. Berlin, Heidelberg: Springer-Verlag, S. 892–905. ISBN: 978-3-642-03654-5. DOI: 10.1007/978-3-642-03655-2_98.

Pederiva, Inés u. a. (2007). „The Beautification Process in Model-Driven Engineering of User Interfaces“. In: *INTERACT (1)*, S. 411–425. DOI: 10.1007/978-3-540-74796-3_39.

Petter, Andreas, Alexander Behring und Max Mühlhäuser (2009). „Solving Constraints in Model Transformations“. In: *Theory and Practice of Model Transformations*. Hrsg. von Richard Paige. Bd. 5563. Lecture Notes in Computer Science. 10.1007/978-3-642-02408-$5_1$0. Springer Berlin / Heidelberg, S. 132–147.

Petter, Andreas, Miroslav Zlatkov und Alexander Behring (Jan. 2010). *Solverational Grammar and a Set of Evaluation Results*. Techn. Ber. TR-12. ISSN 1864-0516. Darmstadt: Telecooperation Research Division, TU Darmstadt.

Petter, Andreas u. a. (Sep. 2008). „Modeling Usability in Model-Transformations“. In: *Proceedings of the 1st International Workshop on Non-functional System Properties in Domain Specific Modeling Languages, NFPinDSML-2008*. Hrsg. von Marko Bokovic u. a. Bd. 394. ISSN 1613-0073. CEUR.

Petter, Andreas u. a. (2009). „Optimizing Non-Functional Properties of a Service Composition Using a Declarative Model-to-Model Transformations“. In: *Acta Universitates Apulensis* 18, S. 15.

Pinheiro da Silva, Paulo (2000). „User Interface Declarative Models and Development Environments: A Survey“. In: *Lecture Notes in Computer Science* 1946, S. 207–226.

Pinheiro da Silva, Paulo und Norman W. Paton (Juli 2003). „User Interface Modeling in UMLi“. In: *IEEE Software* 20.4, S. 62–69. ISSN: 0740-7459.

Plaisant, Catherine u. a. (1996). „LifeLines: visualizing personal histories“. In: *CHI '96: Proceedings of the SIGCHI conference on Human factors in computing systems*. New York, NY, USA: ACM, S. 221–227. ISBN: 0-89791-777-4. DOI: 10.1145/238386.238493.

Puerta, Angel und Jacob Eisenstein (1999). „Towards a general computational framework for model-based interface development systems“. In: *IUI '99: Proceedings of the 4th international conference on Intelligent user interfaces*. New York, NY, USA: ACM Press, S. 171–178. ISBN: 1-58113-098-8. DOI: 10.1145/291080.291108.

— (2001). *XIML: A Universal Language for User Interfaces*. Available at http://www.ximl.org, last access 21.02.2009.

— (2002). „XIML: a common representation for interaction data“. In: *Proceedings of the 7th international conference on Intelligent user interfaces*. IUI '02. New York, NY, USA: ACM, S. 214–215. ISBN: 1-58113-459-2. DOI: 10.1145/502716.502763.

Puerta, Angel R., Michael Micheletti und Alan Mak (2005). „The UI pilot: a model-based tool to guide early interface design“. In: *IUI*. Hrsg. von Robert

St. Amant, John Riedl und Anthony Jameson. ACM, S. 215–222. ISBN: 1-58113-894-6.

Puerta, A.R. (Juli 1997). „A model-based interface development environment“. In: *Software, IEEE* 14.4, S. 40–47. ISSN: 0740-7459. DOI: 10.1109/52.595902.

Rathsack, Robert, Andreas Wolff und Peter Forbrig (2006). „Using HCI-Patterns with Model-based Generation of Advanced User-Interfaces“. In: *MDDAUI*.

Reenskaug, Trygve (Sep. 2003). *The Model-View-Controller (MVC) Its Past and Present*. Talk at Java Zone, Oslo 18–19 September 2003 and JAOO, Aarhus 22–25 September 2003.

Reuther, Achim (2003). *useML - Systematische Entwicklung von Maschinenbediensystemen mit XML*. Techn. Ber. Fortschritt-Berichte pak, Band 8. Kaiserslautern: Technische Universität Kaiserslautern.

Schaefer, Robbie, Steffen Bleul und Wolfgang Mueller (2006). „Dialog Modeling for Multiple Devices and Multiple Interaction Modalities“. In: *TAMODIA*, S. 39–53.

Schlungbaum, Egbert (1996). *Model-based User Interface Software Tools Current state of declarative models*. Techn. Ber. GRAPHICS, VISUALIZATION und USABILITY CENTRE, GEORGIA INSTITUTE OF TECHNOLOGY, GVU TECH REPORT.

Schneider, Kevin A. und James R. Cordy (2001). „Abstract User Interfaces: A Model and Notation to Support Plasticity in Interactive Systems“. In: *Proceedings of the 8th International Workshop on Interactive Systems: Design, Specification, and Verification-Revised Papers*. London, UK: Springer-Verlag, S. 28–48. ISBN: 3-540-42807-0.

Seffah, Ahmed, Peter Forbrig und Homa Javahery (2004). „Multi-devices "Multipleüser interfaces: development models and research opportunities“. In: *J. Syst. Softw.* 73.2, S. 287–300. ISSN: 0164-1212. DOI: 10.1016/j.jss.2003.09.017.

Seissler, Marc, Gerrit Meixner und Kai Breiner (Aug. 2010). „Using HCI Patterns within the Model-Based Development of Run-Time Adaptive User Interfaces“. In:

Selic, B. (Sep. 2003). „The pragmatics of model-driven development“. In: *Software, IEEE* 20.5, S. 19–25. ISSN: 0740-7459. DOI: 10.1109/MS.2003.1231146.

Shaer, Orit u. a. (2008). *User Interface Description Languages for Next Generation User Interfaces*. Techn. Ber.

Sharp, Helen, Yvonne Rogers und Jenny Preece (2007). *Interaction Design: Beyond Human Computer Interaction*. 2nd. New York, NY, USA: John Wiley & Sons, Inc. ISBN: 0470018666.

Sottet, Jean-Sébastien, Gaëlle Calvary und Jean-Marie Favre (Okt. 2006). „Models at Runtime for Sustaining User Interface Plasticity“. In: *Workshop Models@run.time in conjunction with MoDELS 2006*. Genova, Italy.

Sottet, Jean-Sébastien u. a. (2008). „A Model-Driven Engineering Approach for the Usability of Plastic User Interfaces“. In: S. 140–157. DOI: 10.1007/978-3-540-92698-6_9.

Souchon, Nathalie, Quentin Limbourg und Jean Vanderdonckt (2002). „Task Modelling in Multiple Contexts of Use“. In: *DSV-IS '02: Proceedings of the 9th International Workshop on Interactive Systems. Design, Specification, and Verification*. London, UK: Springer-Verlag, S. 59–73. ISBN: 3-540-00266-9.

Stachowiak, Herbert (1973). *Allgemeine Modelltheorie*. Wien [u.a.]: Springer. ISBN: 3-211-81106-0.

Stahl, Thomas u. a. (2007). *Modellgetriebene Softwareentwicklung: Techniken, Engineering, Management*. 2. Heidelberg: dpunkt. ISBN: 978-3-89864-448-8.

Stanciulescu, Adrian u. a. (2005). „A transformational approach for multimodal web user interfaces based on UsiXML“. In: *ICMI '05: Proceedings of the 7th international conference on Multimodal interfaces*. New York, NY, USA: ACM Press, S. 259–266. ISBN: 1-59593-028-0. DOI: 10.1145/1088463.1088508.

Sukaviriya, Noi u. a. (2007). „Model-Driven Approach for Managing Human Interface Design Life Cycle“. In: *MoDELS*. Hrsg. von Gregor Engels u. a. Bd. 4735. Lecture Notes in Computer Science. Springer, S. 226–240. ISBN: 978-3-540-75208-0.

Szekely, Pedro A. (1996). „Retrospective and Challenges for Model-Based Interface Development“. In: *DSV-IS*. Springer, S. 1–27. ISBN: 3-211-82900-8.

Szekely, Pedro A. u. a. (1995). „Declarative interface models for user interface construction tools: the MASTERMIND approach“. In: *EHCI*. Hrsg. von Leonard J. Bass und Claus Unger. Bd. 45. IFIP Conference Proceedings. London, UK, UK: Chapman & Hall, S. 120–150. ISBN: 0-412-72180-5.

Szekely, Pedro, Ping Luo und Robert Neches (1992). „Facilitating the exploration of interface design alternatives: the HUMANOID model of interface design“. In: *CHI '92: Proceedings of the SIGCHI conference on Human factors in computing systems*. New York, NY, USA: ACM Press, S. 507–515. ISBN: 0-89791-513-5. DOI: 10.1145/142750.142912.

Thevenin, David und Joëlle Coutaz (1999). „Plasticity of User Interfaces: Framework and Research Agenda“. In: *Proceedings of the IFIP Conference on Human-Computer Interaction (INTERACT99)*. Hrsg. von A.M. Sasse und C. Johnson. IOS Press, S. 110–117.

Thiel, Matthias, Andreas Petter und Alexander Behring (2007). „Usability Aware Model Driven Development of User Interfaces“. In: *Proceedings of the AmI-Workshop on Ambient Intelligence*. To appear.

Tidwell, Jenifer (Apr. 2007). *Designing Interfaces: Patterns for Effective Interaction Design*. O'Reilly.

Van den Bergh, Jan und Karin Coninx (Nov. 2004). „Model-based design of context-sensitive interactive applications: a discussion of notations“. In: *TAMODIA '04: Proceedings of the 3rd annual conference on Task models and*

diagrams. New York, NY, USA: ACM Press, S. 43–50. ISBN: 1-59593-000-0. DOI: 10.1145/1045446.1045456.

Vanderdonckt, Jean M. und François Bodart (1993). „Encapsulating knowledge for intelligent automatic interaction objects selection“. In: *CHI '93: Proceedings of the INTERACT '93 and CHI '93 conference on Human factors in computing systems*. New York, NY, USA: ACM, S. 424–429. ISBN: 0-89791-575-5. DOI: 10.1145/169059.169340.

Vanderdonckt, Jean u. a. (2008). „Multimodality for Plastic User Interfaces: Models, Methods, and Principles“. In: *Multimodal user interfaces: signals and communication technology*. Hrsg. von Dimitrios Tzovaras. D. Tzovaras (ed.), Lecture Notes in Electrical Engineering, Springer-Verlag, Berlin, 2007. Springer. Kap. 4, S. 61–84.

Veit Schwartze Sebastian Feuerstack, Sahin Albayrak (2009). „Behavior-sensitive User Interfaces for Smart Environments“. In: *HCII 2009 - User Modeling*.

Zadmajid, Maryam (Feb. 2010). „Klassifikation und Nutzung von Aufgaben“. Diplomarbeit. Darmstadt, Germany: TU Darmstadt.

Zaplata, Sonja u. a. (März 2009). „Abstract User Interfaces for Mobile Processes“. In: *16. Fachtagung Kommunikation in Verteilten Systemen (KiVS 2009)*. Hrsg. von Gurt Geihs Klaus David. Gesellschaft für Informatik. Springer, S. 129–140.

Zhai, Shumin (2003). *Evaluation is the worst form of HCI research except all those other forms that have been tried*. essay published at CHI Place. http://www.almaden.ibm.com/u/zhai/papers/EvaluationDemocracy.htm, last accessed 26.01.2010.

Ziegert, Thomas, Markus Lauff und Lutz Heuser (2004). „Device Independent Web Applications - The Author Once - Display Everywhere Approach“. In: *ICWE*, S. 244–255.

Zühlke, Detlef (2004). *Useware-Engineering für technische Systeme*. Berlin, Germany: Springer.

Definitionsverzeichnis

Dieser Anhang enthält eine Liste aller in der Arbeit vorkommenden Definitionen.

Abbildungsverzeichnis

Tabellenverzeichnis

Schlagwortverzeichnis

Wissenschaftlicher Werdegang des Verfassers[3]

10/1998 – 06/2004	Studium der Physik an der TU Darmstadt Abschluss: Diplom-Physiker Nebenfächer: Astrophysik und Datenbanken Freiwillige Prüfungsleistung: Projektmanagement Diplomarbeitsthema: Charakterisierung der Absorptionseigenschaften Azo-Farbstoff-dotierter Flüssigkeiten durch Pump-Probe-Experimente
11/2005 – 03/2010	Wissenschaftlicher Mitarbeiter im Fachbereich Informatik an der TU Darmstadt

[3] gemäß §20 Abs. 3 der Promotionsordnung der TU Darmstadt