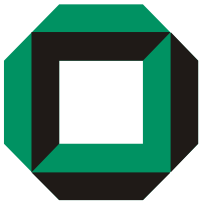
Universität Karlsruhe – Fakultät für Informatik – Bibliothek – Postfach 6980 – 76128 Karlsruhe

Modellgetriebene Software-Entwicklung - Architekturen, Muster und Eclipse-basierte MDA

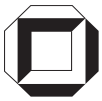
Autoren:
Steffen Becker, Aleksander Dikanski, Nils Drechsel,
Aboubakr Achraf El Ghazi, Jens Happe,
Ihssane El-Oudghiri,
Heiko Koziolek, Michael Kuperberg, Andreas
Rentschler, Ralf Reussner, Roman Sinawski,
Matthias Thoma, Marko Willsch

Interner Bericht 2006-18



Universität Karlsruhe
Fakultät für Informatik

ISSN 1432 - 7864



Universität Karlsruhe (TH)
Forschungsuniversität • gegründet 1825

Modellgetriebene Software-Entwicklung – Architekturen, Muster und Eclipse-basierte MDA

Seminar im Sommersemester 2006

Universität Karlsruhe (TH)
Fakultät für Informatik
Institut für Programmstrukturen und Datenorganisation
Lehrstuhl für Software-Entwurf und Qualität
Prof. Dr. Reussner

`http://sdq.ipd.uni-karlsruhe.de`

Steffen Becker, Aleksander Dikanski, Nils Drechsel, Aboubakr Achraf El Ghazi, Jens Happe, Ihssane El-Oudghiri, Heiko Koziolek, Michael Kuperberg, Andreas Rentschler, Ralf Reussner, Roman Sinawski, Matthias Thoma, Marko Willsch

Vorwort

Modellgetriebene Software-Entwicklung ist in den letzten Jahren insbesondere unter Schlagworten wie MDA und MDD zu einem Thema von allgemeinem Interesse für die Software-Branche geworden. Dabei ist ein Trend weg von der Code-zentrierten Software-Entwicklung hin zum (Architektur-) Modell im Mittelpunkt der Software-Entwicklung festzustellen. Modellgetriebene Software-Entwicklung verspricht eine stetige automatisierte Synchronisation von Software-Modellen verschiedenster Ebenen. Damit einher geht eine mögliche Verkürzung von Entwicklungszyklen und mehr Produktivität. Primär wird nicht mehr reiner Quellcode entwickelt, sondern Modelle und Transformationen übernehmen als eine höhere Abstraktionsebene die Rolle der Entwicklungssprache für Software-Produkte.

Derweil ist eine Evolution von Werkzeugen zur modellgetriebenen Entwicklung festzustellen, die einen zusätzlichen Gewinn an Produktivität und Effizienz ermöglichen sollen. Waren die Werkzeuge zur Jahrtausendwende in ihrer Mächtigkeit noch stark eingeschränkt, weil die Transformationssprachen nur eine begrenzte Ausdrucksstärke besaßen und die verfügbaren Werkzeuge eine nur geringe Integration von modellgetriebenen Entwicklungsprozessen boten, so ist heute mit den Eclipse-basierten Werkzeugen rund um EMF ein deutlicher Fortschritt spürbar. In der Eclipse-Plattform werden dabei als Plugins verschiedenste Aspekte der modellgetriebenen Entwicklung vereint:

- Modellierungswerkzeuge zur Erstellung von Software-Architekturen
- Frameworks für Software-Modelle
- Erstellung und Bearbeitung von Transformationen
- Durchführung von Transformationen
- Entwicklung von Quellcode

Der Seminartitel enthält eine Reihe von Schlagworten: „MDA, Architekturen, Muster, Eclipse“. Unter dem Dach von MDA ergeben sich zwischen diesen Schlagworten Zusammenhänge, die im Folgenden kurz skizziert werden.

Software-Architekturen stellen eine allgemeine Form von Modell für Software dar. Sie sind weder auf eine Beschreibungssprache noch auf eine bestimmte Domänen beschränkt. Im Zuge der Bemühungen modellgetriebener Entwicklung lassen sich hier Entwicklungen hin zu Standard-Beschreibungssprachen wie UML aber auch die Einführung von domänen-spezifischen Sprachen (DSL) erkennen. Auf diesen weiter formalisierten Beschreibungen von Software lassen sich schließlich Transformationen anwenden. Diese können entweder zu einem weiteren Modell („Model-to-Model“) oder einer textuellen Repräsentation („Model-to-Text“) erfolgen. In beiden Fällen spielen Muster eine wichtige Rolle. Transformationen kapseln in gewisser Weise wiederholt anwendbares Entwurfs-Wissen („Muster“) in parametrisierbaren Schablonen.

Eclipse stellt schließlich eine freie Plattform dar, die in letzter Zeit zunehmend Unterstützung für modellgetriebene Entwicklung bietet. In die Bemühungen zur Unterstützung modellgetriebener Entwicklung fällt auch das im Mai 2006 angekündigte „Eclipse Modeling Project“, das als „top level project“ auf die Evolution und Verbreitung modellgetriebener Entwicklungs-Technologien in Eclipse zielt.

Das Seminar wurde wie eine wissenschaftliche Konferenz organisiert: Die Einreichungen wurden in einem peer-to-peer-Verfahren begutachtet (vor der Begutachtung durch den Betreuer) und in verschiedenen „Sessions“ wurden die „Artikel“ an zwei „Konferenztagen“ präsentiert. Es gab „best paper awards“ und einen eingeladenen Gastredner, Herrn Achim Baier von der itemis AG & Co KG, der dankenswerter Weise einen aufschlussreichen Einblick in Projekte mit modellgetriebener Entwicklung in der Praxis gab. Die „best paper awards“ wurden an Herrn El-Ghazi und Herrn Rentschler verliehen, denen hiermit nochmal herzlich zu dieser herausragenden Leistung gedankt wird.

Gliederung

Die Seminarthemen des Seminars spiegeln das Feld der modellgetriebenen Entwicklung wider. Dieser technische Bericht gliedert sich dabei wie folgt.

Im ersten Abschnitt wird der Bereich der Architekturen und Muster beleuchtet. Hierunter fällt der Vergleich von Architekturevaluationsverfahren sowie eine Betrachtung von Entwurfsmustern für Enterprise Applications und nebenläufige Systeme.

Der zweite Abschnitt befasst sich mit Software-Komponentenmodellen in Form von „Fractal“ und der „Service Component Architecture“, die das Paradigma komponentenbasierter Software-Entwicklung mit Service Orientierten Architekturen (SOA) verknüpft.

Im dritten Abschnitt werden zwei Formen von Transformationen beleuchtet: zum einen die bereits oben angesprochene Modell zu Text Transformation, zum anderen die Transformation von Software-Architekturen in Warteschlangenmodelle.

Die Transformation in Warteschlangenmodelle leitet schließlich in den vierten Abschnitt zur Performanz-Modellierung über. Hier werden die Performance-Modellierung mit Queuing Networks sowie UML Profile für Qualitätsanforderungen behandelt.

Dank

Wir möchten uns an dieser Stelle bei allen Teilnehmern des Seminars für ihre engagierte Mitarbeit sehr herzlich bedanken. Ein mehrstufiger Begutachtungs-Prozess bestehend aus „peer-to-peer-Reviews“ sowie Gutachten durch die Betreuer ermöglichte die Auswahl qualitativ hochwertiger Paper. Insgesamt wurden acht Ausarbeitungen für diesen technischen Bericht angenommen. Auf der Homepage[1] zu diesem Seminar sind daneben auch die Vortragsfolien der Seminarteilnehmer zu finden, die auf der Abschlusskonferenz des Seminars vorgestellt wurden.

Ganz besonders möchten wir uns bei Herrn Achim Baier von der itemis AG & Co. KG für seine Keynote auf der Abschlusskonferenz bedanken.

Karlsruhe, 24. August 2006

Steffen Becker
Jens Happe
Heiko Koziolek
Klaus Krogmann
Michael Kuperberg
Ralf Reussner

[1] `http://i43pc12.ipd.uni-karlsruhe.de/wiki/Seminar_MDA_SS06`

Inhaltsverzeichnis

Teil I.

Architektur

Architekturevaluationsverfahren: ATAM, SAAM und ALMA

Matthias Thoma

Betreuer: Prof. Dr. Ralf H. Reussner

Zusammenfassung

Architekturevaluationsverfahren sollen Software-Architekten erlauben, in einer frühen Phase der Software-Entwicklung ihre Entwürfe auf Qualitätsmerkmale hin zu prüfen und Risiken aufzudecken. Diese Arbeit gibt einen Überblick über die szenariobasierten Architekturevaluationsverfahren „Software Architecture Analysis Method“ (SAAM), „Architecture Tradeoff Analysis Method“ (ATAM) und „Architecture-level modifiability analysis“ (ALMA). Über ein Klassifikationsschema werden die wichtigsten Unterschiede der jeweiligen Methoden herausgearbeitet. Zusätzlich wurden Ansätze für die Einbindung von Architekturevaluationsverfahren in den Software-Entwicklunsprozess entwickelt. Im besonderen wird eine mögliche Einbindung der Verfahren in den Extreme Programming (XP) Prozess betrachtet.

1 Einführung

1.1 Motivation

Software-Architekturevaluation ist ein noch recht junges Forschungsgebiet in der Software-Technik. Seit der Einführung von SAAM [11] im Jahre 1994 wurde Architekturevaluation als wichtiges Hilfsmittel zur Steigerung der Qualität von Architekturen erkannt, systematisch in der Industrie eingesetzt und von der Forschung in zunehmendem Maße untersucht. Vor allem in jüngerer Zeit werden neue Evaluierungstechniken vorgestellt die die Erfahrungen aus der Anwendung ihrer Vorgänger verarbeiten.

Es ist inzwischen allgemein bekannt, dass sich Software dauernd ändert und verändert. Spätestens seit den Veröffentlichungen von Lientz und Swanson im Jahre 1980 ist die Tatsache, dass die höchsten Kosten und die meiste Zeitbedarf nicht bei der initialen Entwicklung entstehen, sondern danach, jedem Software-Architekten und Entwickler bekannt. Software entwickelt sich, sie wächst und muss dauernd an neue Bedürfnisse angepasst werden.

Natürlich haben Architekten schon immer Wert auf Dinge wie Erweiterbarkeit, Modifizierkeit und Portabilität gelegt. Ausgangspunkt der Entwicklung von Evaluierungsverfahren war die Frage, wie man abseits von Beteuerungen des Architekten — der unter Umständen, da kein Experte der Domäne, die notwendigen Änderungen nicht vorhersehen kann — diese Attribute **strukturieren**, **formalisieren** und vielleicht sogar **messen** könne.

1.2 Software-Architekturen

Viele Software-Entwickler und Architekten haben ein mehr oder minder intuitives Verständnis für den Begriff „Software-Architektur“ und dessen Wesen. Betrachtet man all diese Meinungen jedoch genauer, so zeigen sich neben einigen Gemeinsamkeiten auch erhebliche Unterschiede. Um die Architekturevaluation auf ein solides Fundament zu stellen muß entsprechend erst einmal geklärt werden was eine Architektur ist und welche verschiedenen Aspekte dabei zu berücksichtigen sind.

Es gibt im Moment keine allgemeingültige Definition des Begriffes „Software-Architektur“. Am SEI (Software Engineering Institute an der Carnegie Mellon University) werden seit geraumer Zeit Definitionen des Begriffes Software-Architektur gesammelt. Es sind im Moment mehr als hundert.

Eine gängige Definition kommt von Bass, Clements, Kazman [2]:

> The software architecture of a program or computing system is the structure or structures of the system, which comprise software elements, the externally visible properties of those elements, and the relationships among them. (Bass, Clements, Kazman: Software Architecture in Practice)

Im ANSI/IEEE Std 1471-2000 Standard mit dem Titel „Recommended Practice for Architectural Description of Software-Intensive Systems“ wird Architektur wie folgt definiert:

> Architecture is defined by the recommended practice as the fundamental organization of a system, embodied in its components, their relationships

to each other and the environment, and the principles governing its design and evolution.

Neben der Frage was Architekturen sind stellt sich natürlich auch die Frage wie man sie am besten beschreibt. Im Laufe der Zeit haben sich verschiedene Notationen gebildet: UML, Petri-Netze, Automaten, diverse formale Beschreibungssprachen und viele andere mehr.

Die hier vorgestellten Evaluationsverfahren gehen in Bezug auf die Notation einen sehr pragmatischen Weg. Sie lassen die Frage offen und überlassen es den Evaluationsteam und dem Architekten Beschreibungssprache zu wählen, die auf die Teilnehmer angepasst ist und den Gebräuchen des Unternehmens entspricht.

1.3 Ziele

Die Evaluierungsverfahren sollen sicherstellen, dass die Architektur die geforderten Qualitätsanforderungen besitzt. Es handelt sich somit um eine Validierung der Architektur gegen die an sie gestellten funktionalen sowie nicht funktionalen Anforderungen. Je früher Fehler entdeckt werden, desto kostengünstiger können sie behoben werden. Falsche Entwurfsentscheidungen erst in der Implementierungsphase oder später zu beheben ist um ein vielfaches teurer als in der Designphase. Im schlimmsten Fall kann eine schlechte Architektur das komplette Projekt zum Scheitern bringen. Man stelle sich nur ein Projekt für ein Atomkraftwerk vor, dessen Software-Architektur nicht die geforderten Zuverlässigkeits- und Sicherheitsanforderungen erfüllt. Aber auch bei Projekten, bei dem Fehler weniger drastische Auswirkungen haben, lohnt es sich in der Regel Software-Evaluierung vorzunehmen. Die Kosten einer Evaluierung sind im Vergleich zu dem erwarteten Nutzen meist vernachlässigbar.

2 Qualitätsmerkmale und -eigenschaften

2.1 Einführung

Software-Qualität muß immer vom Standpunkt des Nutzers und des Einsatzgebietes der Software aus betrachtet werden. Qualität ist die Einhaltung von Anforderungen. Diese Anforderungen können sich jedoch von Nutzer zu Nutzer unterscheiden. Eine Tabellenkalkulation bei der Sinusfunktion und Kosinusfunktion eine geringe Genauigkeit haben, wird das Produkt für einen bestimmten Nutzerkreis wertlos machen. Die Software hat vom Standpunkt dieser Nutzer eine sehr niedrige Qualität. Für eine andere Nutzergruppe die stattdessen, die gut ausgebaute finanzmathematischen Funktionen schätzt, hat ein und dieselbe Software eine sehr hohe Qualität. Das Software-Qualität immer eine Validierung des Tatsächlichen gegen die Anforderungen ist liegt der folgenden Definition zugrunde.

Software-Qualität Die Gesamtheit der Merkmale eines Software-Produktes, die sich auf dessen Eignung beziehen, festgelegte oder vorausgesetzte Erfordernisse zu erfüllen. (DIN 66272 [7])

Wesentlich sind neben der Anforderung die beiden Begriffe **Qualitätsmerkmal** und **Qualitätseigenschaft**.

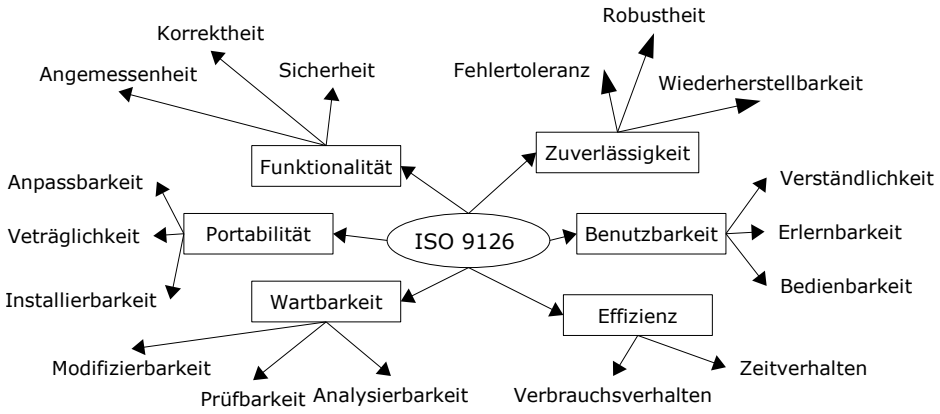


Abbildung 1: Qualitätsmodell ISO 9126

Qualitätsmerkmal Ein Qualitätsmerkmal ist eine Eigenschaft, die zur Unterscheidung von Produkten, Bausteinen oder Herstellungsprozessen in qualitativer (subjektiver) oder quantitativer (messbarer) Hinsicht herangezogen werden kann. ([7])

Qualitätseigenschaft Die konkrete Ausprägung eines Qualitätsmerkmales.

Der Begriff **Software-Qualität** selbst ist ohne weitere Erläuterungen und Verfeinerungen zu unspezifisch, deshalb werden Qualitätsmodelle verwendet. Ein solches Qualitätsmodell ist die DIN 66272 (ISO 9126). Sie definiert die sechs Qualitätsmerkmale **Funktionalität**, **Zuverlässigkeit**, **Effizienz**, **Benutzbarkeit**, **Änderbarkeit** und **Portabilität** welche wiederum in Untermerkmale aufgeteilt werden können (Abbildung 1).

2.2 Ausgewählte Qualitätsmerkmale

Im Folgenden werden, um ein besseres Gefühl für Software-Qualitätsmerkmale zu bekommen, einige gesondert betrachtet. Vor allem die Modifizierbarkeit wird bei den hier beschriebenen Evaluierungsverfahren eine besondere Rolle spielen.

2.2.1 Funktionalität

Funktionalität gibt an, in welchem Maß ein System die funktionalen Anforderungen der Nutzer erfüllt. Funktionalität beinhaltet die Korrektheit, Angemessenheit und Sicherheit eines Systems.

2.2.2 Zuverlässigkeit

Die Zuverlässigkeit einer Software beschreibt die Fähigkeit dieser unter den gegebenen Bedingungen fehlerfrei zu arbeiten. Es gibt mehrere Maße um Zuverlässigkeit im Betrieb zu messen. Gebräuchlich ist - unter anderem - die durchschnittliche Zeit bis zum Auftreten eines Fehlers (mean time to failure).

2.2.3 Effizienz

Eine Menge von Merkmalen, die sich auf das Verhältnis zwischen dem Leistungsniveau der Software und dem Umfang der eingesetzten Betriebsmittel unter festgelegten Bedingungen (DIN 66272 [7]) beziehen.

2.2.4 Modifizierbarkeit

Unter Modifizierbarkeit wird die Eignung einer Architektur, Änderungen der Anforderungen möglichst schnell und kostengünstig umzusetzen, zusammengefasst.

2.2.5 Portabilität

Portabilität ist die Anpassbarkeit einer Architektur an eine andere Umgebung (z. B. eine neue Plattform oder Architektur). Das Qualitätsmerkmal Portabilität im architektonischen Sinne misst, in welchem ausmaß, Annahmen über die zu Grunde liegende Umgebung von der Architektur gekapselt werden. In den übrigen Fällen kann sie als eine Untermenge der Modifizierbarkeit betrachtet werden.

DIN 66272 versteht unter Portabilität zusätzlich die Anpassung an eine neue organisatorische Umgebung. Dieser Aspekt wird im Rahmen einer Architekturevaluation im Allgemeinen nicht betrachtet.

3 Szenarien

Die hier beschriebenen Evaluationsmethoden haben mindestens eines gemeinsam: Sie basieren alle auf **Szenarien**. Die im vorangegangen Abschnitt erläuterten Qualitätsmerkmale sind zum großen Teil **nicht-funktional**. Wie lässt sich Zuverlässigkeit nur anhand einer Architektur prüfen? Wie Erweiterbarkeit? Es sind mehrere Strategien denkbar um diese nicht funktionalen Merkmale fassbar zu machen. Eine ist Szenarien zu verwenden.

Zu jedem zu untersuchenden Qualitätsmerkmal werden verschiedene Szenarien gebildet. Um beispielsweise die Modifizierbarkeit eines Systems zu testen, könnten mehrere wahrscheinliche Änderungen festgehalten und anhand der Architektur diskutiert werden. Im Fall eines bisher rudimentären eMail Clients könnte man sich z. B. fragen was passieren müsste um neben dem bereits vorhandenen POP (Post Office Protocol) Protokoll zusätzlich IMAP (Internet Message Access Protocol) zu unterstützen. Oder was getan werden müsste um einen weiteren Importfilter zu dem bereits vorhanden einzubauen.

Man kann die Szenarien grob in drei Kategorien einteilen, die verschiedene Sichten auf ein System wiederspiegeln: Die typische Nutzung, wahrscheinliche Änderungen, weitgehende Modifizierungen und Extremfälle.

Anwendungsfall (Use case) Ein Szenario das den gewöhnliche Gebrauch des Systems wiederspiegelt. Es sollte in der vorhandenen Architektur ohne Änderungen ausführbar sein.

Wachstumsszenario (growth scenario) Ein Wachstumsszenario ist eine erwartete und wahrscheinliche Modifizierung des Systems. Im Falle des einfaches eMail Cilent würde es sich um die Erweiterung um den IMAP Client handeln oder um eine mehrbenutzer Erweiterung.

Erkundungsszenario (exploratory scenario) Diese Szenarien beinhalten weitgehende Modifizierungen, Randfälle und Stresstests. Sie können dazu verwendet werden, die Grenzen der Architektur auszutesten. Für den einfachen eMail Client könnte ein solches Erkundungsszenario der Übergang vom Desktop PC auf einen Handheld sein.

Die Szenerien und die von UML bekannte Bezeichnung „use case“, verleiten dazu anzunehmen, dass Szenarien nur aus der Sicht eines Anwenders geschrieben werden. Dem ist nicht so. Es ist sogar eher die Ausnahme. Ein typisches Wachstumsszenerario wäre auch der Übergang vom 32 Bit zum 64 Bit Betriebssystem.

Szenarien und auf Szenarien basierte Evaluationsmethoden sind nicht für alle Qualitätsmerkmale gleichermassen geeignet. So lassen sich Zuverlässigkeit, Performance und Sicherheit eher schwer umsetzten. Diese Qualitätsmerkmale erfordern meist andere Methoden, auf die im Abschnitt Ausblick kurz eingegangen wird.

4 Interressensgruppen

4.1 Einführung

Die Evaluierung von Software-Architekturen ist keine rein technische Angelegenheit. Sie hängt stark von den beteiligten Personen und ihren Interressen und ihren jeweiligen Vertretern ab. Die Interressensgruppen sind in entscheidendem Maß für den Erfolg der Evaluation mitverantwortlich. Sie erstellen und wählen die Szenarien aus.

4.2 Stakeholder

Unter dem Begriff „Stakeholder“ (engl. Stake=Anteil) werden alle Personen zusammengefasst, die eine bestimmte Position oder eine bestimmte Interessensgruppe vertreten. Ein Stakeholder kann Mitglied in mehr als einer solchen Interresensgruppe sein. Typische Stakeholder sind Anwender, Projektverantwortliche und natürlich auch die Architekten und Entwickler des Systems.

5 Evaluationsteam

Das Evaluationsteam besteht aus mehreren Personen, die nicht direkt in das Projekt involviert sein sollten. Im günstigsten Fall hat man ein — externes oder aus einer anderen Abteilung stammendes — eingespieltes Evaluationsteam.

Innerhalb des Evaluationsteams nehmen verschiedene Personen eine oder mehrere Rollen ein. Eine Person kann mehr als eine dieser Rollen einnehmen und jede Rolle kann mit mehr als einer Person besetzt sein. Es haben sich folgende Rollen herausgebildet [5], die in dieser oder einer leicht abgewandelten Version in fast jedem Evaluationsteam zu finden sind.

Der **Teamleiter** iniziert die Evaluation. Er führt die Verhandlungen mit dem Kunden und stellt sicher, dass die Anforderungen des Kunden verstanden und erfüllt werden. Er ist für die Auswahl des Teams verantwortlich.

Im Gegensatz dazu führt der **Evaluationsleiter** den kompletten Evaluationsprozess. Zu seinen Aufgaben gehört es, dafür so sorgen, dass die einzelnen Schritte der Evaluation bestmöglich ausgeführt werden. Diese Rolle sollte deshalb von einem erfahrenen Teammitglied eingenommen werden, das sowohl Erfahrung mit Evaluationen als auch mit Architekturen hat.

Dem Evaluationsleiter stehen der **Process Enforcer** und der **Zeitnehmer** zur Seite. Der **Process Enforcer** behält die Übersicht über die einzelnen Prozessschritte und drängt nötigenfalls über den Evaluationsleiter auf ihre Einhaltung. Der **Zeitnehmer** unterstützt den Evaluationsleiter dabei, die Evaluation innerhalb des gesetzten Zeitrahmens zu halten. Es ist dem Zeitnehmer erlaubt eine Diskussion nötigenfalls zu beenden oder auf ihre rasche Beendigung zu drängen.

Es sollte zwei **Protokollanten** geben. Einer ist für die Szenarien zuständig. Er hält, während die Szenarien erarbeitet werden, die Ergebnisse fest. Er muss sicherstellen, dass ein Szenario vollständig und konsistent ist bevor zum Nächsten übergegangen wird. Der zweite Protokollant ist für das Protokollieren der einzelnen Besprechungen verantwortlich. Er hält alles Wesentliche der einzelnen Besprechungen fest. Bei der Erhebung der Szenarien protokolliert er die Rohfassung und die Motivation der Szenarien

und später deren jeweilige Lösung.

Der **Prozessbeobachter** ist ein stiller Zuhörer, der das Gelingen oder Mißlingen der einzelnen Prozesschritte beobachtet und protokolliert. Seine Aufgabe ist nach der Evaluation Verbesserungsvorschläge zu machen und so zu der kontinuierlichen Verbesserung des Evaluationsteams bei zu tragen.

Der **Fragesteller** ist in der Regel ein erfahrener Architekt, dessen Aufgabe es ist, Fragestellungen von Relevanz für die Architektur aufzuwerfen, die von den Stakeholdern nicht oder nur unzureichend betrachtet wurden.

6 Klassifizierung

In dieser Arbeit wird ein einfaches aber zweckmäßiges Klassifikationsschema verwendet, das im Hinblick auf die zu untersuchenden Verfahren — SAAM, ATAM und ALMA — entwickelt wurde und die Unterschiede und Gemeinsamkeiten der Verfahren herausarbeitet. In der Literatur werden weitere (teils umfangreichere) Klassifikationen vorgeschlagen. Kazman et. al. [10] nennen als wesentliche Kriterien die betrachtet werden sollten, den Kontext und Zielfindung (Context and goal identification), den Fokus und betrachtete Teile (Focus and Properties under Examination), die Unterstützung der Analyse (Analysis Support) und die Bestimmung der Analyseergebnisse (Determining analysis outcome). Neben dieser sehr grobgranularen Betrachtungsweise sind die weiteren Arbeiten auf diesem Gebiet weitaus feingranularer (u. a. [1] und [6]). Das nun folgende Klassifkationsschema geht einen Mittelweg.

6.1 Vorbedingungen

Es werden etwaige Vorbedingungen vorgestellt, die erfüllt sein müssen, um das Verfahren anzuwenden. Dabeí wird sowohl die theoretische Sicht, als auch die praktische — in Fallstudien beschriebener — Sicht betrachtet.

6.2 Ziele und Qualitätsmerkmale

Welche Ziele verfolgt eine Methode und für welche Qualitätsmerkmale ist sie geeignet?

6.3 Zielbestimmung

Das Evaluationsteam ist in der Regel mit dem Projekt und der Domäne nicht vertraut. Hier wird zusammengefasst welche Schritte ein Verfahren vorsieht um das Team mit den Geschäftszielen einer Architektur vertraut zu machen.

6.4 Analyse- und Erhebungstechniken

Verschiedene Evaluationsverfahren sind unterschiedlich formal in Bezug auf die verwendeten Analyse- und Erhebungstechniken. Brainstorming oder Checklisten sind gebräuchliche Erhebungstechniken. Innerhalb eines Verfahrens können natürlich je nach den Vorlieben des Evaluationsteam verschiedene Analyse- und Erhebungstechniken zum Einsatz kommen. Hier werden jedoch nur die Techniken entsprechend klassifiziert, die verbindlich von den Verfahren vorgeschrieben sind.

6.5 Einbeziehung der Interessensvertreter

Unterschiedliche Evaluationsverfahren beziehen die Interessensvertreter in unterschiedlichem Maße mit ein. Teilweise sind Gruppensitzungen erwünscht, in anderen Fällen werden sie nur vom Evaluationsteam befragt. Für den praktischen Einsatz der Methoden ist es von erheblicher Bedeutung in welcher Weise die Interessensvertreter einbezogen werden.

6.6 Zeitaufwand

Der Zeitaufwand ist für die Planung einer Evaluation bedeutend, deshalb wurde sie als zu betrachtendes Kriterium aufgenommen.

6.7 Ergebnis

Es wird das Ergebnis des kompletten Evaluationsprozesses betrachtet, insbesondere erstellte Dokumente und Empfehlungen.

7 Software Architecture Analysis Method

7.1 Einführung

Die Software Architecture Analysis Method (SAAM) ist eine der ältesten publizierten Evaluierungsmethoden. Sie wurde von Rick Kazman, Len Bass, Mike Webb und Gregory Abowd entwickelt und 1994 publiziert ([11]). Sie wurde seither vielfach in der Industrie und im akademischen Umfeld praktisch eingesetzt ([13], [2]). Wie alle hier beschriebenen Methoden basiert SAAM auf Szenarien. Die Qualitätsmerkmale werden auf die Szenarien abgebildet und mit deren Hilfe evaluiert.

7.2 Zielsetzung

SAAM wurde entwickelt, um die Modifizierbarkeit (Modifiability) von Software-Architekturen — basierend auf den gegenwärtigen und antizipierten Bedürfnissen der Stakeholder — zu untersuchen. Es hat sich gezeigt, dass SAAM auch für andere Qualitätsmerkmale wie Erweiterbarkeit, Portabilität, Wiederverwendbarkeit und Funktionalität eingesetzt werden kann. Zuweilen wird SAAM auch für Performance und Sicherheit eingesetzt, in diesem Bereich haben sich jedoch andere Verfahren (die nicht auf Szenarien basieren) als zweckmäßiger erwiesen.

7.3 Ablauf

SAAM läuft in sieben Schritten ab, die sequentiell durchlaufen werden sollen. Bei Bedarf kann ein Schritt auch mehrmals ausgeführt werden oder, wenn es erforderlich ist, auf einen vorangegangen Schritt zurück gesprungen werden. Der hier beschriebene Ablauf orientiert sich an [5].

Für den Evaluationsprozess sollten zwei bis drei Tage vorgesehen werden. Am ersten Tag werden die Szenerien erhoben und priorisiert. An den folgenden die Architektur gegen die Szenarien validiert und das Ergebnis präsentiert.

Voraussetzungen

Zur Evaluation wird eine Beschreibung der Architektur benötigt. In der Regel wird die bereits vorhandene Architekturbeschreibung verwendet. Sollte die Dokumentation unzureichend sein, wird sie im Laufe des Prozesses (vor allem im zweiten Schritt) erstellt. Dies führt zu einer verbesserten Dokumentation am Ende des Evaluationsverfahrens.

Schritt 1: Szenarien ermitteln

Im ersten Schritt werden Szenarien gesammelt. Nach Möglichkeit sollten in diesem Schritt alle Stakeholder ihre Sichtweisen einbringen können. Die Autoren von SAAM empfehlen die „Brainstorming“ Technik [4]. Brainstorming verläuft in zwei Phasen. In der ersten werden Ideen (in diesem Fall Szenarien) gesammelt ohne sie zu diskutieren oder anderweitig zu werten. In einer zweiten Phase — die in SAAM im dritten Schritt erfolgt — werden die Ideen sortiert und bewertet.

Die Szenarien dürfen und sollen nicht nur an gegenwärtigen Bedürfnissen ausgerichtet sein, sondern sollen auch zukünftige Erweiterungen und Änderungen mit einschließen. Das „Aussortieren“ unwahrscheinlicher Szenarien und die Auswahl für den Evaluationsprozess sinnvoller Szenarien folgt später.

Schritt 2: Architektur beschreiben

Im zweiten Schritt wird die Architektur oder die vorhandenen Architekturen mit möglichen Designalternativen allen Projektbeteiligten vorgestellt, nach Möglichkeit vom Architekten oder dem Architekturteam selbst. Die Vorstellung sollte für alle verständlich und die verwendete Notation (von formal über semiformal bis hin zu umgangsprachlich) den Teilnehmern angepasst sein.

Es hat sich gezeigt, dass die Beschreibung der Architektur oftmals Ideen für wichtige neue Szenarien aufwirft. Deshalb können die Schritte 1 und 2 mehrmals durchgeführt werden.

Schritt 3: Szenarien klassifizieren und Priorisieren

Die Szenarien werden in zwei Klassen unterteilt: **Direkte** Szenarien und **indirekte** Szenarien. In die erste Klassen fallen alle Szenarien, die ohne Änderungen der Architektur ausgeführt werden können. Alle funktionalen Anforderungen, die das System erfüllen muss, sollten in diese Klasse fallen.

In die zweite Klasse fallen entsprechend alle Szenarien, zu deren Ausführung Änderungen an der Architektur vorgenommen werden müssen.

Da aus dem ersten Schritt sehr viele Szenarien hervorgegangen sein können und diese auch unterschiedliche Dringlichkeiten und Wahrscheinlichkeiten haben werden, müssen den Szenarien Prioritäten zugeordnet werden. Was Priorität hat, entscheiden allein die Interressenvertreter. Die Autoren von SAAM schlagen vor, pro Interressenvertreter eine gewisse feste Anzahl von Stimmen zu vergeben, die diese auf die einzelnen Testfälle frei verteilen dürfen.

Schritt 4: Bewertung indirekter Szenarien

Die direkten Szenarien werden in diesem Schritt lediglich angewendet. Der Architekt zeigt anhand der Beschreibung der Architektur, wie diese das direkte Szenario umsetzt.

Für alle indirekten Szenarien hingegen wird ermittelt, welche Teile der Architektur geändert werden müssten um das Szenario zu unterstützen. Die erwarteten Änderungen werden festgehalten und der geschätzte Aufwand sollte ermittelt werden.

Schritt 5: Szenario Interaktionen ermitteln

Falls zwei oder mehrere der indirekten Szenarien Änderungen an der selben Komponente erfordern, so *interagieren* sie in SAAM Terminologie. Eine hoher Interaktionsgrad semantisch unterschiedlicher Szenarien ist ein Zeichen mangelnder Dekomposition (eine Komponente ist für mehrere unabhängige Aufgaben verantwortlich) oder einer zu grobgranularen Architekturbeschreibung (die Komponenten sollten dekomponiert werden).

Die Komponenten mit hohem Interaktionsgrad sind Erfahrungsgemäß die fehleranfälligsten. Deshalb sollte der Architekt diesen Komponenten im Weiteren besondere Aufmerksamkeit widmen.

Schritt 6: Allgemeine Bewertung

Im letzten Schritt werden die einzelnen Szenarien von den Stakeholdern gewichtet. Diese reflektieren die dahinter liegenden Geschäftsziele. Mögliche Gewichtungen sind unter anderem erwartete Kosten, Wichtigkeit für den Kunden usw. Mit Hilfe der gewichteten Szenerien können verschiedene Architekturen oder Entwurfsalternativen einer einzigen Architektur, gegeneinander abgewogen werden. Aufgrund dieser Daten sollen die Verantwortlichen in die Lage versetzt werden, die richtigen Architekturentscheidungen zu treffen.

7.4 Ergebnis

Das Ergebnis von SAAM ist eine verbesserte Architekturbeschreibung, Szenerien und die damit verbundenen durch Prioritäten gewichteten Qualitätsmerkmale.

7.5 Kritik

SAAM hängt naturgemäß stark von der Qualität der Szenarien und somit von den Stakeholdern ab. Der Verlauf ist relativ frei gehalten. Eine Erhebung dessen, was die Architektur leisten soll, wird nicht explizit vorgenommen, sondern eher implizit über die Abstimmungen.

7.6 Zusammenfassung

SAAM ist das älteste der hier beschriebenen Verfahren. Sie läuft in sechs einfachen Schritten ab: Szenerien werden erhoben, in direkte und indirekte Szenarien eingeteilt und priorisiert auf die Architektur angewendet. Die Ergebnisse werden durch das Evaluationsteam ausgewertet. Die **Zielbestimmung** erfolgt implizit über Erhebung der Szenarien durch die Interressensvertreter und deren spätere Priorisierung.

Die **Einbindung der Interessensverteter** ist durch den gesamten Prozess hindurch groß. Sie stellen die Szenarien zusammen und müssen über ihre Gewichtung entscheiden. Die Entscheidungsfindung ist im originalen SAAM eine Gruppenaktivität und die Entscheidung fällt letztlich durch Abstimmung.

Die einzige **Analyse- und Erhebungstechnik** ist die Generierung von Szenarien durch Brainstorming der Interessensverterter, damit hängt das komplette Verfahren sehr stark von der Erfahrung und dem Können des Evaluationsteams ab. Der **Zeitaufwand** ist mit zwei bis drei Tagen eher moderat.

8 Architecture Tradeoff Analysis Method

8.1 Einführung

Die Erfahrungen mit SAAM und ihre Schwächen und Unzulänglichkeiten haben zur Entwicklung der Architecture Tradeoff Analysis Method (ATAM) geführt. Sie wurde im Jahr 2000 von Kazman, Klein und Clements vorgestellt. Gegenüber der SAAM sollte der *Analyse-Teil* stärker betont werden. Die Methode hat sich seit ihrer ersten Veröffentlichung weiterentwickelt, weshalb gelegentlich von ATAM 1 (aus [9]) und ATAM 2 (aus [5]) gesprochen wird. Hier wird die neueste Version betrachtet.

8.2 Zielsetzung

ATAM analysiert Architekturentscheidungen in Bezug auf verschiedene Qualitätsmerkmale unter Berücksichtigung der wechselseitigen Auswirkungen einer solchen Entscheidung auf die verschiedenen Qualitätseigenschaften.

8.3 Ablauf

Der ATAM Prozess verläuft in vier Phasen. In der ersten Phase (den Schritten 1 bis 6) wird *präsentiert*. In der zweiten Phase *untersucht und analysiert*. In der dritte Phase (Schritte 7 und 8) werden Untersuchung und Analyse, mit Hilfe aller Stakeholder, auf Richtigkeit hin `getestet`. Die letzte Phase besteht aus der *Auswertung*. Die erste Phase wird mit einer kleineren Gruppe durchgeführt, die Phasen 2 bis 4 dagegen unter Beteiligung aller Stakeholder. Für die erste Phase wird ungefähr ein Tag benötigt, für die folgenden Phasen zwei Tage.

Die Phasen und einzelnen Schritte sind in ATAM nicht strikt festgelegt. Es kann nach Maßgabe des Evaluationsteam mehrfach über Schritte und Phasen iteriert werden, oder falls es sich als nötig erweist, zurück gesprungen werden.

Schritt 1: Vorstellung von ATAM

Das Evaluationsteam stellt allen Teilnehmern die die Methode vor. Beschreibt was erwartet werden kann und beantwortet Fragen.

Schritt 2: Vorstellung der Ziele (Business Drivers)

Die Geschäftsziele werden (idealerweise) vom Auftraggeber oder Projektmanager vorgestellt. Dabei werden die Geschäftsziele (Business Drivers) definiert, die die Architektur motiviert hat. Im folgenden wird die Evaluierung in Bezug auf diese Geschäftsziele durchgeführt.

Schritt 3: Vorstellung der Architektur
Der Architekt oder das Architekturteam stellt die Architektur vor und beschreibt, wie sie die verschiedenen Geschäftsziele adressiert.

Schritt 4: Vorstellung der Architekturentscheidungen
Der Architekt oder das Architekturteam stellt die getroffenen Architekturentscheidungen und Herangehensweisen vor; sie werden jedoch noch nicht analysiert oder bewertet.

Schritt 5: Quality Attribute Utility Tree
Der Qualitätsattributbaum (Quality Attribute Utility Tree) ist ein Analysehilfmittel. Er besteht aus der Wurzel, die mit „Utility“ beschriftet wird. Die Knoten auf der ersten Ebene sind die als wichtig identifizierten Qualitätsmerkmale (z. B. Performance, Modifizierbarkeit und Verfügbarkeit). Diese werden dann auf feingranulare Anforderungen heruntergebrochen. Die Blätter des Qualitätsattributbaums sind die jeweiligen Szenarien. Ihnen werden Prioritäten (Hoch, Mittel oder Niedrig) und durch den Architekten der notwendige Aufwand zugeordnet. Ein beispielhafter Qualitätsattributbaum ist in Abbildung 2 zu sehen.

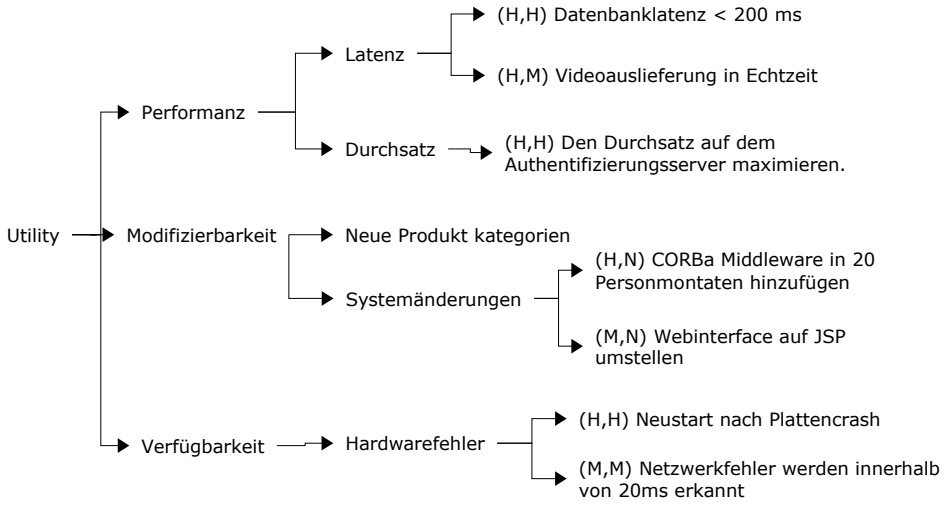


Abbildung 2: Qualitätsattributbaum

Schritt 6: Analyse der Architekturentscheidungen
Der Qualitätsattributbaum und die Architekturentscheidungen werden einander gegenübergestellt. Dabei werden ihre Risiken, Sensitivity- und Trade-off Points festgehalten. Zu jedem Szenario des Qualitätsattributbaums werden die entsprechenden Architekturentscheidungen gemäß ihrer zugeordneten Qualitätsmerkmale betrachtet.

Risiken Risiken sind Punkte, die je nach zukünftigem Verlauf, potentiell Probleme bereiten könnten.

Empfindliche Punkte Empfindliche Stellen (sensitivity points) sind Teile der Architektur bei der geringe Änderungen weitreichende Auswirkungen haben werden.

Kompromisse Kompromisse oder Trade-off Points geben an, inwiefern eine Designentscheidung zwei oder mehrere Qualitätsmerkmale wechselseitig beeinflusst.

Am Ende dieses Schrittes sollten zu jedem Szenario des Qualitätsattributbaums mit hoher Priorität die relevanten Architekturentscheidungen herausgearbeitet worden sein.

Schritt 7: Szenarien priorisieren

Der Qualitätsattributbaum wurde nur von einem Teil der Stakeholder erstellt. Nun werden äquivalent zu SAAM mittels der Brainstorming Technik von allen Stakeholder weitere Szenarien erhoben. Diese werden anschließend von den Stakeholder priorisiert. Jeder Stakeholder bekommt eine bestimmte Anzahl an Stimmen, die frei auf die Szenarien verteilt werden dürfen.

Schritt 8: Analyse der Architekturentscheidungen

Der zweite Analyseschritt wird genau so ausgeführt wie der erste (Schritt 6) – allerdings dieses mal mit den Szenarien, die aus dem vorangegangenen Schritt hervorgegangen sind.

Schritt 9: Präsentieren der Ergebnisse

Im neunten und letzten Schritt werden die Resultate vorgestellt. Die Ergebnisse der einzelnen Schritte (Szenarien, die Attribut basierten Fragen, der Qualitätsattributbaum, die Risiken, Nichtrisiken, sensitive Stellen und eingegangene Kompromisse) werden in einer Zusammenfassung vorgestellt. Zusätzlich werden die Risiken gruppiert. Alle Risiken, die sich mit der Portierung auf eine neue Plattform beschäftigen, könnten so in einer Gruppe zusammengefasst werden. Diese Risikogruppen werden dann den Geschäftszielen gegenübergestellt.

8.4 Ergebnis

Das Ergebnis des ATAM (Architecture Tradeoff Analysis Method) Prozesses sind Szenarien, dokumentierte Designentscheidungen (inklusive Risiken, empfindlichen Stellen und den eingegangenen Kompromisse) und einen Qualittsattributbaum (Utility Tree).

8.5 Kritik

ATAM hängt in hohem Maße vom Können des Evaluierungsteams ab. Wie auch SAAM hängt das Verfahren in hohem Maße von der Qualität der Szenarien und somit von den Stakeholdern ab.

8.6 Zusammenfassung

Die **Einbindung der Interessensverteter** ist in der ersten Phase mäßig, in der zweiten hingegen sehr groß. Sie stellen die Szenarien zusammen und müssen über ihre

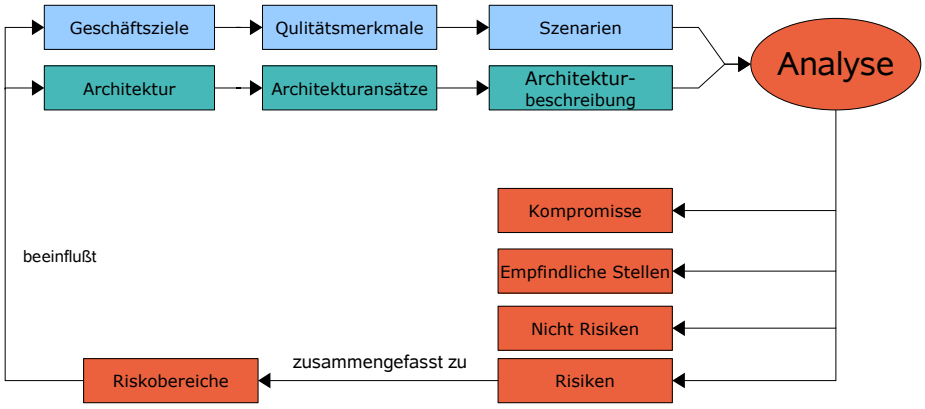


Abbildung 3: Der ATAM Prozess (nach [8])

Gewichtung entscheiden. Die Entscheidungsfindung findet auch in ATAM als Gruppenaktivität statt und die Entscheidung fällt letztlich durch Abstimmung. Jedoch helfen dabei Analysehilfmittel wie der Qualitätsattributbaum.

Die **Analyse- und Erhebungstechniken** sind das Festlegen der Geschäftsziele, darauf aufbauend die Generierung von Szenarien aus denen der Utility Tree generiert wird und abschließend das Brainstorming der Interessensverterter zum finden weiterer Szenarien.

Der **Zeitaufwand** ist mit durschnittlich drei bis vier Tagen etwas höher als der einer SAAM Analyse.

9 Architecture-level modifiability analysis

9.1 Einführung

Die Architecture-level modifiability analysis wurde von Nico Lassing, Per Olof Bengtsson, Hans van Vliet und Jan Bosch im Jahr 2004 vorgestellt [3]. Sie entand aus zwei Vorläufern von Lassing und van Vliet. Sie ist eine Erweiterung der SAAM Methode mit klarem Fokus auf Modifizierbarkeit.

9.2 Zielsetzung

ALMA setzt darauf, nur eine einziges Qualitätsmerkmal (Modifizierbarkeit) zu betrachten. Alle anderen Merkmale und die Abhängigkeiten der Merkmale untereinander werden nicht betrachtet. Weiterhin soll ALMA möglichst früh im Design-Prozess eingesetzt werden können und möglichst unkompliziert in der Ausführung sein.

9.3 Ablauf

Voraussetzungen

Für eine ALMA Evaluation wird eine Beschreibung der Architektur(en) benötigt. Je nach Ziel werden zusätzlich eine Methode zur Abschätzung von Wartungskosten benötigt.

Schritt 1: Ziele setzten
Zuerst werden die zu erreichenden Ziele festgelegt. ALMA unterstützt drei mögliche Zielsetzungen: Vorhersage der Wartungskosten, Risikoabschätzung und Entscheidungsfindung zwischen möglichen Design-Alternativen.

Schritt 2: Vorstellung der Architektur
Wie bei SAAM und ALMA wird in diesem Schritt vom Architekten oder dem Architekturteam die Software-Architektur vorgestellt. Es sind nur die Architekten und das Evaluationsteam anwesend.

Schritt 3: Szenarien erheben und wählen
In diesem Schritt werden die Szenarien erhoben. Anders als bei SAAM und ATAM ist dies kein dynamischer Gruppenprozess. Je nach Zielsetzung werden die zu interviewenden Stakeholder ermittelt. Das Evaluationsteam interviewt daraufhin die Stakeholder und ermittelt die wahrscheinlichen Änderungen (change cases), die auf das System zukommen könnten, sowie – falls gewünscht – die zu betrachtenden Risiken. Um die relativ große Anzahl an Szenarien handhabbar zu halten, werden diese in Äquivalenzklassen und Kategorien eingeteilt.

Die Kategorien können entweder vor dem Erheben der Szenarien festgelegt werden (top-down) oder sie werden nach und nach aus den Szenarien gewonnen (bottom-up). Nach den Erfahrungen der Autoren kann die Erhebung abgebrochen werden sobald neue Szenarien keine neuen Kategorien mehr bilden und sich zum größten Teil in die bereits vorhandenen Äquivalenzklassen einbinden lassen.

Je nach Zielsetzung werden nun die zu evaluierenden Szenarien gewählt. Falls die Vorhersage der Wartungskosten wichtig ist, werden die Szenarien gewählt von denen anzunehmen ist, dass sie innerhalb der Lebenszeit des Systems mit hoher Wahrscheinlichkeit auftreten werden. Entsprechend werden bei einer Risikoanalyse primär die Szenarien gewählt, die die identifizierten Risiken am besten wiederspiegeln. Sollen mehrere Architekturen oder Design-Alternativen miteinander verglichen werden, so sollen die Szenarien gewählt werden, die in den jeweiligen Alternativen besonders zur Geltung kommen.

Schritt 4: Szenarien evaluieren
Alle Szenarien werden einzeln in Hinblick auf die Systemkomponenten vom Evaluationsteam in Kooperation mit den Architekten evaluiert. Es werden die Systemkomponenten ermittelt, die bei einer Umsetzung der Änderung (change case) angepasst werden, hinzukommen oder wegfallen. Den einzelnen Szenarien werden quantitativ gemäß ihres Einflußes bewertet (Einwirkungsanlyse) — beispielsweise mit einer Skala von 1 bis 10. Zusätzlich ist es möglich, den geschätzten Aufwand zur Umsetzung des jeweiligen Szenarien anzugeben.

Schritt 5: Ergebnisse auswerten
Im letzten Schritt werden die Ergebnisse der Evaluation abhängig von den gesetzten Zielen ausgewertet und bewertet.

9.4 Ergebnis

Das Resultat des ALMA Verfahrens sind die Ergebnisse der Einwirkungsanlyse (Impact Analysis) und je nach Ziel Schätzungen für den Wartungsaufwand oder die Risken einer Architektur. Zusätzlich ergibt sich eine verbesserte Dokumentation der Architektur.

9.5 Kritik

Das Auswählen der Szenerien ist kritisch innerhalb der ALMA Methode. Die verwendete Interviewtechnik hat selbstverständlich den Vorteil, dass nicht alle Stakeholder zur gleichen Zeit anwesend sein müssen. Allerdings ist damit ein Verlust der Gruppendynamik verbunden. Es findet keine gegenseitige Befruchtung der Stakeholder statt. Auch das Verständnis der Architekten für die Nöte der restlichen Stakeholder ist durch die Trennung nicht so ausgeprägt wie bei SAAM und ATAM.

9.6 Zusammenfassung

ALMA ist eine relativ einfache Evaluationsmethode. Sie verläuft in fünf Schritten von der Zielsetzung, der Vorstellung der Architektur über das Szenarien erheben und der Evaluation bis hin zur Auswertung der Ergebnisse. Die **Zielbestimmung** erfolgt zu Beginn durch das Evaluationsteam in Zusammenarbeit mit dem Auftraggeber. Es kann eines oder mehrere der folgenden Ziele verfolgt werden: Vorhersage der Wartungskosten, Risikoabschätzung und Entscheidungsfindung zwischen möglichen Design-Alternativen. Der **Zeitaufwand** ist durch die Einfachheit der Methode moderat.

Die **Einbindung der Interessensverteter** ist geringer als bei SAAM/ATAM. Die angewandte **Analyse- und Erhebungstechnik** ist die Generierung von Szenarien durch Interviews der Interessensverterter. Die Analyse der Szenarien erfolgt mittels einer Einwirkungsanlyse (Impact Analysis). Dabei wird für jedes Szenario ermittelt, auf welche Systemkomponenten es einwirkt. Es kann hierzu eine formale Methode verwendet werden. ALMA selbst schreibt jedoch keine vor.

10 Einbindung in den Software-Entwicklungsprozess

Noch weitgehend unerforscht ist die Einbindung von Software-Evaluierungstechniken in vorhandene Software-Entwicklungsprozesse. Erst in jüngerer Zeit sind entsprechende Artikel erschienen. Ziel soll es sein, Architekturevaluation zu einem unabdingbaren Teil der Design-Phase(en) zu machen, so wie Codereviews und Inspektionen heute natürlicher Teil der Implementationsphase sind. Nachteilig wirkt sich aus, dass ein zusätzliches Team benötigt wird, während die Inspektionen von den Entwicklern selbst ausgeführt werden. Die Evaluationsverfahren können von der Erhebung der Anforderungen im Requirements Engineering profitieren.

Im Folgenden werden das traditionelle Modell und ein agiler Prozess betrachtet: Als Vertreter des traditionellen Ansatzes das Wasserfallmodell sowie die agile Methode Extreme Programming.

10.1 Wasserfallmodell

Das traditionelle Wasserfallmodell (mit zurückspringen) ist in Abbildung 4 dargestellt. Von der Analyse (Requirements Engineering) wird zum Entwurf übergegangen, von dort zur Implementierung. Im Folgenden wird in dieses Modell ein Evaluationsverfahren eingefügt.

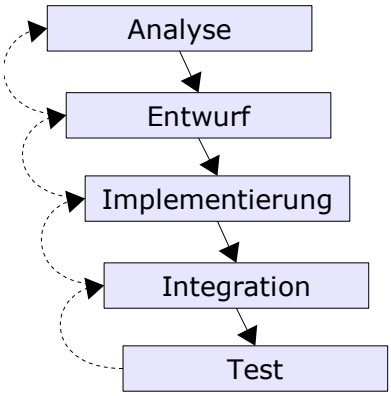


Abbildung 4: Wasserfallmodel

Bereits im Requirements Engineering (RE) können Informationen für die spätere Evaluation gewonnen werden. Die im RE Prozess gewonnenen Informationen über die zu erfüllenden Qualitätsmerkmale, können in die spätere Evaluation eingebracht werden. Weiterhin können bereits jetzt Szenarien entworfen werden. In RE-Prozessen, in denen die Use Cases zum natürlichen Erhebungsprozess gehören, können diese um die später zu betrachtenden Qualitätsmerkmale erweitert werden. Teil dieses neuen Prozesses muss es sein, die wichtigen Qualitätsmerkmale und Qualitätsanforderungen möglichst früh zu ermitteln. Ein Vorschlag dazu kommt vom SEI — eine Methode namens Quality Attribute Workshop (QAW).

Ziel des Quality Attribute Workshop (QAW) ist es, in einem frühen Stadium die treibenden Qualitätsmerkmale einer Architektur zu ermitteln. Der QAW benötigt lediglich eine grobe Idee der Architektur. Daraufhin werden mit Hilfe der Stakeholder Szenarien entwickelt und priorisiert. Aus diesen Szenarien werden dann die treibenden Qualitätsmerkmale bestimmt.

Fallen während des Design-Prozesses mehrere mögliche Design-Alternativen an, so kann mittels einer einfachen Methode – beispielsweise SAAM oder ALMA – eine Entscheidung getroffen werden.

Nach Fertigstellung des finalen Entwurfs fungiert die Evaluationsmethode als letzte Qualitätssicherungsmaßnahme, bevor die Architektur von den Entwicklern implementiert wird.

10.2 Extreme Programming

Extreme Programming ist unter den agilen Software-Entwicklungsprozessen eine der ältestesten, best dokumentiertesten und erforschten. Sie wurde von Kent Beck, Ward Cunningham und Ron Jeffries entwickelt.

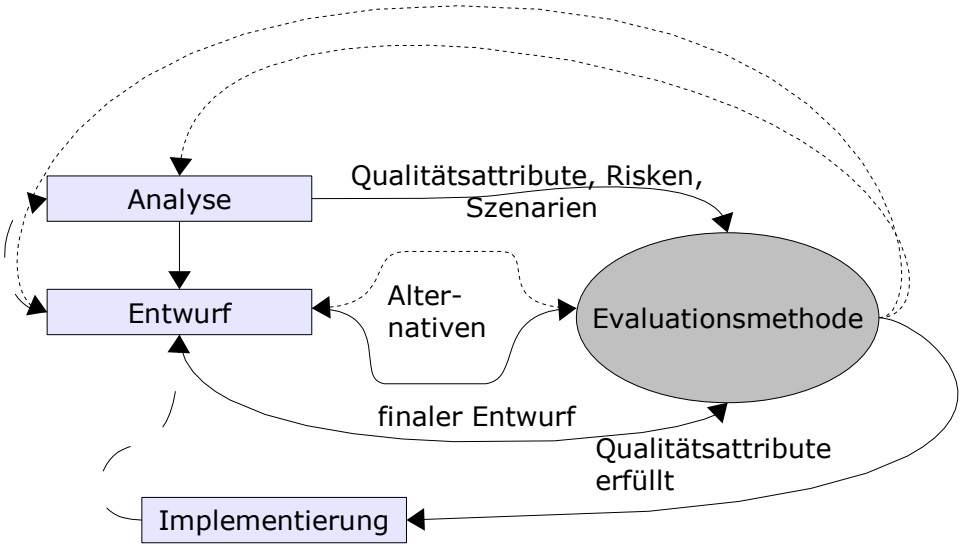


Abbildung 5: Wasserfallmodel mit Evaluationsverfahren

Der folgende Entwicklungsprozess baut architekturzentrierte Aspekte in den XP Prozess ein, ohne seine grundlegendes Wesen zu verändern. Der von Nord und Tomayko [12] vorgeschlagene Prozess geht in eine ähnliche Richtung, jedoch mit der Tendenz, einen iterativen architekturzentrierten Entwicklungsprozess mit XP Methoden zu entwerfen. Der hier vorgestellte Prozess geht in die andere Richtung. Einige Risiken von XP werden durch architekturzentrierte Methoden gemildert, ohne die Agilität von XP auch in Hinblick auf die Architektur in höherem Maße aufzugeben als notwendig.

Eines der hartnäckigsten Vorurteile gegenüber XP ist die Annahme, dass keine Architektur existiert. Natürlich existiert eine Architektur, jedoch nur für die jeweilige Iteration. Für kleinere bis mittlere Systeme können Whiteboards verwendet werden, so dass sich die Architektur mit dem Code und den Requirements ändern kann. Die Architektur ist also genauso „agil“ wie der Rest des XP Prozesses. Es gibt jedoch keine „overall architecture“ – also eine Architektur für das komplette System.

Im XP Prozess werden die Anforderungen der Kunden mit Hilfe von Szenarien – den Storycards – erhoben. Zu Beginn jeder Iteration bestimmt das Team in Zusammenarbeit mit dem Kunden, welche Szenarien in der jeweiligen Iteration umgesetzt werden sollen.

Es besteht hier die Gefahr, dass die Entwickler aufgrund der Storycards eine Architektur entwerfen, die Aufgrund nicht beachteter oder nicht bekannter Qualitätsanforderungen in späteren Iterationen komplett umgestellt werden müssen. Dies stellt innerhalb von XP einen „worst case“ dar. Dieses Risko kann jedoch durch den Quality Attribute Workshop gemindert werden. Je besser die XP Entwickler die treibenden Qualitätsmerkmale eines Systems verstehen, desto größer ist die Wahrscheinlichkeit, bereits in der ersten Iteration eine adäquate Architektur zu entwerfen.

Sobald die Architektur einer Iteration steht, kann Sie in eine SAAM oder ATAM Sitzung eingefügt werden. Diese systematischen Ansätze helfen sowohl Stakeholdern als auch den XP Entwicklern sicherzustellen, dass die Qualitätsanforderungen in der Architektur entsprechend gewürdigt werden.

Der Prozess lässt sich, wie folgt, zusammenfassen:

1. Als Teil der Storycard Erhebung wird ein Quality Attribute Workshop (QAW) durchgeführt. Es werden die Qualitätsmerkmale von den Stakeholdern, gemäß ihrer Wichtigkeit gewichtet (z. B. auf einer Skala von 1 bis 10).

2. Die Storycards werden auch im Hinblick auf die Qualitätsmerkmale erhoben und entsprechend erweitert.

3. Die erste Iteration wird durch eine SAAM, ATAM oder ALMA Session mit Fokus auf Modifizierbarkeit und den als wichtig identifizierten Qualitätsmerkmalen durchgeführt, um das Risiko einer völlig verfehlten ersten Iteration zu mindern.

4. In den folgenden Iterationen sollten SAAM Sessions (ähnlich den Code Reviews heute) zu einem integralen und regelmäßig durchgeführten Bestandteil des Entwicklungsprozesses werden.

11 Zusammenfassung

Software-Evaluierungsverfahren helfen frühzeitig sicherzustellen, dass alle Qualitätsanforderungen, die an eine Architektur gestellt werden auch erfüllt sind. Die meist nicht funktionalen Qualitätsmerkmale müssen in einer Art und Weise aufbereitet werden, die es erlaubt, sie mit Hilfe der jeweils vorhandenen Architekturbeschreibung zu evaluieren. Die hier vorgestellten Verfahren verwenden zu diesem Zweck Szenarien. Dabei werden die Qualitätsanforderungen auf zu erfüllende Szenarien abgebildet, die vom Evaluationsteam mithilfe der zur Verfügung stehenden Architekturbeschreibung untersucht werden.

SAAM ist eine der ältesten auf Szenarien basierenden Evaluationsverfahren. Sie basiert auf sieben einfachen Schritten mit dem Ziel die Design-Entscheidungen in Hinblick auf die Qualitätsmerkmale zu validieren. Das Hauptelement hierzu ist die Erhebung von Szenarien durch Brainstorming. Die Nachfolgemethode ATAM baut auf den Lehren von SAAM auf, formalisiert die Vorgehensweise jedoch ein wenig. Hinzugekommen sind Analysehilfen wie der Qualitätsattributbaum, der die wichtigsten Qualitätsmerkmale sowie ihre zugehörigen Szenarien zusammenfasst.

ALMA hingegen besteht aus einfachen und klaren Vorgehensweisen. Die Methode erlaubt es praxisnah eine Architektur auf Modifizierbarkeit zu untersuchen. Es unterstützt dabei Vorhersage der Wartungskosten, Risikoabschätzung und Entscheidungsfindung zwischen möglichen Design-Alternativen.

Alle Evaluationsmethoden bauen auf die Erfahrenheit des Evaluationsteams. Es gibt kaum Analysehilfsmittel, die es dem ungeübten erlauben würden, zielsicher eine Evaluation durchzuführen. Naturgemäß hängt das Ergebnis stark von den Szenarien und somit von den Stakeholdern ab.

Diese Evaluierungsverfahren lassen sich auf verschiedene Weise in vorhanden Software-Entwicklungsprozesse einfügen. Im traditionellen Wasserfallmodell dienen sie als Hilfsmittel, um zwischen Design-Alternativen zu wählen. Weiterhin können sie als Wächter fungieren, der sicherstellt, dass nur in die nächste Phase übergegangen wird, wenn die Qualitätseigenschaften den Anforderungen entsprechen. In den agilen Methoden ist es möglich, mit Hilfe der Architekturevaluation und anderen architekturzentrierten

Methoden (z. B. dem Quality Attribute Workshop), die entsprechenden Prozesse zu unterstützen und eine ihrer Schwächen zu beseitigen.

12 Ausblick

In den hier beschriebenen Verfahren spielten Kosten im betriebswirtschaftlichen Sinne eine eher untergeordnete Rolle. Zwar enthält z. B. SAAM eine Kostenschätzung, jedoch ist nicht weiter ausgeführt, wie und auf welcher Basis geschätzt wird. Eine einzelne isolierte Änderung zu betrachten und deren Kosten zu schätzen, wird nicht immer zum gewünschten Ergebnis führen. Weiterhin sollten die Kosten gegen die Nutzen abgewägt werden. Selbstverständlich kann man sich auf die Position zurückziehen, dass — zumindest in einer gesunden Organisation — bereits Methoden existieren, die einfach verwendet werden können. Eine dieser Methoden ist CBAM, die Cost Benefit Analsysis Method. Dort werden die Kosten einer Design-Entscheidung (Costs) dem zu erwarteten Nutzen (benefits) gegenüberstellt. CBAM eignet sich besonders als Ergänzung zu ATAM.

Neben den hier beschriebenen szenariobasierten Evaluationsmethoden gibt es noch weitere Möglichkeiten, die nicht auf Szenarien basieren. Vor allem im Bereich der Performanz sind Quantitative Methoden (Vorhersagen durch Messungen und Simulation) im Einsatz.

Die Einbindung in den Software-Entwicklungsprozess ist zur Zeit noch kaum dokumentiert. Es fehlt noch an Vorschlägen, Untersuchungen und Fallstudien über den effektiven Einsatz der Evaluierungsmethoden. Hier wurden in der jüngeren Vergangenheit erste Arbeiten veröffentlicht. Es ist zu erwarten, dass in näherer Zukunft einige Lücken auf diesem Gebiet geschlossen werden.

Abbildungsverzeichnis

Literatur

[1] M. A. Babar and I. Gorton, "Comparison of Scenario-Based Software Architecture Evaluation Methods," in *APSEC*, 2004, pp. 600–607.

[2] L. Bass, P. Clements, and R. Kazman, *Software Architecture in Practice, Second Edition*. Addison-Wesley Professional, 2003.

[3] P. Bengtsson, N. H. Lassing, J. Bosch, and H. van Vliet, "Architecture-level modifiability analysis (alma)," *Journal of Systems and Software*, vol. 69, no. 1-2, pp. 129–147, 2004.

[4] C. Clark, *Brainstorming: How to Create Successful Ideas*. Wilshire Book Company, 1989.

[5] P. Clements, R. Kazman, and M. Klein, *Evaluating Software Architectures: Methods and Case Studies*. Addison-Wesley Professional, 2002.

[6] L. Dobrica and E. Niemel, "A survey on software architecture analysis methods," *IEEE Trans. Softw. Eng.*, vol. 28, no. 7, pp. 638–653, 2002.

[7] D. I. für Normung, "Bewertung von Softwareprodukten - Qualitätsmerkmale und Leitfaden zu ihrer Verwendung," 1994.

[8] S. E. Institute, "www.sei.cmu.edu/architecture/atam_method.html," Stand: Juli, 2006.

[9] R. Kazman, M. Klein, and P. Clements, "ATAM: Method for Architecture Evaluation," 2000. [Online]. Available: citeseer.ifi.unizh.ch/kazman00atam.html

[10] R. Kazman, L. Bass, M. Klein, T. Lattanze, and L. Northrop, "A Basis for Analyzing Software Architecture Analysis Methods," *Software Quality Control*, vol. 13, no. 4, pp. 329–355, 2005.

[11] R. Kazman, L. Bass, M. Webb, and G. Abowd, "Saam: a method for analyzing the properties of software architectures," in *ICSE '94: Proceedings of the 16th international conference on Software engineering*. Los Alamitos, CA, USA: IEEE Computer Society Press, 1994, pp. 81–90.

[12] R. L. Nord and J. E. Tomayko, "Software architecture-centric methods and agile development," *IEEE Softw.*, vol. 23, no. 2, pp. 47–53, 2006.

[13] M. D. Simone and R. Kazman, "Software architectural analysis: an experience report," in *CASCON '95: Proceedings of the 1995 conference of the Centre for Advanced Studies on Collaborative research*. IBM Press, 1995, p. 18.

Muster für betriebliche Anwendungen - Objekt-Relationale Muster

Aleksander Dikanski

Betreuer: Prof. Dr. Ralf H. Reussner

Zusammenfassung

Betriebliche Anwendungen bilden, aufgrund ihrer umfangreichen Anwendungsgebiete und heterogenen Anforderungen, einen Schwerpunkt in der Softwareentwicklung. Zur Trennung der Belange Präsentation, Geschäftslogik und Datenhaltung wird in den meisten betrieblichen Anwendungen eine Schichtenarchitektur verwendet, um die Komplexität der Software zu beherrschen. Aufgrund dessen herrscht eine Dissonanz zwischen der Repräsentation der Daten als Objekte in der Geschäftslogikschicht und als Relationen in der Datenhaltungsschicht. Eine geeignete bijektive Abbildung von Objekten auf Relationen ist seit Langem eine Fragestellung bei der Entwicklung von betrieblichen Anwendungen und führte zur Entdeckung von verschiedenen best-practice Ansätzen, die als Muster dokumentiert wurden. In dieser Arbeit wird ein Großteil dieser Muster zur objektrelationalen Abbildung vorgestellt und die Kombinationsmöglichkeiten der Muster untersucht.

1 Einleitung

1.1 Kontext

Informationssysteme finden in der heutigen Zeit vielfältig Anwendung, sei es in Wirtschaft, Forschung oder Industrie. Die Bereiche stellen jeweils unterschiedliche Anforderungen an die zu verwendende Software. Diese grundsätzlichen Unterschiede spiegeln sich unter anderem in der Architektur eines Software-Systems wieder.

Eine spezielle Art von Software stellen betriebliche Anwendungen (*enterprise applications*) dar. Es gibt keine hinreichende Definition was genau eine betriebliche Anwendung ausmacht. Dafür ist die Zahl der Einsatzgebiete, z.B. Banken, Versicherungen und Handel, zu groß und die Anforderungen (z.B. Lagerverwaltung, Rechnungswesen und Bestellwesen) zu unterschiedlich. Trotzdem gibt es in allen Fällen ähnliche Fragestellungen, die es gilt zu beantworten.

Eine Grundarchitektur, die sich bei betrieblichen Anwendungen herausgebildet hat, ist die Schichtenarchitektur. Sie bietet viele Vorteile wie z.B. Trennung von Präsentations-, Anwendungs und Datenhaltungsbelangen und bietet eine lose Kopplung zwischen den Schichten, welche sich dadurch unabhängig entwickeln können.

1.2 Problemstellung

Eine der größten Herausforderungen für betriebliche Anwendungen basierend auf Schichtenarchitekturen ist der Austausch von Daten zwischen den Schichten. Ganz besonders herauszustellen ist hierbei der Übergang von der Datenhaltungsschicht zur Datenverarbeitungsschicht.

Die persistente Datenhaltung wird heutzutage meist von *relationalen Datenbankensystemen* übernommen. Diese speichern Daten in Relationen mit bestimmten Attributen ab. Beziehungen zwischen Daten werden über Schlüssel und Fremdschlüssel zwischen den Relationen hergestellt. Die Verwaltung von Daten und der schnelle Zugriff auf die Daten erfolgt mit relationalen Datenbanken effizient und effektiv, so dass sie schwer zu ersetzten sind.

Dagegen hat sich in der Datenverarbeitungsschicht die Objektorientierung durchgesetzt. Daten werden als Objekte verschiedener Klassen dargestellt, die unterschiedliche Eigenschaften und Methoden besitzen.

Auf dem ersten Blick sind die Unterschiede zwischen Relationen und Klassen nicht sehr groß. Allerdings treten beim bijektiven Abbilden von Objekten auf Relationen schwierige Fragestellungen auf: Wie behandelt man z.B. objektorientierte Merkmale wie etwa Vererbung und Assoziationen in Relationen?

1.3 Ziel dieser Arbeit

Im Laufe der Entwicklung von betrieblichen Anwendungen kristallisierten sich bestimmte häufig auftretende Fragestellungen heraus. Viele dieser Fragestellungen wurden bereits auf effiziente Weise gelöst und als best-practice Ansatz in verschiedenen Mustern dokumentiert.

In dieser Arbeit sollen Muster für die Entwicklung von betrieblichen Anwendungen aufgezeigt werden, die es ermöglichen, effektiv Objekten bijektiv auf Relationen abbilden zu können.

1.4 Aufbau dieser Arbeit

In Abschnitt 2 werden grundlegende Themen behandelt, die für das weitere Verständnis der Arbeit wichtig sind.

Abschnitt 3 präsentiert einen Katalog von objektrelationalen Mustern. Dabei werden Hinweise zum Gebrauch und Einsatz gegeben, Grenzen diskutiert und Verbindungen zu anderen Mustern aufgezeigt.

Viele der präsentierten Muster können einfach miteinander kombiniert werden. Andere dagegen müssen erst angepasst werden oder erlauben keine effiziente Zusammenarbeit. In Abschnitt 4 werden Kombinationsmöglichkeiten von Mustern dargestellt, die in unterschiedlichen Anwendungsfällen eine gute Zusammenarbeit erlauben.

Die objektrelationale Abbildung ist seit Langem eine Problematik in Rahmen der betrieblichen Anwendungen. Daher sind viele kommerzielle und Open-Source Werkzeuge entwickelt worden, um diese Abbildung zu vereinfachen. Ein Großteil dieser Werkzeuge verwenden die vorgestellten Muster. Einige davon werden in Abschnitt 5 besprochen und die Verwendung von Mustern aufgezeigt.

2 Grundlagen

2.1 Muster in Software-Architekturen

Muster werden seit Langem in der Software-Technik eingesetzt. Ursprünglich wurde der Begriff in der Architekturwelt geprägt. In [1] wird ein Muster als eine Beschreibung eines wiederkehrenden Problems und die Beschreibung der Kernlösung zu diesem Problem dargestellt. Diese Lösung kann dann, mit bestimmten Modifikationen, immer wieder auf gleiche oder ähnliche Probleme angewendet werden. Wichtig ist also, dass ein Muster immer zwei Bestandteile besitzt: das Problem und die Basislösung. Daher kann nicht vom Erfinden von Mustern geredet werden, sondern Muster werden als wiederkehrende Lösung zu einem Problem entdeckt. Gleichzeitig bieten Muster keine Standardlösung. Viele der Muster müssen je nach Problem angepasst und verändert werden, so dass sie ihre Funktion erfüllen.

Die grundlegende Arbeit über Muster in objektorientierter Software ist [5]. Die Autoren beschreiben grundlegende Muster, die auch heute noch weitläufige Anwendung finden. Sie unterteilten die Muster in drei Kategorien: Erzeugungsmuster, Strukturmuster und Verhaltensmuster.

Die Basismuster, die in [5] beschrieben wurden, zielen auf die Lösung von Problemen im relativ kleinen Kontext. Eine Einführung von architekturellen Mustern, d.h. dem eher grobgranularen Aufbau von Anwendungen, in Software gelang mit [3].

Neben den Basismustern wurden über die Zeit verschiedene weitere Muster dokumentiert, die für verschiedene Anwendungsgebiete bestimmt waren. So auch für betriebliche Anwendungen. Obwohl Muster relativ unabhängig dargestellt werden, treten bestimmte Muster in Kombination auf, wobei ein Muster häufig das Vorhandensein Weiterer bestimmt. Dies führt mit zu der Einführung von Mustersprachen. Dies sind best-practive Beschreibungen für die Kombination von Mustern in einem bestimmten Anwendungsbereich.

Muster erlauben Software-Architekten, sich gemeinsames Vokabular aufzubauen. Sie vereinfachen somit die Kommunikation zwischen den Entwicklern. Gleichzeitig wird die

Struktur und Architektur eines Software-Systems übersichtlicher, wenn Muster einsetzt werden. Die Verwendung von Mustern in Software dokumentiert somit den Stand der Technik bei der Software-Entwicklung. Mit dem nötigen Wissen über Muster ist es einfacher, gute Software-Architekturen zu entwickeln.

2.2 Betriebliche Anwendungen

Betriebliche Anwendungen sind eine spezielle Klasse von Software-Systemen. Sie zählen zu der größten Gruppe von Anwendungen und werden den in unterschiedlichsten Bereichen eingesetzt. Durch diese Heterogenität ihrer Anforderungen wächst die Herausforderung, gute Architekturen für betriebliche Anwendungen zu entwerfen. Es lassen sich grundsätzliche Gemeinsamkeiten dieser Klasse von Anwendungen herausstellen.

Zunächst arbeiten betriebliche Anwendungen fast immer auf *persistenten Daten*. Dies ist nötig, weil die Daten mehrfach, über Zeit und Raum verteilt, gebraucht werden. Desweiteren ändern sich Anwendungen, die diese Daten verarbeiten, oder neue kommen hinzu. Daher werden die Daten separat von den jeweiligen Anwendungen gehalten. Es ist häufig der Fall, dass diese Daten jahrelang gespeichert werden und länger existieren als die Anwendungen, die auf ihnen arbeiten.

Hinzu kommt, dass häufig die *Quantität* und die *Komplexität* der Daten unüberschaubar ist.

Meist ist nicht nur ein schnelles sondern auch ein *gleichzeitiges Zugreifen* auf die Daten nötig. Dabei ist die Anzahl der gleichzeitigen Zugriffe im Voraus meist nicht bekannt. Eine Anwendung, die nur intern in einer mittelgroßen Firma verwendet wird, muss eventuell mehrere hundert Zugriffe gleichzeitig verarbeiten können. Bei einem web-basiertem System dagegen, ist die Anzahl der Benutzer im Voraus unbekannt.

Daten haben meist auch eine *unterschiedliche Bedeutung* für die Benutzer und wollen unterschiedlich von ihnen verarbeitet werden. Dem entsprechend haben Daten oft *unterschiedliche Darstellungen* für unterschiedliche Zwecke.

Weiterhin sind betriebliche Anwendungen keine Recheninseln, sondern sind *integriert mit anderen betrieblichen Anwendungen*. Daher müssen Anwendungen miteinander kommunizieren, die mit unterschiedlichen Technologien zu unterschiedlichen Zeiten entwickelt wurden.

2.3 Grundlegende Muster in Betrieblichen Anwendungen

Eine grundlegende Architektur ist in fast allen betrieblichen Anwendungen vorhanden: die logische Aufteilung in Schichten. Die Anzahl der Schichten variiert je nach Nomenklatur, aber es lassen sich drei wesentliche Schichten festlegen: *Präsentation*, *Geschäftslogik* und *Datenhaltung*.

Die Datenhaltung, als unterste Schicht, verwaltet persistente Daten und ermöglicht den Zugriff auf diese. Die Geschäftslogik, als darauf aufbauende Schicht, kapselt die eigentliche Funktionalität der Anwendung. Hier werden die Daten aufbereitet, die dann in der darüberliegenden Präsentationsschicht dargestellt werden. Häufig kommt es in der Präsentation auch zu einer Interaktion mit dem Anwender, wodurch wiederum die Geschäftlsogik genutzt wird und Datenzugriffe erfolgen.

Der Vorteil der Schichtenarchitektur liegt in der Trennung von verschiedenen Belangen. Die Schichten sind unabhängig voneinander und greifen nur über bestimmte

Schnittstellen auf die Dienste der darunterliegenden Schichten zu. Ein Zugriff von unteren Schichten auf obere Schichten ist unerwünscht, da es zu zyklischen Abhängigkeiten führen kann. Theoretisch ist es in einer Schichtenarchitektur möglich, eine ganze Schicht auszutauschen, ohne dass das System darunter leidet. Dies allerdings nur unter der Voraussetzung, dass sich die Schnittstellen der Schicht nicht ändern.

2.4 Objekt-Relationale Abbildung in betrieblichen Anwendungen

Wie bereits erwähnt, liegt eine der Herausforderungen in Schichtenarchitekturen im Transport der Daten zwischen den Schichten. Allerdings wird heutzutage die Präsentationsschicht wie Geschäftslogik mit objektorientierter Technologie implementiert, so dass dieser Datentransfer keine konzeptionelle Brücke benötigt. Dagegen stellt die etablierte Technik der relationalen Datenbanken auf der Seite der Datenhaltung immer noch eine große Herausforderung bei der Abbildung von Objekten dar.

Die meisten der im folgenden Abschnitt 3 vorgestellten Muster beschäftigen sich mit der Abbildung von objektorientierten Strukturen auf relationale Strukturen und umgekehrt. So etwa bieten *Foreign Key Mapping* (3.4.2) und *Association Table Mapping* (3.4.3) Möglichkeiten, Assoziationen zwischen Objekten auf Fremdschlüsselbeziehungen zwischen Tabellen abzubilden. Ein besonderes Interesse liegt häufig in der Speicherung von Vererbungsstrukturen, die in Relationen nicht unterstützt werden. Hier hilft die Verwendung eines *Inheritance Mappers* (3.4.7) weiter.

Ein weiterer Schritt besteht darin, die Daten effizient aus der Datenquelle in Anwendung zu transportieren. *Gateways* (3.2.1) bietet hier eine Alternative zwischen dem Zugriff auf eine ganze Tabelle (*Table Data Gateway*) oder eine ausgewählte Zeile (*Row Data Gateway*). Komplexe Domänen verlangen meist mehr Funktionalität beim Datenzugriff. In solchen Situationen liefern *Active Records* (3.2.2) oder *Data Mapper* (3.2.3) eine mögliche Alternative.

Bei der Verarbeitung von persistenten Daten, muss immer wieder auf Konsistenz der Daten geachtet werden. Dazu wird eine *Unit of Work* (3.3.1) verwendet, die alle geladenen Daten verwaltet und veränderte Objekte kennzeichnet. Um die Identität zwischen den Objekten zu gewährleisten und Daten nicht mehrfach aus der Datenbank zu laden, kann z.B. eine *Identity Map* (3.3.2) eingesetzt werden.

Bei der Implementierung des Datenzugriffs fallen üblicherweise ähnliche Arbeiten an, die wiederholt durchzuführen sind. Um die Arbeit mit den Anfragen zu vereinfachen und um eine lose Kopplung zwischen der Anwendung und der Datenhaltung einzuführen, eignen sich Metadaten. Mit einem *Metadata Mapping* (3.5.1) lässt sich automatisch Code generieren, der die objektrelationale Abbildung vornimmt. Mit einem *Query Object* (3.5.2) oder ein *Repository* (3.5.3) lassen sich weiter Indirektionsstufen integrieren, welche die Arbeit mit Anfragen angenehmer machen und von der Datenquelle abstrahieren.

Die verwendeten Muster hängen oft von der Komplexität und Struktur des verwendeten Domänenmodells ab. Während ein reichhaltiges Domänenmodell (*Domain Model*, 3.1.2) häufig in komplexen Anwendungen zu finden ist, liefert ein *Transaction Script* (3.1.1) oft genug Funktionalität in kleinen bis mittleren Anwendungen. Ein *Table Module* (3.1.3) bietet einen oft geeigneten Kompromiss zwischen den beiden anderen Mustern.

3 Katalog für Objekt-Relationale Muster

3.1 Muster zum Strukturieren der Domäne

3.1.1 Transaction Script

Bündelt Geschäftslogik in Prozeduren, wobei jede Prozedur eine Anfrage bearbeitet [4, S. 110]

Aufbau und Funktionsweise Üblicherweise werden Anfragen an die Datenschicht in Transaktionen gebündelt. Das Transaction Script Muster greift diesen Gedanken auf und kapselt für die Anwendung übliche Transaktionen in Prozeduren.

Die Strukturierung ist relativ offen. Ein Transaction Script kann direkt in der Präsentationsschicht vorhanden sein, z.B. in einer Server Page. Allerdings ist es sinnvoll die Scripts separat zu kapseln. Eine mögliche Vorgehensweise ist, zueinandergehörige Transaktionen in einer Klasse zu bündeln.

Verwendung Die Stärke eines Transaction Scripts liegt in seiner Einfachheit. Bei einer Anwendung mit einem sehr einfachen Domänenmodell bietet das Transaction Script eine unkomplizierte und performante Lösung.

Bei zunehmender Komplexität des Domänenmodells kommt es allerdings häufig zu doppelten Codefragmenten. Hiergegen kann mit sorgfältigem Refactoring vorgegangen werden. Bei sehr komplexen Domänenmodellen ist es allerdings angebrachte, mächtigere Muster zu verwenden.

3.1.2 Domain Model

Ein Objektmodell der Domäne, dass Verhalten und Daten vereint [4, S. 116]

Aufbau und Funktionsweise Meist ist die Domäne einer betrieblichen Anwendung komplex und verlangt mächtige Strukturen zur Repräsentation in der Anwendung. Für die nötigen umfangreichen Datenverarbeitung und -manipulation ist ein objektorientiertes Geschäftsmodell geeignet. Durch Assoziationen, Vererbung und Polymorphie entsteht ein weitläufiges Netz verbundener Objekte, welche Artefakte der Domäne wiederspiegeln.

Jeder Klasse der Domäne kapselt neben den Attributen und Daten die zu ihr gehörige Funktionalität aus der Domäne. Dadurch wird der Aufwand der Implementierung der Geschäftslogik auf mehrere Stellen verteilt. Somit können verschiedene Versionen und Strategien für Teile der Geschäftslogik vorhanden sein (z.B. durch unterschiedliche Implementationen einer Schnittstelle). Dadurch kann die Geschäftslogik an verschiedenen Stellen weiterentwickelt und erweitert werden, ohne die gesamte Funktionalität zu beeinflussen.

Der Nachteil eines solchen komplexen Objektgraphen liegt in der Abbildung auf die Datenbank, denn objektorientierte Strukturen existieren nicht in relationalen Datenbanken. Desweiteren liefert die Anfrage an die Datenbank oft einen sehr großen Teilbaum des Objektgraphen zurück. Hierbei kann es zu Speicherengpässen kommen.

Verwendung Eine komplexe Geschäftslogik mit umfangreicher oder sich häufig ändernder Funktionalität ist am einfachsten mit einem Domain Model zu verwalten. Der menschliche Faktor spielt hier eine nicht unwesentliche Rolle. Sind die Entwickler nicht geübt im Umgang mit Objekten, kann es sehr schwierig sein, ein komplexes Domain Model zu entwickeln.

3.1.3 Table Module

Eine einzige Instanz behandelt die Geschäftslogik für alle Zeilen in einer Datenbanktabelle oder -sicht. [4, S. 125]

Aufbau und Funktionsweise Die Komplexität eines Domain Models (3.1.2) ist Vorteilhaft für die Funktionalität der Anwendung. Allerdings ist die Abbildung auf Relationen häufig recht schwierig. Ein Table Module bietet eine ähnliche Kapselung von Verhalten und Daten wie ein Domain Model, ohne die Identiät von Objekten zu beachten, d.h. eine Klasse ist für die Erzeugung und die Speicherung aller ihr Daten in Objekte zuständig. Dadurch ist ein Table Module ein Kompromiss zwischen Transaction Script (3.1.1) und Domain Model (3.1.2).

Ein Table Module wird häufig in Verwendung mit einer tabellenorientierten Datenstruktur genutzt. Das macht eine Abbildung auf Relationen simpel. Das Table Module bietet dazu ein einfaches Interface auf die Daten an.

Ein Table Module vertritt üblicherweise eine Tabelle in der Datenbank, aber auch komplexe Anfragen und Sichten können damit gekapselt werden. Oft müssen verschiedene Table Module auf gemeinsamen Daten arbeiten, damit eine bestimmte Funktionalität erreicht werden kann.

Verwendung Ein Table Module sollte dann eingesetzt werden, wenn Daten in tabellarischer Form aus der Datenbank extrahiert werden (z.B. in einem Record Set ([4, S. 508ff]). Dies ist häufig der Fall bei Datenbank-APIs wie ODBC oder JDBC.

Eine komplexe Verarbeitung der Daten ist in dieser Form sehr aufwendig. Die Vorteile des Table Module Musters liegen daher vor allem bei der Darstellung der Daten in der Präsentationsschicht. Wenn Daten in Form von Listen, Tabellen oder Formularen dargestellt werden sollen, muss keine unnötige Konvertierung zwischen dem tabellenorientierten Darstellungsformat der Präsentationskomponenten und dem Table Module vorgenommen werden.

Der Einsatz von Table Module setzt die Datenstruktur in den Mittelpunkt, was den Austausch mit der Datenhaltung vereinfacht. Allerdings sind damit die Möglichkeiten, komplexe Geschäftslogiken zu implementieren eingeschränkt. In solchen Fällen eignet sich ein Domain Model (3.1.2).

3.2 Architekturmuster für den Zugriff auf die Datenschicht

3.2.1 Gateways - Table und Row Data Gateway

Ein Objekt, dass den Zugriff auf externe Systeme oder Ressourcen kapselt [4, S. 466]

Der Zugriff auf relationale Datenbanken erfolgt durch den weit verbreiteten SQL Standard. Diese Sprache unterscheidet sich allerdings grundsätzlich von der objektorientierten Syntax. Somit ist es häufig unerwünscht, SQL und Code in der Anwendung zu mischen.

Die einfache Lösung hierzu ist, den SQL Code hinter einer einfachen API zu kaspeln. Verallgemeinert man dies auf alle externen Systeme und Ressourcen, die ein Mischen von unterschiedlichem Code mit sich bringen, führt dies zu dem Ansatz eines Gateways.

Bei der Integration mit Datenbanken lassen sich dabei zwei grundsätzliche Arten von Gateways klassifizieren. Zum Einen können Gateways zu ganzen Tabellen in der Datenbank existieren. Solche *Table Data Gateways* ([4, S. 144ff]) behandeln den Zugriff auf alle Zeilen der Tabelle. Zum Anderem gibt es den Ansatz der *Row Data Gateways* ([4, S. 152ff]). Hier existiert ein Gateway zu einem Eintrag in der Tabelle, d.h. es existiert ein Objekt pro Zeile.

Aufbau und Funktionsweise Ein Table Data Gateway bietet einen einfachen Zugriff auf Daten in einer Datenbanktabelle. Üblicherweise werden sie durch mehrere Methoden zur Suche und zum Einfügen, Aktualisieren und Löschen, hinter denen sich SQL Aufrufe kaspeln lassen, implementiert.

Der Rückgabewert der Methoden eines Table Data Gateways hängt zum Teil von der Domäne ab. Eine SQL Anfrage liefert häufig ein Record Set ([4, S. 508]) zurück. Solche eine datenbanknahe Struktur im objektorientierten Code, erzeugt meist eine Abhängigkeit zu der Datenbank. Andererseits ist in manchen Plattformen, wie etwa .NET (siehe 5.1), eine Unterstützung für Record Sets integriert.

Ist die Domäne durch ein reiches Domain Model repräsentiert, können Table Data Gateways auch entsprechende Objekte zurückliefern. Dadurch kann aber eine Abhängigkeit zwischen dem Objekt und dem Table Data Gateway entstehen.

Ein Row Data Gateway bietet einen Zugriff auf einen Eintrag einer Datenbanktabelle, gekapselt in ein Objekt pro Eintrag. Attribute in der Tabelle werden auf Attribute des Objektes abgebildet und eventuell wird eine Konvertierung von Datenbanktypen auf Anwendungstypen vorgenommen. Zusätzlich bieten diese Objekte Methoden zum Einfügen, Aktualisieren und Löschen an.

Häufig findet man anstelle von statischen Suchmethoden, spezielle Finderobjekte, die die Row Data Gateway Objekte in verschiedenen Datenquellen finden und erzeugen können.

Verwendung Table Data Gateway ist ein sehr einfach einzusetzendes Muster, wenn es darum geht, einen einfachen Zugriff auf eine Datenquelle zu erlangen. Es bietet eine gute Möglichkeit Zugriffslogik zu kapseln und ist dennoch einfach auf darunter liegenden Tabellen abzubilden. Ein Kriterium zur Anwendung von Table Data Gateway ist die Unterstützung des Musters in der verwendeten Plattform. Ist eine tabellarische Repräsentation von Daten, z.B. durch Record Sets, vorgesehen so eignet sich die Anwendung von Table Data Gateway in Verbindung mit einem Table Module (3.1.3).

Ein Row Data Gateway, wie auch ein Table Data Gateway, eignet sich gut in Verbindung mit einem Transaction Script (3.1.1). Dadurch kann sich häufig wiederholender Zugriffscode wiederverwendet werden und die typischen Codeduplikate in Transactions Scripts reduziert werden.

Gateways sollten dagegen beim Vorhandensein eines Domain Models (3.1.2) selten eingesetzt werden, denn hier sind Zugriffe durch Active Record (3.2.2) oder Data Mapper (3.2.3) geeigneter, die Komplexität des Domain Models auf die unterliegende Datenquelle abzubilden.

3.2.2 Active Record

Ein Objekt, dass ein Eintrag in einer Datenbanktabelle oder Sicht darstellt, den Datenbankzugriffscode kapselt und zusätzliche Funktionalität auf den Daten anbietet.
[4, S. 160]

Aufbau und Funktionsweise Ein Active Record ähnelt in gewisser Weise einem Row Data Gateway (3.2.1). Es kapselt den Datenbankzugriff und die Daten in einem Objekt. Der Unterschied besteht in der zusätzlichen Geschäftslogik, die ein Active Record anbietet, um die Daten zu manipulieren. Ähnlich wie beim Row Data Gateway lassen sich auch statische Anfragemethoden aus dem Objekt in eigene Klassen extrahieren.

Active Records sind ähnlich zu den Tabellen der angefragten Datenbank aufgebaut und verstecken daher die Datenquelle relativ schlecht.

Verwendung Bei einfacher Geschäftslogik ist ein Active Record eine gute Wahl aufgrund seiner Einfachheit. Daher wird der Datenzugriff in einem einfachen Domain Model (3.1.2) mit einem Active Record realisiert. Einfache Anfragen und Validierungen können ohne Umstände implementiert werden.

Auch ein Transaction Script (3.1.1) lässt sich gut mit einem Active Record kombinieren. Dadurch lassen sich Codeduplikate vermeiden. Häufig findet man zunächst ein Gateway (3.2.1) als erstes Zugriffsmuster. Bei zunehmender Komplexität der Transaction Scripts, wird mehr Funktionalität in die Gateways ausfaktorisiert, wodurch sie zu Active Records werden.

Der Nachteil von Active Records liegt an der engen Bindung zur Struktur der Datenquelle. Komplexere Geschäftslogiken verlangen umfangreichere, objektorientierte Strukturen, die schlecht auf ein Active Record abzubilden sind.

3.2.3 Data Mapper

Eine Schicht von Mappers, die Daten zwischen Objekten und der Datenbank transportieren und dabei beide Schichten unabhängig voneinander und von den Mappers lassen
[4, S. 165]

Aufbau und Funtionsweise Ein Data Mapper ist eine Indirektion, um die objektorientierte Geschäftslogik von der relationalen Datenhaltung in den Datenbanken zu entkoppeln. Um diese komplexe Aufgabe zu erfüllen, sind viele Entwurfsentscheidungen bei der Verwendung von Data Mapper zu treffen. Daher existieren viele Varianten von Data Mapper und man findet weitere objektrelationale Muster bei der Verwendung von Data Mapper wieder.

Die Dinge, die ein Data Mapper bei der Abbildung von Objekten auf Relationen beachten muss, umfassen u.a. die Abbildung von Attributen der Objekte auf mehrere Tabellenspalten, Klassen, die aus mehreren Tabellen entstehen, die Behandlung von Vererbung und Assoziationen zwischen Objekten.

Das Erzeugen von Objekten kann hier zu Problemen führen, da die Objekte mit den Daten aus dem Anfrageergebnis initialisiert werden müssen. Dafür müssen die Objekte öffentliche Methoden dem Data Mapper zu Verfügung stellen, die aber eventuell nicht der Geschäftslogik zu Verfügung stehen sollen.

Eine mögliche Lösung besteht darin, die Sichtbarkeitsregeln der Plattform auszunutzen und die Geschäftsobjekte und Data Mapper nahe beieinander zu kaspeln. Dies führt aber dazu, dass Teile des Systems, welche die Geschäftsobjekte benutzen, auch die Data Mapper kennen, was unerwünscht ist. Eine weitere, wenn auch langsamere, Lösung wäre die Nutzung von Reflection, um die Sichtbarkeiten von Feldern zu umgehen.

Ein anderer Ansatzpunkt ist der Konstruktor. Einerseits bietet ein umfangreicher Konstruktor eine komfortable Möglichkeit, ein Objekt mit den nötigsten Attributen zu initialisieren. Hierbei können aber zyklische Abhängigkeiten entstehen, wenn mehrere Objekte, die über Assoziationen verbunden sind, initialisiert werden. Hier lohnt es sich, zunächst ein leeres Objekt zu erzeugen und zwischenzuspeichern (z.B. in einer Identity Map, 3.3.2). Falls ein Zyklus vorhanden ist, wird das Objekt nicht mehr aus der Datenbank geladen, sondern ist bereits als leere Hülle in der Anwendung vorhanden.

Beim Einfügen und Aktualisieren muss sichergestellt werden, dass der Data Mapper nur veränderte Objekte berücksichtig, neu erzeugte Objekte in die Tabellen einträgt und Gelöschte aus der Tabelle entfernt.

Da Objekte in einem Objektgraph interagieren, wird mit einer Anfrage häufig ein großer Teilbaum in den Speicher geladen. Das Ziel hierbei ist dann, die Datenbankzugriffe zu minimieren. Hierbei können Muster wie Lazy Load (3.3.3) helfen.

Oft werden mehrere Data Mapper verwendet. Der Aufwand, der hier für Implementierung und Wartung gebraucht wird, lässt sich durch den Einsatz von Metadata Mapping (3.5.1) verringern.

Verwendung Der Vorteil von Data Mapper ist die zusätzliche Indirektionsschicht, die es erlaubt, dass sich Datenbankschema und die Geschäftsobjekte unäbhängig voneinander entwickeln können. Daher findet man Data Mapper oft in Anwendungen mit einem reichhaltigen Domain Model (3.1.2).

In anderen Szenarien lohnt sich der Aufwand durch die zusätzlich zu verwaltende Schicht der Data Mapper oft nicht. Allerdings existieren bereits viele Werkzeuge, die eine konfigurierbare Data Mapper Schicht anbieten, so dass der Aufwand der Implementierung verringert werden kann.

3.3 Objekt-Relationale Verhaltensmuster

3.3.1 Unit of Work

Zeichnet durch Transaktionen beeinflusste Objekte auf, koordiniert die Speicherung von Veränderungen und löst nebenläufige Probleme [4, S. 184]

Aufbau und Funktionsweise Die Geschäftslogik einer betrieblichen Anwendung führt häufig zu Änderungen an Objekten der Domäne, die auch persistent gespeichert werden sollen. Jede Änderung einzelnd zu speichern, kann zu Performanzproblemen führen, da Datenbankzugriffe langsam sind.

Die Aufgabe einer Unit of Work besteht darin, Veränderungen an Objekten zu vermerken. Somit kann die Datenbank in einer Transaktion aktualisiert werden. Bei der Verwendung von einer Unit of Work, lassen sich auch Inkonsistenzen beim gleichzeitigen Laden von Objekten vermeiden.

Damit die Veränderungen vermerkt werden, müssen die zu speichernden Objekte bei der Unit of Work registriert werden. Eine Möglichkeit dies zu tun, ist dem Benutzer des Objektes die Registrierung vornehmen zu lassen (*caller registration*, [4, S. 185f]). Dadurch lassen sich auch Änderungen durchführen, welche später nicht gespeichert werden sollen. Allerdings können diese Aufrufe auch unabsichtlich vergessen werden, wodurch die Änderungen verloren gehen.

Eine andere Möglichkeit ist, es dem Objekt zu überlassen, seine Änderungen der Unit of Work mitzuteilen. Dafür muss dem Objekt seine Unit of Work übergeben werden. Eine weitere Methode besteht darin, bei der Erzeugung eines Objektes eine Kopie des ursprünglichen Zustandes zu speichern und bei einer Speicherung den Orginalzustand mit der veränderten Version zu vergleichen.

Die eigentliche Arbeit beim Persistieren der Veränderungen bleibt der Unit of Work überlassen. Die Unit of Work übernimmt die Überprüfung von nebenläufigen Zugriffen, referenzieller Integrität u.a. Zusicherungen. Dadurch bleibt es dem Entwickler erspart, sich um derlei datenbankspezifische Überprüfungen zu kümmern.

Verwendung Objekte in einer Anwendung verändern ihren Zustand über die Zeit und häufig will man diesen Zustand abspeichern. Unit of Work bietet eine zentrale Stelle um Änderungen zu vermerken und Gültigkeitsprüfungen durchzuführen. Dadurch ist sie sowohl für einfache als auch für komplexe Situationen geeignet.

3.3.2 Identity Map

Stellt sicher, dass jedes Objekt nur einmal geladen wird, indem jedes geladene Objekt in einer Map gespeichert wird. Wenn Objekte angefordert werden, wird zunächst in der Map nach einer verfügbaren Kopie geschaut. [4, S. 195]

Aufbau und Funktionsweise Eine Identity Map verhindert, dass ein Objekt mehrfach aus der Datenbank geladen und unterschiedlich verändert wird. Geladene Objekte werden in einer Map vermerkt und jeder weitere Zugriff auf dieses Objekt geht über die Identity Map anstelle eines erneuten Ladens aus der Datenbank. Üblicherweise werden dadurch auch Datenbankzugriffe eingespart, was aber nicht das Hauptziel einer Identity Map ist.

Bei der Implementierung einer Identity Map muss zunächst festgelegt werden, wie der Schlüssel zu den Objekten aussieht. Eine mögliche Wahl ist, den Primärschlüssel der zugrundeliegenden Tabelle zu verwenden, vorausgesetzt, er ist einspaltig und unveränderbar und das Objekt wird nur aus einer Tabelle erzeugt.

Eine weitere Implementierungsentscheidung besteht darin, explizite oder generische Identity Maps zu verwenden. Explizite Identity Maps verfügen über Methoden für jedes gespeicherte Objekt und haben somit den Vorteil statischer Typsicherheit. Generische Identity Maps bieten die gleichen Methoden für alle Objekte an, inklusive eines Parameters, der den zu suchende Objekttyp angibt. Der Vorteil hierbei liegt in der Wiederverwendung einer einzelnen Identity Map. Moderne Programmierumgebungen

bieten auch die Möglichkeit, generische Datentypen zu verwenden, was die Vorteile beider Möglichkeiten kombiniert.

Eine weitere Entscheidung betrifft die Anzahl der Identity Maps. Eine einzige Identity Map funktioniert nur bei datenbankweit eindeutigen Schlüsseln. Üblicherweise existieren genauso viele Identity Maps wie Tabellen oder Klassen, wenn die beiden Strukturen ähnlich sind. Bei Unterschieden empfiehlt sich eine Identity Map pro Klasse, da die Objekte von der darunterliegenden Datenbank unabhängig sein sollen.

Weiter sollte überlegt werden, in welchem Kontext eine Identity Map existieren soll. Der beste Ansatz hierbei ist die Verbindung mit einem sitzungsspezifischen Objekt. Wird eine Unit of Work (3.3.1) verwendet, so ist sie ein geeigneter Ort für eine Identity Map, da alle Schreib- und Leseoperationen hier ausgelöst werden.

Verwendung Eine Identity Map wird haupsächlich dafür eingesetzt, um Inkonsistenzen durch mehrfaches Lesen und Verändern eines Objektes zu verhindern. Ein weiterer Vorteil von Identity Maps ist ihre Funktion als Cachespeicher. Dadurch kann ein gewisser Performanzgewinn durch Einsparung von Datenbankzugriffen erreicht werden.

Eine Identity Map ist für unveränderbare Objekte oder bei einem Dependent Mapping (3.4.4), bei dem die Persistierung eines Objektes durch ein Elternobjekt durchgeführt wird, unnötig.

3.3.3 Lazy Load

Ein Objekt, dass nicht alle benötigten Daten enthält, aber weiß, wie es sie beschaffen kann. [4, S. 200]

Aufbau und Funktionsweise Häufig führt das Laden eines Objektes zum Laden weiterer verwandter Objekte. Dies ist meist der Fall in einem reichen Domain Model (3.1.2). Das vereinfacht die Arbeit des Entwicklers, der sonst alle Objekte, die er benötigt, explizit laden müsste. Allerdings kann es passieren, dass große Teile des Objektgraphen geladen werden, die meist nicht benötigt werden und zu Speicherengpässen führen.

Ein Lazy Load verhindert das voreilige Laden von unnötigen Objekten, indem ein Platzhalter für das eigentliche Objekt konstruiert wird. Dadurch lassen sich auch Performanzprobleme verhindern, da Datenbankzugriffe eingespart werden.

Für die Implementierung des Platzhalters gibt es vier mögliche Varianten. Bei *Lazy Initialization* ([2]), wird bei Zugriff auf ein Feld zuerst überprüft ob dieses gültig ist.

Ein *virtueller Proxy* ([5]) dagegen stellt einen leeren Platzhalter an die Stelle des erwarteten Objektes. Erst bei Zugriff auf das Objekt, wird es geladen. Das Problem hierbei ist, die Objektidentität von mehreren Proxies eines Objektes zu gewährleisten, da jeder Proxy eine eigenes Objekt darstellt.

Ein *Value Holder* ist ein Wrapper für ein Objekt. Beim Zugriff, wird der Value Holder nach dem Objekt gefragt und dieser lädt beim ersten Zugriff das Objekt aus der Datenbank. Dazu ist es notwendig, die Klasse wissen zu lassen, dass ein Value Holder für ein Objekt existiert, was meist unerwünscht ist.

Die letzte Variante für die Implementierung eines Platzhalters ist ein *Ghost*, der ein echtes Objekt in einem halbinitialisierten Zustand darstellt. Beim Laden wird ein

Ghost nur mit seiner ID initialisiert. Beim Zugriff auf die Felder des Objektes, werden die Daten geladen.

Ein großes Performanzproblem bei Lazy Load stellt das sogenannte *ripple loading* ([4, S. 202] dar. Beim Iterieren über eine Liste von Lazy Load Objekten, wird jedes Objekt einzeln geladen. Eine mögliche Alternative ist, die gesamte Liste als ein Lazy Load Objekt zu betrachten. Bei größeren Listen ist eventuell ein seitenweises Laden angebrachter.

Oft kann man die Größe des zu ladenden Objektgraphen im Voraus abschätzen. Hierfür ist es empfehlenswert, mehrere Stufen des verzögerten Ladens zu implementieren, wobei jede Stufe eine bestimmte Anzahl von assozierten Objekten lädt. Meist reichen aber zwei Stufen, ein komplettes Laden und ein Laden zum ausreichenden Identifizierung des Objektes, aus.

Verwendung Die Entscheidung ein Lazy Load zu verwenden, basiert grundsätzlich auf der Entscheidung wie viele Daten und wann Daten aus der Datenbank geladen werden.

Sinn macht ein Lazy Load nur dann, wenn der Zugriff auf ein Attribut einen weiteren Datenbankzugriff nötig macht. Wie oben angeführt hat Lazy Load den großen Vorteil des verzögerten Ladens, aber es ist kompliziert zu implementieren und sollte daher nur wenn nötig eingesetzt werden.

3.4 Objekt-Relationale Strukturmuster

3.4.1 Identity Field

Speichert einen Datenbankschlüssel in einem Objekt, so dass die Identität zwischen Objekt und Daten sichergestellt ist. [4, S. 216]

Aufbau und Funktionsweise Datenbanken identifizieren die Daten in einer Tabelle durch Schlüssel. Objekte brauchen einen solchen Schlüssel nicht, da die Identität der Objekte durch die zugrundeliegende Plattform sichergestellt wird. Um die Identität zwischen Daten der Datenbank und der Objekte sicherzustellen, wird der Datenbankschlüssel in den Objekten gespeichert.

Die Entscheidungen, die bei der Verwendung von Identity Field getroffen werden, beziehen sich haupsächlich auf den zu verwendenden Schlüssel in der Datenbank, die Speicherung des Schlüssels in einem Objekt und die Generierung neuer Schlüssel.

Die Wahl eines geeigneten Schlüssels ist eine vieldiskutierte Entwurfsentscheidung im Bereich des Datenbankentwurfs. Dass der Schlüssel auch in Objekten verwendet werden soll, hat einen nicht unerheblichen Einfluss auf die Entscheidung. So muss zunächst die Wahl zwischen einfachen Schlüsseln, die nur eine Spalte der Relation beanspruchen und aus mehreren Spalten zusammengesetzten Schlüsseln getroffen werden. Zusammengesetzte Schlüssel beinhalten häufig eine semantische Bedeutung. Dadurch lässt sich oft die Eindeutigkeit und die Unveränderlichkeit der Schlüssel nicht garantieren.

Eine weitere Entscheidung betrifft die Wahl zwischen tabellenweit oder datenbankweit eindeutigen Schlüsseln. Ein datenbankweit eindeutiger Schlüssel erlaubt die Verwendung einer Identity Map (3.3.2).

Von dem verwendeten Datenbankschlüssel hängt die Repräsentation des Schlüssels in den Objekten ab. Zusammengesetzte Schlüssel benötigen meist eine eigene Klasse, welche die entsprechende Menge von Objekten kaspelt. Die Hauptaufgabe dieser Schlüsselklasse ist es, die Gleichheit der Schlüssel zu püfen. Die Klasse kann generisch sein, was die Wiederverwendbarkeit erhöht, oder explizit die Felder des Schlüssels beinhalten. Dabei besteht jedoch die Gefahr, dass viele kleine Klassen mit wenig Funktionalität zu erhalten.

Das nächste Problem besteht in der Generierung neuer Schlüssel. Eine Möglichkeit besteht darin, den Schlüssel automatisch von der Datenbank erzeugen zu lassen. Das führt zu Problemen beim Einfügen von abhängigen Objekten, da der neu generierte Schlüssel von den Datenbanken während der Transaktion zur Verfügung gestellt wird und somit nicht als Fremdschlüssel eingetragen werden kann.

Die Verwendung von GUIDs (Global Unique IDentifier) führt zu eindeutigen, von der jeweiligen Hardware abhängigen Schlüsseln. Diese haben allerdings den Nachteil, das der String sehr lang ist und es zu Performanzproblemen führt, z.B. bei Vergleichen oder Indizes.

Neue Schlüssel lassen sich auch selbst generieren. Dazu existieren verschiedene Vorgehensweisen. Eine davon besteht darin, eine eigene Schlüsseltabelle zu verwenden, in welcher der zuletzt generierte Schlüssel jeder Tabelle gespeichert wird.

Verwendung Identity Field wird verwendet, wenn die Identität zwischen Spalten in einer Datenbanktabelle und den Daten in den Objekten sichergestellt sein soll. Daher findet es häufig zusammen mit einem Domain Model (3.1.2) Anwendung.

Unnötig wird ein Identity Field bei kleinen Objekten, die keine eigene Tabelle besitzen. Hier ist die Verwendung eines Embedded Value (3.4.5) angebrachter. Existiert ein komplexer Objektgraph in der Domäne, in dem in der Datenbank nicht selektiert wird, so läßt sich der Graph effizienter mit einem Serialized LOB (3.4.6) speichern.

Soll der Schlüssel nicht in dem Domänenobjekt gespeichert werden, so kann eine Veränderung einer Identity Map (3.3.2) die Identität sicherstellen.

3.4.2 Foreign Key Mapping

Bildet die Assoziation zwischen Objekten auf Fremdschlüsselbeziehungen zwischen Tabellen ab. [4, S. 236]

Aufbau und Funktionsweise Objektorientierte Systeme beinhalten auf unterschiedlichste Art verknüpfte Objekte. Da die Daten in den Objekten meist von diesen Assoziationen abhängen, müssen diese mit gespeichert werden. Hierfür eignet sich idealerweise ein Identity Field (3.4.1). Dadurch lassen sich Assoziationen relativ einfach auf Fremdschlüsselbeziehungen abbilden.

Schwieriger wird es, ein Update durchzuführen, wenn eine Menge von Objekten an mindestens einem Ende der Assoziation steht. Dabei muss herausgefunden werden, welche Daten verändert wurden. Die einfachste Variante ist es, alle Daten zu löschen und die unveränderten mit den veränderten Daten zurückzuschreiben. Das funktioniert jedoch nur, solange keine weitere Abhängigkeit zu diesen Daten besteht. Verändert man das Objektmodell und macht eine Assoziation bidirektional, können solche Probleme einfach umgangen werden.

Eine dritte Variante besteht darin, die Veränderungen durch Abgleich mit den ursprünglichen Daten herauszufinden. Dabei kann entweder mit dem aktuellen Datenbankzustand verglichen werden, was ein erneutes Lesen der Daten voraussetzt, oder mit dem ursprünglich Gelesenem, was ein Speichern des gelesenen Zustandes voraussetzt.

Besondere Vorsicht verlangt das Laden von abhängigen Objekten aus der Datenbank. Vor Allem bei zyklischen Abhängigkeiten kann es vorkommen, dass es zu einem rekursiven Laden kommt. Hierbei erweist sich ein Lazy Load (3.3.3) oder eine Identity Map (3.3.2) als nützlich.

Verwendung Beliebige Assoziationen zwischen Objekten können mittels eines Foreign Key Mappings abgebildet werden. Einzig eine n:m Beziehung benötigt ein Association Table Mapping (3.4.3).

Bei einer 1:n Beziehung sollte die Anwendung eines Depending Mappings (3.4.4) berücksichtigt werden. Dies vereinfacht die Behandlung von Mengen von assozierten Objekten.

Ist das assozierte Objekt ein einfaches Objekt, das nur Daten enthält, so sollte eine Anwendung eines Embedded Values (3.4.5) berücksichtig werden.

3.4.3 Association Table Mapping

Speichert eine Assoziation als Tabelle mit Fremdschlüsseln der Tabellen, welche durch die Assoziation verbunden sind. [4, S. 248]

Aufbau und Funktionsweise Die Behandlung von Mengen ist in objektorientierten Sprachen durch die Benutzung von entsprechenden Klassen einfach. Relationale Datenbanken verfügen nicht über ein solches Konzept. Um eine n:m Beziehung zu modellieren, muss eine neue Tabelle angelegt werden, welche die Fremdschlüssel der assozierten Tabellen beinhaltet. Ein Association Table Mapping wird verwendet, um eine Abbildung von n:m Beziehungen zwischen Objekten auf die zusätzliche Tabelle vorzunehmen.

Die zusätzliche Tabelle besitzt kein entsprechendes Objekt im System und hat daher auch keine ID. Um Objekte der Assoziation zu laden, sind mindestens zwei Datenbankanfragen nötig. Eine zum Laden der Fremdschlüssel und eine Zweite zum Auflösen der Fremdschlüssel auf die assozierten Daten.

Idealerweise sind alle Daten bereits im Speicher; andernfalls können die Anfragen an die Datenbank sehr teuer werden. Alternativ können die Tabellen auch in der Datenbank durch einen Join vereint werden. Das hat den Nachteil, dass die Daten mehrerer Objekte aus einer Tabelle herausgesucht werden müssen.

Durch die Verwendung eines Dependent Mappings (3.4.4) für die Verknüpfungstabelle, werden eventuelle Schwierigkeiten beim Ändern der Assoziationsdaten verhindert.

Verwendung Die hauptsächliche Verwendung von einem Association Table Mapping ist bei einer n:m Beziehung zwischen Objekten. Es kann auch für andere Beziehungen eingesetzt werden. Da es aber komplexer ist als z.B. ein Foreign Key Mapping (3.4.2), lohnt sich der Aufwand häufig nicht.

Andererseits kann ein Association Table Mapping nützlich sein, wenn man keine Änderungen an einem existierenden Datenbankschema vornehmen kann, z. B. um Ta-

bellen nachträglich zu verknüpfen oder eine existierende Verknüpfungstabelle abzubilden. Trägt die Verknüpfungstabelle weitere Informationen über die Beziehung, so sollte ein eigenes Domänenobjekt für die Tabelle erstellt werden.

3.4.4 Dependent Mapping

Läßt eine Klasse die Datenbankabbildung für ein abhängiges Objekt durchführen. [4, S. 262]

Aufbau und Funktionsweise Objekte sind häufig abhängig von der Existenz von anderen Objekten. Wenn auch in der Datenbank keine weitere Tabelle auf diese abhängigen Daten verweist, läßt sich die Abbildung durch die Anwendung eines Dependent Mappings vereinfachen.

In einem Dependent Mapping verläßt sich eine abhängige Klasse auf eine andere Klasse, den Eigentümer, welche die Datenbanklogik für die abhängige Klasse implementiert.

Der Aufbau kann sehr variieren. In einem Active Record (3.2.2) oder einem Row Data Gateway (3.2.1) liegt die gesamte Datenbanklogik beim Eigentümer. Bei der Verwendung von Data Mapper (3.2.3) existiert kein Mapper-Objekt für die abhängige Klasse. Bei einem Table Data Gateway (3.2.1) existiert die abhängige Klasse nicht einmal.

Beim Laden eines Eigentümer-Objektes, werden auch alle abhängigen Daten geladen. Durch Anwenden eines Lazy Loads (3.3.3) kann das Laden verzögert werden.

Die Hierarchie von Eigentümer und abhängigen Klassen läßt sich schachteln, wobei dann der primäre Eigentümer für das Laden aller abhängigen Klassen verantwortlich ist.

Ein Nachteil der Verwendung von Dependent Mapping liegt in der Verfolgung von Veränderungen. Alle Veränderungen der abhängigen Klassen müssen auch den Eigentümer als verändert deklarieren, so dass die Veränderungen gespeichert werden.

Verwendung Ein Dependent Mapping läßt sich am Besten verwenden, wenn ein Objekt nur von einem anderen Objekt referenziert wird. Dadurch lassen sich die bereits erwähnten Schwierigkeiten beim Update von Mengen von Objekten umgehen.

3.4.5 Embedded Value

Bildet ein Objekt auf mehrere Spalten der Tabelle eines Objektes ab. [4, S. 268]

Aufbau und Funktionsweise Oft besitzt ein Objektgraph viele kleine nützliche Objekte, welche aber als eigenständige Tabelle kaum Sinn machen würden. Besitzt ein Objekt, welches in einer Tabelle gespeichert wird eine Referenz auf ein solches Objekt, so läßt sich dieses auf mehrere Spalten der Tabelle abbilden. Dadurch werden die abhängigen Objekte beim Laden oder Speichern der eigentlichen Objekte ebenfalls geladen oder gespeichert.

Verwendung Die Schwierigkeit beim Verwenden eines Embedded Values liegt darin, eine geeignete Situation zu erkennen.

Eine mögliche Verwendung besteht darin, in einem vorgegebenem Datenbankschema eine Tabelle in mehrere Objekte aufzuteilen, die in dem Domänenmodell vorkommen.

Ausgehend von einem Domänenmodell ist es sinnvoll, Embedded Value bei 1:1 Beziehungen einzusetzen. Weitere Fälle lassen sich durch das Betrachten von Speicher- und Ladevorgängen erschliessen. Wird ein Objekt nur dann geladen oder gespeichert, wenn ein anderes Objekt erzeugt wird, dann ist es möglicherweise ein Kandidat für ein Embedded Value.

Eine Alternative zum Embedded Value ist ein Serialized LOB (3.4.6), welches den Vorteil hat, dass die Objekte in einer Anfrage verwendet werden können.

3.4.6 Serialized LOB

Speichert ein Graph von Objekten, indem eine serialisierte Version in einem einzigen großen Objekt (Large OBject, LOB) in einem Eintrag abgelegt wird. [4, S. 272]

Aufbau und Funktionsweise Ähnlich wie bei einem Embedded Value (3.4.5) wird auch ein Serialized LOB bei vielen kleinen Objekten angewendet, die als eigenständige Tabellen wenig sinnvoll erscheinen.

Allerdings werden die Daten eines kompletten Objektgraphen oder eines Ausschnittes eines Graphen serialisiert und als ein Eintrag in der Datenbank gespeichert. Dies ist vor allem bei Objektgraphen, die selten über eine SQL Anfrage selektiert werden, effizient. Eine weitere Anwendungsmöglichkeit von Serialized LOB besteht bei hierarchisch angeordneten Objekten, da relationale Datenbanken nur sehr umständlich mit hierarchischen Daten umgehen können.

Es gibt zwei Arten von Serialisierung von Objekten: in Binärdarstellung oder in Zeichendarstellung. Die Binärdarstellung als BLOB (*Binary Large OBject*) ist die einfache Variante, da viele Plattformen bereits Serialisierungsfunktionen bereitstellen und diese effizient komprimieren. Der Nachteil von BLOBs ist zum einen, dass die Datenbank einen Binärdatentyp unterstützen muss und zum anderen, dass der Graph nur als Ganzes rekonstruiert werden kann.

Als alternative Darstellung eignen sich CLOBs (*Character Large OBjects*), bei der der Graph als Zeichenkette serialisiert wird und somit lesbar und durchsuchbar wird. Häufig wird eine XML Darstellung gewählt, da dies ein weit verbreiteter Standard ist. Der Nachteil hierbei liegt eindeutig in dem enormen Platzverbrauch und den damit verbundenen Performanzeinbußen bei der Übertragung.

Verwendung Ein Serialized LOB eignet sich für das Speichern eines Teils eines Objektgraphen, welcher nur selten in SQL Anfragen gebraucht wird. XML ist hier hilfreich, um einen einfachen textuellen Ansatz zu implementieren und um mit XPath eventuell auftretende Anfragen in den Objektgraphen zu stellen.

3.4.7 Inheritance Mappers - Single, Class und Concrete Table Inheritance

Organisiert Datenbankabbildungen in eine Struktur, die Vererbung unterstützt
[4, S. 302]

Aufbau und Funktionsweise Eine der am aufwendigsten abzubildenden Strukturen objektorientierter Systeme ist die Vererbung. Relationale Datenbanken unterstützen keine Vererbungsstrukturen. Es existieren verschiedene Ansätze, mit dem Ziel mit möglichst geringem Aufwand Vererbungshierarchien abzubilden, welche auf einem Grundschema basieren.

Die prinzipielle Lösung besteht darin, entlang der Hierarchie der zu speichernden Objekte eine Hierarchie von Mappern (3.2.3) aufzubauen, die für das Laden und Speichern des jeweiligen konkreten Objektes zuständig ist. Dabei wird von den Laden- und Speichermethoden der Oberklassen Gebrauch gemacht. Dadurch ist es möglich, dass jede Klasse der Hierarchie nur ihre Daten verwalten muss.

Eine Herausforderung stellen abstrakte Klassen dar. Möchte man auch diese Klassen speichern, muss sichergestellt werden, dass der Mapper der abstrakten Klasse einen Mapper einer konkreten Klasse benutzt. Daher ist es sinnvoll, diese Mapper aus der Mapper-Hierarchie auszugliedern.

Das generelle Schema sagt noch nichts darüber aus, wie die Objekte in der Datenbank abgelegt werden. Dazu existieren drei populäre Methoden, die auch gemischt verwendet werden können.

Bei *Single Table Inheritance* ([4, S. 278ff]) werden die Daten der Objekthierarchie in einer einzigen Tabelle gespeichert. Dadurch werden teuere Joins verhindert, welche sonst gebraucht werden, um die Klassendaten zu erhalten. Allerdings muss auch eine Spalte vorgesehen werden, die auf das zu erzeugende Objekt hinweist. Sonst besteht die Gefahr, dass ein Objekt nicht vollständig initialisiert werden kann, da entsprechende Spalten leer sein können.

Im Gegensatz dazu, wird bei *Class Table Inheritance* ([4, S. 285ff]) für jede Klasse in der Hierarchie eine eigene Tabelle erzeugt. Dafür ist es notwendig, die Zeilen der Tabelle zu verknüpfen, die zu einem Pfad im Objektbaum gehören. Dazu können Schlüssel verwendet werden, die in jeder Tabelle auf das gleiche Objekt verweisen. Alternativ kann jede Tabelle eigene Schlüssel verwenden und über Fremdschlüssel auf zusammengehörige Daten verweisen. Die Herausforderung liegt hier bei der Zusammenführung der nötigen Daten, sei es durch mehrfache Datenbankabfragen auf die verschiedenen Tabellen oder durch Verwendung von Joins. Beide Ansätze können zu Performanzproblemen führen.

Beim letzten Ansatz, dem *Concrete Table Inheritance* ([4, S. 293ff]) Muster, existiert eine Tabelle pro konkreter Klasse in der Datenbank. Jede Tabelle enthält hierbei Spalten für alle Attribute des Objektes und der Oberklassen, so dass die Spalten in den Tabellen mehrfach vorkommen.

Verwendung Bei der Abbildung von Vererbungsstrukturen auf Tabellen, sollte der Aufwand möglichst gering gehalten werden. Hier bieten Inheritance Mappers ein einfaches Grundprinzip mit verschiedenen Implementierungsmöglichkeiten, die komplementär zueinander eingesetzt werden können. So macht es z.B. Sinn, für einige Unterklassen Concrete Table Inheritance zu verwenden und Single Table Inheritance für die restlichen Klassen der Hierarchie.

Unter den Optionen zeichnet sich Single Table Inheritance durch die einfache Verwaltung der Daten in einer einzigen Tabelle aus. Dadurch wird das Finden und Einfügen der Daten enorm vereinfacht. Allerdings kann es zu leeren Enträgen in der Tabelle kommen, was je nach Datenbank ein Platzproblem darstellt. Außerdem kann die Ta-

belle sehr groß werden und muss bei Operationen eventuell gesperrt werden, was der Performanz schaden kann.

Class Table Inheritance ist aus der objektorientierte Sichtweise eine sehr natürliche Abbildung von Vererbung, da alle Spalten den jeweiligen Feldern entsprechen und die Beziehungen zwischen den Tabellen den Vererbungsstrukturen ähneln. Allerdings kann das Zusammenführen der Daten umständlich und langwierig sein und das Datenbankschema wird zusätzliche anfällig für Refactorings in der Anwendung.

Conrete Table Inheritance scheint auf dem ersten Blick ein guter Kompromiß zwischen den ersten beiden Ansätzen, Single und Class Table Inheritance, zu sein, da jede Tabelle alle nötigen Informationen zum Erzeugen der Objekte enthält. Allerdings kann die Schlüsselverwaltung kompliziert sein und bei Veränderungen des Domänenmodells muss die Datenbank angepasst werden.

3.5 Muster für die OR-Abbildung mittels Metadaten

3.5.1 Metadata Mapping

Hält Details der Objekt-Relationen-Abbildung in Metadaten. [4, S. 306]

Aufbau und Funktionsweise Der Code, der zur Abbildung von Objekten auf Relationen entsteht, ist meist sehr ähnlich zueinander, aber sehr mühsam zu schreiben. Mit der Auslagerung von Abbildungsregeln in Metadaten läßt sich nicht nur der Programmieraufwand reduzieren, sondern auch die Wartbarkeit der Abbildungsregeln verbessern.

Der Aufwand bei der Nutzung von Metadaten entsteht beim Einbinden der Daten in die Anwendung. Dazu kann ein Codegenerationsverfahren oder Reflection verwendet werden.

Bei der Codegeneration werden aus den Metadaten bei der Kompilierung die Mapper-Klassen automatisch erzeugt. Eine Änderung der Metadaten zieht somit ein erneutes Erzeugen und Ausliefern der Klassen mit sich.

Dagegen liest ein Reflection-Programm die Metadaten der zu persistierenden Klasse ein und ruft dynamisch die notwendigen Attribute oder Methoden der jeweiligen Objekte auf, um die Daten zu erhalten. Reflection ist sehr flexibel, da bei einer Änderung der Metadaten, keine Änderungen am Quellcode vorgenommen werden müssen. Es wäre sogar möglich, diese Änderungen der Metadaten zur Laufzeit vorzunehmen. Allerdings ist dieser Ansatz meist sehr langsam, doch im Kontext einer langsamen Datenbankanfrage ist die Geschwindigkeit eventuell vertretbar.

Zur Speicherung von Metadaten eignet sich XML als Format, da es lesbar ist und bereits zahlreiche Parser existieren, was den Aufwand zum Erstellen und Einlesen der Daten vereinfacht. Die Komplexität der Metadaten kann je nach Anwendung variieren. Ein generisches Metadatenmodell ist meist sehr komplex, doch die meisten Anwendungen nutzen nicht alle objektrelationalen Abbildungsmöglichkeiten. Hier kann ein einfaches Metadatenmodell den Aufwand verringern.

Verwendung Metadata Mapping kann den Aufwand bei der objektrelationalen Abbildung sehr vereinfachen. Allerdings gibt es Anwendungen, bei denen eine Abbildung mittels Metadaten nicht akzeptabel ist. Der Aufwand zwischen selbstgeschriebene Abbildungslogik und Metadaten sollte daher für die jeweilige Anwendung selbst abgeschätzt

werden. Ein eigene Lösung, die gut entworfen und an die Domäne angepasst ist, kann eventuell besser sein als eine generische Metadatenlösung.

Metadaten sind häufig anfällig für Refactorings. Ändert sich der Code so kann ein Metadaten-Framework eventuell die nötigen Felder nicht mehr ansprechen. Die anschließende Fehlersuche in erzeugtem Code oder in Reflectioncode ist meist sehr mühsam.

3.5.2 Query Object

Ein Objekt, das eine Datenbankanfrage darstellt. [4, S. 316]

Aufbau und Funktionsweise Zur Kommunikation mit einer Datenbank wird hauptsächlich SQL als Standardsprache eingesetzt. Häufig sind Entwickler wenig oder gar nicht mit SQL vertraut. Mit einem Query Object lassen sich Datenbankanfragen mit Hilfe von Objekten und Klassen darstellen. Ein Query Object ist somit eine Anwendung des Interpreter Musters ([5]) auf SQL Anfragen.

Query Object ist ein sehr komplexes Muster und es ist sinnvoll, zunächst nur die gewünschte Funktionalität zu implementieren und je nach Bedarf das Muster zu erweitern.

Ein Query Object arbeitet mit den Objekten im Speicher anstelle der Datenbankschemata, daher wird es meist in Zusammenhang mit Metadata Mapping (3.5.1) verwendet, damit die Objekte auf die Tabellen abgebildet werden können.

Ein fortgeschrittenes Query Object unterstützt z.B. die SQL Dialekte verschiedener Datenbanken, um Daten auf mehreren Datenbanken zu speichern oder ein Austausch zwischen unterscheidlichen Datenbanken zu vereinfachen. Eine weitere Eigenschaft ist das Wiederverwenden von ähnlichen Anfragen. So ist es relativ einfach eine Anfrage zu laden, welche eine Einschränkung einer bereits gestellten Anfrage aus einer Identity Map (3.3.2) ist.

Verwendung Query Object ist ein sehr komplexes Muster, welches aber die Arbeit mit den Daten sehr vereinfachen kann. Es wird häufig in Verbindung von Metadata Mapping (3.5.1) verwendet und bei komplexen Domänenmodellen (3.1.2) angewendet, die mittels Data Mapper (3.2.3) abgebildet werden. Erst in solchen Situtationen lässt sich das Potenzial von einem Query Object voll ausnutzen.

Mit einem Query Object wird meist eine lose Kopplung zur verwendeten Datenbank und zum Datenbankschema sowie häufig eine Optimierung von Anfragen verwirklicht. Diese Anforderungen sind oft in komplexen Anwendungen nötig.

3.5.3 Repository

Vermittelt zwischen der Domäne und der Abbildungsschicht, indem es eine Schnittstelle zum Auffinden von Domänenobjekten zur Verfügung stellt. [4, S. 322]

Aufbau und Funktionsweise Ein Repository dient als Vermittler zwischen einem komplexen Domänenmodell und der dazugehörigen Schicht von Data Mapper (3.2.3).

Eine solche Zwischenschicht ist vor allem dann nötig, wenn viele Anfragen an die Datenbank gestellt werden und Anfragen wiederholt auftreten. Ein Repository verhält

sich wie eine kleine objektorientierte Datenbank, die eine einfache Schnittstelle bietet, um Objekte nach bestimmten Kriterien aufzusuchen. Die Objekte werden allerdings nicht im Speicher gehalten, sondern mit Hilfe der Data Mapper aus der Datenbank geladen.

Dabei können die Selektionsmethoden der Data Mapper (3.2.3) durch eine Funktionalität ersetzt werden, welche mittels Kriterienobjekten die Domänenobjekte auffindet. Eine Repository ist ein komplexes Muster, das viele andere Muster verwendet. Zum Einen ist es eine Weiterentwicklung des Query Object Musters (3.5.2) und zum Anderen bietet es in Zusammenarbeit mit einem Query Object eine wesentliche Vereinfachung des Datenzugriffs an. Um das nötige SQL zur Datenbankabfrage aus den Kriterien zu generieren, wird bei einem Repository häufig das Query Object Muster (3.5.2) mit dem Metadata Mapping (3.5.1) kombiniert.

Ein weiteren Vorteil, den das Repository bietet, ist die Unabhängigkeit von der zugrundeliegenden Datenhaltung. So kann ein Repository genutzt werden, um aus verschiedenen Quellen (Datenbanken, XML Dateien, etc.) Daten zu erhalten.

Verwendung Ein Repository wird meist in komplexen Anwendungen mit einem reichhaltigen Domänenmodell und häufigen Anfragen verwendet. Der Schwerpunkt liegt hierbei insbesondere in der Verwendung mehrerer Datenquellen. Auch beim Ausführen von Tests kann ein Repository hilfreich sein, indem es als echte Datenbank im Speicher fungiert und somit langsame Datenbankabfragen vermeidet.

Es sei noch erwähnt, dass hinter der Repository-Schnittstelle die Strategien zum Auffinden der Daten beliebig variiert werden können, so dass unveränderliche, häufig gebrauchte Objekte z.B. im Speicher gehalten werden.

4 Kombination von Objekt-Relationalen Mustern

Die Kombinationsmöglichkeiten der im letzten Abschitt vorgestellten Muster sind vielfältig. Abhängig von der Komplexität der Anwendung werden unterschiedliche Muster verwendet, um ein effiziente Abbildung der Objekte auf Relationen zu erreichen.

Um die Wahl der zu verwendenden Muster einzuschränken, sollten verschiedene Aspekte in der Entwicklung und beim Zusammenspiel der Muster berücksichtigt werden.

Der Einsatz der Muster ist abhängig von bereits bestehenden Software-Artefakten. So ist die Verwendung von komplexen Mustern, wie Repository (3.5.3) oder Metadata Mapping (3.5.1) in umfangreichen Software-Architekturen eher denkbar als in kleinen Anwendungen.

Bei der Refaktorisierung bestehender Anwendungen sollten daher nur die notwendigsten Implementierungen vorgenommen werden, die Probleme bei der Weiterentwicklung der Anwendung lösen. Der Einsatz unnötig komplexer Muster verzögert meist nur die Refaktorisierung und führt nicht selten zu unnötigen weil ungenutzten Code.

Eine weitere Einschränkung in der Anwendung und der Kombination der Muster besteht in der Unterstützung der verwendeten Plattform. Einige Plattformen bietet eine integrierten Persistenzlösung, die eine effizientere Leistung bietet als eine aufwändige Eigenentwicklung.

Auch der Einsatz von Werkzeugen zur Persistierung der Daten sollte bedacht werden. Diese Werkzeuge bieten eine großen Funktionalität und hervorragende Leistung

in verschiedensten Anwendungsfällen. Dadurch können sich die Entwickler auf die Implementierung der eigentliche Geschäftslogik konzentrieren.

Im Folgendem sollen zwei typische Kombinationsmöglichkeiten der in Abschnitt 3 vorgestellten Muster näher betrachtet werden und mögliche Anwendungsszenarien der kombinierten Muster dargelegt werden.

4.1 Table Module und Gateways

Der Einsatz von Table Module und Gateways kommt relativ selten in der Entwicklung von betrieblichen Anwendungen vor. Allerdings bietet die Kombination der beiden Muster viele Vorteile. Aufgrund der Struktur von Table Module und den jeweilig verwendeten Table oder Row Data Gateway gelingt der Austausch mit der Datenquelle besonders einfach. Deshalb sind hier auch kaum weitere Muster notwendig, da eine umständliche Abbildung auf Relationen vermieden wird. Dies vermindert die Komplexität der Anwendung enorm.

Die .NET Plattform bietet mit ADO .NET (siehe Beschreibung in 5.1) eine Referenzbeispiel für die Verwendung von Table Module und Gateways und liefert gleichzeitig einen Hauptanwendungsfall für die Verwendung dieser Muster. Die einfache Darstellung und das Editieren von Daten in Form von Listen, Tabellen oder Eingabeformularen ist in .NET über die Bindung der jeweiligen Steuerelemente an die Daten möglich.

Für solche einfachen Verarbeitungen und Darstellungen, die bereits in Tabellenform vorliegen, eignet sich die Verwendung von Table Module besonders. Eine Transformation der Daten in Objekte, wie sie z.B. bei der Verwendung von Domain Model geschieht, und die spätere Rücktransformation in Tabellenform zur Darstellung ist hier aus Performanzgründen kaum sinnvoll.

4.2 Metadata Mapping mit Domain Model und Data Mapper

Der hauptsächliche Verwendungszweck der persistenten Daten in betrieblichen Anwendungen besteht in der Verarbeitung dieser. Dazu ist es vom Vorteil, die Daten in eine Objektstruktur zu transformieren, damit sie leichter in einer objektorientierten Programmierumgebung verwendet werden können.

In dieser Situation eignet sich der Einsatz eines, je nach Anwendung einfachen oder komplexen, Domain Models (3.1.2). Wie bereits erwähnt, bietet dieses Objektmodell einen relevanten Ausschnitt aus der realen Geschäftswelt. Zusammen mit den objektorientierten Strukturen läßt sich so die Anwendungslogik einfach realisieren.

Der Datenzugriff erfolgt bei einem einfachen Domain Model meist mit einem Active Record (3.2.2), in dem die Objekte Methoden zum Laden und Speichern anbieten. Bei einem komplexeren Domain Model eignet sich allerdings der Einsatz von Data Mapper (3.2.3), da hier die umfangreichenden objektorientierten Strukuren besser auf die darunterliegenden Tabellen abgebildet werden können.

Aufgrund der Komplexität des Domain Models, findet man beim Einsatz von Data Mappers auch fast alle Muster zur Abbildung von objektorientierten Strukturen (3.4) wieder. Diese werden genutzt um die verschiedenen objektorientierten Strukturen auf relationale Strukturen abzubilden und dabei möglichst wenig Information über die Verbindungen der Objekte zu verlieren.

Wie bereit in 3.5.1 erwähnt, ist die Pflege einer solchen Schicht von Mappers sehr aufwändig und nicht selten führt eine Änderung des Domain Models eine Änderung der jeweiligen Mapper mit sich.

Aufgrund dessen werden solche Mappers und zum Teil auch die Objekte des Domain Models über Metadaten (3.5.1) erzeugt und können so relativ einfach ausserhalb der Anwendung und zum Teil auch zur Laufzeit aktualisiert werden.

Eine ausgereifte Metadatenlösung kann auch die Abbildung von objektorientierten Strukturen übernehmen (siehe etwa 5.2). Hierbei werden die notwendigen Muster (3.4) ebenfalls von dem Metadata-Framework generiert.

5 Beispiele für Anwendungen von Mustern

Wie aus an der Beschreibung der Muster in Abschnitt 3 deutlich wird, ist das Entwerfen und Entwickeln einer angemessenen Datenbankzugriffslogik ein, aufgrund der Komplexität, schwieriges Unterfangen. Aus dieser Tatsache heraus, haben viele Firmen vorgefertigte Werkzeuge entwickelt, welche die Kommunikation zwischen Objekten und Relationen automatisieren und vereinfachen.

Oftmals ist es sinnvoll, in eines dieser Werkzeuge zu investieren, anstatt Zeit für eine Eigenentwicklung aufzubringen. Die Funktionalität dieser Werkzeuge ist meist sehr umfangreich, allerdings lohnt sich eine Anschaffung bei einer Weiterentwicklung einer betrieblichen Anwendung.

Auch verschiedene Open-Source Projekte beschäftigen sich mit der objektrelationalen Abbildung. Auch hier findet man mächtige Werkzeuge, die den kommerziellen Werkzeugen in nichts nachstehen.

Allen objektrelationalen Anwendungen ist es gemeinsam, dass sie viele der hier vorgestellten Muster integrieren. An einem proprietären (Microsoft ADO .NET) und einem Open-Source (Hibernate) Werkzeug, sollen im Folgenden verschiedene Anwendungen der vorgestellten Muster dargestellt werden.

5.1 ADO .NET - Table Module und Repository

ADO (*Access Data Objects*) ist eine von Microsoft entwickelte Datenzugriffstechnologie, die hauptsächlich auf Microsoft kompatiblen Plattformen verwendet wird. Mit Aufkommen der .NET Technologie wurde die ADO Technik überarbeitet und bietet in der Version ADO .NET ([7]) einen enormen Funktionsumfang.

Im Grundsatz allerdings ist ADO .NET eine Implementierung des Table Module Musters in Verbindung mit einem Table Data Gateway. Im Zentrum des Datenzugriffs steht die *DataSet* Klasse. Sie bietet eine tabellarischen Darstellung der Daten an. Dazu werden verschiedene *DataTable* Klassen in einem DataSet gekapselt und über *DataRelation* Klassen werden die Tabellen verknüpft. Die Tabellen können auch hierarchisch angeordnet werden.

Ein DataTable Objekt ist die Repräsentation einer Datenquelle ohne Information über Herkunft der Daten. Die DataTable Klasse stellt im Wesentlichen ein Table Module zur Manipulation der Daten zur Verfügung. Die tabellarische Darstellung wird über *DataColumn* Objekte und *DataRows* Objekte verwaltet. Erstere dienen der Gliederung der Tabelle, Letztere speichern die eigentlichen Daten. *Constraints* dienen dazu Einschränkungen auf den Daten, z.B. Primärschlüsseleigenschaften, zu verwirklichen.

Der eigentliche Datenzugriff wird über sogenannte *Managed Provider* übernommen. Diese bieten den Zugriff auf verschiedene Datenquellen. Neben Datenbanken können so auch XML Dateien unterstützt werden und in eineinander umgewandelt werden. Dabei können auch offline Daten in einem DataSet verändert werden. Die Verbindung zum DataProvider wird nach dem Einlesen der Daten geschlossen.

Ein DataSet wirkt dabei wie ein Cache oder eine im Speicher gehaltene gemischt hierarchisch-relationale Datenbank und stellt somit auch eine Variation eines Repositories dar.

ADO .NET bietet einen mächtigen Funktionsumfang und integriert sich nahtlos in die restlichen .NET Platform (z.B. lassen sich Präsentationssteuerelement direkt an eine Datenquelle anbinden).

5.2 Hibernate - Domain Model und Metadata Mapping

Hibernate ([6]) ist ein Open-Source Persistenzwerkzeug für die Programmiersprache Java ([8]), das sich großer Beliebtheit in der Entwicklergemeinde erfreut. Es bietet einen großen Funktionsumfang bei der Abbildung von Objekten und Relationen, stellt aber gleichzeitig eine einfache Schnittstelle zur Verfügung.

Hibernate ist ein Beispiel für Werkzeuge, die eine Vielzahl der vorgestellten Muster implementieren. Daran erkennt man, dass eine reiche Persistenzschicht den Umfang einer eigenen Anwendung annehmen kann.

Der Kern von Hibernate stellt ein auf Metadata Mapping (3.5.1) basierendes Prinzip dar. Metadaten werden für einfache Javaobjekte (*Plain Old Java Objects*, *POJOs*) erstellt, die meist das Domain Model (3.1.2) der Anwendung darstellen. Dabei wird eine XML Syntax verwendet. Mit Java 1.5 können die Metadaten per Annotationen direkt in dem jeweiligen POJO notiert werden.

In den Metadaten werden verschiedenen Abbildungsarten der Objekte unterstützt. Optional können die POJOs über ein Identity Field (3.4.1) verfügen. Hibernate bietet in diesem Fall unterschiedliche Algorithmen zur Generierung neuer Schlüssel an. Desweiteren gibt es auch die Möglichkeit Assoziationen abzubilden. Einfache Assoziationen (1:1, 1:n) nutzen hierfür ein Foreign Key Mapping (3.4.2), Mehrfachassoziationen nutzen ein Association Table Mapping (3.4.3). Eng damit verbunden ist die Abbildung von Mengen von Objekten. Hibernate bietet hier Metadatenelemente zur Abbildung von Listen, Mengen, Hashmaps, Bags und Arrays.

Auch die Abbildung von Vererbungsstrukturen wird unterstützt. Hierbei kann der Benutzer zwischen den Strategien Single Table Inheritance, Class Table Inheritance und Concrete Table Inheritance (3.4.7) wählen. Bei den letzten beiden Strategien wird von Fremdschlüsselbeziehungen und Joins Gebrauch gemacht, um die Daten für ein Objekt zu erhalten. Auch die gemischte Verwendung der Strategien zur Abbildung von verschiedenen Zweigen eines Hierarchiebaumes wird unterstützt.

Zur Abbildung der Objekten auf Tabellen, werden durch die Metadaten automatisch Data Mappers (3.2.3) erzeugt, welche die Abbildung vornehmen.

Neu erstellte oder veränderte Objekte werden einem Repository (3.5.3) zur Speicherung übergeben. Auch die Suche nach Objekten, die sowohl über natives SQL als auch über Query Objects (3.5.2) geschehen kann, wird über das Repository (3.5.3) abgewickelt. Die Ergebnisse können in unterschiedlichen Formaten betrachtet werden.

Beim Laden der Objekte kann zur Perfomancesteigerung ein Lazy Load (3.3.3) ver-

wendet werden. Auch der Einsatz eines Caches, einer fortgeschrittenen Variante einer Identity Map (3.3.2), zum Zwischenspeichern häufig verwendeter Objekte, ist vorgesehen.

Hibernate ist ein sehr mächtiges Werkzeug, das seinem Benutzer eine umfangreiche Funktionalität bei der Abbildung von Objekten und Relationen zur Verfügung stellt. Obwohl vieles in Hibernate automatisch geschieht, muss einiges an die jeweilige Anwendung angepasst werden. Hierfür bietet Hibernate umfangreiche Konfigurationsmöglichkeiten und Werkzeugunterstützung an. Allerdings sollte dieser Aufwand mit dem Umfang der Erstellung einer einfachen Persistenzschicht einer kleinen Applikation je nach Anwendungsfall verglichen werden.

6 Fazit und Ausblick

Die persistente Speicherung von Objekten gehört zu einer der wichtigsten Aufgaben von betrieblichen Anwendungen. Die Aufgabe wird umso schwieriger, wenn sich die Strukturen der meist relationalen Datenquelle von denen der Objektorientierung unterscheiden.

Zu Überbrückung dieser Diskrepanz wurden verschiedene Lösungsstrategien entwickelt und in Muster dokumentiert. Die Muster vereinfachen die Kommunikation zwischen Objekten und nicht objektorientierten Datenquellen und sind für unterschiedlich komplexe Anwendungsszenarien einsetzbar.

XML als plattformunabhängiges Datenaustauschformat bietet eine weitere Vereinfachung der Kommunikation zwischen Geschäftslogik und Datenhaltung. Während allerdings XML in der Anwendungsschicht bereits fest integriert ist, bieten nur wenige Datenbankhersteller geeignete XML-Erweiterungen an. Bei der Verwendung von XML sollte der zusätzliche Platzbedarf und Performanzverlust gegenüber einem eventuell datenbankabhängigem Binärformat berücksichtigt werden.

Obwohl relationale Datenbanken heutzutage zu den meistgenutzten, weil meisterforschten, Datenquellen gehören, existieren auch Objekt-Relationale und Objekt-Orientierte Datenbanken, welche die Problematik entschärfen könnten. Allerdings sind sie noch nicht lange im Einsatz. Da jedoch die Daten der Unternehmen die Ertragsgrundlage darstellen, werden hier meist keine Risiken eingegangen.

Bei der Implementierung der Muster reduzieren neue Technologien den Aufwand der Codierung. So bietet etwa die aspektorientierte Progammierung Möglichkeiten, sich wiederholenden, querschnittlichen Code an einer Stelle zu kapseln und zur Kompilierungszeit an verschiedenen Stellen in die Anwendung einzubinden. Auch die Unterstützung von generischen Datentypen in verschiedenen Programmiersprachen kann die Entwicklung der Muster vereinfachen. Als Beispiel sei hier eine generische Identity Map (3.3.2) angeführt.

Bei der Entwicklung einer betrieblichen Anwendung sollte auch immer der Einsatz eines bestehenden Persistenz-Frameworks berücksichtigt werden. Mit dem Preis variiert die Funktionalität der verfügbaren Werkzeuge, obwohl auch leistungsfähige Open-Source Werkzeuge existieren. Häufig ist ein großer Aufwand nötig, um dieselbe Funktionalität manuell nachzuprogrammieren. Zudem sind die Werkzeuge meist in anderen Anwendungen integriert und somit bereits im Einsatz erprobt.

Literatur

[1] C. Alexander, S. Ishikawa, M. Silverstein, M. Jacobson, I. Fiksdahl-King, and S. Angel, *A Pattern Language*. Oxford University Press, 1977.

[2] K. Beck, *Smalltalk Best Practice Patterns*. Prentice Hall, 1997.

[3] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal, *Pattern Oriented Software Architecture*. Wiley, 1996(3).

[4] M. Fowler, *Patterns of Enterprise Application Architecture*, 9th ed. Addison Wesley, 2003.

[5] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns, Elements of Reusable Object Oriented Software*. Addison Wesley, 1995.

[6] Hibernate, "Hibernate Reference Documentation," Hibernate Website, April 2006, http://www.hibernate.org.

[7] Microsoft, "Datenzugriff mit ADO .NET," Microsoft MSDN Library, April 2006, http://www.microsoft.com/germany/msdn/library/net/adonet.

[8] S. Microsystems, "JAVA Tutorial," SUN Java Website, Stand April 2006, April 2006, http://java.sun.com.

Teil II.

Komponentenmodelle

Das Fractal Komponentenmodell

Ihssane El-Oudghiri

Betreuer: Steffen Becker

Zusammenfassung

In der Welt der komponentenbasierten Software-Entwicklung steht der Begriff „Komponente“ im Mittelpunkt. Trotz seiner höhren Verwendung gibt es für diesen Begriff keine allgemeingültige Definition. Aus diesem Grund kann man die Wichtigkeit von Komponentenmodellen erkennen, da sie nicht nur das nötige Framework anbieten um Komponenten überhaupt zu definieren, sondern auch sie spezifizieren wie man Komponente erstellt, wie man sie miteinander verknüpft, und wie man zusammengesetzte Komponente baut.

Diese Arbeit liefert einen kurzen Überblick über Komponentenmodelle insbesondere das Fractal Komponentenmodell. Es wird vor allem das abstrakte Modell und die Spezifikation von Fractal vorgestellt, danach wird an Hand eines Beispiels das Programmieren mit Fractal erläutert. Schließlich werden ein paar konkrete Fractal Implementierungen und Werkzeuge, die auch zum Fractal-Projekt gehören vorgestellt.

1 Einleitung

1.1 Was ist ein Komponentenmodell ?

Ein Komponentenmodell genau zu definieren ist eine sehr schwierige Aufgabe, statt dessen könnte man einfach die wichtigen Aspekte, die ein Komponentenmodell festlegen kann, betrachten und erklären. Hauptsächlich wird bei einem Komponentenmodell folgendes definiert [1]:

1.1.1 Die Syntax

Durch die Definition einer Syntax, ist man in der Lage Komponente zu repräsentieren. Außerdem ist es möglich die Art und Weise zu beschreiben, wie Komponente konstruiert werden,

1.1.2 Semantik

Allgemein könnte man eine Komponente als eine Software-Einheit betrachten, die aus einem Namen, einer Schnittstelle und dem Quelltext besteht. Im Quellcode sind die Implementierungen der angebotenen Operationen enthalten.Diese dürfen von außen nicht zugänglich oder sichtbar sein. Die Schnittstelle bietet die einzige Möglichkeit mit der Komponente von außen zu kommunizieren, deshalb sollte sie alle Informationen zur Verfügung stellen, die notwendig sind, um die Komponente zu benutzen. Insbesondere sollte sie die Dienstleistungen spezifizieren, die durch die Komponente erfordert werden, um die Dienstleistungen anzubieten, die sie zur Verfügung stellt.

1.1.3 Die Komposition

Die Komposition von Komponenten ist eine wichtiger Aspekt bei Komponentenmodellen. Es wird vorallem definiert, wie Komponenten erstellt werden und wie sie untereinander zusammengesetzt sind. Außerdem werden die Verbindungen zwischen den Komponenten spezifiziert.

1.2 Das Komponentenmodell Fractal

Fractal [2] ist ein modulares und erweiterbares Komponentenmodell, das mit verschiedenen Programmiersprachen verwendet werden kann, um verschiedene Systeme und Applikationen (Betriebssystemen, Middelware-Plattformen, GUI..) zu entwerfen, implementieren, konfigurieren und zu rekonfigurieren. Es wurde bei der France Telecom entwickelt und hat inzwischen mehrere konkrete Implementierungen in verschiedenen Sprachen (z.B. Julia: eine Java Implementierung [3]). Was ist genau Fractal, seine Spezifikation und Werkzeuge werden in dieser Arbeit erfahren.

2 Fractal Spezifikation

Fractal ist nicht als eine große festgelegte Spezifikation zu betrachten, der alle Komponenten folgen müssen, sondern als ein erweiterbares System von Konzepten und die dazugehörigen APIs, die Fractal-Komponenten implementieren oder auch nicht. Das hängt von der Dienstleistungen ab, die die Komponente zur Verfügung stellt. Die Spezifikation von Komponenten ist in so genannten Kontrollebenen (Level of control) unterteilt. Im folgenden Abschnitt wird auf die Ebenen detaillierter eingegangen.

2.1 Basis

In der untersten Kontrollebene, wird eine Komponente als eine einfache ausführbare Einheit betrachtet, die keine Introspektion-Funktionalität für andere Komponenten bereitstellt. Eine solche Komponente wird als **elementare Komponente** bezeichnet und könnte als ein Objekt angesehen werden.
Bevor man Operationen von einer Komponentschnittstelle aufrufen kann, muß zuerst die Schnittstelle identifizieren. In diesem Zusammenhang werden drei Punkte betrachtet:

- Der Name: Eine Komponentenschnittstelle wird durch einen Namen eindeutig bestimmt, dabei kann der Name mehrere Formen annehmen (z.B. eine Java Referenz) und gibt nicht unbedingt den direkten Zugang zur Schnittstelle. Er kann auch serialisiert werden (z.B. eine Java Referenz wird durch String repräsentiert, das die Speicheraddresse des Objekts enthält) mit Hilfe der *encode* Methode. Die encode Methode und eine weitere Methode, nämlich die *getNamingContext* Methode, sind in der Schnittstelle *Name* definiert (siehe Listing 1).

- Der Namenskontext: Ein Name ist immer mit einem Kontext assoziiert und er ist meistens außerhalb dieses Kontextes nicht mehr gültig. Ein Namenskontext ist durch die Schnittstelle *NamingContext* repräsentiert und übernimmt die Aufgabe Namen zu erstellen und zu verwalten. In *NamingContext* (siehe Listing 1) sind zwei Methoden definiert. Eine erstellt Namen für die Schnittstellen: *export*. Die andere ist für die Deserialisierung zuständig ist: *decode*

- Der Binder: Um den Zugang zur einer Schnittstelle, die durch einen Namen bestimmt ist, zu ermöglichen, muß man vorher eine Bindung zu dieser Schnittstelle etablieren. Das geschieht mit Hilfe eines Binders. Die Funktionen des Binders sind durch die Schnittstelle *Binder* repräsentiert (siehe Listing 1).

Listing 1: Naming API.

```
package org.objectweb.naming;
interface Name {
   NamingContext getNamingContext ();
   byte[] encode () throws NamingException;
}
interface NamingContext {
   Name export (any o, any hints) throws NamingException;
```

```
    Name decode (byte[] b) throws NamingException;
}
interface Binder extends NamingContext {
    any bind (Name n, any hints) throws NamingException;
}
```

2.2 Introspektion

Wenn man eine Fractal-Komponente von außen ansieht und ohne ihre innere Organisation zu betrachten, dann sieht man nur eine Menge von Zugangspunkten zu dieser Komponente, genannt externe Schnittstellen, die wiederum in zwei Arten unterteilt sind. Eine Client-Schnittstelle um Operationsaufrufe zu senden und eine Server-Schnittstelle um sie abzufangen.
In dieser Kontrollebene, die man als Introspektionsebene bezeichnet, kann ein Fractal-Komponente eine Standardschnittstelle zur Verfügung stellen, die es ermöglicht, alle ihre externen Schnittstellen zu ermitteln. Dafür sind zwei Schnittstellen vorgesehen: Die Schnittstelle *Component*, um die Menge alle Schnittstellen bzw. eine bestimmte Schnittstelle zu bekommen, und die Schnittstelle *Interface*, um die Schnittstelle selbst zu „introspektieren“(siehe Listing: 2).

Listing 2: Komponent Introspektion API.

```
package org.objectweb.fractal.api;
interface Component {
    Object[] getFcInterfaces ();
    Object getFcInterface (String itfName) throws
       NoSuchInterfaceException;
    Type getFcType ();
}

interface Type {
    boolean isFcSubTypeOf (Type t);
{

interface Interface {
    string getFcItfName ();
    Type getFcItfType ();
    Component getFcItfOwner ();
    boolean isFcInternalItf ();
}
```

In der Schnittstelle *Component* sind zwei Methoden definiert: *getFcInterfaces* und *getFcInterface*. Die erste Methode ist parameterlos und gibt ein Array, das alle externen Schnittstellen der Komponente enthält, zurück. Die zweite nimmt einen Namen einer Schnittstelle als Parameter und gibt diese Schnittstelle, wenn sie existiert, zurück.
Diese beiden Methoden liefern meistens nur eine Referenz einer Schnittstelle zurück,

aus diesem Grund ist in der Schnittstelle *Interface* weitere Methoden definiert, um mehr Informationen über die Schnittstelle zu erfahren. Die Methode *getFcItfName ()* z.B. gibt den Namen der Schnittstelle zurück.

2.3 Konfiguration

Intern gesehen besteht eine Komponente aus zwei Teilen, dem Controller und dem Inhalt. Der Inhalt kann wiederum aus anderen Komponenten bestehen, die Unter-Komponenten (*Sub-Component*) genannt werden. Im Fractal-Modell wird eine Komponente, die Unter-Komponenten besitzt, zusammengesetzte Komponente (*Composite Component*) genannt. Außerdem bezeichnet man eine Komponente als *primitiv*, wenn sie keine andere enthält und mindestens eine Kontroll-Schnittstelle implementiert.
Es ist auch ganz wichtig, zwischen externen und internen Schnittstellen eines Controllers zu unterscheiden. Während die externen Schnittstellen von außen zugänglich sind, sind die internen Schnittstellen nur innerhalb der Komponente zugänglich. Außerdem übt der Controller Kontrolle über die Komponente aus, d.h. er kann Dienste weiterleiten oder blockieren. Fractal definiert eine Reihe von Kontroll-Schnittstellen, die eine Komponente implementieren kann. Im Folgenden werden die einzelnen Schnittstellen erklärt.

2.3.1 Attribut-Controller

Ein Attribut ist eine konfigurierbare Eigenschaft einer Komponente, die meistens einen primitiven Typ hat. Sie wird benutzt um den Zustand der Komponente zu konfigurieren. In Fractal kann eine Komponente die Schnittstelle *AttributeController* (siehe Listing: 3) zur Verfügung stellen, damit seine Attribute von außen gelesen und geändert werden können.

Listing 3: Schnittstelle AttributeController.

```
package org.objectweb.fractal.api.control;

interface AttributeController { }
```

Wie man sieht ist die Schnittstelle *AttributeController* leer, deshalb muß die Komponente eine Unter-Schnittstelle von AttributeController implementieren, die mindestens eine getter und/oder eine setter-Methode für jedes Attribut enthält. Zum Beispiel will eine Komponente eine AttributeController Schnittstelle anbieten, um das String Attribut *foo* zu lesen und zu schreiben, muß sie eine Unter-Schnittstelle von AttributeController zur Verfügung stellen, die die zwei Methoden: String getFoo() und setFoo(String s) enthält.

2.3.2 Bindung-Controller

Eine Komponente kann die Schnittstelle *BindingController* bereitstellen, um ihre Client-Schnittstellen mit anderen Komponenten entweder zu binden oder zu lösen (siehe Listing: 4).

Listing 4: Schnittstelle BindingController.

```
package org.objectweb.fractal.api.control;

interface BindingController {
   string[] listFc ();
   Object lookupFc (string clientItfName)
      throws NoSuchInterfaceException;
   void bindFc (string clientItfName, Object serverItf)
      throws NoSuchInterfaceException,
         IllegalBindingException, IllegalLifeCycleException
         ;
   void unbindFc (string clientItfName)
      throws NoSuchInterfaceException,
         IllegalBindingException, IllegalLifeCycleException
         ;
}
```

Die Schnittstelle BindingController enthält die folgenden Methoden:

- *listFc*: gibt die Namen aller Client-Schnittstellen der Komponente zurück.
- *lookupFc*: nimmt der Name einer Client-Schnittstelle der Komponente als Parameter, und gibt die Server-Schnittstelle, die mit dieser Schnittstelle gebunden ist zurück, oder *null* falls sie nicht existiert.
- *bindFc*: nimmt der Name einer Client-Schnittstelle der Komponente und eine Server-Schnittstelle als Parameter, und bindet die beiden Schnittstellen zusammen.
- *unbindFc*: nimmt der Name einer Client-Schnittstelle der Komponente als Parameter, und löst diese Schnittstelle.

2.3.3 Inhalts-Controller

Um die Manipulation von Unterkomponenten zu ermöglichen, d.h. Unterkomponenten hinzufügen oder löschen, kann eine Komponente die Schnittstelle *ContentController* implementieren (siehe Listing: 5).

Listing 5: Schnittstelle ContentController.

```
package org.objectweb.fractal.api.control;
interface ContentController {
   Object[] getFcInternalInterfaces ();
   Object getFcInternalInterface (string tfName) throws
      NoSuchInterfaceException;
   Component[] getFcSubComponents ();
   void addFcSubComponent (Component c)
      throws IllegalContentException,
         IllegalLifeCycleException;
   void removeFcSubComponent (Component c)
```

```
        throws IllegalContentException,
           IllegaLifeCycleException;
}
```

Die Schnittstelle definiert vor allem die folgenden Methoden:

- *getFcSubComponents*: liefert eine Liste von Unterkomponenten, die die Komponente enthält.
- *addFcSubComponent*: nimmt eine Referenz vom Typ *Component* als Parameter, und fügt diese Komponente zum Inhalt hinzu.
- *removeFcSubComponent*: nimmt auch eine Referenz vom Typ *Component* als Parameter, und entfernt diese Komponente vom Inhalt.

Es könnte passieren, daß eine Komponente zu mehreren Komponenten hinzugefügt wird. Eine solche Komponente nennt man *Shared Component*.

2.3.4 Lebenszyklus-Controller

Während der Ausführungszeit kann eine Änderung eines Attributs, einer Bindung oder die Entfernung von Unterkomponenten zu Problemen führen. Z.B. können Daten verloren gehen oder die Applikation in einen inkonsistenten Zustand übergehen oder sogar abstürzen. Um die Implementierung solchen dynamischen Rekonfigurationen zu ermöglichen, kann eine Komponente die Schnittstelle *LifeCycleController* anbieten (siehe Listing: 6).

Listing 6: Schnittstelle LifeCycleController.

```
package org.objectweb.fractal.api.control;
interface LifeCycleController {
   string getFcState ();
   void startFc () throws IllegalLifeCycleException;
   void stopFc () throws IllegalLifeCycleException;
}
```

Die Schnittstelle *LifeCycleController* definiert vorallem die zwei Methoden: startFc und stopsFc. Wie der Name schon sagt, sind für das Starten und das Anhalten von Komponenten zuständig. Jedoch ist nicht spezifiziert was sie genau tun.

2.4 Instanzierung

Das im letzten Abschnitt präsentierte Framework gibt uns die Möglichkeit schon existierende Komponenten zu verwenden,zu konfigurieren und zu rekonfigurieren. Der folgende Abschnitt beschäftigt sich mit der Erzeugung von Komponenten.

2.4.1 Fabriken (Factories)

Für die Instanzierung neuer Komponenten, sind die so genannten Fabrik-Komponenten zuständig. Im Fractal Modell unterscheidet man zwischen generischen Fabrik-Komponenten,

die verschiedene Sorten von Komponenten erzeugen können, und Standard Fabrik-Komponenten, die nur eine einzige Sorte von Komponenten erzeugen können. Generische und Standard Fabrik-Komponenten können die Schnittstelle *GenericFactory* bzw. *Factory* bereitstellen (siehe Listing: 7).

Listing 7: Die Schnittstellen GenericFactory und Factory.

```
package org.objectweb.fractal.api.factory;
interface GenericFactory {
   Component newFcInstance (Type t, Object controllerDesc,
      Object contentDesc)
       throws InstantiationException;
}
interface Factory {
   Type getFcInstanceType ();
   Object getFcControllerDesc ();
   Object getFcContentDesc ();
   Component newFcInstance () throws InstantiationException
      ;
}
```

Die Schnittstelle *GenericFactory* definiert nur eine einzige Methode nämlich *newFcInstance*. Sie nimmt den Typ, eine Beschreibung des Controllers und eine Beschreibung des Inhalts der Komponente, die mann erstellen möchte, als Parameter, und gibt ihre *Component* Schnittstelle zurück. Die gleiche Methode ist auch in der Schnittstelle *Factory* zu sehen, aber ohne Parameter. Das bedeutet, daß man hier nur Komponenten erstellen kann, die den gleichen Typ, die gleiche Controllerbeschreibung und die gleiche Inhaltbeschreibung haben. Diese Informationen kann man mit der Methoden: *getFcInstanceType*, *getFcControllerDesc*, *getFcContentDesc* abfragen.

2.4.2 Vorlagen (Templates)

Eine Vorlage ist eine speziele Sorte von Standard Fabrik-Komponenten, die eine quasi zu ihr isomorphe Komponente erstellt. Genauer gesagt, eine von einer Vorlage erstellte Komponente muß die gleichen funktionalen Client und Server-Schnittstellen (außer die Factory Schnittstelle) haben wie die Vorlage, aber kann beliebige Kontroll-Schnittstellen besitzen.

Links in der Abbildung 1 steht die Vorlage-Komponente und rechts eine mögliche

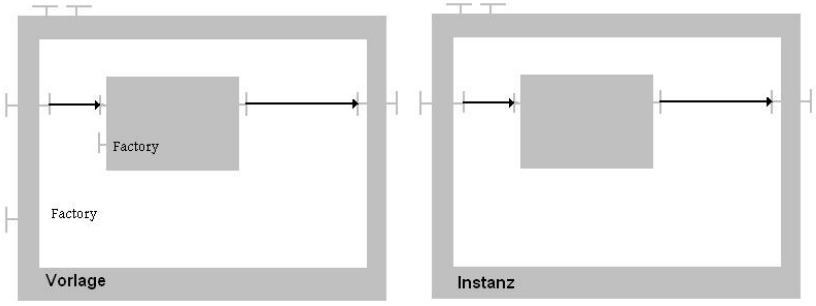


Abbildung 1: Darstellung einer Vorlage und ihrer Instanz.

Instanz dieser Vorlage. Wie man auch leicht bemerkt, besitzt die Instanz alle funktionellen Schnittstellen der Vorlage, wohl aber nicht die *Factory* Schnittstellen.

2.4.3 Bootstrap

Bis jetzt ist die einzige Möglichkeit Komponenten zu erstellen, ist die Benutzung von Fabrik-Komponenten. Es stellt sich die Frage, wie sind überhaupt diese Fabriken entstanden? Eine mögliche Antwort wäre: aus anderen Fabriken, aber diese Lösung führt zu einer unendlichen Rekursion. Um dieses Problem zu lösen, sind Bootstrap-Komponenten, die eine explizite Erstellung nicht brauchen, notwendig. Außerdem müsen sie in der Lage sein, alle Sorten von Komponenten, darunter Fabrik-Komponenten, zu erstellen.
In Fractal wird die Methode *getBootstrapComponent* bereitgestellt, um die Bootstrap-Komponente zu bekommen. In Java z.B. diese Methode ist static, hat keine Parameter und befindet sich in der Klasse *Fractal*.

2.5 Typisierung

In diesem Abschnitt beschäftigen wir uns mit der Definition eines einfachen Typsystems für Komponenten und Schnittstellen. Als erstes führen wir neue Begriffe ein, die wir später brauchen : Contingency und Kardinalität.

- **Die Contingency** einer Schnittstelle deutet an, ob man garantieren kann, daß die Funktionalität dieser Schnittstelle während der Ausführung der Komponente verfügbar ist oder nicht. Mit anderen Worten, es wird festgestellt, ob diese Schnittstelle optional ist oder nicht.

- **Die Kardinalität** eines Schnittstellentyps **T** bestimmt die Anzahl der Schnittstellen vom Typ **T**, die eine Komponente haben kann. *Singleton* bedeutet, daß die Komponente genau eine Schnittstelle vom Typ **T** besitzen muß. Im Gegensatz dazu, bedeutet *Collection*, daß die Komponente eine beliebige Anzahl von Schnittstellen vom Typ **T** haben kann. Collection-Schnittstellen sind nützlich für Komponenten, die eine Variable Anzahl von Komponenten gleicher Typ brauchen.

In Fractal unterscheidet man zwischen Komponententyp und Schnittstellentyp. Während Komponententyp einfach als eine Menge von Schnittstellentypen definiert ist, ist Schnittstellentyp aus einem Namen , Signatur, Rolle, Contingency und Kardinalität zusammengesetzt. Komponententyp und Schnittstellentyp werden durch die Schnittstellen *ComponentType* bzw. *InterfaceType* repräsentiert (Listing 8). Komponenten und SchnittstellenTypen können mit Hilfe einer Typfabrik erstellt werden, die durch die Schnittstelle *TypeFactory* repräsentiert (siehe Listing: 8).

Listing 8: Die Schnittstellen ComponentType InterfaceType und TypeFactory.

```
package org.objectweb.fractal.api.type;
interface ComponentType extends Type {
   InterfaceType[] getFcInterfaceTypes ();
```

```
    InterfaceType getFcInterfaceType (string itfName) throws
        NoSuchInterfaceException;
}
interface InterfaceType extends Type {
    string getFcItfName ();
    string getFcItfSignature ();
    boolean isFcClientItf ();
    boolean isFcOptionalItf ();
    boolean isFcCollectionItf ();
}
interface TypeFactory {
    InterfaceType createFcItfType (string name, string
       signature, boolean isClient,
       boolean isOptional, boolean isCollection) throws
          InstantiationException;
    ComponentType createFcType (InterfaceType[] itfTypes)
       throws InstantiationException;
}
```

Das Erstellen von Schnittstellen- bzw Komponententypen geschieht mit Hilfe der in der Schnittstelle *TypeFactory* definierten Methoden *createFcItfType* und *createFcType*. Die *createFcItfType* Methode werden folgende Parameter übergeben: Namen, Signatur (z.B. ein Java Interface) und drei boolsche Werte. Der erste Wert bestimmt die Rolle der Schnittstelle (True bedeutet Client). Der zweite bestimmt die Contingency der Schnittstelle. Schließlich bestimmt der dritte Wert die Kardinalität der Schnittstelle (True bedeutet Collection).

3 Programmieren

Dieser Abschnitt zeigt, wie das API, das in der Fractal Spezifikation definiert wird, verwendet werden kann, um Komponenten zu erzeugen, zusammenzubauen und zu rekonfigurieren. Das hier verwendete Beispiel ist eine einfache Applikation bestehend aus zwei primitiven Komponenten, die sich innerhalb einer zusammengesetzten Komponente befinden (siehe Abbildung 2). Die erste Komponente ist eine Server-Komponente, die eine Schnittstelle zur Verfügung stellt, um Meldungen auf der Konsole auszudrucken. Die zweite Komponente ist eine Client-Komponente, die diese Schnittstelle verwendet, um Meldungen auszudrucken.

Was Kontroll-Schnittstellen angeht, implementieren alle drei Komponenten die Schnittstelle *Component* (wird mit **C** dargestellt). Die Schnittstelle *BindingController* wird nur von den Haupt- und Server-Komponenten implementiert (wird mit **BC** dargestellt). Schließlich implementiert die Hauptkomponente noch die Schnittstelle *ContentController*, um ihre Unterkomponenten zu verwalten (wird mit **CC** dargestellt).

Zusätzlich verfügt die Server-Komponente über die Server-Schnittstelle „s“ vom Typ **S**. Diese Schnittstelle enthält die Methode *print*. Außerdem verfügt die Client-Schnittstelle über eine Server-Schnittstelle „m“ vom Typ **M**, die die Methode *main* enthält, und über eine Client-Schnittstelle „s“ vom Typ **S**. Die *main* Methode wird aufgerufen, wenn man die Applikation startet.

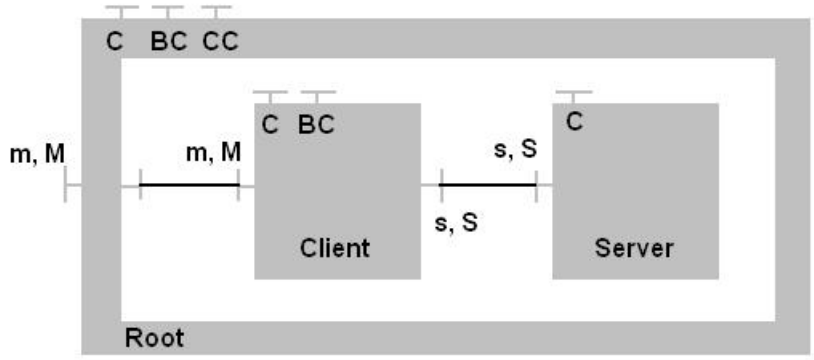


Abbildung 2: Eine einfache komponentenbasierte Applikation.

Die Schnittstellen **S** und **M**, sowie eine Mögliche Implementierung der Client und Server-Komponentenklassen *ClientImpl* bzw *ServerImpl* finden sie im Anhang 7.

3.1 Instanzierung

Der erste Schritt um die Komponenten zu instanzieren, ist die Erzeugung der Komponenten und der Komponententypen. Dafür brauchen wir die Bootstrap-Komponente und die Typ-Fabrik.

```
Component boot = Fractal.getBootstrapComponent();
TypeFactory tf = (TypeFactory)boot.getFcInterface("type-factory");
```

Jetzt können wir mit Hilfe der TypeFactory-Instanz *tf* den Typ der Hauptkomponente erstellen. Dafür rufen wir die Methode *createFcType* auf. Weil diese Methode ein Array von Schnittstellentypen nimmt, müssen wir zuerst die Typen der Schnittstellen der Komponente erstellen. Da diese Komponente nur eine Client-Schnittstellen hat, übergeben wir die Methode *createFcType* ein Array bestehend aus einem Element. Die Methode *createFcItfType* wird verwendet, um Schnittstellentypen zu erstellen. In diesem Fall werden folgenden Parameter die Methode übergeben:

- „m“: Der Name der Schnittstelle.
- „M“: Die Java Signatur der Schnittstelle.
- false: Diese Schnittstelle ist eine Server-Schnittstelle.
- false: Diese Schnittstelle ist nicht optional, d.h sie ist immer verfügbar.
- false: Diese Schnittstelle ist *singetlon*.

```
ComponentType rType = tf.createFcType(new InterfaceType[] {
   tf.createFcItfType("m", "M", false, false, false)
});
```

Analog, können wir den Typ der Client-Komponente erstellen:

```
ComponentType cType = tf.createFcType(new InterfaceType[] {
   tf.createFcItfType("m", "M", false, false, false),
   tf.createFcItfType("s", "S", true, false, false)
});
```

und auch der Server-Komponente:

```
ComponentType sType = tf.createFcType(new InterfaceType[] {
   tf.createFcItfType("s", "S", false, false, false)
});
```

Zwar können wir die Komponente jetzt direkt erzeugen, aber wir wollen in diesem Beispiel Vorlagen verwenden. Deshalb erzeugen wir als nächstes die Vorlagen. Dazu brauchen wir zu nächst eine *GenericFactory* Instanz:

```
GenericFactory gf = (GenericFactory)boot.getFcInterface("generic-factory");
```

Mit der Methode *newFcInstance* ist die Erstellung der Vorlagen möglich. Fangen wir mit der Vorlage der Hauptkomponente an:

```
Component rTmpl = gf.newFcInstance(
   rType, "compositeTemplate", new Object[] {"composite", null});
```

Die Erstellung der Vorlage für die Client- und Server-Komponente kann analog erfolgen. Der kleine Unterschied ist, daß hier die Komponenten Inhalt besitzt und zwar die Klassen *ClientImpl* bzw *ServerImpl*, die in diesen Komponenten gekapselt sind:

```
Component cTmpl = gf.newFcInstance(
   cType, "template", new Object[] {"primitive", "ClientImpl"});

Component sTmpl = gf.newFcInstance(
   sType, "template", new Object[] {"primitive", "ServerImpl"});
```

Hier, beschreibt der Descriptor „template“ bzw. „compositeTemplate„ eine Komponente mit einer *BindingController* Schnittstelle bzw eine Komponente mit einer *BindingController* Schnittstelle und einer *ContentController* Schnittstelle. Der „primitiv“ und „composite“ Descriptoren beschreiben die gleichen Komponenten wie eben, aber mit einer zusätzlichen *LifeCycleController* Schnittstelle.
Jetzt können wir entweder jede Vorlage alleine instanzieren, die entstanden primitiven Komponenten innerhalb der zusammengesetzten Komponente einbauen und schließlich die Komponenten verbinden, oder die primitiven Vorlagen innerhalb der zusammengesetzten Vorlage setzen, die beiden miteinander verbinden, danach die gesamte zusammengesetzte Vorlage instanzieren. Die letzte Möglichkeit ist hier verwendet.
Mit dem Inhalt-Controller fügt man die beiden primitiven Vorlagen zur Hauptkomponente hinzu:

```
ContentController cc = (ContentController)rTmpl
    .getFcInterface("content-controller");
cc.addFcSubComponent(cTmpl);
cc.addFcSubComponent(sTmpl);
```

Mit dem Bindung-Controller der Hauptvorlage verbindet man jetzt die Schnittstelle „m“ der Hauptvorlage mit der Schnittstelle „m“ der Client-Vorlage:

```
((BindingController)rTmpl.getFcInterface("binding-controller"))
   .bindFc("m", cTmpl.getFcInterface("m"));
```

Analog verbindet man die Schnittstelle „s“ der Client-Vorlage mit der Schnittstelle „s“ der Server-Vorlage:

```
((BindingController)cTmpl.getFcInterface("binding-controller"))
   .bindFc("s", sTmpl.getFcInterface("s"));
```

Am Ende wird die Hauptkomponente von der Hauptvorlage instanziert:

```
Component r = ((Factory)rTmpl.getFcInterface("factory"))
     .newFcInstance();
```

Schließlich kann man die Applikation starten. Das geschieht indem man einfach die *startFc* Methode der Hauptkomponente aufruft, die alle Komponenten startet, und die Methode *main*. Um die Applikation zu starten, muß man zuerst die *startFc* Methode des Lebenszyklus-Controllers der Hauptkomponente, die alle Komponenten startet, und danach die *main* Methode der Schnittstelle m der Hauptkomponente aufrufen:

```
((LifeCycleController)r.getFcInterface("lifecycle-controller"))
       .startFc();
((M)rComp.getFcInterface("m")).main(null);
```

3.2 Rekonfiguration

Angenommen wir wollen die Server-Komponente wechseln. Dafür müssen wir die Klient-Komponente lösen, die Server-Komponente entfernen, eine neue Server-Komponente erzeugen, sie zu der Hauptkomponente hinzufügen, und schließlich die Klient-Komponente mit der neuen Server-Komponente binden. Das Lösen und die Entfernung einer Komponente kann jedoch nicht während der Ausführungszeit erfolgen, deshalb müssen die Komponenten zuerst gestoppt, die Änderungen vornehmen und die Komponenten wieder gestartet werden.

```
((LifeCycleController)r.getFcInterface("lifecycle-controller"))
                       .stopFc();

Component c = ((Interface)((BindingController)r.
  getFcInterface("binding-controller")).lookupFc("m")).getFcItfOwner();
Component s = ((Interface)((BindingController)c.
  getFcInterface("binding-controller")).lookupFc("s")).getFcItfOwner();

((BindingController)c.getFcInterface("binding-controller"))
                     .unbindFc("s");
((ContentController)r.getFcInterface("content-controller"))
                     .removeFcSubComponent(s);
```

```
Component newS = gf.newFcInstance(sType, "primitive", "NewSImpl");

((ContentController)r.getFcInterface("content-controller"))
                     .addFcSubComponent(newS);
((BindingController)c.getFcInterface("binding-controller"))
                     .bindFc("s", newS);
((LifeCycleController)r.getFcInterface("lifecycle-controller"))
                     .startFc();
```

4 Implementierungen

Neben **Julia**, einer Java Implementierung des Fractal Komponentenmodells, existieren noch andere Projekte, die die Fractal Spezifikation in verschiedenen Sprachen implementieren, beispielsweise :

- **FracTalk** : eine Smalltalk Implementierung.
- FractNet : eine .Net Implementierung.
- Plasma : eine C++ Implementierung.
- THINK : eine C Implementierung.

Wie man sieht gibt es verschiedene Implementierungen des Fractal Komponentenmodells, aber in diesem Abschnitt konzentrieren wir uns nur auf das Julia Projekt von Objectweb [3].

4.1 Ziele von Julia

Mit der Julia Implementierung will man verschiedene Ziele erreichen. Man will voralem ein erweiterbares Framework erstellen, das hauptsächlich die Programmierung von Controllern ermöglicht. Außerdem sollen unterschiedliche Sorten von Kontrollobjekten angeboten werden, die ein Kontinuum zwischen statischer Konfiguration und dynamischer Rekonfiguration garantieren. Darüberhinaus sollte die Integration von Kontrollobjekten einen geringeren Einfluß auf den Zeit- und Speicheraufwand der Applikation haben. Schließlich soll das Framework auf jeder JVM / JDK (auch mit Beschränkungen, wie z.B. ohne ClassLoader Klasse, ohne Reflection API, ...) laufen können.

4.2 Verwendete Datenstrukturen in Julia

In Julia ist eine Fractal Komponente durch mehrere Java Objekte repräsentiert, die in drei Arten unterteilt sind (siehe Abbildung 3):

1. Objekte, die die Komponentenschnittstelle implementieren.
2. Objekte, die den Controller der Komponente implementieren.
3. Objekte, die den Inhalt der Komponente implementieren.

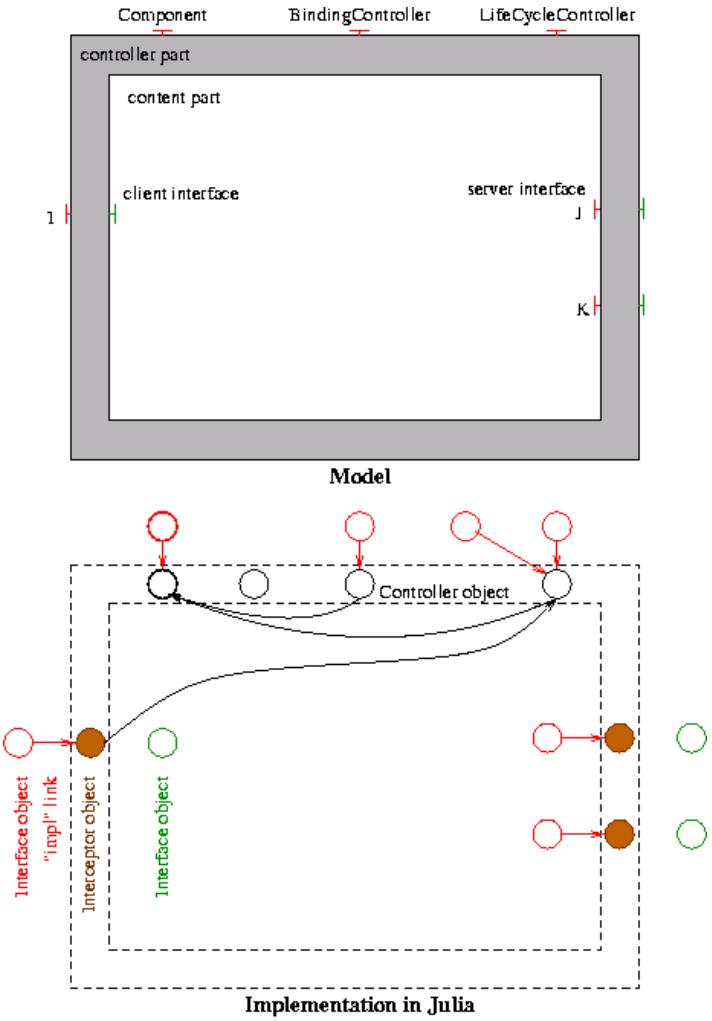


Abbildung 3: Eine abstrakte Komponente und ihre mögliche Implementation in Julia.

4.3 Julia Konfigurationsdatei

Bevor man überhaupt das Julia Framework startet, soll man zuerst eine Konfigurationsdatei [4] schreiben, die bei der Initialisierung von Julia mitberücksichtigt wird. In dieser Datei sollen die in der Methode *newFcInstance* verwendeten Aliase, genannt *Controller descriptors*, definiert werden (z.B. „template“, „compositeTemplate“ siehe Abschnitt 3.1). Dabei soll auch für jeden Alias jeweils eine GeneratorKlasse Schnittstelle (da es keine ClassLoader Klasse vorhanden ist), eine Menge von Kontrollschnittstellen Typen (welche Kontrollschnittstellen enthält dieser Controller), eine Menge von Klassen, die diese Schnittstellen implementieren (da es in Julia mehrere Implementierungen für die gleiche Kontrollschnittstelle gibt), eine Menge von Interceptoren und am Ende noch ein Optimierungsgrad definiert werden.
Schon die Konfigurationsdatei eines kleinen Programms, wie das HalloWorld-Beispiel, ist sehr lang und aufwändig. Deshalb wird hier auf die Julia-Tutorial-Seite [4] verwiesen, wenn man genau die einzelnen Bestanden der Konfigurationsdatei verstehen möchte. Beispiel gibt es auch dort.

5 Werkzeuge

5.1 Fractal-ADL Werkzeug

Die Fractal Architektur-Beschreibungssprache (ADL) ist eine erweiterbare Sprache, die Komponenten des Fractal Komponentenmodells definiert. Genauer gesagt, besteht Fractal ADL aus einem erweiterbaren Satz von ADL Modulen. Jedes Modul definiert eine abstrakte Syntax für einen gegebenen Architekturaspekt (wie Schnittstellen, Bindung, Attribute ...).
FractalADL ist ein Werkzeug bestehend aus mehreren Fractal Komponenten, das die Fractal Architektur-Beschreibung Sprache parsen kann und dabei die dazugehörige Komponenten instanziert. Siehe auch [5].

5.2 FractalGUI

FractalGUI ist eine Fractal Applikation, die aus Fractal Komponenten besteht, und eine graphische Bearbeitung und Konfiguration von Fractal-Komponenten anbietet [6].

6 Fazit

Zusammengefasst kann man sagen, dass das Fractal-Komponentenmodell ein erweiterbares System von Konzepten repräsentiert durch APIs ist, das zur Entwicklung, Implementierung, Konfiguration und Rekonfiguration von Applikationen entwickelt wurde. Im Fractal Modell besitzt eine Komponente funktionelle und nicht funktionelle (d.h. Kontrollschnittstellen) Schnittstellen. Durch die funktionellen Schnittstellen werden die Dienste der Komponente angeboten, während die nicht funktionellen, Kontrollschnittstellen genannt, eine Introspektion der Komponente ermöglichen. Eine Komponente selbst besteht aus einem Controller und einem Inhalt. Zusätzlich wird auch definiert wie die Komponenten mit einander kommunizieren, wie man sie erstellt und wie man sie konfiguriert und rekonfiguriert.
Außer der Homepage von Objectweb gibt es wenig Literatur und Dokumentationen, die sich mit Fractal beschäftigen. Es deutet darauf hin, daß Fractal im Vergleich mit anderen Komponentenmodelle (wie EJB, COM ...) weltweit nicht sehr bekannt ist und Anerkennung nur in einem kleinen Kreis findet. Trotzdem gibt immerhin eine gewisse Anzahl von konkreten Fractal-Implementierungen und Erweiterungen, die in der Praxis Einsatz gefunden haben.

7 Anhang

```
/**
 * Interface Main
 */
public interface Main {
  void main(String [] args);
}

/**
 * Interface Service
 */
public interface Service {
  void affiche(String msg);
}

/**
 * Class ClientImpl
 */
public class ClientImpl implements Main, BindingController{
  private Service service;

  public void main (final String[] args) {
    service.affiche("Hello World");
  }

  public String[] listFc() {
    return new String[]{"s"};
  }

  public Object lookupFc(final String cItf){
    if (cItf.equals("s"))
      return service;
    return null;
  }

  public void binfFc(final String cItf, final Object sItf){
    if (cItf.equals("s"))
      service = (Service)sItf;
  }

  public void unbindFc(final String cItf){
    if (cItf.equals("s"))
      service = null;
  }
}
```

```
/**
 * Class ServerImpl
 */
public class ServerImpl implements Service{
  private int count = 1;
  private String entï¿½e = ">>";

  public void affiche(final String msg) {
    for (int i = 0; i < count ; ++i)
      System.err.println(entï¿½e + msg);
    System.err.println("Server: affichage effectuï¿½);
  }

}
```

Abbildungsverzeichnis

Literatur

[1] K.-K. Lau and Z. Wang, "A taxonomy of software component models," 2005.

[2] E. Bruneton, T. Coupaye, *http://fractal.objectweb.org/specification/index.html*. Fractal Homepage, 2004.

[3] *Julia Documentation*. http://fractal.objectweb.org/current/doc/javadoc/julia/overview-sammary.html.

[4] E. Bruneton, *Julia Tutorial*. http://fractal.objectweb.org/tutorials/julia/index.html, 2003.

[5] *FractalADL Homepage*. http://fractal.objectweb.org/current/doc/javadoc/fractal-adl/overview-sammary.html.

[6] *FractalGUI Homepage*. http://fractal.objectweb.org/current/doc/javadoc/fractal-gui/overview-sammary.html.

Service Component Architecture

Marco Willsch

Betreuer: Heiko Koziolek

Zusammenfassung

Das Modell der Service Component Architecture (SCA) ist eine Neuentwicklung von führenden Unternehmen aus der Anwendungsentwicklung, BEA, IBM, IONA, Oracle, SAP, Siebel Systems und Sybase. Basierend auf dem Konzept der serviceorientierten Architektur (SOA) wird ein Vorgehen aufgezeigt, diese Architekturen zu entwickeln. Dazu wird die serviceorientierte Architektur (SOA) vorgestellt, die es erlaubt auf Basis von gekoppelten Diensten, Geschäftsprozesse eines Unternehmens zu unterstützen. Da dieses Konzept kein Vorgehen zum entwickeln von SOA Anwendungen aufzeigt, stellt die SCA ein Modell zur Verfügung mit dem eine Anwendungsentwicklung ermöglicht wird. Diese Anwendung soll dann über die Vorteile, die eine SOA mit sich bringt, verfügen. Dazu gehört die flexible Reaktion auf sich ändernde Geschäftsprozesse im Unternehmen. Leichtere Zusammenarbeit der IT Infrastruktur mit Partnerunternehmen und eine Entkoppelung der Technologie, die zum Implementierung benutzt wird, von der Geschäftslogik. Das neu entwickelte Konzept SCA bildet dabei den Kern der Arbeit und wird über ein Beispiel erläutert.

1 Einleitung

Heutzutage sind komplexe Informationssysteme aus unserem Alltag nicht mehr wegzudenken. Der Trend geht in die Richtung von Systemen, die schnell auf wechselnde Anforderungen reagieren können. Sie sollen Geschäftsprozesse möglichst vollständig unterstützen.

Im Rahmen der Entwicklung des Managementkonzepts Serviceorientierte Architekturen (SOA) [9], wurde sich dieser Aufgabenstellung angenommen. Das Ziel ist eine an die jeweiligen Geschäftsprozesse angepasste IT-Infrastruktur. Die Serviceorientierte Architektur sieht dazu den Einsatz voneinander unabhängiger Dienste vor, durch deren Aneinanderreihung komplexe Geschäftsprozesse abgebildet werden können.

Die SOA bietet allerdings kein Vorgehen, um eine derartige Architektur zu entwickeln. Hier setzt die Service Component Architecture (SCA) an. Diese möchte den Prozess der Erstellung einer Serviceorientierten Architektur vereinfachen. Für die Entwicklung einer Serviceorientierten Anwendung geht die Service Component Architecture in zwei Schritten vor. Es werden die benötigten Dienste implementiert, und darauf aufbauend werden diese miteinander gekoppelt. Dabei gibt die SCA eine Anleitung für diese Schritte an und wie man das Konzept der SOA umsetzen kann. Für die SCA ist es dabei sehr wichtig, möglichst eine große Bandbreite an Technologien zur Umsetzung zu unterstützen, sodass für Unternehmen, die bereits verschiedenste etablierte Entwicklungsumgebungen und Programmiersprachen einsetzen, die vorgestellten Konzepte eingesetzt werden können. Das Ziel ist ein standardisiertes Modell zur Entwicklung von Anwendungen auf Basis von Serviceorientierten Architekturen.

Dazu wird in Kapitel 2 der Begriff SOA und SCA vorgestellt und erklärt. Die SOA bietet dabei die Grundlage für das vom SCA Konsortium, derzeit bestehend aus BEA, IBM, IONA, Oracle, SAP, Siebel Systems und Sybase, entwickelten SCA.

Darauf folgend stellt Kapitel 3 das SCA System vor. Es werden die Spezifikationen aufgezeigt, die ein SCA System ausmachen und aus welchen Teilen sich ein solches zusammensetzt.

Kapitel 4 bietet dann einen Einblick in die Service Data Objects (SDO). Mit ihnen lassen sich unterschiedlichste Datenquellen auslesen und die Daten in einem einheitlichen Format zur Verfügung stellen.

Zusammengeführt werden die vorgestellten Konzepte in einer Beispielanwendung in Kapitel 5. Hier werden die Verfahren anhand eines imaginären Finanzdienstleisters vorgestellt, der einen Dienst entwickeln möchte.

2 Grundlagen

2.1 Serviceorientierte Architektur

SOA [6] steht für serviceorientierte Architektur und ist ein Architekturmuster, das den Aufbau einer Anwendung aus einzelnen Diensten (Services) beschreibt. Bei der SOA steht die Integration aus Sicht des Anbieters im Vordergrund. Ein Dienst ist eine feste, definierte Leistung und wird so gestaltet, dass er von möglichst vielen Konsumenten (Service Consumer) verwendet werden kann. Die Idee der SOA sind die Trennung der Zuständigkeiten nach fachlichen Gesichtspunkten, sowie die Kapselung technischer Details. Die Dienste sind dabei lose miteinander gekoppelt [7] [39ff] und bilden damit

die Geschäftsprozesse ab. Diese Möglichkeit Softwarekomponenten lose miteinander zu koppeln, bringt der Anwendung eine deutlich höhere Agilität. Damit sind SOAs so flexibel, dass sie im Rahmen von unterschiedlichsten Anwendungen in Anspruch genommen werden können.

Die SOA ermöglicht es externe Funktionen anzubieten oder zu nutzen. Dabei ist ein wichtiger Punkt die Entkoppelung der Komponenten. Die Dienste lassen sich dabei über einen einheitlichen Mechanismus aufrufen, der die Anwendungsteile plattformunabhängig miteinander verbindet und alle technischen Details der Kommunikation verbirgt. Diese Kapselung der Implementierung hinter definierten Schnittstellen hilft es die Komplexität deutlich zu reduzieren. Damit erleichtert die SOA, die IT Infrastruktur zu beherrschen, und die Kosten für Entwicklung und Wartung des Systems zu reduzieren. Einmal entwickelte Komponenten können immer wieder verwendet werden.

Leider existiert derzeit keine standardisierte Methode zur Entwicklung von klar definierten Diensten. Hier setzen verschiedenste Verfahren an. Zum Beispiel arbeitet die Oasis an einem Referenzmodell für SOAs [8] und die Service Component Architecture (SCA) möchte eine Methodik aufzeigen mit der man SOAs entwickeln kann.

2.2 Service Component Architecture

Eine Service Component Architecture (SCA) spezifiziert ein Modell, mit dessen Hilfe man Anwendungen entwickeln kann die auf der Idee der SOA beruhen. SCA möchte das Erstellen und Integrieren von Geschäftsanwendungen, die eine SOA verwenden, vereinfachen. Dazu ermöglicht SCA, dass sich grobkörnige Komponenten aus feingranularen Komponenten zusammensetzen. Grobkörnig bedeutet dabei, dass die Schnittstelle (Interface) über wenige Methoden verfügt, die ihre Parameter im Wesentlichen über eine Dokument ähnliche Struktur, z.B. XML Dokumenten, austauschen. Feingranulare Komponenten hingegen, können über eine Vielzahl von Methoden verfügen, die wiederum einfache Parameter verwenden. Dabei gibt SCA ein Vorgehen vor, um die eingesetzten Technologien zur Entwicklung von Diensten und Komponenten, von der Entwicklung der Geschäftslogik zu trennen. Der Entwickler soll sich ganz auf die zu entwickelnde Geschäftslogik konzentrieren können. Der Vorteil ist dadurch die vereinfachte Erstellung von Geschäftskomponenten. Diese wiederum können in Form von Diensten leicht zusammengesetzt werden, und somit kann ein Geschäftsprozess unterstützt werden. Zudem ermöglicht die Entwicklung der Geschäftslogik unabhängig von der darunter liegenden Technologie das, wenn sich diese Technologie ändert, man trotzdem noch die Geschäftslogik weiterverwenden kann. Dadurch erreicht man eine höhere Flexibilität und Kostenersparnis. Um dieses zu ermöglichen spezifiziert die SCA ein Modell. Dieses Modell besteht aus den verschiedensten Teilkomponenten, die zusammen das SCA System bilden. Die einzelnen Teile erfüllen dabei die folgenden Aufgaben.

3 SCA System

Ein SCA System besteht aus Modulkomponenten (Module Component) und möglichen Einstiegspunkten (Entry Points), externen Diensten (External Service) und den dazugehörigen Verbindungen (Wires). Das SCA System stellt die größte Einheit der SCA [2] dar. Es beinhaltet eine Menge von Diensten, die lose miteinander gekoppelt sind.

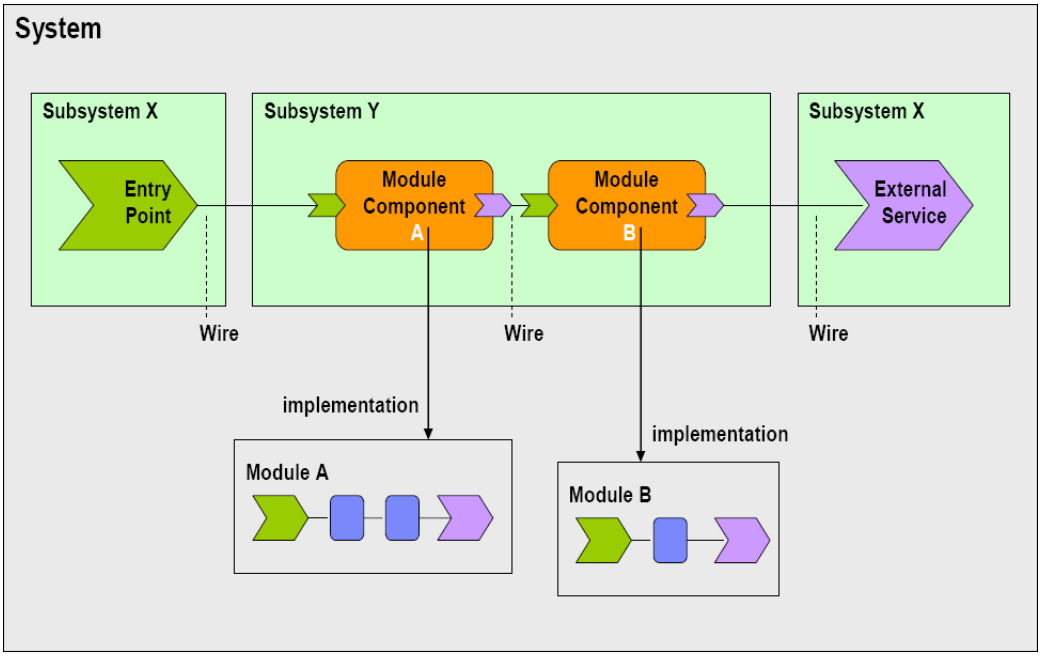


Abbildung 1: SCA System

Zusammen bilden diese eine Geschäftsfunktionalität ab. Dazu kann das SCA System [3] so genannte Teilsysteme (Subsystems) verwenden. Diese wiederum kapseln verschiedenste Funktionalitäten. Ein SCA Subsystem kann dabei Einstiegspunkte, Modulkomponenten oder externe Dienste kapseln. Darüber hinaus enthält es Informationen, wie diese Teilsysteme miteinander verbunden (Wire) sind. Eine Modulkomponente besteht seinerseits aus einem Einstiegspunkt, einer Komponente und einem externen Dienst.

3.1 SCA Teilsystem (Subsystem)

Ein SCA System kann in kleinere SCA Teilsysteme unterteilt werden, die unabhängig voneinander aktualisiert und eingesetzt werden können. Damit erreicht man eine höhere Flexibilität, da sich die Teilsysteme schneller anpassen lassen. Die Teilsysteme können ihrerseits alle Elemente enthalten, die auch ein SCA System enthält. Darüber hinaus können sie auch nur aus Verbindungen bestehen, die einzelne Dienste und Referenzen aus anderen Teilsystemen miteinander verbindet. Ein Teilsystem kann damit dafür verwendet werden, Module die ähnliche Funktionen bereitstellen, über die Konfiguration von Modulkomponenten, Einstiegspunkten, externen Diensten und den zugehörigen Verbindungen dem SCA System bereitzustellen.

3.1.1 Einstiegspunkte (Entry Points)

Einstiegspunkte werden dazu verwendet, einen Dienst nach außen zur Verfügung zu stellen. Dabei unterscheidet die SCA zwischen Einstiegspunkten, die in Teilsystemen verwendet werden, und denen, die in Modulen zum Einsatz kommen. Die Namen für die Einstiegspunkte müssen einmalig in dem jeweiligen Teilsystem sein. Es dürfen also keine Namensübereinstimmungen zwischen Teilsystem-Einstiegspunkten und Einstiegspunkten, die von Modulkomponenten in dem jeweiligen Teilsystem verwendet werden, vorhanden sein.

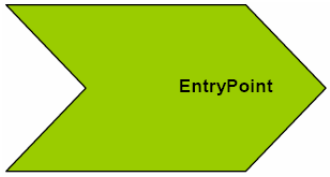


Abbildung 2: SCA Einstiegspunkte

3.1.2 Externe Dienste (External Services)

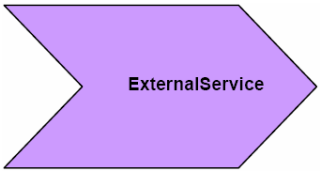


Abbildung 3: SCA Externe Dienste

Mit einem externen Dienst werden Dienste von außerhalb des SCA Systems in dieses eingeführt. Diese Dienste können dann innerhalb des SCA Systems genutzt werden, um dessen Funktionalität zu erweitern. Wie diese Dienste in das SCA System einzufügen sind, beschreiben entsprechende Anbindungen (Bindings) in Kapitel 3.1.4.

3.1.3 Verbindungen (Wires)

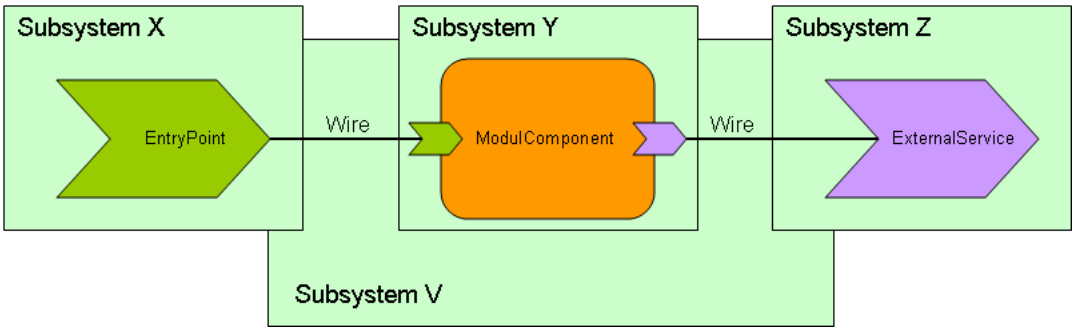


Abbildung 4: SCA Verbindungen

In einem SCA Teilsystem werden Verbindungen dazu genutzt, um externe Dienste mit Einstiegspunkten zu verbinden. Dabei ist es möglich, dass die Verbindungen, die eine Quelle mit einem Ziel verbinden, nicht in demselben Teilsystem definiert sind, indem sich die Quelle oder das Ziel befindet. Es ist also möglich Teilsysteme zu haben, in denen nur Verbindungen definiert sind. Dazu ist es nicht möglich, Quellen und Ziele mit Anbindungen, die nicht zueinander kompatibel sind, miteinander zu verbinden.

3.1.4 Anbindungen (Bindings)

Anbindungen werden von externen Diensten und Einstiegspunkten benutzt. Sie beschreiben den Zugang zu dem Diensten, wenn dieser angebunden wird. Ein SCA System muss dazu wenigstens zwei unterschiedliche Anbindungen unterstützen. Zum einen

ist dies der SCA service binding type und zum anderen der Web service binding type. Bei ersterem handelt es sich um eine Möglichkeit, Dienste zwischen verschiedenen SCA Modulen miteinander interagieren zu lassen. Bei der zweiten Art geht es darum, Dienste auf einer höheren Abstraktionsebene miteinander verbinden zu können, sodass diese über die SCA Systemgrenzen hinaus miteinander zusammenarbeiten können. Anbindungen werden bei externen Diensten dazu verwendet, um zu beschreiben wie der externe Dienst aufzurufen ist. Bei Einstiegspunkten beschreiben Anbindungen wie der Zugriff auf den bereitgestellten Dienst vonstatten geht. Anbindungen werden unabhängig von dem in der Komponente implementierten Code konfiguriert. Diese Trennung ist ein wichtiges Konzept, es erlaubt verschiedenste Zugangsmöglichkeiten zu demselben Dienst zu entwickeln und den Zugangsmechanismus jederzeit anpassen zu können.

3.1.5 Modulkomponente (Module Component)

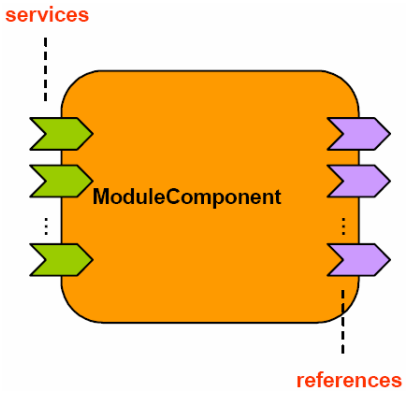


Abbildung 5: SCA Modulkomponente

Modulkomponenten setzten sich aus Einstiegspunkten, externen Diensten und Modulen zusammen. Die Module, die von einer Modulkomponente referenziert werden, werden von dieser ausgeführt. Die Modulkomponente stellt dabei einen Dienst in Form von Einstiegspunkten zur Verfügung, dabei ist der Dienst durch den Einstiegspunkt in das Modul definiert. Modulkomponenten bezeichnet man somit als konfigurierte Module innerhalb eines Teilsystems.

3.2 SCA Modul (Module)

Ein SCA Modul ist die Grundlage für ein SCA System. Es stellt dem SCA System Dienste zur Verfügung, die wiederum lose gekoppelt werden können. Dabei ist ein SCA Modul die größte Einheit von eng miteinander verbundenen Komponenten. Es enthält Einstiegspunkte, externe Dienste, Komponenten und Verbindungen. Komponenten enthalten dabei die Logik der Module. Einstiegspunkte stellen den Dienst nach Außen zur Verfügung. Externe Dienste beschreiben die Abhängigkeit des Moduls von Diensten die von außerhalb des Moduls genutzt werden. Die Verbindungen werden dazu genutzt, Dienste und Komponenten miteinander zu verbinden.

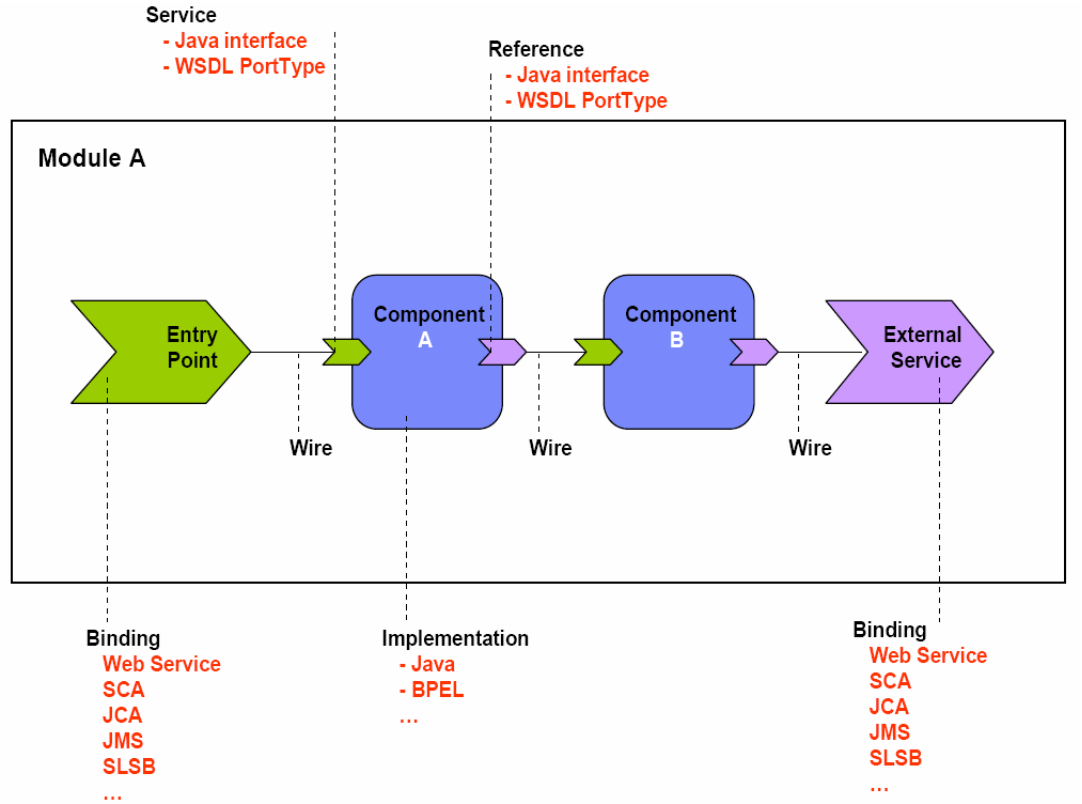


Abbildung 6: SCA Modul

3.2.1 Einstiegspunkte (Entry Points)

Einstiegspunkte in einem Modul können den Dienst, der durch eine im Modul enthaltene Komponente spezifiziert ist, nach außen zur Verfügung stellen oder sie können Dienste, die über externe Dienste an das Modul angebunden sind, in z.B. geänderter Form dem SCA System zur Verfügung stellen. Ein Einstiegspunkt nutzt dazu spezielle Anbindungen.

3.2.2 Externe Dienste (External Services)

Mit der Hilfe von externen Diensten kann sich das SCA Modul jede Art von Funktionalität sich hinzufügen. Solche eingebundenen Dienste können genauso angesprochen werden, wie Dienste die durch interne Komponenten zur Verfügung gestellt werden.

3.2.3 Verbindungen (Wires)

Mit SCA Verbindungen kann man die Teile eines SCA Moduls miteinander verknüpfen. Dazu ist es möglich, Komponentenverweise (References) oder Einstiegspunkte mit Komponenten Diensten (Services) oder externen Diensten zu verbinden. Beachten muss man dabei, ob die Quelle und das Ziel der Verbindung sich im selben Modul befinden, indem auch die Verbindung definiert ist und die beiden Schnittstellen zueinander kompatibel sind. Es ist grundsätzlich möglich Schnittstellen miteinander zu verbinden, die z.B. in unterschiedlichen Sprachen implementiert sind. Hierbei muss allerdings beachtet werden, dass die beiden Schnittstellen sich gleich zueinander verhalten, hinsichtlich

Parameter, Rückgabewerte, etc.

3.2.4 Komponentenarten (Component Type)

Das SCA System versteht unter den Komponentenarten den konfigurierbaren Teil einer Implementierung. Sie stellt einen Dienst zur Verfügung, über den man Eigenschaften der Implementierung einstellen kann. Dieser Dienst wird von der Komponente, die die Implementierung verwendet, genutzt. Erstellen kann man Komponentenarten auf zwei verschiedene Weisen: Zum einen kann die Implementierung zugrunde gelegt und anhand dessen eine Komponentenart entwickelt werden. Zum anderen ist es möglich, eine Komponentenartinformation in Form einer speziellen Datei zu nutzen.

3.2.5 Komponente (Component)

Wenn man eine Komponentenart nutzt um eine Implementierung einzustellen, erhält man eine Komponente. Diese ist eine konfigurierte Instanz der Implementierung. Von daher ist es möglich, dass ein und dieselbe Implementierung von mehreren Komponenten genutzt wird und dabei jedes Mal anders konfiguriert ist. Eine Komponente stellt dabei Dienste zur Verfügung und kann ihrerseits Dienste nutzen.

3.2.6 Implementierung (Implementation)

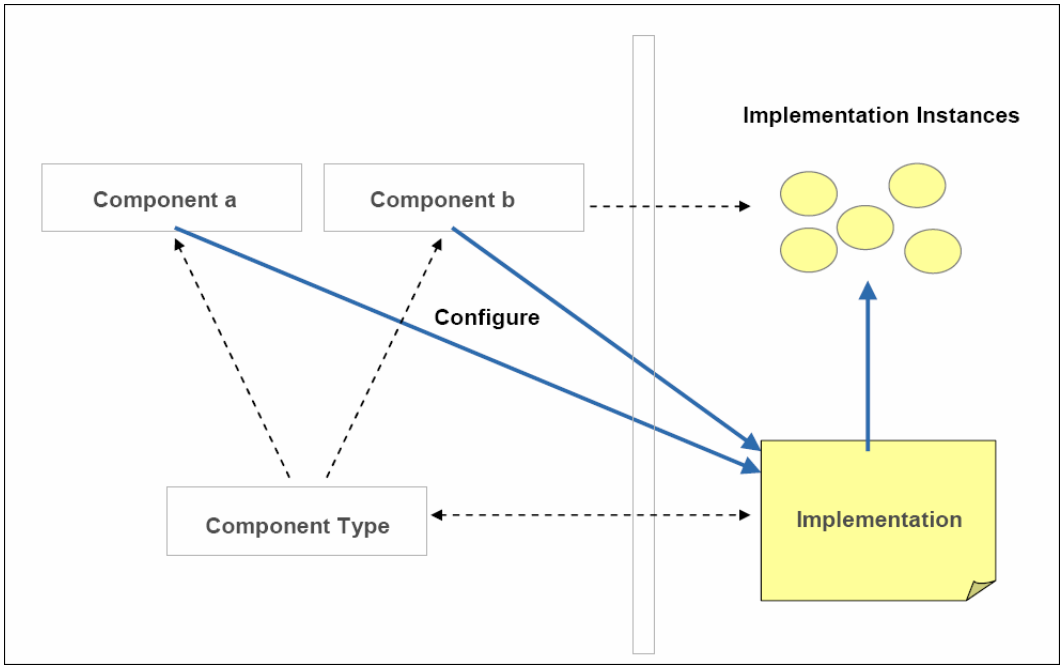


Abbildung 7: SCA Implementierung

Die Implementierung stellt einer Komponente im SCA System Funktionalität zur Verfügung. Sie kann auf der Basis von verschiedensten Technologien entwickelt sein. Dabei besteht die Möglichkeit, dass man die Implementierung über gewisse Parameter konfigurieren kann. Während der Laufzeit wird eine Instanz der Implementierung

angelegt. Die Geschäftslogik erhält diese Instanz von der zugrunde liegenden Implementierung. Die Parameter und Werte die verändert werden können, kommen von der Komponente, die die Implementierung verwendet.

4 Service Data Objects

Service Data Objects (SDO) [1] sollen einen einheitlichen Zugang zur Datenzugriffsschicht ermöglichen. Derzeit sind die Technologien, die eingesetzt werden um auf Daten zuzugreifen, auf die jeweiligen Datentypen festgelegt. Allerdings kommen in den meisten Anwendungen vielfältige Datenquellen zum Einsatz. Diese Vielfalt erschwert die Entwicklung der Anwendungen natürlich, da verschiedenste Technologien und Modelle angewendet werden müssen, um auf die Daten zuzugreifen. Außerdem erschwert die Vielfalt von Datenquellen es Werkzeugen automatisch Komponenten, mit zugrunde liegenden Datenquellen, miteinander zu verbinden. Deswegen soll die einheitliche Darstellung von Daten aus verschiedensten Datenquellen hier Abhilfe schaffen. Die SDO

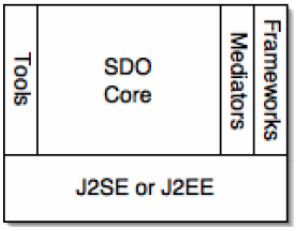


Abbildung 8: SDO Architektur

Architektur setzt dabei in derart auf die Java Umgebung auf, dass sie einen SDO Core zur Verfügung stellt. Der SDO Core enthält dabei die Funktionen, die benötigt werden, um mit den SDO umzugehen. Damit man Zugriff auf Datenquellen erhält, bietet die Architektur SDO Data Mediator Services an. Diese können Daten aus Datenquellen lesen und aktualisieren. Das SDO Framework ist in der Lage Daten zu einzelnen Komponenten zuzuordnen. Ermöglicht wird dies von der SDO Architektur. Die Komponenten in einer SDO Architektur arbeiten wie in Abbildung 9 zusammen. Eine Datenquelle (Data Source) stellt Daten zur Verfügung. Der Data Mediator Service ermöglicht den Zugriff auf diese Datenquelle und erstellt aus den gelesenen Daten, so genannte abgekoppelte Datengraphen, die die Daten der Quelle in Form von Datenobjekten und Metadaten enthalten. Deswegen verfügt die SDO Architektur über eine Vielzahl von Data Mediator Services, die mit unterschiedlichen Datenquellen umgehen können. Diese können dann von der Anwendung verarbeitet werden. Veränderungen in den Daten werden dabei in den Datengraphen bewerkstelligt und vom Data Mediator Service ausgewertet und in der Datenquelle aktualisiert.

4.1 Datengraphen

Die Datengraphen sind in einer SDO Architektur für die Darstellung der Daten zuständig. Sie enthalten dazu Datenobjekte. Die Datengraphen werden zwischen den einzelnen Komponenten ausgetauscht. Sollten dabei in einer Komponente Änderungen in dem

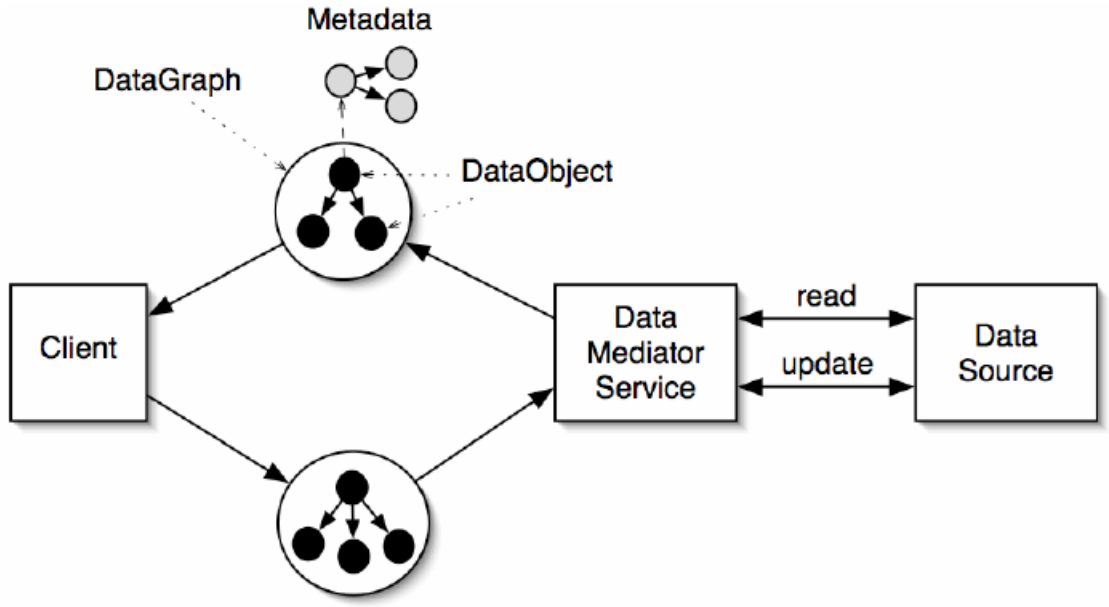


Abbildung 9: SDO Komponenten

Datengraphen vorgenommen werden, so muss die Struktur diese Veränderungen in einer Zusammenfassung aufzeichnen. Diese kann dann vom Data Mediator Service ausgewertet werden, damit die Änderungen in der Datenquelle aktualisiert werden. Danach beinhaltet die Zusammenfassung über Veränderungen in den einzelnen Datenobjekten eine Übersicht über die hinzugefügten, gelöschten und aktualisierten Datenobjekte. Wobei für die aktualisierten Datenobjekte Informationen über alte und neue Werte verfügbar sind.

4.2 Datenobjekte

Datenobjekte enthalten eine Vielzahl von unterschiedlichen Daten. Sie können zum einen einfach nur Werte enthalten, aber auch auf andere Datenobjekte verweisen und somit Listen abbilden. Um Datenobjekte zu analysieren verfügt die SDO Architektur über Metadaten. Über diese können Informationen über die Daten, die in den Datengraphen und Datenobjekten enthalten sind, erhalten werden. Dabei unterstützen Datenobjekte verschiedene Zugriffsformen. Zum einen kann auf Datenobjekte dynamisch über XPath [5] zugegriffen werden oder die Datenobjekte können im Vorfeld statisch über Modelle und Schemata generiert werden. Der statische Zugriff ermöglicht dabei dem Entwickler einen einfacheren und direkteren Zugriff auf die Daten in dem zugehörigen Datengraph.

4.3 Metadaten

Mit den Metadaten erhält der Entwickler einen veränderten Blick auf die in einem Datengraphen enthalten Datenobjekte und die darin befindlichen Daten. Dieser wird benötigt, um die Daten zu inspizieren. Dies kann durch den Entwickler geschehen oder aber auch von Werkzeugen übernommen werden. Die Metadaten ermöglichen es, Werkzeugen Datenbindungen zwischen Benutzerschnittstellen und Datenquellen durch-

zuführen. Dazu wird aus den zugrunde liegenden Daten ein Datenmodell aufgebaut. Dieses Datenmodell wird dann durch die Metadaten beschrieben.

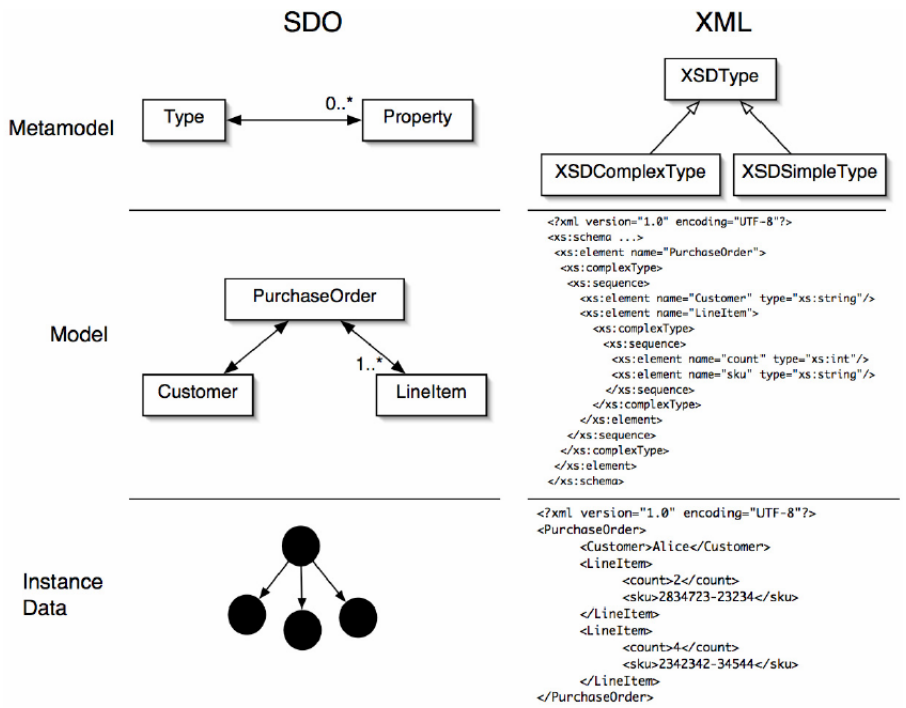


Abbildung 10: Beziehung zwischen Daten,Metamodellen und Metadaten

5 SCA Beispielanwendung

Die im folgenden vorgestellte Beispielanwendungen, soll Einblicke darin geben, wie eine Service Component Architecture (SCA) entwickelt wird. Dazu werden Konzepte einer SCA aufgegriffen und anhand von Beispielen eingesetzt. Es werden Implementierungen erstellt, die mit Hilfe von Konfigurationsparametern eine SCA Komponente ergeben. Um Dienste verfügbar zu machen und nutzen zu können, werden Einstiegspunkte und externe Dienste exemplarisch entwickelt. Diese einzeln erstellten Elemente werden dann in Form von SCA Modulen zusammengeführt und in ein SCA System als ein SCA Teilsystem eingesetzt. Als Beispiel wird ein Finanzinstitut [4] zugrunde gelegt, das seinen Endkunden verschiedene Dienste zur Verfügung stellen möchte. Es soll möglich sein, verschiedene Kontoinformationen abrufen zu können und auch Börsenkurse zu nutzen. Initial wird entschieden, dass die Anwendung in zwei Teilen entwickelt wird. Ein Modul wird die verschiedenen Kontendienste ermöglichen. Das zweite Modul ist die Webschnittstelle, die genutzt werden kann, um die Dienste des ersten Moduls zu nutzen. Abbildung 11 zeigt das SCA Modul für die Kontenverwaltung "bigbank.accountmodule". Es verfügt über einen Entry Point "Account Service", der einen Einstiegspunkt entspricht und somit den Dienst nach außen zur Verfügung stellt. Die "Account Service Component"stellt die Implementierungen bereit, die es dem Kunden ermöglichen, Konteninformationen zu bekommen. Damit diese Informationen erhalten werden können, wird von der "Account Service Component"die "Account Data Service Component"verwendet. Die "Account Data Service Component"übernimmt dabei die

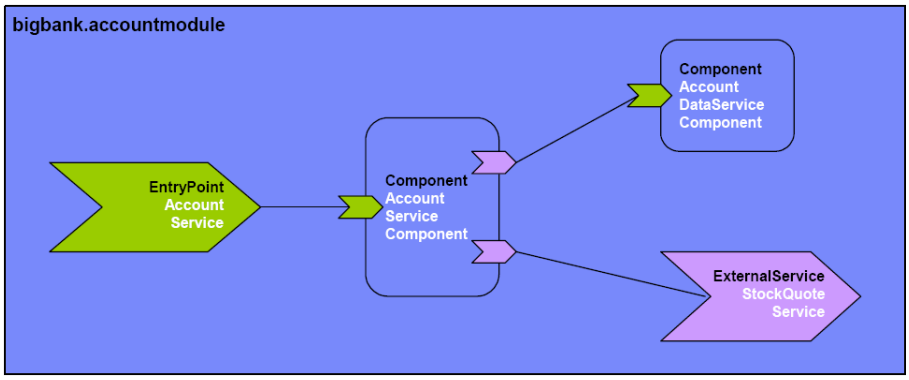


Abbildung 11: Modul zur Accountverwaltung

Aufgabe Konteninformationen abzurufen. Zusätzlich erhält die Anwendung einen externen Dienst. Der "External Stock Quote Service"bindet einen Dienst in das Modul ein, damit aktuelle Wertpapierkurse abgerufen werden können.

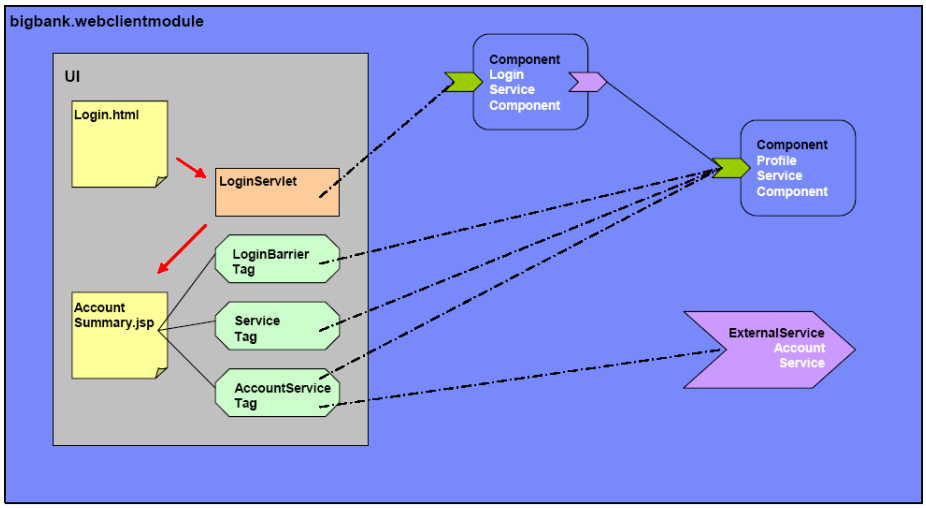


Abbildung 12: Modul für die Internetanbindung

Das SCA Modul "bigbank.webclientmodule"in Abbildung 12 stellt die Funktionen bereit, damit sich Kunden über einen Browser in das System einloggen und Zugriff auf ihre Kontoinformationen erhalten können. Dazu verfügt das System über eine Benutzerschnittstelle, wobei die einzelnen Teile, Login.html, LoginServlet und Account-Summary.jsp, für den Informationsaustausch zwischen dem System und dem Benutzer zuständig sind. Die beiden Komponenten "Login Service Component"und "Profile Service Component"enthalten Informationen über den aktuellen Zustand, indem sich der Benutzer und das System befindet. Als externen Dienst bindet der "Account Service"das "bigbank.accountmodule"ein.

Begonnen wird mit der Entwicklung des "bigbank.accountmodule". Die Umgebung für das SCA "accountmodule"wird als erstes erstellt. Danach folgen die einzelnen Implementierungen, die für die Komponenten benötigt werden. Damit die "Account Data Service Component"Konteninformtionen zur Verfügung stellen kann, benötigt sie eine Implementierung. Die "Account Data Service Implementation"verfügt in unserem Beispiel über drei Methoden. Mit diesen können der Kontostand geprüft werden, Spar-

einlagen angezeigt und Wertpapierinformationen abgerufen werden. Die Methoden sind darüber hinaus in einer Schnittstelle definiert. Die SCA Implementierung setzt sich somit aus der Implementierung der Methoden und der zugehörigen Schnittstelle zusammen. Damit kann die "Account Data Service Component"erstellt werden. Sie nutzt eine Instanz der Implementierung aus der "Account Data Service Implementation", die dazu über verschiedene Parameter konfiguriert werden kann. In einem weiteren Schritt wird der externe Dienst "Stockquote Service"erstellt. Die Funktionalität, um Wertpapierinformationen abrufen und auswerten zu können, wird hierbei von einem außenstehenden Dienst erbracht, der von dem Modul genutzt werden soll. Dazu wird dieser Dienst über den "Stockquote Service"dem Modul zur Verfügung gestellt. Die beiden so erstellten Teile werden in der "Account Service"Implementierung referenziert. Die "Account Service"Komponente erstellt Kontenberichte und Kontenzusammenfassungen. Dafür wird eine Implementierung mit Methoden erstellt, die diese Berichte erzeugen können und es wird auf die zuvor entwickelten Dienste zugegriffen, siehe Abbildung 13.

In das "bigbank.accountmodule"wird die Implementierung als "Account Service Component"eingebracht (Abbildung 14).

Um die entwickelten Dienste außerhalb des Moduls zur Verfügung zu stellen, benötigt das Modul einen Einstiegspunkt. Hierfür dient der "Account Service Entry Point". Er ermöglicht den externen Zugriff auf die Dienste. Diese Dienste müssen dazu im Einstiegspunkt genau definiert sein.

Das "bigbank.webclientmodul"wird in äquivalenter Weise aufgebaut. Zuerst wird das Modul erstellt und danach werden sukzessive die einzelnen Teile im Modul entwickelt. Damit sich ein Benutzer in das System einloggen kann, verfügt das Modul über die "Login Service Component". Dieser wiederum liegt eine "Login Service"Implementierung zugrunde. Die Implementierung hat eine Schnittstelle mit verschiedenen Zustandsvariablen und einer Methode die es ermöglicht auf das System Zugriff zu nehmen. Um unseren Benutzer persönlich ansprechen zu können und seinen Zustand im System mitverfolgen zu können, verfügt das Modul über die "Profile Service Component. Die Implementierung verfügt über verschiedene Methoden, um unter anderem, den Nachnamen und Vornamen sichern zu können und um Informationen darüber zu haben, wo sich der jeweilige Benutzer befindet. Für die Anbindung der eigentlichen Funktionalität, die dem Kunden zur Verfügung gestellt werden soll, ist der externe Dienst "Account Service"zuständig. Dieser bindet der Dienst, den wir über den Einstiegspunkt aus dem "bigbank.accountmodule"öffentlich gemacht haben, in das "bigbank.webclientmodule ein. Somit sind die beiden Module fertig erstellt und können in das SCA System ausgeliefert werden. Dazu wird aus den beiden Modulen jeweils eine Modulkomponente erstellt, die in ein SCA Teilsystem eingepflegt werden kann. Die beiden SCA Teilsysteme können dann in das SCA System eingepflegt werden und ergeben die angeforderte Anwendung. Ein Dienst, der es Kunden ermöglicht nach Anmeldung, verschiedenste Kontoinformationen und Kontenberichte abzurufen.

```
package services.account;

import java.util.List;

import commonj.sdo.DataFactory;

import org.osoa.sca.annotations.Property;
import org.osoa.sca.annotations.Reference;

import services.accountdata.AccountDataService;
import services.accountdata.CheckingAccount;
import services.accountdata.SavingsAccount;
import services.accountdata.StockAccount;
import services.stockquote.StockQuoteService;

public class AccountServiceImpl implements AccountService {

    @Property
    private String currency = "USD";

    @Reference
    private AccountDataService accountDataService;
    @Reference
    private StockQuoteService stockQuoteService;

public AccountReport getAccountReport(String customerID) {

 DataFactory dataFactory = DataFactory.INSTANCE;
 AccountReport accountReport = (AccountReport)dataFactory.create(AccountReport.class)
 List accountSummaries = accountReport.getAccountSummaries();
    CheckingAccount checkingAccount = accountDataService.getCheckingAccount(customerID);
    AccountSummary checkingAccountSummary = (AccountSummary)dataFactory.create(AccountSummary.class);
    checkingAccountSummary.setAccountNumber(checkingAccount.getAccountNumber());
    checkingAccountSummary.setAccountType("checking");
    checkingAccountSummary.setBalance(fromUSDollarToCurrency(checkingAccount.getBalance()));
    accountSummaries.add(checkingAccountSummary);

    SavingsAccount savingsAccount = accountDataService.getSavingsAccount(customerID);
    AccountSummary savingsAccountSummary = (AccountSummary)dataFactory.create(AccountSummary.class);
    savingsAccountSummary.setAccountNumber(savingsAccount.getAccountNumber());
    savingsAccountSummary.setAccountType("savings");
    savingsAccountSummary.setBalance(fromUSDollarToCurrency(savingsAccount.getBalance()));
    accountSummaries.add(savingsAccountSummary);

    StockAccount stockAccount = accountDataService.getStockAccount(customerID);
    AccountSummary stockAccountSummary = (AccountSummary)dataFactory.create(AccountSummary.class);
    stockAccountSummary.setAccountNumber(stockAccount.getAccountNumber());
    stockAccountSummary.setAccountType("stock");
    float balance= (stockQuoteService.getQuote(stockAccount.getSymbol()))*stockAccount.getQuantity();
    stockAccountSummary.setBalance(fromUSDollarToCurrency(balance));
    accountSummaries.add(stockAccountSummary);

    return accountReport;
   }

   private float fromUSDollarToCurrency(float value){

    if (currency.equals("USD")) return value; else
    if (currency.equals("EURO")) return value * 0.8f; else
    return 0.0f;
   }
}
```

Abbildung 13: Account Service Implementierung

```
<?xml version="1.0" encoding="ASCII"?>
<module     xmlns="http://www.osoa.org/xmlns/sca/0.9"
            xmlns:v="http://www.osoa.org/xmlns/sca/values/0.9"

    name="bigbank.accountmodule" >

    <component name="AccountServiceComponent">
            <implementation.java class="services.account.AccountServiceImpl"/>
            <properties>
                    <v:currency>EURO</v:currency>
            </properties>
            <references>
                    <v:accountDataService>AccountDataServiceComponent</v:accountDataService>
                    <v:stockQuoteService>StockQuoteService</v:stockQuoteService>
            </references>
    </component>

    <component name="AccountDataServiceComponent">
            <implementation.java class="services.accountdata.AccountDataServiceImpl"/>
    </component>


    <externalService name="StockQuoteService">
            <interface.java interface="services.stockquote.StockQuoteService"/>
            <binding.ws port="http://www.quickstockquote.com/StockQuoteService#
                                    wsdl.endpoint(StockQuoteService/StockQuoteServiceSOAP)"/>
    </externalService>

</module>
```

Abbildung 14: Account Service Komponente

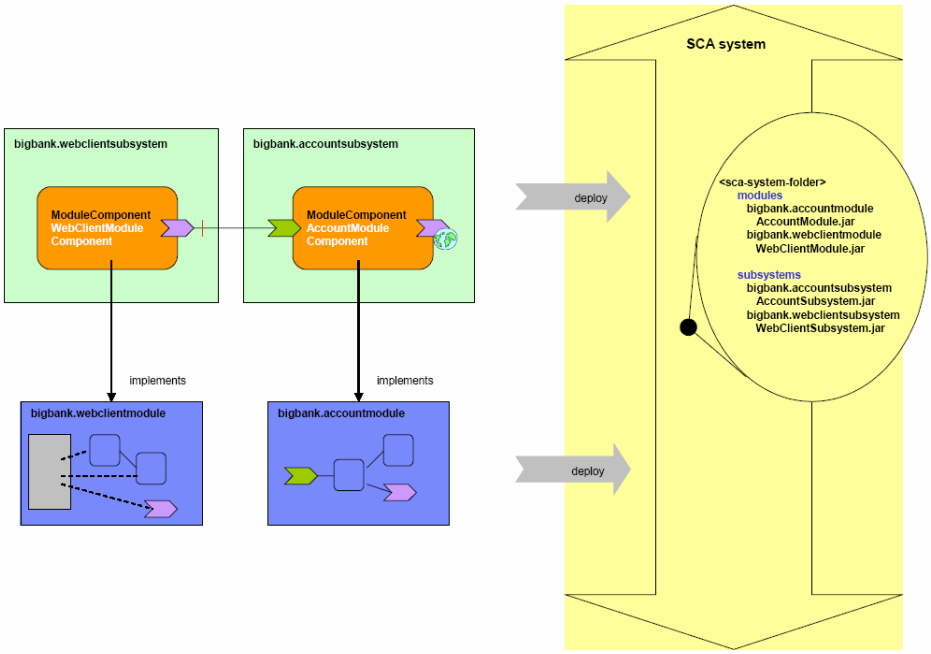


Abbildung 15: Einbindung der Module ins SCA System

Abbildungsverzeichnis

Literatur

[1] J. Beatty, S. Brodsky, M. Nally, and R. Patel, *Next-Generation Data Programming: Service Data Objects*. BEA, IBM, 2003. [Online]. Available: www-128.ibm.com/developerworks/library/specification/ws-sdo/

[2] M. Beisiegel, H. Blohm, D. Booz, J.-J. Dubray, A. Colyer, M. Edwards, D. Ferguson, B. Flood, M. Greenberg, D. Kearns, J. Marino, J. Mischkinsky, M. Nally, G. Pavlik, M. Rowley, K. Tam, and C. Trieloff, *Building Systems using a Service-Oriented Architecture*. BEA, IBM, Interface21, IONA, Oracle, SAP, Siebel, Sybase, 2005. [Online]. Available: www-128.ibm.com/developerworks/library/specification/ws-sca/

[3] M. Beisiegel, H. Blohm, D. Booz, J.-J. Dubray, M. Edwards, B. Flood, B. Ge, O. Hurley, D. Kearns, M. Lehmann, J. Marino, M. Nally, G. Pavlik, M. Rowley, A. Sakala, C. Sharp, and K. Tam, *Assembly Model Specification*. BEA, IBM, IONA, Oracle, SAP, Siebel, Sybase, 2005. [Online]. Available: www-128.ibm.com/developerworks/library/specification/ws-sca/

[4] M. Beisiegel, H. Blohm, D. Booz, J.-J. Dubray, M. Edwards, A. Karmarkar, J. Marino, M. Nally, G. Pavlik, M. Rowley, K. Tam, and L. Waterman, *Building Your First Application - Simplified BigBank*. BEA, IBM, Oracle, SAP, Sybase, 2005. [Online]. Available: www-128.ibm.com/developerworks/library/specification/ws-sca/

[5] J. Clark and S. DeRose, *XML Path Language (XPath)*. W3C, 1999. [Online]. Available: www.w3.org/TR/xpath

[6] H. He, *What Is Service-Oriented Architecture*. O'Reilly, 2003. [Online]. Available: www.xml.com/pub/a/ws/2003/09/30/soa.html

[7] M. Keen, A. Acharya, S. Bishop, A. Hopkins, S. Milinski, C. Nott, R. Robinson, J. Adams, and P. Verschueren, *Patterns: Implementing an SOA Using an Enterprise Service Bus*. IBM, 2004. [Online]. Available: http://publib-b.boulder.ibm.com/abstracts/sg246346.html?Open

[8] C. M. MacKenzie, K. Laskey, F. McCabe, P. F. Brown, and R. Metz, *Reference Model for Service Oriented Architecture 1.0*. OASIS, 2006. [Online]. Available: www.oasis-open.org/committees/tc_home.php?wg_abbrev=soa-rm

[9] Wikipedia, *Serviceorientierte Architektur*. Wikimedia Foundation Inc., 2006. [Online]. Available: de.wikipedia.org/wiki/Serviceorientierte_Architektur

Teil III.

Transformationen

Model-To-Text Transformation Languages

Andreas Rentschler

Betreuer: Steffen Becker

Zusammenfassung

Transformationssprachen, welche aus Modellen plattformspezifischen Code oder andere textuellen Artefakte generieren können, stellen das letzte Glied im modellgetriebenen Softwareentwicklungsprozess dar. Auf Grund der noch ausgebliebenen Standardisierung gibt es auf dem kommerziellen wie auch quelloffenen Markt unzählige proprietäre Lösungen und zur Standardisierung eingereichte Vorschläge.

Diese Ausarbeitung gibt nach einer kurzen Einführung in die Grundlagen der modellgetriebenen Softwareentwicklung einen Überblick über einige derzeit existierende Modell-zu-Text-Transformationssprachen und wie sie in die Architektur des jeweiligen MDA-Werkzeugs eingebettet sind. An Hand von Beispielszenarien wird des Weiteren herausgearbeitet, wie im konkreten Anwendungsfall der Einsatz dieser Sprachen aussehen kann.

1 Einleitung

Diese Einleitung soll zunächst *die* Basiskonzepte erläutern, welche Modell-zu-Text-Transformationssprachen zu Grunde liegen. Ein historischer Abriss zeigt die Ideologie hinter der modellgetriebenen Softwareentwicklung und welche Rolle die Standards der OMG hier spielen. Anschließend wird auf die modellgetriebene Architektur (MDA) eingegangen, insbesondere ihr Verständnis für Transformationen zwischen Modellen und Code. Am Ende des Kapitels wird der von der OMG initiierte Prozess zur Erarbeitung einer standardisierten Modell-zu-Text-Transformationssprache genauer betrachtet.

1.1 Paradigmenwechsel in der objektorientierten Softwareentwicklung

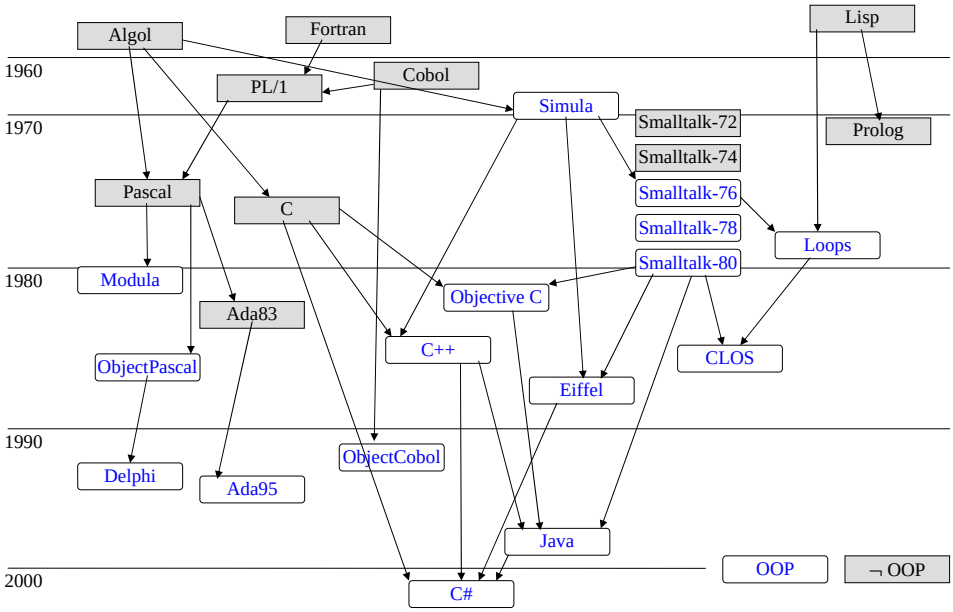


Abbildung 1: Historie objektorientierter Sprachen

In den 60er Jahren wurde mit der Erfindung der Programmiersprache Simula das objektorientierte Programmierparadigma in die Welt der Softwareentwicklung eingeführt. Grund für die vor allem in den 80er Jahren wachsende Popularität ist vor allen Dingen die Tatsache, dass sich Probleme in der objektorientierten Programmierung (OOP) wesentlich eleganter lösen lassen als in herkömmlichen modularen Programmiersprachen. Beim modularen Entwurf wird durch Trennung von Schnittstelle und Implementierung das Prinzip der Kapselung von Daten unterstützt. Der objektorientierte Entwurf erlaubt dies ebenfalls und lässt den Programmierer Systeme auf eine Art beschreiben, „wie menschliches Denken die reale Welt begreift“ [1]. Zahlreiche Konzepte wie Objekt, Klasse, Nachricht/Methode, Vererbung, Verkapselung, Abstraktion und Polymorphismus ermöglichen ein derartiges Vorgehen. Die objektorientierte Programmierung (OOP) stellt einen Paradigmenwechsel vom Subjekt – der Information – zum Objekt dar. Ein Softwaresystem lässt sich mit der OOP durch Klassen von Objekten repräsentieren welche untereinander über Botschaften kommunizieren können, zustandsbehaftet sind und voneinander Eigenschaften erben können.

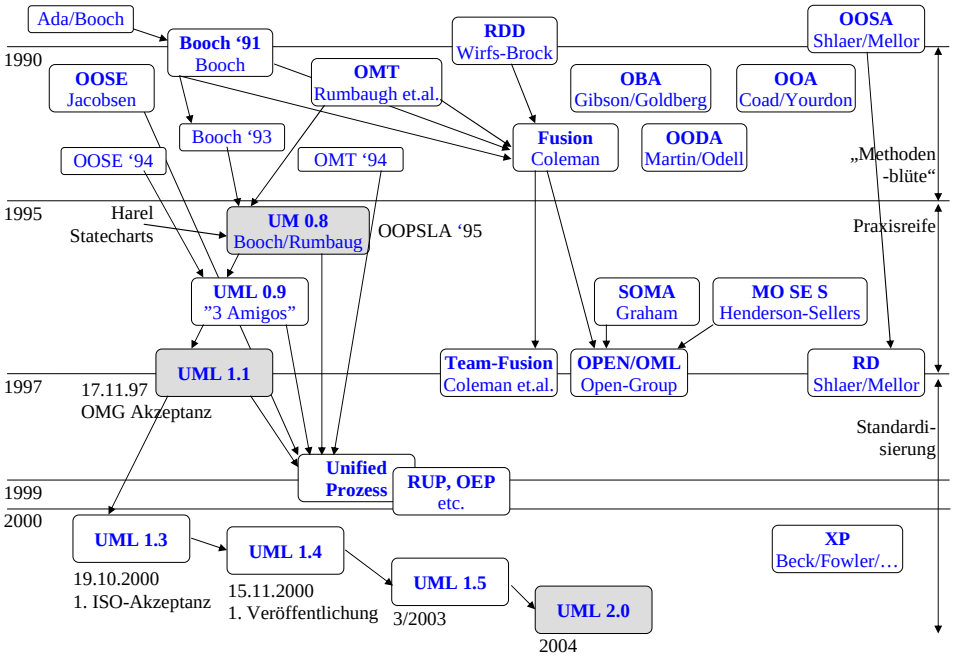


Abbildung 2: Historie der UML

Später in den 90er Jahren entwarfen die „drei Amigos" Grady Booch, James Rumbaugh und Ivar Jacobson die Sprache Unified Modeling Language (UML) als Hilfsmittel zur Visualisierung von objektorientierten Strukturen in Planung und Definition. Komplizierte Zusammenhänge lassen sich mit ihr einfacher begreifen wenn sie grafisch notiert sind und unterstützen die Kommunikation im Team. Die „drei Amigos" übergaben die Sprache an die Object Management Group (OMG), welche bis heute an deren Weiterentwickelung und Standardisierung arbeiten. (Abbildung 2)

Die OMG ist ein unabhängiges Konsortium von mittlerweile über 800 Mitgliedern, dazu gehören auch viele namhafte Firmen wie Apple, IBM und Sun. Das Hauptziel dieser Vereinigung ist die Etablierung von Industrierichtlinien für objektorientierte Technologien, um Interoperabilität zwischen modellgetriebenen Entwicklungswerkzeugen unterschiedlicher Hersteller zu gewährleisten, sowie die Verabschiedung von Objektmanagement-Spezifikationen, weche die Entwicklung von verteilten Applikationen über alle Plattformen und Betriebssysteme hinweg ermöglichen.

Die UML definiert zunächst Begriffe und mögliche Beziehungen zwischen diesen Begriffen aus der Welt der Modellierung, die Begriffe sind mit Schlüsselwörtern aus „gewöhnlichen" Sprachen wie Java vergleichbar. Die UML ist damit zunächst eine textuelle Sprache um Beziehungen zwischen Objekten, deren Eigenschaften und deren Verhalten zu beschreiben. Darauf aufbauend definiert die UML grafische Notationen sowohl für die Begriffe als auch aus den Begriffen formulierbare Modelle. Grafische Diagramme bieten lediglich eine Sichtweise (View) auf einen Teil eines Modells.

Eine Definition des Modell-Begriffs im informationstechnischen Kontext findet sich in [2] mit Ergänzungen frei nach [3]:

Definition. Ein *Modell* ist eine in einer wohldefinierten Sprache geschriebene Beschreibung eines Systems oder eines Teils davon. Modelle sind zielorientiert, d.h. gewisse Merkmale des Systems können weggelassen werden, wenn sie der Autor des Modells für

den geplanten Einsatz als nicht relevant betrachtet. Modelle erfüllen also die Merkmale der Abbildung, der Verkürzung und des Pragmatismus.

Definition. Eine *wohldefinierte Sprache* ist eine Sprache in einer wohldefinierten Form (Syntax) und Bedeutung (Semantik), welche zur automatisierten Interpretation durch einen Computer geeignet ist.

Mit der Zeit entwickelte die OMG UML in den Versionen 1.1, 1.3, 1.4, 1.5 weiter, und präsentierte schließlich im März 2005 die UML in der Version 2.0 mit mehreren Neuerungen:

- Der neue Begriff *Metamodellierung* verkörpert die Eigenschaft der Sprache, sich aus wenigen vorgegebenen Grundbestandteilen heraus selbst definieren zu können. Die Grundbestandteile der UML werden Meta Object Facility (MOF) genannt.
- *Profile* stellen domänenspezifische Erweiterungen des vorgegebenen Metamodell um Stereotypen, Tagged Values, Constraints und Custom Icons dar. Damit lässt sich die Sprache an spezifische Begriffe aus der Geschäftswelt anpassen.
- Das bereits bekannte XML Metadata Interchange (XMI) Format, das neben Serialisierung den Austausch von UML-Modellen über Anwendungsgrenzen hinweg erlaubt, wurde um das Format „UML 2.0 Diagram Interchange“ erweitert. Nun lassen sich auch grafische Daten von Diagrammen an andere UML-Werkzeuge weitergeben. Wer mehr über die Fähigkeiten der UML2 wissen möchte, findet weitere Informationen in [4].

Auch mit der Unterstützung von UML in Planung und Definition bleibt die Softwareentwicklung mit OOP immer noch eher ein Handwerk denn eine Ingenieurswissenschaft. Viele wichtige Implementierungsentscheidungen lassen sich kaum ohne Erfahrungswerte machen, aus diesem Grund können Zeitbedarf und die damit verbundenen Kosten eines Softwareprojektes nur näherungsweise geschätzt werden.

1.2 Modellgetriebene Softwareentwicklung

Die modellgetriebene Softwareentwicklung (MDSD) hat das Paradigma der generativen Programmierung zum Vorbild. Entwicklungszeit (und damit -kosten) entstehen neben der Wartung und Pflege vor allem in der Implementierungsphase. Da von Hand implementiert wird, bietet es sich an, an dieser Stelle Automatismen einzufügen. Genau dieses Ziel verfolgt die MDSD unter dem Leitspruch „Automatismus durch Formalismus“: Im idealen Fall muss lediglich eine formale Spezifikation des Problems angegeben werden, durch Wahl entsprechender Generatoren kann festgelegt werden, auf welcher Zielplattform (.NET, Java Virtual Machine (JVM)) das Programm laufen wird, und welche Technologien zur Lösung eingesetzt werden (z.B. Enterprise Java Beans (EJB), Hibernate, Struts,...)

Eine Ausprägung der MDSD ist die **modellgetriebene Architektur (MDA)** der OMG, ein Rahmenwerk für die Anwendungsentwicklung. Sie besteht vornehmlich aus einer Familie von herstellerneutralen Spezifikationen, welche seit 2002 von der OMG verabschiedet werden. Für einige Bestandteile der MDA stehen Standards noch aus [5].

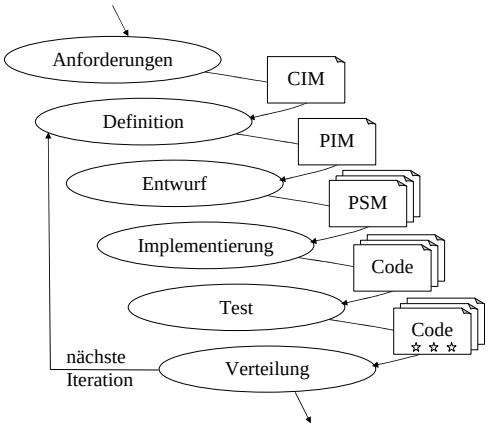


Abbildung 3: Zyklus der Softwareentwicklung in der MDA

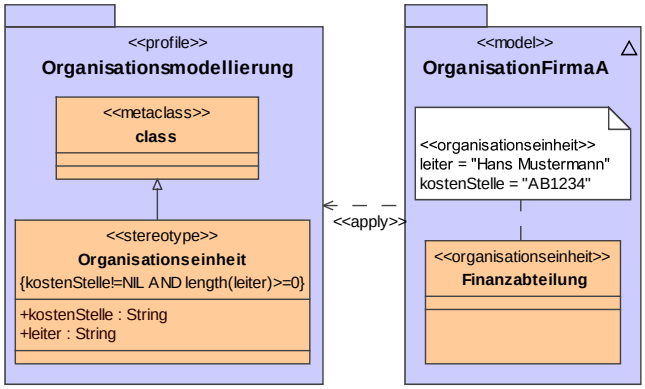


Abbildung 4: Definition und Anwendung eines UML Profiles

Betrachtet man die einzelnen Phasen, welche beim Vorgehensmodell der MDA durchlaufen werden (Abbildung 3), findet man im Vergleich zum traditionellen Softwareentwicklungszyklus [6] keine Unterschiede: Anforderungsanalyse, Definition, Entwurf, Implementierung, Test und Verteilung werden in gleicher Weise durchlaufen. Auch bei der MDA erstellen, verwenden und erweitern die einzelnen Phasen (Zwischen-)Ergebnisse, sogenannte Artefakte. Beim traditionellen Vorgehensmodell jedoch bestehen diese aus Diagrammen und informellen bzw. halbformalen Texten. Im Gegensatz dazu sind ab der Definitionsphase des MDA-Entwicklungsprozesses alle Artefakte formale Modelle um damit einen höheren Grad an Automatismus zu erreichen, da formale Artefakte auch von Rechnern verarbeitet werden können.

Ein weiterer Unterschied zum traditionellen Vorgehen in der Softwareindustrie liegt in der strikten Trennung von Entwurf und Architektur des Systems. Bei der Anforderungsanalyse wird lediglich den funktionalen Anforderungen Rechnung getragen, ohne Aussagen über die Architektur des Systems zu machen. Ergebnis ist das berechnungsunabhängige Modell (CIM), eine Art umgangssprachliche Black-Box-Spezifikation mit festgelegter Semantik, es kann mit Anwendungsfall-, Interaktions- und Aktivitätsdiagrammen der UML beschrieben werden. Das zweite, und jetzt vollständig formale Artefakt im Entwurfsprozess ist das plattformunabhängige Modell (PIM), ein abstraktes Lösungsmodell. In diesem Modell werden ausschließlich die funktionalen Anforderungen umgesetzt, wie sie im CIM spezifiziert sind, während nicht-funktionale Anforderungen erst später in das plattformspezifische Modell (PSM) eingearbeitet werden. Plattformspezifische Modelle sind auf die Architektur des Zielsystems ausgerichtet, so ist – rein exemplarisch – die Spezialisierung eines PSMs auf die plattformbedingten Möglichkeiten und Einschränkungen die Java und JBoss bieten denkbar. In Abbildung 5 wird der Weg vom CIM über PIM und PSM zum Code verdeutlicht.

Die UML erlaubt erst durch ihre Erweiterbarkeit und Anpassbarkeit ab Version 2 eine effektive automatisierte Nutzung von Modellen, wie sie von der MDSD gefordert wird. Die bereits erwähnten *UML-Profile* erlauben das Definieren von Benutzergruppen (*Stereotypen*), um Modellelementen derselben Metaklasse unterschiedliche Rollen bei seiner Verwendung im Modell zuweisen zu können. Diese Metainformation kann später bei der rechnergestützten Transformation entsprechend verarbeitet werden. Eine mit dem Stereotyp `«EJBEntity»` behaftete Klasse, beispielhaft angeführt, würde in eine

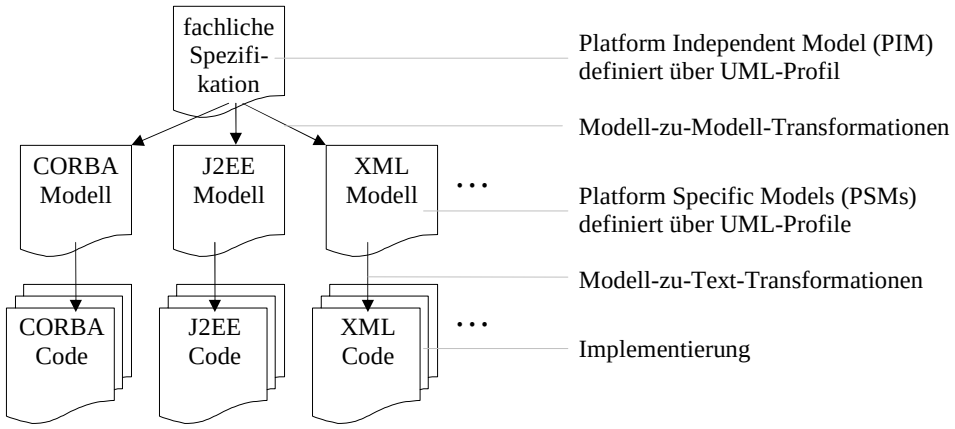


Abbildung 5: Grundprinzip der MDA

Klasse mit EJB-Annotationen übersetzt werden, während eine Klasse ohne Stereotypen einer gewöhnlichen Javaklasse entspräche. Abbildung 4 zeigt schematisch wie ein Stereotyp mit zwei Werten, *Tagged Values* oder einfach *Tags* genannt, definiert und eingesetzt wird. Einschränkende Bedingungen in geschweiften Klammern schließen eine inkonsistente Verwendung der Tagged Values aus.

1.3 Transformationssprachen

Nach [2] definiert sich eine Transformation wie folgt:

Definition. Eine *Transformation* ist das automatische Generieren eines Zielmodells aus einem Quellmodell entsprechend einer Transformationsdefinition, mit Erhalt der Semantik sofern die Sprache des Zielmodells dies zulässt.

Definition. Eine *Transformationsdefinition* ist eine Menge von Transformationsregeln, zusammen beschreiben sie wie ein Modell in der Quellsprache in ein Modell in der Zielsprache transformiert werden kann.

Definition. Eine *Transformationsregel* ist eine Beschreibung, wie eines oder mehr Konstrukte in der Quellsprache in eines oder mehr Konstrukte in der Zielsprache transformiert werden können.

Wie bereits erläutert, sieht die modellgetriebene Architektur (MDA) zwischen den Artefakten der einzelnen Phasen automatische Transformationen vor. Je nach Grad der Formalität des CIM lässt sich bereits dieses teilautomatisiert in ein PIM transformieren. Für gewöhnlich enthält das CIM jedoch lediglich die in menschlicher Sprache geschriebenen Anforderungen mit einigen erläuternden Diagrammen. Die Transformation von PIM nach PSM findet fast immer mit Hilfe einer rechnergestützten Transformation statt. Bei beiden Transformationen handelt es sich sowohl bei Quell- wie auch Zielsprache um Modellbeschreibungen, daher wird diese Art der Umwandlung *Modell-zu-Modell-Transformation* (M2M-Transformation) genannt. Die andere Klasse von Abbildungen generiert aus einem PSM textuelle Artefakte, das kann Quellcode

sein, aber auch Dokumentationen des Codes oder der Anwendung in HTML oder PDF sind als Ausgabe denkbar. Transformatoren, welche aus einem oder mehreren Eingabe-Modellen eine oder mehrere Textdateien als Ausgabe generieren, führen eine sogenannte *Modell-zu-Text-Transformation* (M2T-Transformation) durch. Genaugenommen stellen sie einen Sonderfall der Modell-zu-Modell-Transformationen dar, schließlich sind Programmiersprachen wie Java ebenfalls wohldefinierte Sprachen, welche ein System zu beschreiben vermögen. Es exisitert eine Vielfalt an Sprachen, mit welchen Modell-zu-Text-Transformationen beschrieben werden können, denn leider klafft noch immer eine Lücke in den Spezifikationen der MDA. Zwar lassen sich mit Hilfe der bereits standardisierten QVT-Sprache Modelle in andere Modelle umwandeln (obgleich bis jetzt eine 100% konforme Implementierung fehlt), doch eine einheitliche Sprache, welche aus Modellen Text-Artefakte generieren kann, wurde noch nicht verabschiedet. Nicht erst seit dem Aufruf der OMG im Jahre 2004 [7], Vorschläge für eine derartige Sprache einzusenden, arbeiteten mehrere Institutionen aus Forschung und Wirtschaft an Lösungen, welche teilweise auch als Kandidaten im noch nicht abgeschlossenen Standardisierungsprozess der OMG dienen.

Der Aufruf verlangt vorgeschlagenen Modell-zu-Text-Transformationen gewisse Merkmale ab:

- Die Generierung von Text aus Modellen welche auf MOF 2.0 basieren muss möglich sein.
- Wo anwendbar, sollen bereits existierende OMG Spezifikationen wiederverwendet werden. Insbesondere die schon jetzt standardisierte M2M-Sprache QVT soll als Vorlage dienen. Der Hintergrund ist der, dass textuelle Artefakte als Spezialisierung gewöhnlicher Modelle verstanden werden können.
- Transformationen müssen auf Metaebene des Quellmodells definierbar sein.
- Die Umwandlung beliebiger Modelldaten in Zeichenketten sollen möglich sein.
- Umfangreiche Operationen auf Zeichenketten wie sie z.B. in der Java Application Programming Interface (API) vorhanden sind.
- Die Transformationen auf den Modelldaten sollen mit hart kodiertem Ausgabetext in derselben Datei kombinierbar sein wie aus den meisten Skriptsprachen für Webanwendungen bekannt, z.B. Java Server Pages (JSP).
- Komplexe Transformationen sollen ebenfalls möglich sein.
- Auch mehrere Meta Object Facility (MOF)-Modelle sollen gleichzeitig als Eingabe dienen können.
- Eine sprachliche Unterstützung für Round-Trip-Engineering, bzw. Erkennung/-Schutz modifizierter Codebereiche in wiederholten Generatordurchläufen ist zwar explizit erwünscht, dennoch nicht zwingend vorgeschrieben.

Verfügbare Code-Generatoren lassen sich in zwei Kategorien einordnen [8]: zum Einen gibt es eine auf Schablonen basierende Methode (*template-based method*), zum Anderen eine auf Besuchern basierende Methode (*visitor-based method*).

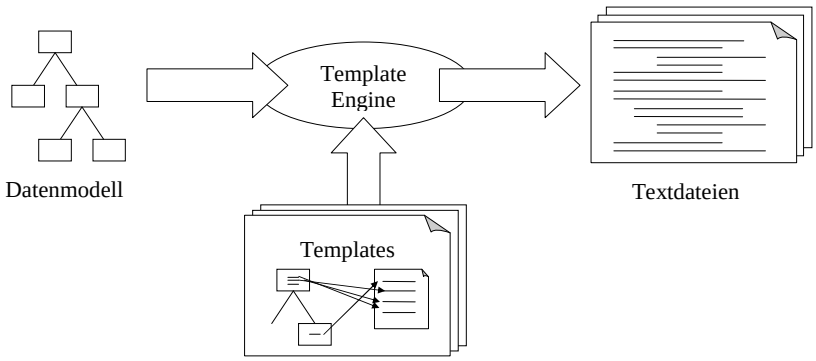


Abbildung 6: Arbeitsweise eines Code-Generators mit Schablonen

Code-Generatoren welche mit **Schablonen** arbeiten sind am Häufigsten anzutreffen. Die Idee ist die, Codefragmente (d.h. sprachliche Beschreibungen) des Zielmodells mit Metamarken (*meta tags*) zu versehen, welche auf Elemente des Quellmodells verweisen. Die Abfrage von Elementen des Quellmodells kann imperativ (z.B. mittels Anweisungen in Java und verfügbarer Java API) erfolgen, oder deklarativ (z.B. OCL-Abfragen). Zusätzlich geben Schablonen an, auf welche Elemente des Quellmodells sie anzuwenden sind. Beim Transformieren wird die Anwendbarkeit einer vorgegebenen Menge von Schablonen auf die Elemente des Quellmodells geprüft. Das Codefragment einer anwendbaren Schablone wird sofort ausgegeben, wobei alle im Code enthaltenen Metamarken aufgelöst werden. Ein Beispiel für eine solche Metamarke ist der Bezeichnername einer Klasse.

Auf der Basis des **Besuchermusters** [9] ist eine alternative Art der Code-Erzeugung möglich. Je Element der internen Struktur des Quellmodells besteht die Möglichkeit eine Methode zu definieren. Die interne Struktur ist i.d.R. eine baumartige Struktur, welche an einen abstrakten Syntaxbaum aus dem Übersetzerbau erinnert. Die Knoten eines solchen Baums können mit einer Tiefensuche durchlaufen werden, und je nach Elementtyp den passenden Besucher aufrufen sofern er definiert worden ist. Dieses Vorgehen ermöglicht die Ausgabe von Text in einen Ausgabestrom mit der Einbeziehung des Kontextes. Verglichen mit der Code-Generierung durch Schablonen ist ein auf Besuchern basierender Code-Generator selbst einfacher zu implementieren, die Implementierung der Besuchermethoden orientiert sich jedoch stark an der Struktur des Quellmodells, während Schablonen es dem Programmierer einer Transformation erlauben, inhaltlich zusammengehörende Teile der Zielsprache in einer Schablone zusammenzufassen. Schablonen verbessern im Allgemeinen die Lesbarkeit einer Code-Generator-Spezifikation. Aus eben diesen Gründen sind auf dem Besuchermuster basierende Text-Generatoren in der MDA höchst selten anzutreffen, daher wird der Typus in der Ausarbeitung nicht weiter vertieft. Klassisches Beispiel eines solchen Generators ist Jamda (`http://jamda.sourceforge.net`).

2 Eine Marktstudie

Dieses Kapitel bietet eine Übersicht über einige Modell-zu-Text-Transformationssprachen, welche in unkommerziellen, quelloffenen MDA-Entwicklungssystemen enthalten sind. In kommerziellen, proprietären Systemen integrierte Transformationssprachen

werden lediglich kurz angesprochen. Die Studie erhebt nicht den Anspruch vollständig zu sein; viel mehr gibt sie einen Überblick über die vielversprechendsten unter den derzeit verfügbaren Sprachen. Eine solche Auswahl kann nicht rein objektiv getroffen werden. Weiter sind die Erörterung von Gemeinsamkeiten und Unterschieden, der praktische Einsatz und die Einbettung in existierende Werkzeuge Gegenstand dieses Kapitels.

2.1 Xpand

2.1.1 openArchitectureWare (oAW) und die Sprache Xpand

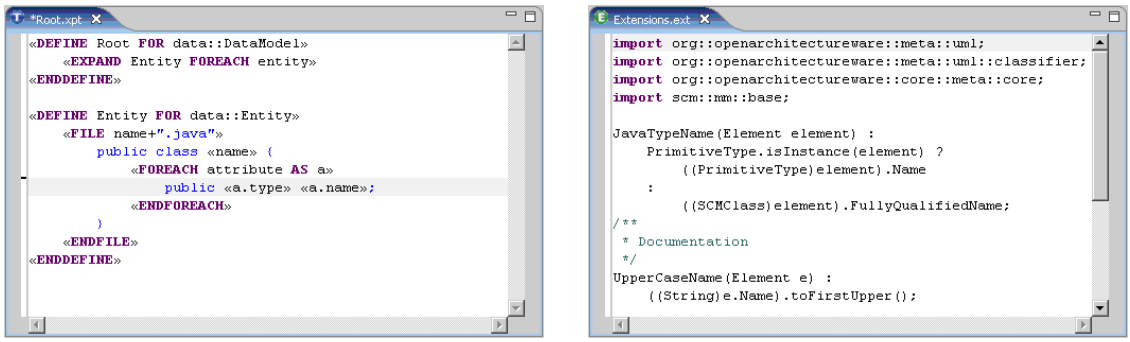


Abbildung 7: Editoren für die Sprachen Xpand und Xtend in Eclipse

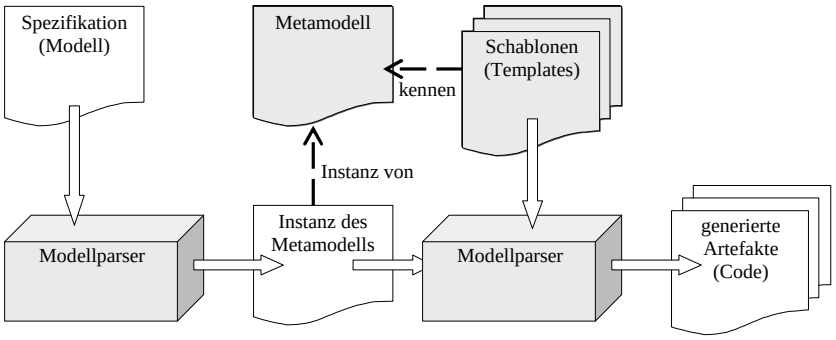


Abbildung 8: Der Aufbau von oAW erinnert an den eines Compilers

Das Entwicklungswerkzeug openArchitectureWare (oAW) ist eine Sammlung aufeinander abgestimmter Werkzeuge und Komponenten zur Unterstützung modellgetriebener Softwareentwicklung. Es baut auf einem modularen Generatorenrahmenwerk auf, das weitgehend mit der MDA kompatibel ist und in Java programmiert wurde. Standards wurden immer dann verwendet, wenn sie für sinnvoll erachtet wurden. Ursprünglich als kommerzielles Projekt *b+m GeneratorFramework* der Kieler Softwarefirma *b+m Informatik* initiiert, wurde das Projekt in der Version 2.0 im Herbst 2003 unter dem Namen *openArchitectureWare* der Open-Source-Gemeinde übergeben. Es genießt mittlerweile einen guten Ruf dank vieler Sponsoren, einer ausgebauten Online-Community, ausführlicher Dokumentation und einer guten Einbindung in die grafische Entwicklungsoberfläche Eclipse. Ab der Version 4.0 ist oAW Subprojekt innerhalb des Generative Model Transformer (GMT) Projektes von Eclipse (`http://www.eclipse.org/`

gmt/oaw). Seine zweite Heimat liegt auf `http://www.openarchitectureware.org`, die Quellen werden auf Sourceforge unter der LGPL Lizenz veröffentlicht. Im Gegensatz zu AndroMDA, dessen Stärke eine „out-of-the-box"-Funktionalität durch eine Vielzahl bereits vorhandener Transformationen ist, setzt oAW vor allen Dingen auf die effizientere Entwicklung eigener Modell-Transformationen, denn oAW bietet Programmierern die volle Unterstützung einer modernen Entwicklungsoberfläche und sie müssen nicht ihre Metamodelle an die Besonderheiten vorgegebener Transformationen anpassen.

Für die Definition eines Metamodells ist ein Meta-Metamodell erforderlich. In oAW kann als Meta-Metamodell das Eclipse-eigene Eclipse Modeling Framework (EMF) verwendet werden. EMF setzt auf Essential MOF (EMOF) auf, einem weitestgehend MOF-kompatiblen Standard von IBM. Alternativ kann ebenfalls das von früheren Versionen bekannte Meta-Metamodell *oAW Classic* verwendet werden.

Der Entwicklungszyklus in oAW gestaltet sich folgendermaßen:

1. Erst durch das Erarbeiten und Verstehen der **Problemdomäne** des Softwaresystems kann anschließend die Domäne strukturiert werden, Regeln und Ausnahmen gefunden werden und ein Glossar entworfen werden. Es hat sich als hilfreich erwiesen, die Domäne als Metamodell zunächst auf Papier zu formalisieren.

2. Das **Metamodell** ist in einem UML-Modellierwerkzeug einzugeben. Das Werkzeug muss das Meta-Metamodell unterstützen (EMOF, Classic oAW). Nach Entwurf der Implementierungsklassen im Modellierwerkzeug müssen die Modelldaten lediglich noch nach oAW exportiert werden. Zur Generatorlaufzeit halten diese Implementierungsklassen das Modell in Form von Instanzen des Metamodells.

3. Die **Einschränkungen** (Constraints) der Metamodellinstanzen müssen entweder in der oAW-eigenen und Object Constraint Language (OCL)-ähnlichen Checks-Sprache formuliert werden, oder aber in echtem OCL unter Verwendung des OSLO-Plugins. Das oAW-eigene *Recipes Framework* hilft bei der manuellen Implementierung der Geschäftslogik. Während dem Generatorlauf werden die Tests instanziiert und danach wird getestet, ob der generierte Code zusammen mit dem manuell erzeugten Code die für das Modell definierten Einschränkungen einhält und gibt direkt in Eclipse Rückmeldung über eventuelle Implementierungs- und Modellfehler. Für nach Java zu transformierende Modelle existieren vordefinierte Rezepttests.

4. **Modell-zu-Modell-Transformationen** können in Java mit den oAW-Classic-Bibliotheken hart kodiert werden, in der funktionalen Sprache Wombat programmiert werden, oder – sofern das Modell auf EMF basiert – in den EMF Transformationstechnologien wie z.B. ATLAS Transformation Language (ATL) spezifiziert werden. Wombat ist fähig, auch Meta-Metamodelle zu transformieren, so existiert bereits eine Transformation von oAW-Classic zu EMF. Eine Implementierung des QVT-Standards ist bald zu erwarten.

5. Mit Hilfe der Xpand-Sprache können unter Eclipse **Modell-zu-Code-Transformatoren** geschrieben werden. Falls hierfür zusätzliche Eigenschaften und Beziehungen benötigt werden sind diese in einer vorgeschalteten Modell-zu-Modell-Transformation hinzufügbar, ohne das Metamodell zu verändern. Das oAW-Rahmenwerk sieht für diese Aufgabe die Sprache Xtend vor. Ein Metamodell welches

z.B. als Quelle für Transformationen in mehrere Code-Artefakte dient, muss somit nicht mit transformationsspezifischen Informationen belastet werden. Das Verwerfen von Informationen ist in Xtend nicht möglich.

6. oAW verwendet zur **Steuerung** seiner internen Abläufe eine *Workflow Engine*. Die Workflow Engine verarbeitet ein eigenes XML Workflow-Schema, das Abhängigkeiten und Parameter von Modellinstanziierung, Modell-Transformationen, Einschränkungen und Codegenerierung für ein Projekt zusammenfasst. Das Schema kann dann von Eclipse oder Ant, einem komfortablen Make-Werkzeug für Javaprojekte, ausgeführt werden.

2.1.2 Eigenschaften der Sprache Xpand

Die Sprache xPand bietet die folgenden Besonderheiten:

- Unterstützung für aspektorientierte Programmierung. Das Programmierparadigma erlaubt die Definition von fachlich zusammengehörigen aber räumlich getrennten Programmbestandteilen an einer zentralen Stelle im Programm. So lässt sich eine Funktionalität welche das gesamte Programm betrifft (ein sog. **Aspekt**) in einer einzelnen Datei implementieren, und zur Übersetzungszeit an vom Programmierer markierten Stellen in das Programm einfügen. Das am häufigsten genannte Beispiel ist die Protokollierungsfunktion einer Software (*Logging*), welche in herkömmlichen Programmiersprachen über das Programm verteilt die Lesbarkeit des Quellcodes verschlechtern würde. Auf xPand bezogen bieten **aspektorientierte Schablonen** die Möglichkeit, gewöhnlichen Schablonen gewisse Aspekte hinzuzufügen, z.B. beim Besuch einer Schablone in einer Ausgabe den Vorgang innerhalb der Schablone zu protokollieren. Diese Funktionalität kann in einer getrennten Datei erfolgen, sofern die eigentliche Schablone per `import`-Befehl eingebunden wird.

- Selbst erstellbare **Modell-Instanziierer** ermöglichen das Einlesen beliebiger Modelle. Es existieren bereits Instanziierer für EMF, eXtensible Meta Language (XML), textuelle Modelle mit JavaCC oder ANTLR Parser als Frontend, diverse UML-Modellierwerkzeuge wie Visio, MagicDraw, Poseidon, Rational Rose und andere.

- Die polymorphe Eigenschaft von xPand erlaubt das **Überladen** von Schablonen. Schablonen müssen sich immer auf ein bestimmtes Element des Eingabemodells beziehen. Beim Parsen des Modells wird für alle von diesem Element erbenden Elemente gleichsam dieselbe Schablone aufgerufen, es sei denn es existiert eine speziellere Schablone dafür, dann wird die speziellste aufgerufen.

- Eine mächtige Sprache für **Ausdrücke** ist in Xpand integriert, mit der umfangreiche Operationen auf den integrierten Basistypen möglich sind.

- Ein Plugin erweitert die integrierte Entwicklungsumgebung (IDE) Eclipse um einen Editor mit farbiger Syntaxmarkierung. Codevervollständigung und Fehlerhervorhebung: Bei der Generierung unverändert in den Zielcode übernommener Text wird blau markiert, Schlüsselwörter werden in Großbuchstaben und in

violetter Farbe angezeigt, Eigenschaften des Metamodells werden automatisch schwarz markiert.

- Xpand unterstützt die Verwendung geschützter Bereiche. Durch das Schlüsselwort `PROTECT` können sie auf ausgewählte Bereiche angewandt werden, und sind später als Metatags mit Kennnummern innerhalb von Kommentaren sichtbar. Die Vergabe von Kennnummern macht sie für den Generator eindeutig zuordenbar und erlaubt das Verschmelzen von modifiziertem mit wiederholt generiertem Code. In der Sprachreferenz [10] wird von einer Verwendung abgeraten, da Änderung und Spezialisierung des Codes durch Vererbung oder besser Muster (Dekorierer, Fabrikmethode, usw.)[9] geschehen sollte und damit leichter in einem eigenen Artefakt untergebracht werden kann.

Die semiformale Referenz zur Sprache ist auf der Projektseite [10] verfügbar.

2.1.3 Die Sprache Xpand am Beispiel

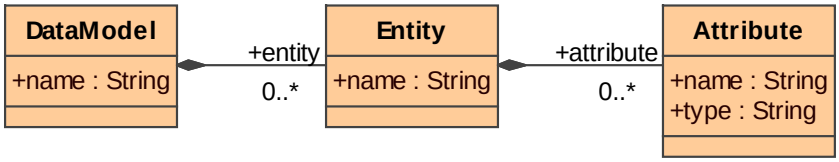


Abbildung 9: Aus diesem Modell sollen Javaklassen generiert werden

Die wichtigsten Befehle der Xpand Sprache lassen sich am Besten an einem Beispiel erklären, das aus einem offiziellen Tutorial der oAW-Macher stammt [11]. Ausgehend von dem einfachen in Abbildung 9 dargestellten Datenmodell sollen Klassenentitäten und enthaltene Attribute auf Javacode abgebildet werden.

Listing 1: Die Datei `root.xpt`

```
1  «DEFINE Root FOR data::DataModel»
2    «EXPAND Entity FOREACH entity»
3  «ENDDEFINE»
4
5  «DEFINE Entity FOR data::Entity»
6    «FILE name+".java"»
7      public class "name" {
8      «FOREACH attribute AS a»
9        // bad practice
10       private "a.type" "a.name";
11   «ENDFOREACH»
12     }
13   «ENDFILE»
14 «ENDDEFINE»
```

Die erste Zeile definiert die zuerst ausgeführte Schablone, die in jeder Transformation enthalten sein muss, ihr Name ist immer `Root`. Sie bezieht sich auf die Klasse `DataModel` des Metamodells. Alternativ ist es auch möglich, den Namensraum `data` zu importieren um Schreibarbeit zu sparen:

```
«IMPORT data»
«DEFINE Root FOR DataModel»
```

```
:
«DEFINE Entity FOR data::Entity»
:
```

Zeile 2 ruft für jede in der Klasse `DataModel` enthaltene Instanz der Klasse `Entity` eine weitere Schablone mit Namen `Entity` auf. Korrekterweise wird in Zeile 5 unter dem selben Namen und im richtigen Modellkontext `data::Entity` eine passende Schablone definiert. Das Skript gibt an, dass eine Datei mit dem Namen des Attributs `name` von `Entity` sowie angehängter Dateiendung zu erstellen ist. In die Datei wird der Kopf der Javaklasse ausgegeben. Anschließend iteriert das Skript in Zeile 8 über alle enthaltenen Attribute und schreibt eine entsprechende Definition in den Ausgabestrom.

Weitergehende Fähigkeiten der Sprache können viel einfacher unter Einbeziehung anderer oAW-Fähigkeiten erklärt werden. Viele beispielhafte Anwendungen sind sehr aufschlussreich in Schritt-für-Schritt-Anleitungen erklärt und liegen auf `http://www.eclipse.org/gmt/oaw/doc` bereit.

2.2 Velocity Template Language (VTL)

2.2.1 AndroMDA und die VTL

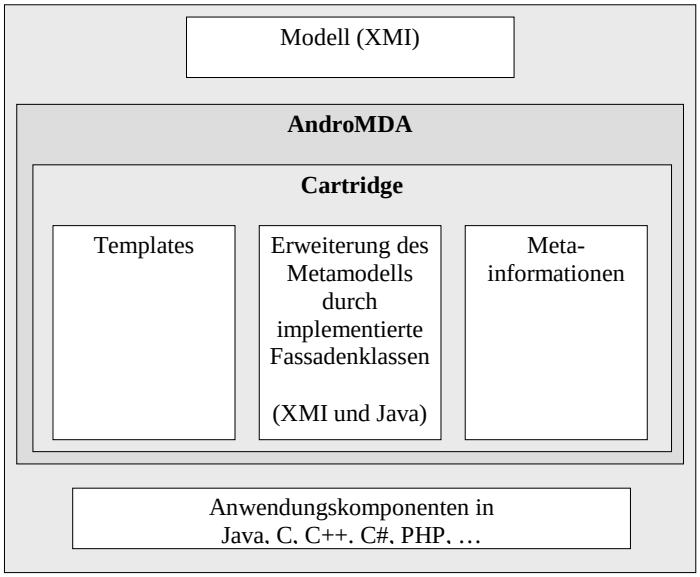


Abbildung 10: Die Architektur von AndroMDA

Bei AndroMDA handelt es sich, wie auch bei oAW, um ein vollständig in Java entwickeltes MDA-Werkzeug. Die Quellen von AndroMDA unterliegen rechtlich der BSD-Lizenz und werden auf SourceForge verwaltet. Sämtliche Dokumentation, die Foren und weitere Informationen zum Projekt können unter `http://www.andromda.org` abgerufen werden. AndroMDA liegt momentan stabil in der Version 3.1 vor. Das Werkzeug hält sich an die Richtlinien der MDA und nimmt dabei die Möglichkeit wahr, auf PSMs als Zwischenschritt in der Entwicklung zu verzichten. Stattdessen ist die fachliche Logik der PIMs um zusätzliche Eigenschaftswerte zu erweitern, welche später während des Codegenerierungsprozesses die interne automatische Generierung von PSMs ermöglichen. Eigenschaftswerte weisen den Stereotypen plattformspezifische Eigenschaften zu.

Aus solchen PIMs ist es möglich über die Verwendung von speziellen AndroMDA Plugins (*Cartridges* genannt) die Artefakte der zu generierenden Anwendung zu erstellen. Die mitgelieferten Cartridges unterstützen Architekturen wie Spring, EJB, .NET, Hibernate und Struts. Einige Funktionen auf welche die Cartridges zurückgreifen sind austauschbar gehalten und lassen damit ein hohes Maß an Flexibilität und Erweiterbarkeit zu. Es gibt Schnittstellen für Translation Libraries, Metafassaden, Repositories und Template Engines, der Zweck dieser Pluginschnittstellen wird im Laufe des Kapitels erklärt.

AndroMDA verwendet für seine Anwendung bevorzugt Maven (`http://maven.apache.org`), ein System ähnlich Ant zur Steuerung des Erstellungsprozesses von Software. Mit Maven lässt sich der Generierungsprozess interaktiv steuern. Einstellungen für statische wie auch durch die Interaktivität bedingte dynamische Abhängigkeiten sind nach vorgegebenem XML-Schema beschreibbar.

Erwähnenswert ist die Arbeit an der Anbindung AndroMDAs an eine grafische Benutzeroberfläche (Graphical User Interface (GUI)). Unter dem Projektnamen *Android* wird an Eclipse-Plugins gearbeitet, um AndroMDA nahtlos in eine moderne Java-Entwicklungsoberfläche einzubetten, inklusive den vielen Vorzügen die Eclipse mit sich bringt: Syntax Highlighting, Code Completion, Fehleranzeige, Projekt- und Konfigurationsverwaltung, usw.

Leider fehlen bei AndroMDA Modell-zu-Modell-Transformationen, in der Version 4 soll über die Sprache ATLAS Transformation Language (ATL) diese Möglichkeit hinzugefügt werden. AndroMDA bietet keine Unterstützung für iterative Entwicklung, weder Round-Trip-Engineering noch die Fähigkeit alten Code mit neuem zu verschmelzen fehlt.

Der Entwicklungsvorgang unter AndroMDA lässt sich grob in die folgenden Schritte einteilen:

- Zuerst ist die **fachliche Logik** zu erörtern. Das Fachklassenmodell der geplanten Anwendung wird in UML modelliert und als XML Metadata Interchange (XMI)-Datei exportiert. Da der XMI-Standard gewisse Freiheiten erlaubt, und zudem unterschiedliche Versionen des Formats existieren, ist der Austausch auch unter standardkonformen Modellierwerkzeugen nicht immer möglich. Die Entwickler von AndroMDA empfehlen die Verwendung der kommerziellen Werkzeuge Poseidon oder MagicDraw. Weiter gilt es die Modellarchitekturvariante zu beachten. Zwar besitzt AndroMDA einen Pluginmechanismus, der unterschiedliche Varianten ermöglicht, doch ist zum jetzigen Zeitpunkt nur das Laden/Speichern von MOF-1.4-Modellen über die Modell-API MDR (`http://mdr.netbeans.org`) möglich, während beispielsweise oAW die von IBM ins Leben gerufene Modell-API EMF verwendet. Die EMOF-Architektur bildet den Kern der EMF-Application Programming Interface (API) und wurde im Mai 2006 als eine der Meta Object Facility (MOF)-Varianten von der OMG anerkannt. Die Entwickler planen die Integration von UML 2.0 in der kommenden Version.

- Nach der Erstellung des Platform-Independent Model (PIM) kümmert sich der Softwarearchitekt um die Beschreibung der Zielplattform. Anhand dieser Erkenntnisse kann das bereits erstellte UML-Profil um plattformspezifische **Eigenschaftswerte** erweitert werden.

- Existiert bereits eine Cartridge für die beabsichtigte Plattform, so gilt es, diese

lediglich noch zu konfigurieren und die nächsten beiden Schritte dürfen übersprungen werden. Ansonsten muss eine eigene geschrieben werden, wie in den beiden nachfolgenden Schritten erläutert.

- Für eine neue Cartridge müssen zunächst **Metafassaden** erstellt bzw. vorhandene modifiziert werden. Metafassaden umhüllen die ursprünglichen Metaklassen des PIM mit neuen Metafassadenklassen. Die vorhandenen Klassen werden dabei mit Klassen desselben Typs umhüllt, welche die Fassadenklassen um zusätzliche Attribute, Assoziationen und Operationen erweitern. Die Operationen sind anschließend in Java manuell zu implementieren inklusive einer Berechnungslogik die den neuen Attributen Werte zuweist. Metaklassen und Metafassadenklassen stellen alle Informationen bereit, auf deren Basis Code generiert werden kann. Es bleibt anzumerken, dass die Metafassaden auch Constraints in Form von OCL-Ausdrücke auswerten können, um die Gültigkeit des Modells auf Metaebene sicherstellen.

- Nach diesem Umformungsprozess bilden Schablonen das erweiterte Modell auf Textartefakte ab. Eine **Template Engine** klinkt sich in die Cartridge ein, und bietet je nach Engine eine Transformationssprache an, in welcher die Modell-zu-Text-Transformation formuliert wird. AndroMDA schlägt hier die Sprache Velocity vor, und bietet nebenbei noch eine Template Engine für die ähnlich mächtige Sprache Freemarker an. Die Unterstützung weiterer Sprachen ist in naher Zukunft absehbar, so ist eine Anbindung an MOFScript durchaus denkbar.

- Die Cartridge wird **konfiguriert**.

- Der **Generator** wird auf der Grundlage der Konfiguration gestartet.

2.2.2 Eigenschaften der Sprache VTL

Die *Velocity Template Engine* verwendet die (VTL), ein Projekt unter den Fittichen des Apache Jakarta Projektes. Die Sprache ist samt Dokumentation und Tutorials unter `http://jakarta.apache.org/velocity` verfügbar. Velocity verwendet Java und Schablonen zur Generation von Text, auf der Heimatseite wird sie als Alternative zu den Sprachen JSP und PHP beschrieben. Alle drei genannten Sprachen sind Skriptsprachen zur Beschreibung von Code mit innerhalb von Metamarken eingebetteten dynamischen Ausdrücken und Anweisungen. Wenn auch der ursprünglich gedachte Verwendungszweck bei der Generierung dynamischer Webseiten liegt, so beschränkt sich ihr Anwendungsfeld nicht nur speziell auf HTML-Code.

Der Sprachumfang von Velocity umfasst einfache Berechnungen auf numerischen und logischen Variablen und Zeichenketten, Verzweigungen, Schleifen, das Einbinden von Text und Velocitycode aus fremden Dateien und die Definition von Makros. Diese Eigenschaften verleihen der Skriptsprache nicht außerordentlich viel Flexibilität, erst der Aufruf von Javamethoden und der Zugriff auf Javaobjekte gibt der Sprache mehr Mächtigkeit. Welche Skripts verwendet werden, und in welche Datei das Generat eines Skriptes bei der Anwendung innerhalb einer Cartridge ausgegeben wird, muss zusammen mit Informationen über die verwendeten Metafassaden in den XML-Deskriptoren der Konfigurationsdatei einer Cartridge, `cartridge.xml`, angegeben werden.

2.2.3 Die Sprache VTL am Beispiel

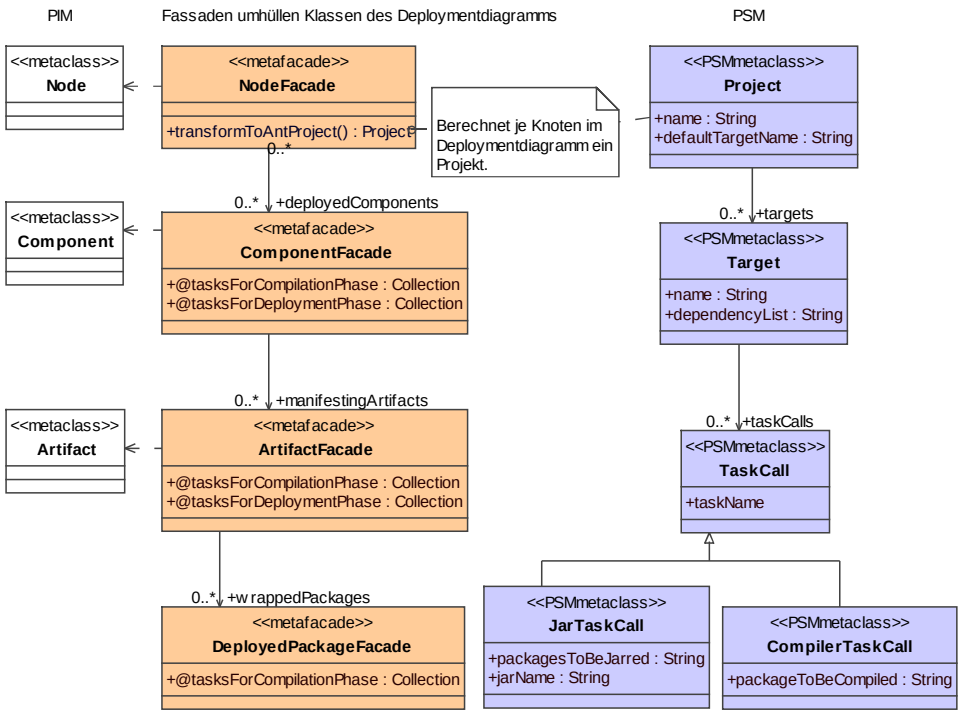


Abbildung 11: Der AST wird von den Metafassen aus dem PIM berechnet

Die Erstellung einer vollständigen Cartridge für die AndroMDA-Engine würde den Rahmen dieser Ausarbeitung sprengen, Interessenten an einer vollständigen Anleitung seien auf das Tutorial „10 steps to write a cartridge“ auf der AndroMDA-Homepage verwiesen. In der Anleitung wird in zehn Schritten gezeigt, wie der Weg zu einer Cartridge aussieht, welche aus einem UML-Verteilungsdiagramm eine Ant-ähnliche Buildkonfiguration generiert. Dennoch soll an Hand eines Ausschnitts aus dieser Anleitung die Anbindung von Velocity an AndroMDA und die syntaktischen Elemente der Sprache Velocity erklärt werden. Das unten gelistete Skript dient der Ausgabe der Ant-ähnlichen Buildkonfiguration im XML-Format. Der Kontext, d.h. in welche Datei die Ausgabe umgeleitet wird und welche Metafassade für die Zusammenstellung des abstrakten Syntaxbaumes im Sichtbarkeitsbereich der Velocity-Template liegt, muss in der Datei `cartridge.xml` angegeben werden. Die Schnittstelle zwischen Skript und Modell befindet sich in der 1. Zeile der Datei, hier wird auf einer Instanz der Klasse `node` die Methode `transformToAntProject` aufgerufen. Durch Rekursion über das Fassaden/Wrapper-Modell berechnet sie notwendige Attribute aus dem Platform-Independent Model (PIM) und gibt den (AST) bzw. das Platform-Specific Model (PSM) in Baumstruktur zurück. Die Klassendiagramme in Abbildung 11 verdeutlichen dies. Wer sich für den Javacode interessiert der die Fassadenlogik implementiert, der sehe sich das vollständige Tutorial an.

Velocity-Quelltext ist recht einfach zu verstehen. Metaanweisungen beginnen mit einer Raute, Variablen werden durch ein vorangestelltes Dollarzeichen gekennzeichnet, sie sind nicht getypt. Interessant ist, dass sie auch Referenzen auf Javaobjekte halten können. Nach der Zuweisung der Datenstruktur an die Variable `$project` kann im Javastil mit dem Punktoperator auf enthaltene Attribute zugegriffen werden. Kom-

mentare werden durch zwei Rauten hintereinander eingeleitet.

Listing 2: Die Datei `AntProject.vsl`

```
#set ($project = $node.transformToAntProject())

<project name="$project.name" default="$project.defaultTargetName">

#foreach ($target in $project.targets)
#if ($target.dependencyList)
  <target name="$target.name" depends="$target.dependencyList">
#else
  <target name="$target.name">
#end

#foreach ($task in $target.taskcalls)
#if ($task.taskName == "javac")
  <javac package="$task.packageToBeCompiled" />
#elseif($task.taskName == "jar")
  <jar name="$task.jarName">
#foreach ($package in $task.packagesToBeJarred)
    <package name="$package" />
#end
  </jar>
#end
#end
  </target>
#end
</project>
```

2.3 MOFScript

2.3.1 Das Eclipse GMT Projekt und die Sprache MOFScript

Eclipse ist mehr als nur eine Entwicklungsumgebung für Java, das Eclipse-Projekt per se (`http://www.eclipse.org/eclipse`) stellt im Subprojekt *Plattform* ein Rahmenwerk zur Erstellung von Rich Client Anwendungen bereit. *Rich Client Anwendungen* sind Anwendungen mit einem kleinen Kern und funktionaler Erweiterbarkeit durch eine ausgeprägte Plugin-Komponenten-Technologie. Ein weiteres Subprojekt, *Java Development Tools (JDT)*, erweitert die Eclipse-Plattform durch eine Reihe von Plugins zur vollständigen Java integrierte Entwicklungsumgebung (IDE). Die anderen Projekte kümmern sich vor allen Dingen um das Hinzufügen neuer Technologien zur Java IDE. Das *Eclipse Modellierungsprojekt* (`http://www.eclipse.org/modeling`) ist im hiesigen Kontext besonders interessant, besteht sein Sinn doch im Vorantreiben modellgetriebener Softwareentwicklung durch das Bereitstellen einer Umgebung für experimentelle Projekte der Forschung. Es beherbergt auch die wichtigen Unterprojekte EMF und Graphical Modeling Framework (GMF). Diese Rahmenwerke stellen Kerntechnologien bereit, u.A. für die Projekte EMF Technologies (EMFT) und GMT.

GMT (`http://www.eclipse.org/gmt`) besteht aus Prototypen für die modellgetriebene Entwicklung mit Eclipse, dazu gehören das bereits besprochene Werkzeug openArchitectureWare und die Referenzimplementierung der MOFScript Sprache.

Die Skriptsprache MOFScript bildet eine Ausnahme unter den Modell-zu-Text-Transformationssprachen, da sie nicht als Teil eines konkreten MDA-Werkzeugs sondern völ-

lig eigenständig entwickelt wird. Das Projekt ist eine Reaktion auf den Aufruf der OMG, Vorschläge für einen zukünftigen M2T-Standard einzusenden. Genaueres zum Standardisierungsprozess und die umrissenen Eigenschaften des baldigen Standards wurden bereits in Kapitel 1.3 aufgezählt.

Sobald der Standard in Kraft treten wird, sind die Entwickler bestrebt, die Sprache MOFScript samt Implementierung daran anzugleichen. Im von der EU unterstützten Projekt MODELWARE wird MOFScript von SINTEF ICT entwickelt, einer Forschungseinrichtung mit Technologietransfer an einer schwedischen Universität. Neueste Informationen zur Sprache und ihrer Spezifikation [12] sind unter `http://www.modelbased.net/mofscript` verfügbar. Die Referenzimplementierung kann unter `http://www.eclipse.org/gmt/mofscript` heruntergeladen werden. Sobald der Quellcode in einem reifen Stadium ist, soll er ebenfalls an dieser Stelle frei zugänglich gemacht werden.

2.3.2 Eigenschaften der Sprache MOFScript

Für einen guten Einstieg ist der „MOFScript User Guide“ [13] empfehlenswert, ein ausführlicher Bericht über die Spracheigenschaften. Im gegenwärtigen Zustand macht sich die Implementierung diese Merkmale zu eigen:

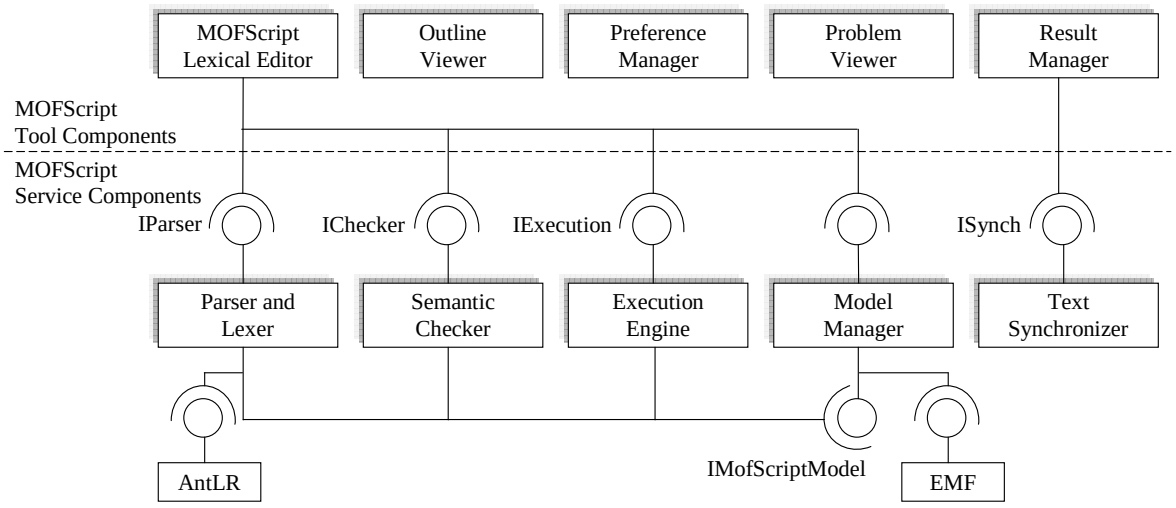


Abbildung 12: Komponentenarchitektur der MOFScript Implementierung

- **Modularer Aufbau**: Wie in Abbildung 12 skizziert, lassen sich die Komponenten logisch in Werkzeug- und Dienstkomponenten einteilen. Auf der Grundlage des Model View Controller (MVC)-Musters [9] gedacht sind die Werkzeugkomponenten für die Sicht d.h. die Kommunikation mit der Oberfläche zuständig. Das Modell wird von der Modellverwaltung repräsentiert, welche ihre Modelle mit Hilfe des EMF einliest. Bei den restlichen Komponenten handelt es sich um Steuerkomponenten, sie übernehmen das Zerteilen, die semantische Überprüfung und Ausführung des eingelesenen (Meta-)Modells. Der Textsynchronisierer versucht Modell und generierten Text konsistent zu halten.

- **IDE-Einbettung**: Plugins sorgen für eine gute Anbindung an die Eclipse-Oberfläche mit der Möglichkeit, Code zu editieren, zu übersetzen und auszuführen. Der Editor bietet Codevervollständigung durch Drop-Down-Kasten, eine Visualisierung des Codebaumes und farbliche Markierung unterschiedlicher syntaktischer Elemente sowie ein eigenes Konsolefenster für Statusinformationen und Logging. Beim Speichern im Editor oder auf expliziten Knopfdruck wird die geladene Datei automatisch übersetzt und auf syntaktische wie semantische Fehler überprüft. Ein Eigenschaften-Dialog erlaubt notwendige Pfadeinstellungen.

- Orientierung an der formalen MOFScript **Spezifikation**: Da die MOFScript Spezifikation selbst als Vorlage beim Standardisierungsprozess der OMG dient und ihr scheinbar besondere Beachtung geschenkt wurde, unterscheidet sich MOFScript nur gering in der Funktionalität von MOF2Text, dem derzeitigen Zwischenergebnis im Standardisierungsprozess. Vielmehr sind die meisten Unterschiede syntaktischer Art, beispielsweise kann in MOF2Text alternativ der Code im Text durch Metatags (z.B. `<%...%>`) gekennzeichnet werden während MOFScript die Kennzeichnung von Text im Code vorschreibt. Zu den funktionalen Unterschieden gehört, dass MOF2Text auf die praktische `foreach`-Anweisung MOFScripts verzichtet. MOF2Text ist eine Verschmelzung aus den der OMG vorgeschlagenen vier Kandidaten (nachzulesen im MOFScript Foliensatz auf `http://www.modelware-ist.org`).

- Aber auch MOFScripts **Implementierung** unterscheidet sich nur marginal von seiner Spezifikation. Das liegt hauptsächlich daran, dass sich MOFScript syntaktisch und semantisch an der bereits standardisierten Modell-zu-Modelltransformationssprache Query/Views/Transformation (QVT) und an der Object Constraint Language (OCL) orientiert. MOFScript erbt von den operationalen Abbildungen von QVT, es übernimmt Abbildungsregeln unter Weglassung des Schlüsselwortes `mapping` samt einer Hauptfunktion `main()` und OCL-konformen Ausdrücken. Die Abbildung 13 zeigt in roter Markierung die Änderungen am Regelschema der QVT auf.

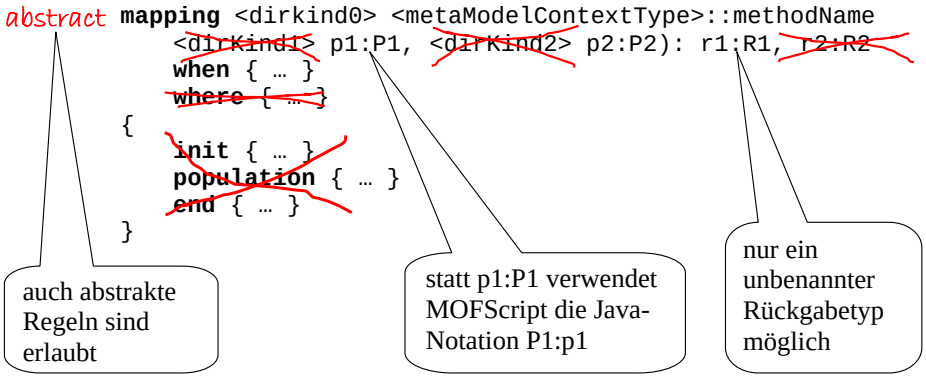


Abbildung 13: MOFScript vereinfacht Regel-Templates der QVT

- Das Schlüsselwort `when` erlaubt das optionale Anheften eines **Wächterausdrucks** in OCL an eine Regel. Zum Beispiel wird die Regel

```
uml.Class::Class2Java() when{self.getStereotype='Entity'}
```

nur dann ausgeführt, wenn die im Quellmodell definierte Klasse den Stereotyp «Entity» besitzt.

- Die Sprache besitzt ein abstraktes **Dateikonzept** um die Textausgabe zu realisieren. Mit `file name (strPfad)` kann einer Variablen ein Dateistrom zugeordnet werden. Das Schreiben in einen solchen benannten Dateistrom erfolgt über den Ausgabebefehl `print(name, text)`. Nebenbei kann auch ein unbenannter Dateistrom erstellt werden, hierbei handelt es sich dann um den Standardausgabestrom. In diesen lässt sich mit `print (text)` schreiben, auch der Text der innerhalb der Marken <%...%> eingebettet ist, wird in diesen Strom ausgegeben.

- **Ausdrücke** in MOFScript stellen Spezialisierungen von OCL-Ausdrücken dar. Deren Funktionalität wurde für das Handhaben von Dateiströmen, Textausgabe usw. erweitert.

- MOFScript enthält mehrere **Funktionsbibliotheken**: Darin enthalten sind Funktionen für die Zeichenkettenmanipulation und für das Formatieren von Text (`toLower`, `size`, `subString`, `indexOf`, `firstUpper` etc.), Operationen auf UML2, die Manipulation von XML-Daten wie auch die Definition der komplexen Datentypen Liste und Wörterbuch (Dictionary, Hashmap).

- Die Sprache zeichnet sich durch einen **einfachen Aufbau** aus. Wenige und unkomplexe Sprachkonstrukte, die Orientierung an bereits existierenden, imperativen, prozeduralen Skriptsprachen machen die Sprache leicht erlernbar und produktiv in der Anwendung.

- Auf Wunsch des Programmierers ist die **Typprüfung** von Variablen möglich: Um die Typprüfung zu aktivieren, muss lediglich bei der Definition der Variablen ihr entsprechender Datentyp angegeben werden, beispielsweise `var myStr :String` statt `var myStr`.

- Neben Variablen sind auch **Konstanten** definierbar, statt `var myStr` ist dann `property myStr` zu schreiben. Die initialisierten Werte können während der Ausführung nicht geändert werden. Variablen und Konstanten können sowohl *global* auf Transformationsebene als auch *lokal* auf Regelebene definiert und zugreifbar sein.

- Der Aufruf von **externem Javacode** ist möglich: Solcher Code muss in statischen Methoden enthalten sein, oder aber in Methoden, deren Klasse einen Standardkonstruktor besitzt. Der Hilfsfunktion `java()` sind Klassenname, Klassenpfad und Methodenname zu übergeben. Optional können Parameter in Form einer geordneten Liste übergeben werden.

- Standardmäßig ist der gesamte generierte Text vor einer manuellen Änderung geschützt. Textbereiche, welche der Programmierer später ändern darf bzw. soll, können bei der Generierung durch das Schlüsselwort `unprotect` gekennzeichnet werden. Ein ungeschützter Bereich spaltet den umliegenden geschützten Bereich in zwei neue auf. Beginn und Ende der geschützten Bereiche werden vom Generator im Augabetext durch spezielle eindeutige Metamarken innerhalb Kommentaren gekennzeichnet. Es sind spezielle Begrenzungszeichen für Kommentare

(in Java z.B. `/*...*/`) spezifizierbar, in die der Generator die Metamarken unterbringt. Derart **ungeschützte Bereiche** werden bei einer erneuten Generierung nach Modifikation des Textes beibehalten. Ebenfalls können Änderungen am Modell auf die geschützten Bereiche des Zielmodells „abgebildet“ werden. Für die Synchronisation in beide Richtungen ist die Komponente *Text Synchronizer* (siehe Abbildung 12) zuständig.

- Das **Vererbungskonzept** erlaubt die Spezialisierung von Transformationen durch Erweiterung, d.h. die Regeln einer vorhandenen Transformation können durch weitere Regeln erweitert/überladen werden. Eine Regel, welche sich auf eine speziellere Klasse des Metamodells als eine andere Regel bezieht, wird bevorzugt angewandt.

Die Entwicklung der Sprache ist noch nicht abgeschlossen. So ist das Einlesen mehrerer Eingabemodelle in einem Durchlauf zwar geplant, aber noch nicht implementiert. Die Spezifikation und Implementierung von Lösungen zu Roundtrip-Engineering und Reverse-Engineering liegt für die Entwickler MOFScripts außerhalb des Rahmens einer Einreichung an die OMG.

2.3.3 Die Sprache MOFScript am Beispiel

Eines der mit MOFScript mitgelieferten Beispielskripte ist eine Transformationsschablone von UML2 nach HTML. Nur die einfachen Elemente Paket, Klasse und Attribut werden in HTML-Code ausgegeben. Basis aller Transformationen ist die Anweisung `texttransformation` mit Name der Schablone und Klasse der einzulesenden Modelle (Zeile 2). Im hier gebrauchten Kontext beschränkt sich das Metamodell auf einen winzigen Ausschnitt (Abbildung 14).

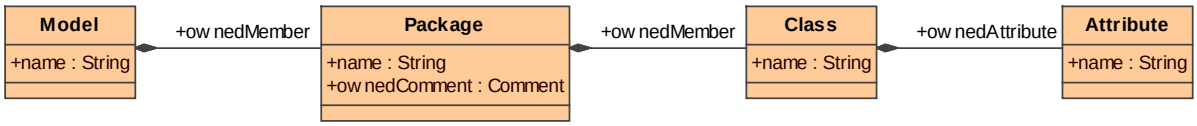


Abbildung 14: Ausschnitt aus dem UML Metamodell

Anschließend folgt die Startregel `main` die den Einstiegspunkt angibt (Zeile 4). `uml.Model` bezeichnet hier den Typ der Elemente im Modell bei denen das Traversieren beginnt. In Zeile 5 statuiert das Skript den Namen der Ausgabedatei, das soll der Modellname in Kleinbuchstaben mit angehängter Endung `.html` sein. Der Object Constraint Language (OCL)-ähnliche Ausdruck `self.name` repräsentiert die Eigenschaft `name` des Modellelementes selbst. MOFScript erlaubt auch die Definition von Hilfsmethoden wie die Methode `stylesheetLink()` in Zeile 60. Sie bezieht sich auf keinen konkreten Elementtyp sondern auf das Modul und kann von überall aus aufgerufen werden. Sehr elegant ist die Iterierung über die Mitglieder eines Elementes mit `foreach` (Zeilen 9-11). Zeile 36 zeigt das Umleiten des Standardausgabestromes in eine neue Datei, hier die im Paket enthaltenen Klassen, welche von der Paketübersicht aus verlinkt werden. Im Gegensatz zu den eingebetteten Ausgabezeichenketten ist auch die Ausgabe mittels `print`/`println`-Funktion möglich wie Zeile 21 demonstriert. In Zeile 39 sehen wir wie Whitespaces auch indirekt an die Ausgabe gesendet werden können: `nl(x)` fügt x Zeilenumbrüche ein, `tab(x)` würde x Tabulaturen einfügen. Bei der Augabe von schön

formatiertem Code ist dieses Hilfsmittel besonders nützlich, da es den Skriptcode unleserlich machen würde, würde man die Formatierungen direkt in das Skript einbinden. Die Sprache Xpand geht noch einen besseren Weg, indem sie Formatierungen während der Generierung ganz außen vor lässt und im Anschluß die Ausführung eines sog. „Code Beautifiers“ bzw. „Pretty Printers“ vorsieht.

Listing 3: Die Datei `uml2html.m2t`

```
1  /* MOF Script Transformation: Simple transformation from UML 2 to HTML
       */
2  texttransformation uml2Html(in uml:"http://www.eclipse.org/uml2/1.0.0/
       UML")
3
4    uml.Model::main() {
5      file (self.name.toLower() + ".html")
6      <%<html><head><title>%> self.name <%</title>%>
7      stylesheetLink()
8      <%<body>%>
9      self.ownedMember->forEach(p:uml.Package) {
10       p.package2html();
11     }
12     <%</body></html>%>
13   }
14
15   /* package2html */
16   uml.Package::Test() {
17   }
18   uml.Package::package2html() {
19     <%<h1>Package %> self.name <%</h1><p>Description: %> self.
           ownedComment <%</p>
20     <hr><h2>The classes</h2><ul>%>
21     print("");
22     self.ownedMember->forEach(c:uml.Class) {
23       c.class2html_link();
24       c.class2html_detail();
25     }
26     <%</ul>%>
27   }
28
29   /* class2html */
30   uml.Class::class2html_link() {
31     <%<li>Class <a href="%> self.name <%.html"> %> self.name <% </a></
           li>%>
32   }
33
34   /* uml.Class::class2html_detail */
35   uml.Class::class2html_detail() {
36     file (self.name.toLower() + ".html")
37     self.class2htmlHeader ()
38     <%Attributes:<br><ul>%> nl
39     self.ownedAttribute->forEach(p:uml.Property) {
40       <%<li>%> p.name <% : some type </li>%>
41     }
42     <% </ul> %>
43     self.class2htmlFooter()
44   }
45
```

```
46  // class2htmlHeader
47  uml.Class::class2htmlHeader() {
48    print("<html><head><title> Class " + self.name + "</title>")
49    stylesheetLink()
50    println("</head><body>")
51  }
52
53  // class2htmlFooter
54  uml.Class::class2htmlFooter() {
55    println("</body>")
56    println("</html>")
57  }
58
59  // Stylesheetlink
60  module::stylesheetLink() {
61    <%<link rel="stylesheet" type="text/css" href="stylesheet.css"
          title="Style">%>
62  }
```

2.4 Java Emitter Templates (JET)

2.4.1 Das Eclipse EMFT Projekt und die JET

Innerhalb der Eclipse-Projekthierarchie liegt das in Kapitel 2.3.1 erwähnte *Eclipse Modeling Framework Technology* Projekt, ein Schwesterprojekt des GMT-Projektes mit dem Bestrebnis das Modellierrahmenwerk EMF in konkreten Anwendungen einzusetzen. Eine existierende Anwendung ist die auf Schablonen basierende Generatorsprache Java Emitter Templates (JET) mit JETEditor. *JETEditor* nennt sich eine Sammlung von Plugins welche die Erstellung von JET-Skripts innerhalb der Eclipse IDE ermöglicht. Der bereitgestellte Editor wartet mit den Features Syntaxfärbung, Fehlermarkierung und Codevervollständigung auf.

Die JET dienen in der Regel dazu, dem Javaprogrammierer das Schreiben repetitiven Codes zu ersparen. Nicht unerhebliche Teile einer Anwendung bestehen aus sogenanntem „Klebercode“, das ist von der Technologie der Zielplattform abhängiger Quelltext, welcher den fachlichen/funktionalen Teil der Anwendung in die Zielplattform integriert, oder anders formuliert, von den technischen Aspekten der Zielplattform abstrahiert. Im Fall des Verteilens einer J2EE-Anwendung kann ein Satz von JET-Schablonen Deployment-Code für unterschiedliche J2EE-Server (JBoss, BEA Weblogic, IBM Websphere,...) generieren.

Der modellbasierte Entwicklungsprozess in EMF gestaltet sich so:

- Zuerst ist das **Modell** zu modellieren. Hierfür gibt es drei unterschiedliche Wege. Das Modell kann in einem UML-Editor erstellt werden, welcher das EMF-eigene Ecore-Format beherrscht. Omondo z.B. stellt einen EMF-fähigen Editor für die Eclipse-Umgebung zur Verfügung. Eine Möglichkeit ist ein eXtensible Meta Language (XML)-Schema zu erstellen, wobei in XSD nur gerichetete Assoziationen zwischen Klassen existieren können. Die andere Möglichkeit ist das Modell in Java mit Annotationen einzugeben, beispielsweise wird mit der Zeile

  ```
  /** @model type="Topic" containment="true" */ List getMembers();
  ```

innerhalb einer Klasse eine unidirektionale Referenz auf die Klasse `Members` gelegt.

- In Eclipse wird nun ein spezielles **EMF-Projekt** erstellt, in das das Modell importiert wird.
- Nun kann das EMF-Modell direkt **ediert** werden. Auch indirekte Änderungen an den Quellen (UML, XSD oder annotierter Java-Code) werden automatisch auf das EMF-Modell übertragen unter Beibehaltung der direkt durchgeführten Änderungen.
- Ist man mit dem Reifegrad des Modells zufrieden, kann aus dem Modell eine Implementierung in Java auf Basis einer abstrakten Fabrik [9], oder aber ein vollständiger Editor in Form eines Eclipse-Plugins generiert werden. Der Javacode erlaubt das Laden, Speichern und Ändern der Modelle mit den Funktionen des EMF-Rahmenwerks. Im Editor sind diesselben Aktionen möglich, das Modell wird in einer baumartigen Struktur visualisiert.
- Das Modell nun in Text zu transformieren ist die Rolle der JET. Wie die Schablonen dafür auszusehen haben, wird im nächsten Kapitel dargestellt.
- Mit JET können beliebige Textartefakte generiert werden: Java Server Pages (JSP), XML, XMI, Java und C++, um nur einige zu nennen. Soll der Generator Javacode erstellen, bietet sich die Option, **JMerge** zu verwenden. JMerge sorgt dafür, dass bereits erstellter Quelltext mit neu generiertem verschmolzen wird. Ähnlich wie bei XDoclet erlauben Annotationen innerhalb von Kommentaren vor Methoden, diese Methoden besonders zu kennzeichnen. Ein `@generated` könnte so Methoden kennzeichnen, welche beim Neuerstellen einfach überschrieben werden. Möchte der Programmierer manuelle Modifikationen permanent machen, entfernt er dieses Tag. Funktionsweise, Auftreten und Aussehen solcher Tags wird in einer XML-Konfigurationsdatei festgelegt. Neben Methoden sind auch andere Elemente von Javacode kennzeichenbar.

2.4.2 Eigenschaften der JET

Die Java Emitter Templates können ihre Verwandtschaft mit den Java Server Pages (JSP) nicht leugnen: JETs Syntax ist eine Teilmenge der JSP mit Erweiterungen, beide müssen zur Ausführung zunächst in Javacode überführt werden, und der eigentliche Zweck beider Sprachen liegt darin, die Sicht von Modell und Steuerung zu separieren. In beiden kann auf externe Javamethoden zugegriffen werden, und die Struktur beider Sprachen richtet sich nach der Struktur des Generats, wobei die dynamischen Inhalte zwischen Marken eingebettet sind. An dieser Stelle ist JET jedoch flexibler, indem es das Umdefinieren dieser Marken (standardmäßig `<%...%>`) erlaubt, damit kann gerade JSP-Code mühelos erstellt werden. Im Gegensatz zu anderen Transformatoren müssen JET-Schablonen erst in einen ausführbaren Code übersetzt werden, um mit ihnen Text generieren zu können. Diese Tatsache sorgt oft für anfängliche Verwirrung. Wie JET genau arbeitet, zeigt die Abbildung 15.

Folgende Punkte erläutern kurz den Aufbau der Skriptsprache:

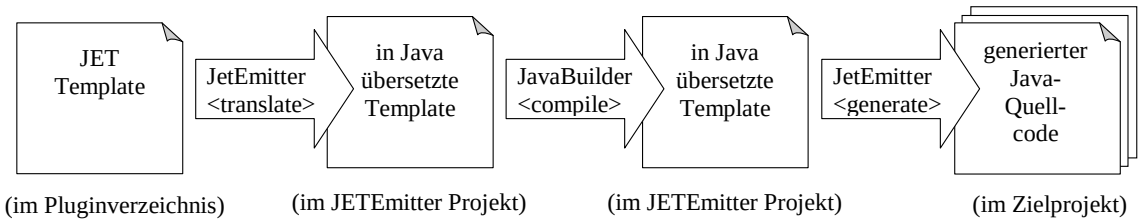


Abbildung 15: Arbeitsweise der JET

- Der Name der Schablonen ist der Name der zu generierenden Datei samt Endung und angehängtem `jet`, z.B. `MyClass.javajet`. Schablonen liegen in einem eigenen Ordner, standardmäßig `templates`. Die Schablonen im Ordner verwandelt der JETBuilder automatisch in ausführbare Generatorklassen und legt sie im Ordner `src` desselben Projekts ab. Eine solche Klasse besitzt eine Funktion `StringBuffer generate(Object argument)` mit den gewünschten Parametern, sie gibt beim Aufruf den zu generierenden Text zurück. Das Skelett der Generatorklassen kann durch eine im Verzeichnis `templates` abgelegte Datei `generator.skeleton` angepasst werden. Standardmäßig kann im Skript auf die Parameter über die Variable `argument` zugegriffen werden, der Rückgabewert wird in der Variablen `stringBuffer` gespeichert.

- Jede JET-Schablone beginnt mit der JET-Direktive in der Art

```
<%@ jet
  package="jsp.demo"
  class="JspTemplate"
  imports="java.io.* java.util.*"
  skeleton="generator.skeleton"
  startTag="<$" endTag="$>"
%>
```

  Hier wird eine Generatorklasse `JspTemplate` im Paket `jsp.demo` erstellt und ihre Importe festgelegt. Skriptcode ist nun innerhalb `<$...$>` gültig und Ausdrücke werden innerhalb `<$=...$>` akzeptiert. Da JSP-Code die voreingestellten Metamarkierungen selbst benutzt müssen sie umdefiniert werden. Das Skelett der erstellten Generatorklasse wird in `generator.skeleton` selbst definiert.

- JET-Direktiven der Form

```
<\%@ include file="copyright.jet" \%>
```

  erlauben das Einbinden von Text und Skriptcode aus anderen Dateien.

- Innerhalb der Metamarken werden Skriplets eingefügt, das sind Bruchstücke von Javaanweisungen, welche durch Textausgaben und Ausdrücke unterbrochen werden können:

```
<% if(Calendar.getInstance().get(Calendar.AM_PM)==Calendar.AM) {
    %>
Good Morning
<% } else { %>
Good Afternoon<% } %>
```

Beginnt der Inhalt zwischen zwei Metamarken mit einem Gleichheitszeichen „=“, wird das Ergebnis des enthaltenen Ausdrucks ausgewertet und ausgegeben:

```
Today's date is <%= (new java.util.Date()).toLocaleString() %>.
```

2.4.3 Die JET am Beispiel

Es gibt exzellente Tutorials, darunter „Model with the Eclipse Modeling Framework“ [14] von Adrian Powell, und „JET Tutorial“ [15] auf der Eclipse-Heimat. Erstgenanntes bietet einen schnellen Einstieg in EMF, JET und JMerge. Letzteres zeigt wie mit JET ein Codewizard für die Eclipse IDE geschrieben wird. Der Codewizard soll spezielle Klassen mechanisch erstellen, welche die Iteratorschnittstelle implementieren und über deren Elemente somit iteriert werden kann. Wie eine solche generierte Klasse aussehen kann, zeigt `Digit.java`:

Listing 4: Die Datei `Digit.java`

```
// an example typesafe enum
package x.y.z;
public class Digit {
  public static final Digit ZERO = new Digit(0, "zero");
  public static final Digit ONE = new Digit(1, "one");
  public static final Digit TWO = new Digit(2, "two");
  // ...
  public static final Digit NINE = new Digit(9, "nine");

  private static final Digit[] ALL =
    {ZERO, ONE, TWO, THREE, FOUR, FIVE, SIX, SEVEN, EIGHT, NINE};

  private final int value;
  private final String name;

  private Digit(int value, String name) {
    this.value = value;
    this.name = name;
  }

  public static Digit lookup(int key) {
    for (int i = 0; i < ALL.length; i++) {
      if (key == ALL[i].getValue()) { return ALL[i]; }
    }
    // lookup failed:
    // we have no default Digit, so we throw an exception
    throw new IllegalArgumentException("No digit exists for " + key);
  }

  public int getValue() { return value; }
  public int getName() { return name; }
  public String toString() { return getName(); }
}
```

Wie die Schablone in JET für die Erstellung derartiger Iterator-Klassen aussieht, zeigt Listing 5. Die Schablone benötigt eine Instanz des in Abbildung 16 skizzierten Modells.

Listing 5: Die Datei `TypeSafeEnumeration.javajet`

```
<%@ jet package="translated" imports="java.util.* article2.model.*"
    class="TypeSafeEnumeration" %>
<% TypesafeEnum enum = (TypesafeEnum) argument; %>
package <%=enum.getPackageName()%>;

/**
 * This final class implements a type-safe enumeration
 * over the valid instances of a <%=enum.getClassName()%>.
 * Instances of this class are immutable.
 */
public final class <%=enum.getClassName()%> {

<% for (Iterator i = enum.instances(); i.hasNext(); ) { %>
<%     Instance inst = (Instance) i.next(); %>

  // instance definition
  public static final <%=enum.getClassName()%> <%=inst.getName()%>
  = new <%=enum.getClassName()%>(<%=inst.constructorValues()%>);
<% } %>

<% for (Iterator i = enum.attributes(); i.hasNext(); ) { %>
<%     Attribute attr = (Attribute) i.next(); %>

  // attribute declaration
  private final <%=attr.getType()%> m<%=attr.getCappedName()%>;
<% } %>

  /**
   * Private constructor.
   */
  private <%=enum.getClassName()%>(<%=enum.
      constructorParameterDescription()%>) {
<% for (Iterator i = enum.attributes(); i.hasNext(); ) { %>
<%     Attribute attr = (Attribute) i.next(); %>
    m<%=attr.getCappedName()%> = <%=attr.getUncappedName()%>;
<% } %>
  }

  // getter accessor methods
<% for (Iterator i = enum.attributes(); i.hasNext(); ) { %>
<%     Attribute attr = (Attribute) i.next(); %>
  /**
  * Returns the <%=attr.getName()%>.
  * @return the <%=attr.getName()%>.
  */
  public <%=attr.getType()%> get<%=attr.getCappedName()%>() {
    return m<%=attr.getCappedName()%>;
  }
<% } %>

  // lookup method omitted...
}
```

Der Quelltext macht anschaulich, wie die notwendigen Parameter aus einem Objekt der Klasse `TypesafeEnum` gelesen werden. Diese Klasse repräsentiert zusammen mit den assoziierten Klassen `Attribute`, `Instance` das Modell der Transformation (Abbildung 16).

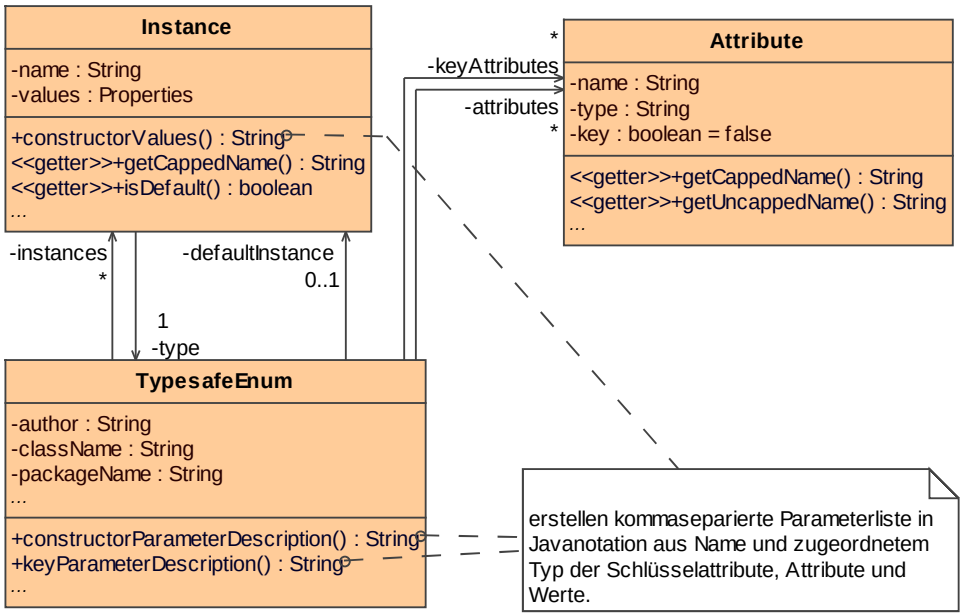


Abbildung 16: Aus Instanzen dieses Modells generiert JET Iteratorklassen in Java

2.5 Kommerzielle Generatorsprachen

Modell-zu-Code-Transformatoren sind im Funktionsumfang kommerzieller UML-Modellierwerkzeuge bereits enthalten. Zu den bekanntesten Werkzeugen dieser Art zählen Borland Together Architect, OptimalJ und ArcStyler. Borland verwendet die JET, OptimalJ kennt hierfür sog. „Implementation Patterns“ und ArcStyler basiert auf AndroMDA-ähnlichen Cartridges als Transformationseinheit. Dokumentation zu den integrierten Transformationsmechanismen von OptimalJ, ArcStyler und vielen anderen Tools ist nicht öffentlich verfügbar, somit kann keine nähere Betrachtung bzw. Bewertung erfolgen.

3 Fazit und Ausblick

Selbst unter den vorgestellten Sprachen fällt es nicht leicht Vergleiche durchzuführen. Die Faktoren Anwendbarkeit, Flexibilität und Mächtigkeit hängen stark vom Kontext ab, d.h. die Integrierbarkeit in das jew. Werkzeug spielt eine große Rolle. Darüberhinaus verfolgen die einzelnen Sprachen unterschiedliche Ziele. Während MOFScript versucht, die Vorgaben der OMG zu erfüllen, ist es das Anliegen der Leute hinter oAW, eine möglichst leicht erlernbare Sprache zu verwenden. AndroMDAs Entwickler machen bei VTL eher Abstriche bei der Integrierbarkeit und Flexibilität anstatt das Rad neu zu erfinden, so scheint es. JET ist eine universell verwendbare Sprache sofern man die Sprache Java nicht scheut, welche als Ausführungsplattform dient. Die beiden verfügbaren Einführungen in JET legen nahe, dass JET mehr der Oberfläche Eclipse unter die Arme zu greifen beabsichtigt als ein Glied in der Prozesskette der MDA zu sein.

Mit dem wachsenden Bekanntheitsgrad von Eclipse und einer immer größer werdenden Eclipse-Entwicklergemeinde in Wirtschaft und Forschung fällt es dieser Plattform immer leichter wegweisende Standards in der Softwareentwicklung zu setzen. Sah es vor einigen Jahren noch so aus, als würde Java wieder in Vergessenheit geraten, so wird

man wohl vor allem dank IBM, den hauptsächlichen Entwicklern von Eclipse, eines Besseren belehrt. Drei der vier besprochenen Sprachen, Xpand, MOFScript und JET, haben entweder ihren Ursprung in Eclipse, oder wurden in die Projekthierarchie von Eclipse integriert. Nur AndroMDAs VTL ist Teil des Apache Jakarta Open-Source-Projektes.

Es bleibt nun abzuwarten, wie der MDA-Standard der Modell-zu-Text-Transformationssprachen aussehen wird und wann er in Kraft tritt. Absehbar ist, dass das angepasste MOFScript die Referenzimplementierung des MOF2Text-Standards sein wird. Sowohl AndroMDA wie auch oAW haben durch ihre Plugin-Fähigkeit die Option, durch eine relativ einfach zu schreibende Adapter-Komponente diese Sprache an ihre Engine anzubinden, und auch andere Werkzeuge werden dem sicher nachkommen, denn offene Standards, vor allem solche mit offener Referenzimplementierung, haben in der Regel die besseren Karten.

Die Tabelle in Abbildung 17 gibt einen abschließenden vergleichenden Überblick über einige angesprochenen Sprachfeatures.

Sprachfeature	**Sprache**			
	Xpand	**VTL**	**MOFScript**	**JET**
Verwendung in Tool	oAW	AndroMDA	—	EMF
Schablonen-Prinzip	ja	ja	ja	ja
Kompatibilität mit OMG M2T-Standard	—	—	—	—
Verfügbarer Kontext (Eingabe)	bel. Modelle, Quellcode	ein Javaobjekt	MOF2-Modell	Javaobjekte
Aufruf von externem Code	Java	Java	Java	Java
Wertemanipulation (subString, toLower,...)	intern (eigene API)	extern (Java-Aufruf)	intern (eigene API)	intern (Java integriert)
Geschützte Bereiche	intern (über Annotationen)	extern denkbar	intern (über Annotationen)	extern (über JMerge)
IDE-Anbindung	Eclipse	bald Eclipse („Android“)	Eclipse	Eclipse (JETEditor)
Sprachreferenz / Tutorials	sehr gut / sehr gut	gut / gut	sehr gut / gut	gut / sehr gut
Transformationen erweiterbar	„extension“-Mechanismus	—	„extend“ und „super“	—
Regeln verfeinerbar / abstrakte Regeln	ja / —	— / —	ja / ja	— / —
Metamarken / ihre Symbolik	«...» / Skript	#... und $... / Befehle und Variablen	<%...%> / Text	<%...%>, umdef.bar / Text

Abbildung 17: Ein abschließender Vergleich der präsentierten Sprachen

Abbildungen

Literatur

[1] Runge, Wolfgang: *Werkzeug Objekt - Kybernetik und Objektorientierung (PhD Dissertation)*. 2001

[2] Bast, Wim ; Kleppe, Anneke ; Warmer, Jos: *MDA Explained*. 1. Addison-Wesley Professional, 2003. – ISBN 0–321–19442–X

[3] Stachowiak, Herbert: *Allgemeine Modelltheorie*. Wien : Springer, 1973. – ISBN 3–211–81106–0

[4] Born, Marc ; Holz, Eckhardt ; Kath, Olaf: *Softwareentwicklung mit UML 2*. München : Addison-Wesley, 2004. – ISBN 3–8273–2086–0

[5] Rossbach, Peter ; Stahl, Thomas ; Neuhaus, Wolfgang: Model Driven Architecture - Grundlegende Konzepte und Einordnung der Model Driven Architecture. (2003), September. `http://www.javamagazin.de/itr/online_artikel/psecom,id,408,nodeid,11.html`

[6] Balzert, Helmut: *Lehrbuch der Software-Technik*. Heidelberg : Spektrum Akademischer Verlag, 1998. – ISBN 3–8274–0065–1

[7] Object Management Group: *MOF Model To Text Transformation Language - Request For Proposal*. `http://www.omg.org/cgi-bin/doc?ad/2004-4-7`. Version: 7. April 2004

[8] Czarnecki, Krzysztof ; Helsen, Simon: Classification of Model Transformation Approaches. In: *OOPSLA 2003 Workshop on Generative Techniques in the context of Model Driven Architecture*. Anaheim : ACM Press, Oktober 2003

[9] Gamma, E. ; Helm, R. ; Johnson, R. ; Vlissides, J.: *Entwurfsmuster: Elemente wiederverwendbarer objektorientierter Software*. Bonn : Addison-Wesley, 1996. – ISBN 3–89319–950–0

[10] Efftinge, Sven ; Kadura, Clemens: *Xpand Language Reference*. `http://www.eclipse.org/gmt/oaw/doc/r20_xPandReference.pdf`. Version: 26. Juni 2006

[11] Völter, Markus: *oAW4 EMF Example*. `http://www.eclipse.org/gmt/oaw/doc/30_emfExample.pdf`. Version: 26. Juni 2006

[12] Object Management Group: *Second Revised Submission for MOF Model to Text Transformation Language RFP*. `http://www.omg.org/cgi-bin/apps/doc?ad/05-11-03.pdf`. Version: 14. November 2005

[13] Jon Oldevik: *MOFScript User Guide*. `http://www.eclipse.org/gmt/mofscript/doc/MOFScript-User-Guide.pdf`. Version: 27. Juni 2006

[14] Powell, Adrian: *Model with the Eclipse Modeling Framework*. `http://www.ibm.com/developerworks/library/os-ecemf1`, `http://www.ibm.com/developerworks/library/os-ecemf2`, `http://www.ibm.com/developerworks/library/os-ecemf3`, Version: 15. April 2004

[15] Popma, Remko: *JET Tutorial*. http://www.eclipse.org/articles/Article-JET/jet_tutorial1.html, http://www.eclipse.org/articles/Article-JET2/jet_tutorial2.html, Version: 31. Mai 2004

Transformation von Software-Architekturen in Warteschlangenmodelle

Nils Drechsel

Betreuer: Michael Kuperberg

Zusammenfassung

Laut "Chaos Report" der Standish Group [1] schlagen 52 Prozent aller IT-Projekte fehl, 15 Prozent werden komplett abgebrochen. Die Gründe sind vielfältig, häufig aber das Nicht-Einhalten von
Performanz-Nebenbedingungen. Kein Kunde möchte ein Produkt haben, das zwar theoretisch all seine Wünsche erfüllt, er es aber nicht benutzen kann, da es zu langsam läuft oder sonstige Performanz-Schwachpunkte hat.

Die bestehenden Lösungen waren früher zu ungenau und boten nicht die gewünschten Lösungen. Mit dem Aufkommen von UML hat sich allerdings ein ganzer Zweig an Optimierungsmöglichkeiten geöffnet, da die Diagramme nicht nur eine bessere Übersicht über das Systemmodell bieten, sondern die Modelle an sich so viel Wissen integriert haben, daß man durch geschickte Umwandlung neue Möglichkeiten bekommt, Engpässe aufzudecken und so früh wie möglich zu beheben.
Bisherige Artikel wie [2], [3] und [4] haben Lösungen vorgestellt, wie an dieses Problem herangegangen werden kann. Sie haben jedoch viel Wissen über Warteschlangen, UML Diagramme, LQNs, XML-Sprachen und Baum-Transformationssprachen vorausgesetzt.

Wir haben einen möglichen Weg, vom UML-Modellierer bis zum LQN-Solver, versucht anschaulich zu erläutern und haben ein Beispiel generiert, an dem die verschiedensten Techniken bzw. Eigenschaften erläutert werden. Das Ergebnis dieser Ausarbeitung ist eine zusammenhängende Darstellung, mit der man einen Einblick in den gesamten Performanz-Evaluations Prozess bekommt.

Zuerst werden Warteschlangen und Warteschlangennetzwerke betrachtet. Es werden Hilfmittel wie XMI und SPT-Profil kurz vorgestellt und danach auf Graphen-Transformationssprachen wie XSLT eingegangen, welche im weiteren Verlauf auf UML-Diagramme angewendet werden. Als große Hilfe haben sich dabei [5], [6], [2] für das Verständnis wie die Umwandlung der

UML-Diagramme in LQNs funktioniert und [7], [8] zum besseren Verständnis von XML Sprachen erwiesen.

1 Einleitung

1.1 Woran scheitern häufig große Softwareprojekte?

Bevor Softwareprojekte gestartet werden, werden vorher die Ziele abgesteckt, die das Projekt erfüllen muss (Requirements engineering). Vor allem Nebenbedingungen wie Geschwindigkeitsanforderungen, Speicherplatzbedarf etc. sind oftmals sehr wichtig für den Prozesse. Es ist allerdings äußerst schwer, diese Nebenbedingungen anfangs abschätzen zu können. Es macht sich meistens erst im Laufe der Entwicklung bemerkbar, wie die Performanz des Projekts tatsächlich aussieht. Erfüllt das Projekt die Nebenbedingungen nicht, kann es möglicherweise vom Prozesse nicht benutzt werden und eine Änderung der Architektur ist notwendig, um den Performanz-Problemen zu begegnen. Allerdings ist zu diesem Zeitpunkt eine Architekturänderung womöglich nicht mehr praktikabel.

1.2 Wie kann man dieses Problem beheben?

Ein gut erforschtes Gebiet zur Performanz-Evaluation sind Warteschlangen. Die Architektur eines Softwareprojekts wird dagegen üblicherweise mit UML beschrieben. Eine Möglichkeit um Engpässe frühestmöglich zu erkennen, ist eine Abbildung der Architektur auf Warteschlangennetzwerke. Es ist dadurch möglich, das Laufzeitverhalten abzuschätzen und frühzeitig Änderungen am Entwurf vorzunehmen, nämlich wenn die Erfüllung der Nebenbedingungen in Gefahr ist.

2 Performanz-Evaluation mit LQN

2.1 Warteschlangentheorie

Im folgenden wird beschrieben, wie eine Warteschlange aufgebaut ist und was ihre Parameter sind. Relevante Kennwerte, die man durch Abbildung der bestehenden Modelle auf Warteschlangen erhalten kann, beschreibt folgende Liste aus [1](Seite 137):

1. Die Aufenthaltszeit eines Prozesses RT: Durchschnittliche Zeit, welche sich ein Prozess im System befindet.

2. Ausnutzung U: Die durchschnittliche Zeit, in der die Warteschlange in Betrieb ist.

3. Durchsatz X: Die durchschnittliche Anzahl der Prozesse pro Zeiteinheit, welche die Warteschlange durchlaufen.

4. Warteschlangenlänge N: Die Anzahl der Prozesse, welche auf Abarbeitung warten.

2.1.1 Startüberlegungen

Um die obigen Kenngrößen abschätzen zu können, benötigt man Informationen darüber, zu welchem Zeitpunkten wieviele Prozesse sich in der Warteschlange wahrscheinlich aufhalten und wie lange sie warten müssen.

Warteschlangen sind typischerweise folgendermaßen aufgebaut:

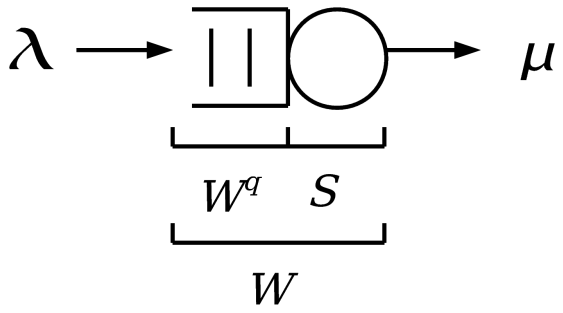


Abbildung 1: Warteschlange

Mit der Ankunftsrate λ und einem Zeitintervall der Länge Δt ist die Wahrscheinlichkeit für den Eintritt eines Prozesses in die Warteschlange $\lambda \cdot \Delta t$ Es gibt eine (unendliche) Quelle, von der Prozesse in die Warteschlange eintreten. Dort verbringen sie eine gewisse Zeit, bis sie zu einer Bedienstation kommen, in der sie eine gewisse Zeit bedient werden. Danach treten sie als abgefertigte Prozesse wieder heraus. Die Austrittswahrscheinlichkeit ist $\mu \cdot \Delta t$.

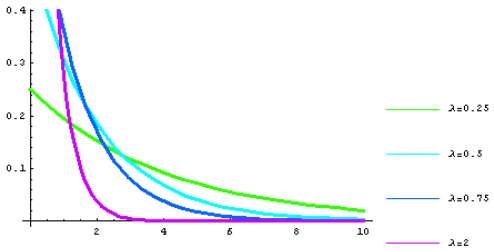


Abbildung 2: Dichtefunktion der Exponentialverteilung

Mithilfe dieser Parameter und der Anzahl der Bedienungsschalter kann eine Warteschlange vollständig beschrieben werden. Die Zeit, die vergeht, in der ein neuer Prozess in die Warteschlange eintritt, wird Zwischenankunftszeit Z genannt. Die Bedienungszeit S, ist die Zeit, in welcher der Prozess am Schalter bedient wird.

Sind Zwischenankunftszeit und Bedienungszeit exponentialverteilt, so spricht man von einem M/M/x System, wobei die M für Exponentialverteilung stehen. Das erste M ist die Verteilung der Zwischenankunftszeit, das zweite, die Bedienungszeit und das x gibt die Anzahl der Bedienungsschalter an. Für die nachfolgenden Betrachtungen wird Z mit dem Parameter λ und S mit dem Parameter μ als exponentialverteilt vorausgesetzt, so wie es in [2] vorgestellt wurde.

2.1.2 Zustandsübergänge

Es gibt 4 mögliche Zustandsübergänge vom Zeitpunkt t zum Zeitpunkt $t + \Delta t$:

1. Es kommt genau ein neuer Prozess an. Die Wahrscheinlichkeit hierfür ist $\lambda \Delta t$
2. Es kommt kein neuer Prozess an. Wahrscheinlichkeit ist das Komplement zu Punkt 1: $1 - \lambda \Delta t$
3. Es wird ein Prozess abgefertigt. Wahrscheinlichkeit hierfür ist $\mu \Delta t$
4. Es kam ein Prozess an und es wurde ein Prozess abgefertigt oder es kam kein Prozess an und es wurde auch kein Prozess abgefertigt: $1 - (\lambda \Delta t + \mu \Delta t)$

Die Wartezeit eines Prozesses der Nummer n im System wird in zwei Abschnitte aufgeteilt: Die Wartezeit W_n^q in der Schlange und die Bedienungszeit S_n, in der er abgearbeitet wird. Die Gesamtwartezeit des n-ten Prozesse berechnet sich durch Summation der beiden Abschnitte $W_n = W_n^q + S_n$

Es interessiert nun die Verteilung der Anzahl der Prozesse über die Zeit hinweg. Das heißt, es müssen die Wahrscheinlichkeiten errechnet werden, dass sich zum Zeitpunkt t 1 Prozess, 2 Prozesse, 3 Prozesse etc. im Wartesystem befinden.
Mit Hilfe der obigen Zustandsübergänge lassen sich zwei Gleichungen aufstellen: Sei $p_j(t)$ die Wahrscheinlichkeit, dass sich zum Zeitpunkt t, j Prozesse im System aufhalten. Das Ereignis, dass sich zu einem späteren Zeitpunkt $t + \Delta t$, 0 Prozesse im System aufhalten, kann nur durch 2 Zustandsübergänge geschehen: Im System war genau 1 Prozess und er wurde abgefertigt $(p_1(t) \cdot \mu \Delta t)$ oder es war überhaupt kein Prozess im System und es kam auch kein neuer hinzu $(p_0(t) \cdot (1 - \lambda \Delta t))$. Also:

$$p_0(t + \Delta t) = p_1(t) \cdot \mu \Delta t + p_0(t) \cdot (1 - \lambda \Delta t) + o(\Delta t) \tag{1}$$

das $o(\Delta t)$ ist ein Landau-Symbol (klein o) und gibt die Wahrscheinlichkeit dafür an, dass 2 Ereignisse gleichzeitig auftreten. Im anderen Fall befinden sich j Prozesse zu einem späteren Zeitpunkt im System und 3 Zustandsübergänge sind möglich. Der 4te wird vom dritten mit abgedeckt:

$$p_j(t + \Delta t) = p_{j-1}(t) \cdot \lambda \Delta t + p_j(t) \cdot (1 - \lambda \Delta t + \mu \Delta t) p_{j+1}(t) \cdot \mu \Delta t + o(\Delta t) \tag{2}$$

Im nächsten Schritt werden die Gleichungen so umgeformt, dass auf der linken Seite ein Differentialquotient steht und die Gleichungen damit in ein Differentialgleichungssystem übergehen:

$$\dot{p}_0(t) = -\lambda p_0(t) + \mu p_1(t) \tag{3}$$

$$\dot{p}_j(t) = \lambda p_{j-1}(t) - (\lambda + \mu) p_j(t) + \mu p_{j+1}(t) \tag{4}$$

2.1.3 Stationäre Wahrscheinlichkeiten

Die Bestimmung der Verteilung von L ist sehr schwierig und rechenaufwendig, da hierzu das Differentialgleichungssystem gelöst werden muss.
Aus diesem Grund nimmt man an, dass sich das Wartesystem nach einer gewissen Laufzeit im Gleichgewicht befinden wird. Diese Annahme ist nur korrekt, wenn die Austrittsrate größer ist als die Eintrittsrate, oder formal ausgedrückt:

$$\rho = \frac{\lambda}{\mu}$$

und

$$\rho < 1$$

Ist diese Bedingung gegeben, kann man die stationären Wahrscheinlichkeiten heranziehen, welche zeitunabhängig sind. Setzt man $\dot{p_j}$ gleich 0, so erhält man das folgende Gleichungssystem, bei dem die $p_j(t)$ in π_j übergehen:

$$(5)$$

$$-\lambda\pi_0 + \mu\pi_1 = 0 \tag{6}$$

$$\lambda\pi_{j-1} - (\lambda + \mu)\pi_j + \mu\pi = 0 \tag{7}$$

Einige Lehrbücher [2],[3] versehen die λ und μ mit Indizes, die identisch mit den zugehörigen Wahrscheinlichkeiten π sind, um auszudrücken, dass bei unterschiedlicher Fülle der Warteschlange, die Ankunftsrate und Abfertigungsrate von der Fülle abhängen. Dies mag z.B. bei Menschen der Fall sein, die sich in eine Kioskschlange anstellen (oder auch nicht), aber nicht für die Betrachtung von Prozessen, die sich wohl kaum entscheiden werden, spontan nicht auf die Festplatte zu speichern, weil der Andrang schon zu groß ist. Die Indizes werden deshalb weggelassen.

Die obigen Gleichungssysteme sind für die Warteschlange M/M/1. Wie weiter unten bei der Beschreibung von LQNs allerdings erläutert wird, ist es möglich, mehrere Bedienungsschalter pro Warteschlange zu haben. Deshalb müssen die Gleichungen noch für das System M/M/s angepasst werden. Dies geschieht folgendermaßen:
Die Ankunftsrate λ bleibt gleich. Die Bedienungsrate μ dagegen wird folgendermaßen aufgeteilt:

$$\mu_j = \begin{cases} j\mu \text{ für } j = 1, ..., s-1 \\ s\mu \text{ für } j \geq s \end{cases}$$

Hier ist der Index bei μ wieder hereingekommen, da es in diesem Fall tatsächlich eine Rolle spielt.
Das Gleichungssystem aus 5 wird nun zu

$$-\lambda\pi_0 + \mu\pi_1 = 0 \tag{8}$$

$$\lambda\pi_{j-1} - (\lambda + j\mu)\pi_j + (j+1)\mu\pi_{j+1} = 0 \text{ für } j = 1, ..., s-1 \tag{9}$$

$$\lambda\pi_{j-1} - (\lambda + s\mu)\pi_j + s\mu\pi_{j+1} = 0 \text{ für } j \geq s \tag{10}$$

Es muss nun nur noch nach den stationären Wahrscheinlichkeiten aufgelöst werden und man bekommt die Verteilung der Anzahl der Prozesse in der Schlange.

2.1.4 Littles Formel

Für den Zweck der Übersichtlichkeit wird folgende, oberhalb schon angesprochene Variable eingeführt:

$$\rho = \frac{\lambda}{\mu} \tag{11}$$

$$\rho_j = \begin{cases} \frac{\lambda}{j\mu} = \frac{\rho}{j} \text{ für } j = 1, ..., s-1 \\ \frac{\lambda}{s\mu} = \frac{\rho}{s} \text{ für } s \geq s \end{cases} \tag{12}$$

$$\pi_1 = \rho\pi_0 \tag{13}$$

$$\pi_2 = \frac{\rho^2\pi_0}{2} \text{ für } j = 1, ..., s-1 \tag{14}$$

$$\pi_3 = \frac{\rho^2\pi_0}{2 \cdot 3} \text{ für } j = 1, ..., s-1 \tag{15}$$

Diesem Schema weiter folgend, bekommt man für π_j

$$\pi_j = \begin{cases} \frac{\rho^j}{j!}\pi_0 \text{ für } j = 1, ..., s-1 \\ \frac{\rho^j}{s!s^{j-s}}\pi_0 \text{ für } j \geq s \end{cases} \tag{16}$$

Die π_js sind Wahrscheinlichkeiten dafür, dass sich j Prozesse im System befinden. Alle π_js aufaddiert, muss also 1 ergeben.

$$\sum_{j=0}^{\infty} \pi_j = \sum_{j=0}^{s-1} \frac{\rho^j}{j!}\pi_0 + \frac{\rho^s}{(s-\rho)(s-1)!}\pi_0 = 1 \tag{17}$$

Dies führt zu der Startwahrscheinlichkeit π_0:

$$\pi_0 = \frac{1}{1 + \sum_{j=1}^{s-1} \frac{\rho^j}{j!} + \frac{\rho^s}{(s-\rho)(s-1)!}} \tag{18}$$

das $1 + \sum$ entsteht durch den Fakt, dass $\rho^0 = 1$ ist und herausgezogen werden kann.

Aus dieser Gleichung lässt sich endlich die mittlere Warteschlangenlänge berechnen:

$$E(L^q) = \sum_{j=s} \infty(j-s)\pi_j = \frac{\rho^j}{j!}\pi_0 + \frac{\rho^{s+1}}{(s-\rho)^2(s-1)!}\pi_0 \tag{19}$$

und damit die mittlere Anzahl der Prozesse im System:

$$E(L) = \sum_{j=0}^{\infty} j\pi_j = L^q + \rho \tag{20}$$

Der Erwartungswert der Verweilzeit (die zweite Größe, die uns interessiert) ist daraufhin:

$$E(W^q) = \frac{E(L^q)}{\lambda} \tag{21}$$

und

$$E(W) = \frac{E(L)}{\lambda} \tag{22}$$

Dies ist Littles Formel.

2.2 Warteschlangennetze

Will man UML Diagramme auf Warteschlangen abbilden, so wird man schnell feststellen, dass diese Aufgabe nicht mit einer einzelnen Warteschlange gelöst werden kann. In dem Beispiel, welches sich durch die gesamte Ausarbeitung zieht, wird ein Webserver betrachtet, welcher vom Browser des Clients Befehle empfängt und zur Ausführung dieser eine Datenbank und einen Fileserver zur Verfügung hat. Webserver, Datenbank und Fileserver stellen als Komponenten eigenständige Warteschlangen da. Verbindet man die einzelnen Warteschlangen zu einem Netzwerk, so ist es nun möglich folgendes Szenario zu simulieren:

1. Ein Webserver erhält eine Anfrage (nach einer Liste aller verfügbaren MP3s, welche in der Datenbank verfügbar sind).
2. Diese Anfrage wird der Datenbank übermittelt.
3. Die Datenbank konsultiert daraufhin die relevanten Tabellen und übergibt die Ausführung wieder an den Webserver.
4. Der Webserver bekommt daraufhin eine neue Anfrage (nach einer bestimmten MP3 Datei).
5. Er übergibt diese an den Fileserver.
6. Der Fileserver sorgt selbstständig dafür, dass dem Client die Datei übermittelt wird.

Ein Warteschlangennetz zu diesem Beispiel besteht aus Knoten, welche Hardware-Komponenten darstellen und aus gerichteten Kanten, welche die Richtung des Prozessflusses darstellen, außerdem kann es Verzweigungen geben.

2.2.1 Offene und geschlossene Warteschlangennetze

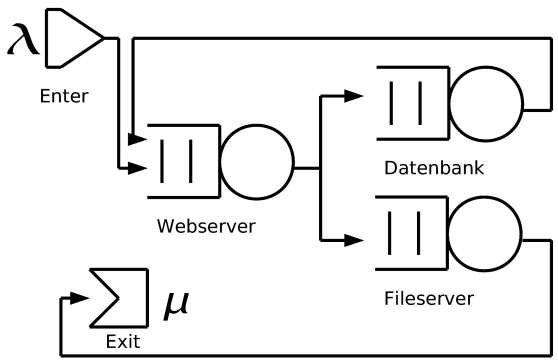


Abbildung 3: Offenes Warteschlangennetz

Offene Warteschlangennetze sind dadurch gekennzeichnet, dass es einen Eintrittsknoten gibt, durch den Prozesse das System betreten und einen Austrittsknoten gibt, durch den sie das System wieder verlassen. Wie für eine einzelne Warteschlange, gibt es für dieses System eine Gesamt-Eintrittsrate und eine Gesamt-Austrittsrate, die für das gesamte System gilt. Die Ein- und Austrittsraten der einzelnen Warteschlangen können dabei natürlich andere Werte haben.
Wenn man folgende Parameter des Netzes kennt, kann man Durchschnittswerte für

verschiedene Kenngrößen berechnen:

1. λ Eintrittsrate
2. V_i Die Anzahl der Besuche des Knotens i
3. S_i Durchschnittliche Bedienungszeit des Knotens

Der Durchsatz des Netzes ist die Eintrittsrate $X_0 = \lambda$. Der Durchsatz eines einzelnen Knoten, ist $X_i = \lambda \cdot V_i$ und die Ausnutzung eines Knotens $U_i = X_i \cdot S_i$. Die Verweilzeit in einem Knoten pro Besuch errechnet sich durch $RT_i = \frac{S_i}{1-U_i}$, die Länge der Warteschlange für einen Knoten $N_i = X_i \cdot RT_i$ und die Gesamtlänge erhält man, wenn man alle Längen aufaddiert: $N = \sum_i N_i$. Die Formeln stammen aus [1], in ihnen wird nur mit Mittelwerten gerechnet, eine echte Betrachtung der Warteschlangennetzwerke und ihre Lösung wird in der Ausarbeitung [4] vorgenommen.

Geschlossene Warteschlangennetze besitzen keinen Ein- und Austrittsknoten. Die Anzahl der Prozesse ist immer konstant, diese Art von Netzen werden wir hier aber nicht weiter behandeln.

2.2.2 Mehrschichtige Warteschlangennetze (LQN)

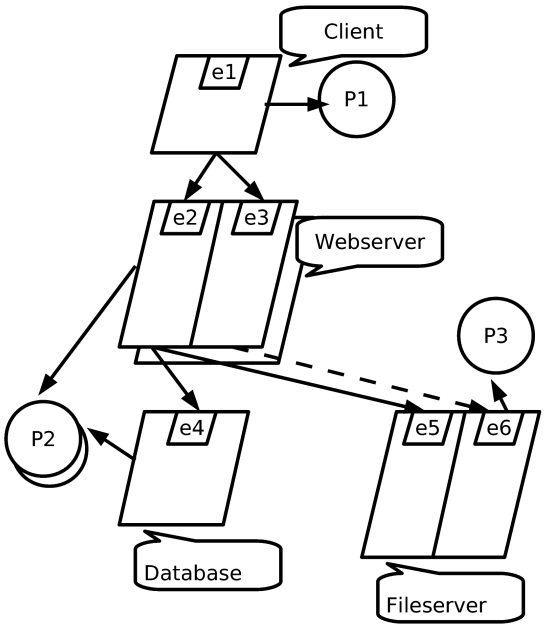


Abbildung 4: LQN

LQNs (Layered Queueing Networks) sind azyklische Graphen. Die Knoten sind entweder Tasks (dargestellt als Parallelogramme), wie beispielsweise der Webserver aus der Abbildung 4, oder Prozessoren (dargesteltt als Kreise), wie P1. Tasks sind eigenständige Prozesse, welche Nachrichten empfangen und versenden können. Tasks werden weiter unterteilt in Pure Clients (Client), Active Server (Webserver) und Pure Servers (Datenbank, Fileserver). Pure Clients sind Knoten, von denen nur Kanten ausgehen, aber

keine ankommen. Pure Server dagegen das genaue Gegenteil, es kommen nur Kanten an, gehen aber keine ab.

Bei Active Servern kommen Knoten an und gehen Knoten ab. Diese Server können Harwareserver oder auch Softwareserver sein und entweder aus nur einem einzigen, oder wie z.B. bei Multi-Core CPUs aus mehreren Prozessoren bestehen. Jedem Server ist also mindestens ein Prozessor zugeordnet (durch eine weitere Kante zum einem Kreis verdeutlicht). Es sind auch Server mit einer unendlichen Anzahl an Prozessoren möglich.

Jeder Knoten besitzt eine eigene Warteschlange, Server mit mehreren Bedienungsschaltern werden wie am Beispiel des Webservers dargestellt. Diese werden so wie M/M/s Systeme modelliert, welche oben vorgestellt wurden. Diese Bedienungsschalter werden auch Entries genannt, und werden, außer, dass sie eine gemeinsame Warteschlange besitzen, so wie normale Server behandelt, das heißt es gehen Kanten hinein und möglicherweise auch heraus. Ausführlich dargestellt in der Masterarbeit [5].

Ein Entry kann weiter in Aktivitäten unterteilt werden. Siehe dazu auch Abbildung 11. Aktivitäten spielen sich also innerhalb der Entries ab. Es sind folgende Aktivitäten möglich:
Der Befehlsfluss kann sich aufspalten und wieder zusammengeführt werden.
Es können Aktivitäten ausgeführt werden.

Wenn sich der Befehlsfluss aufteilt, wird das Fork genannt. Möglich sind Or-Forks und And-Forks, bei den ersteren geht der Befehlsfluss mit einer Wahrscheinlichkeit π den linken, bzw. mit $1 - \pi$ den rechten Weg. Beim And-Fork gibt es eine parallele Ausführung der Befehlsflüsse.

Das Zusammenführen der Befehlsflüsse geschieht über Joins. Hier gibt es auch wieder zwei Möglichkeiten: Beim Or-Join reicht es, wenn ein Befehlsfluss ankommt, beim And-Join allerdings müssen beide Befehlsflüsse ankommen, das heißt, hier wartet ein Fluss auf den anderen: Sie synchronisieren sich.

Aktivitäten haben eigene Ausführungszeiten und können Anfragen zu anderen Entries stellen. Ausführlich ist dies erläutert in den Artikeln [5] und [6].

Kanten repräsentieren Anfragen, wobei zwischen drei Anfragetypen unterschieden wird: Synchrone, asynchrone und weitergeleitete Anfragen. Wenn ein Server eine Anfrage akzeptiert, arbeitet er sie in mehreren Schritten ab, Phasen genannt. Bei einer synchronen Anfrage schickt der Server dem Klienten nach Phase 1 eine Antwort. Bevor er diese Antwort nicht erhalten hat, kann der Klient nicht weiterarbeiten. Bei einer asynchronen Anfrage, kann der Klient direkt weiterarbeiten, ohne auf den Server warten zu müssen. Eine weitergeleitete Anfrage funktioniert so wie eine synchrone Anfrage, mit der Ausnahme, dass der Server die Anfrage nicht selber bearbeitet, sondern an einen anderen Server weiterschickt. Erst der Server, welcher sich für zuständig hält, schickt dem Klienten eine Antwort.

3 Der Weg vom Unified Modeling Language Modell (UML) zum LQN

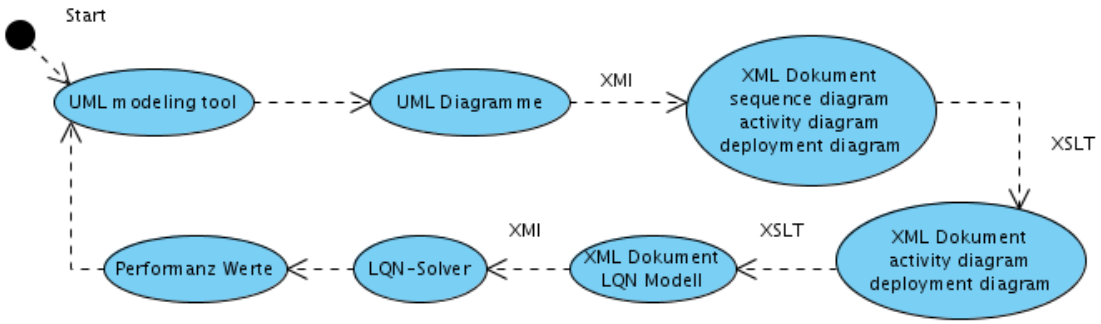


Abbildung 5: Der Prozess

Der hier beschriebene Weg ist nur einer von vielen möglichen. Er wurde ausgewählt, da es offensichtlich ist, dass XML-basierte Schnittstellen und Übertragungswege immer attraktiver werden und die alten Formate nach und nach ablösen. Der Weg führt vom UML Modeler (dies hilft, die einzelnen UML-Diagramme zu entwerfen) über XML-Dokumente, welche von XMI (einer XML-Sprache, die später noch vorgestellt wird) aus den Diagrammen erstellt wurden zu LQN-Netzwerken, welche durch die Stylesheet-Sprache XSLT erzeugt werden.

Es sind noch eine ganze Reihe weiterer Wege möglich, z.B. gibt es Fallstudien wie in [7] erwähnt, welche direkt aus den Diagramm-Objekten aus dem Hauptspeicher des Computers die betreffenden LQNs erzeugen können oder auch Studien, welche statt der Stylesheet-Sprache XSLT, die Graphenumwandlungssprache PROGRES [5] bzw. Graph-Grammar Ansätze [6] verwenden. PROGRES wird im Unterkapitel 3.6.1 noch einmal konkret angesprochen.

3.1 Der UML-Modeler

Derzeit sind über 100 UML Modeler kommerziell oder auch frei verfügbar, darunter z.B. Rose Rational oder Visual Paradigm (mit letzterem habe ich die Beispiele in dieser Ausarbeitung erzeugt). Der UML Modeler dient dabei als Hilfestellung beim Erzeugen von UML-Diagrammen. Die unterschiedlichen Modeler unterscheiden sich weniger in der Art, die Diagramme zu erstellen, sondern in der Art der Integration und der Weiterverwendbarkeit der erstellten Diagramme. Viele der Modeler haben die Möglichkeit, die Diagramme als XML-Dateien abzuspeichern; dies öffnet erst den Weg (falls der Modeler nicht selbst schon Möglichkeiten zur Performanz-Evaluation bietet) der weiteren Umwandlung und der Generierung von LQNs.

3.2 Das UML Meta-Modell

UML ist ein Regelwerk, mit dem man UML-Instanzen, also UML Diagramme erstellen kann. Wie diese Diagramme aufgebaut sind, definiert die UML. Die UML selber unterliegt allerdings auch gewissen Regeln, welche vom UML Meta-Modell (MOF) definiert werden [8]. Die UML Diagramme sollen in XML Dateien abgespeichert werden und möglichst auch wieder heraus gelesen werden. Jede XML-Sprache wie z.B. die unten vorgestellte XMI, kann XML-Schemata oder XML-DTDs besitzen, mit denen es möglich ist, die Korrektheit einer XML Datei zu bestimmen. Um die Korrektheit eines abgespeicherten Diagramms zu bestimmen, benötigt man den Aufbau der Sprache, mit dem dieses Diagramm beschrieben wird - also das UML-Meta Modell.

3.3 UML Diagramme

3.3.1 Sequenzdiagramm

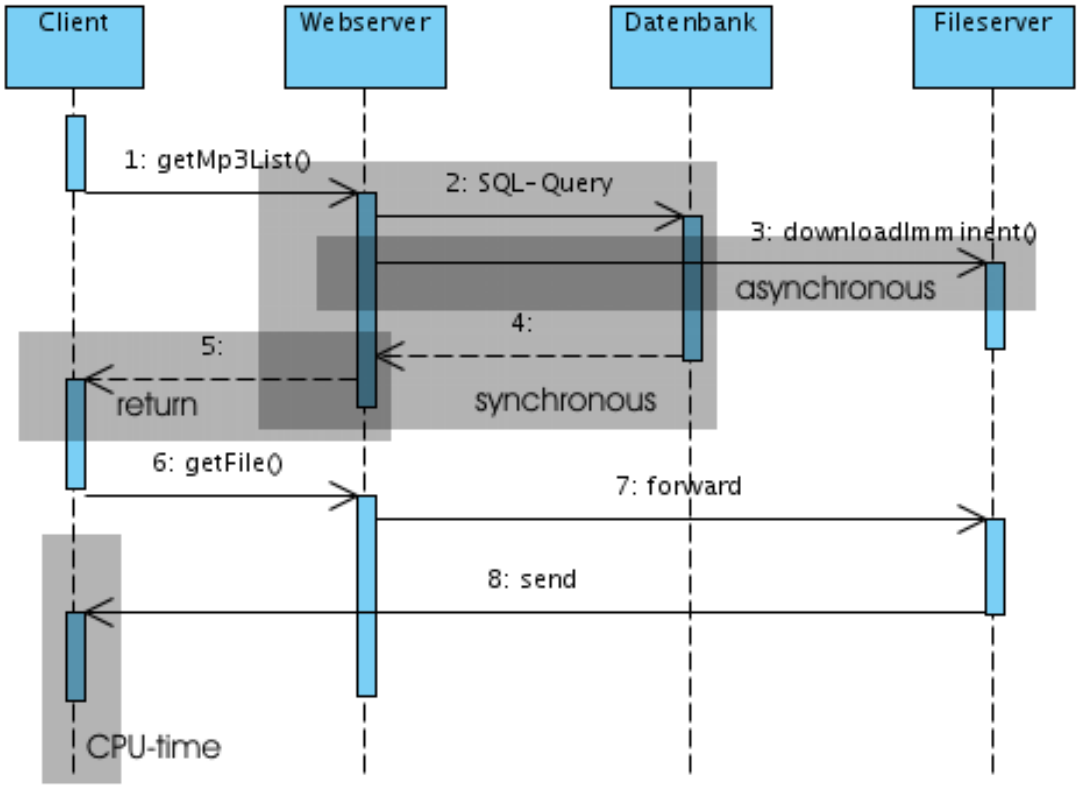


Abbildung 6: Sequenzdiagramm

Ein Sequenzdiagramm kann man sich als visualisierten Quellcode vorstellen. Es ist zweidimensional, in der Horizontalen sind nebeneinander kommunizierende Objekte angeordnet und in der Vertikalen ist der Ausführungsfluss dargestellt. Wenn immer ein Objekt mit der Ausführung von Operationen beschäftigt ist, wird im Diagramm die gestrichelte Lebenslinie durch einen Balken ersetzt. Ruft das Objekt Methoden anderer Objekte auf (d.h. schickt es eine Nachricht), gehen von ihm Pfeile aus, welche beim

anderen Objekt eintreffen. Das andere Objekt fängt nun an zu arbeiten und bekommnt auch einen Balken. Man unterscheidet zwischen mehreren Arten von Nachrichten:
1. Eine Nachricht kann synchron verschickt werden, dann kann das aufrufende Objekt so lange nicht arbeiten, bis eine Antwort zurückkommt.
2. Sie kann asynchron verschickt werden, das aufzurufende Objekt kann weiterarbeiten, während ihre Nachricht von einem anderen Objekt bearbeitet wird.
3. Es gibt erzeugende Operationen, welche Objekte generieren (die dann eine Lebenslinie bekommen)
4. Durch zerstörende Operationen wird ihre Lebenslinie terminiert.
5. Wenn ein Objekt mit der Bearbeitung einer Nachricht fertig ist, schickt es eine Antwort (return) zurück.

3.3.2 Kollaborations-Diagramm

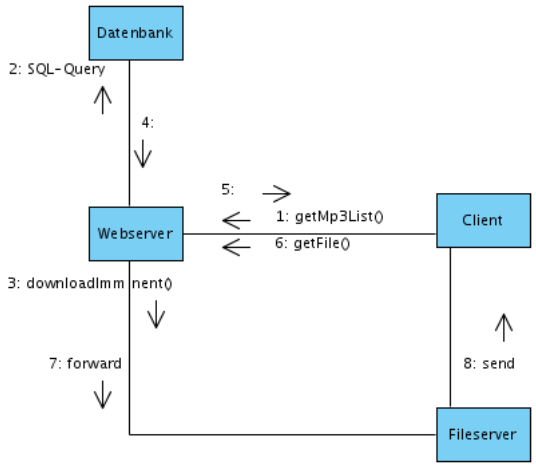


Abbildung 7: Kollaborationsdiagramm

Kollaborationsdiagramme stellen Objekte und ihre Beziehungen zueinander dadurch dar, dass Nachrichten abgebildet werden. Diese Diagramme können aus Sequenzdiagrammen generiert werden. Um die zeitliche Abfolge der Nachrichten nicht zu verlieren, sind diese mit Nummern bezeichnet. Die Nachricht mit der niedrigsten Nummer wird im Sequenzdiagramm als erste abgeschickt.

3.3.3 Deployment-Diagramm

Ein Deployment-Diagramm beschreibt, wie Komponenten örtlich verteilt sind, also auf welchen Computern bzw. Hardwarekomponenten sie sich befinden und wie sie miteinander verbunden sind (Internet, LAN). Wie in Abbildung 8 dargestellt, gibt es Quader, welche die Hardwarekomponenten symbolisieren und Rechtecke (teilweise auch in Komponentendarstellung innerhalb der Quader plaziert), welche die Artefakte beschreiben, die mit den Komponenten assoziiert sind.
Das Bild wurde mit Performanz-Annotationen versehen, welche im Kapitel UML Profile for Scheduling, Performanz and Time erläutert werden.

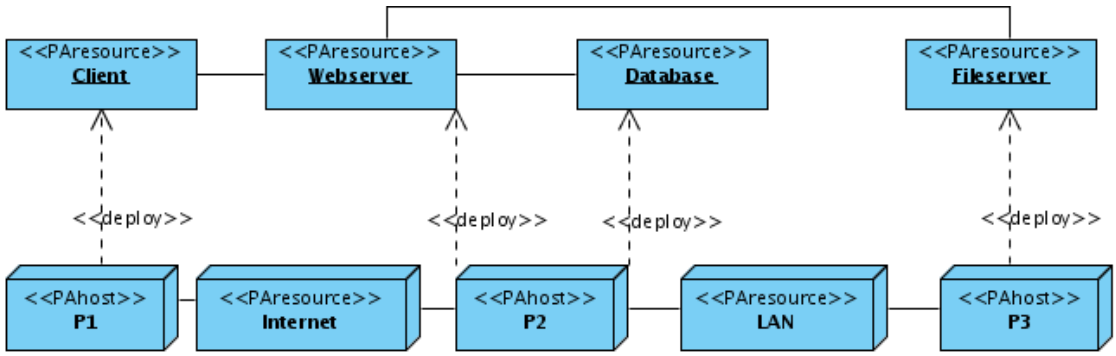


Abbildung 8: Deployment-Diagramm

3.3.4 Aktivitätsdiagramm

Ein Aktivitätsdiagramm stellt, ähnlich wie beim Sequenzdiagramm, den Befehlsfluss dar. Der Befehlsfluss kann verzweigen, was als Fork bezeichnet wird, zwei Flüsse können sich vereinigen, was man Join nennt. Der Befehlsfluss trifft immer wieder auf Actions, welche Code-Befehle symbolisieren.
Es gibt zwei verschiedene Sorten von Forks:
1. Beim And-Fork teilt sich der Befehlsfluss auf und läuft auf beiden Kanten weiter (wenn man z.B. einen Thread erstellt).
2. Beim Or-Fork teilt sich auch der Fluss auf, aber nur ein Weg wird beschritten (eine If-Verzweigung)
Es gibt auch zwei Sorten von Joins:
1. Beim And-Join werden die Befehlsflüsse synchronisiert: sobald ein Fluss den Join erreicht, muss er auf den anderen warten.
2. Beim Or-Join genügt es, wenn ein Fluss eintrifft, dieser läuft dann weiter. Ein Warten ist nicht nötig.

Aktivitäten strukturieren Actions, indem sie sie als Einheit kenntlich machen. Dargestellt wird dies durch abgerundete senkrechte Rechtecke, welche nach dem Objekt benannt sind, aus dem die Actions hervor gegangen sind.

3.4 UML Profile for Scheduling, Performanz and Time

In Abbildung 10 wurde das Aktivitätsdiagramm durch einige Annotationen erweitert. Das SPT-Profile ist in den meisten UML-Modelern noch nicht eingebaut, so dass man diese Erweiterung nur durch Annotationen dem Diagramm hinzufügen kann. Wie in Abschnitt 3.5 beschrieben wird, werden Diagramme mit Hilfe von XMI in XML Dateien umgewandelt. Annotationen werden dort ebenfalls abgespeichert und stehen daher einem LQN-Solver zur Verfügung. In den Annotationen sind deshalb die SPT-Profile Angaben angegeben.

Die Angaben geben darüber Auskunft, wie lange z.B. Actions im Aktivitätsdiagramm zur Ausführung benötigen, oder auch welche Quader im Deployment-Diagramm Prozessoren und welche Ressourcen wie z.B. Übertragungsmedien darstellen.
Die Annotationen werden dafür mit Stereotypen wie `<<PAstep>>`, `<<PAressource>>`

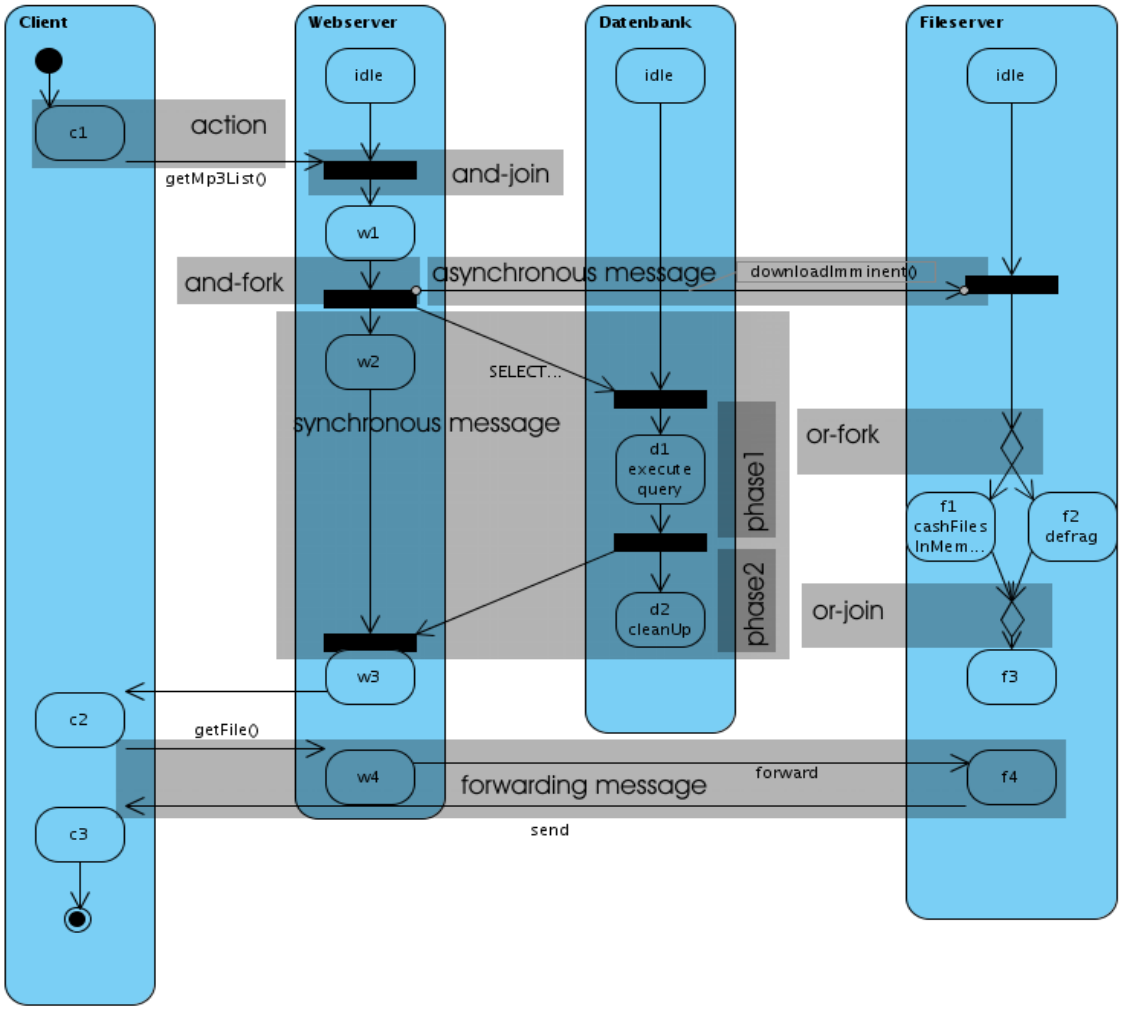


Abbildung 9: Aktivitätsdiagramm

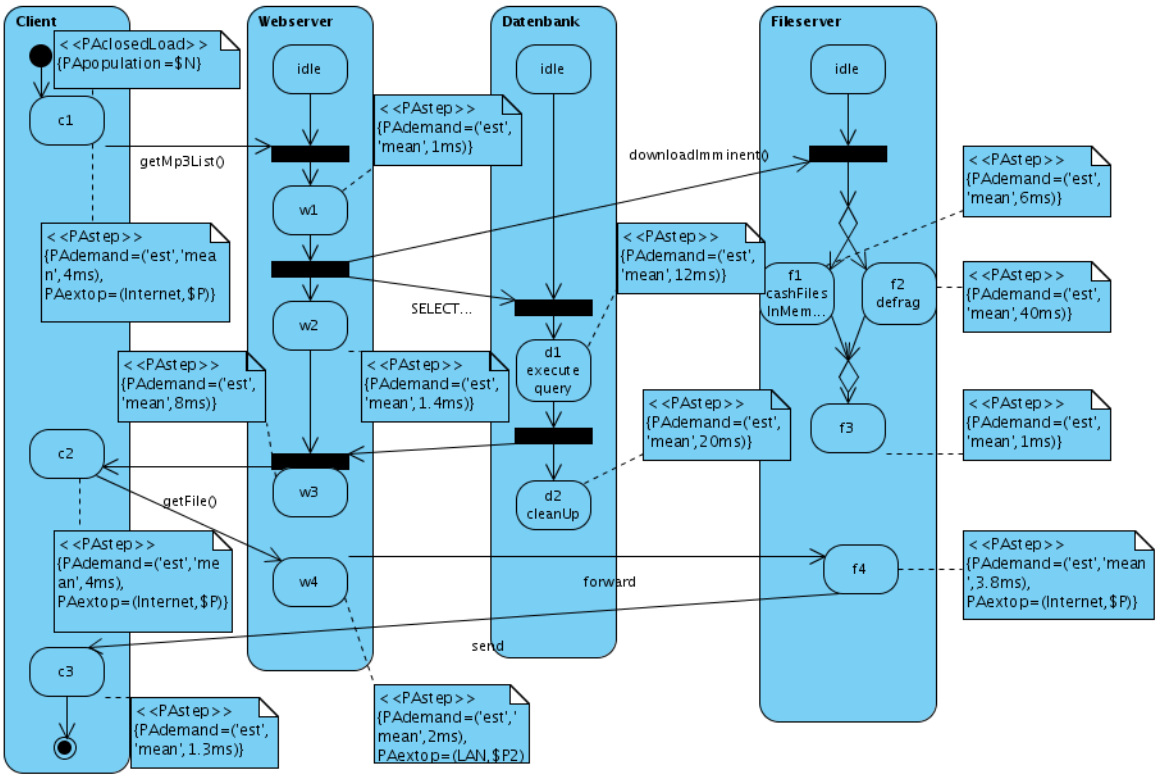


Abbildung 10: Annotiertes Aktivitätsdiagramm

und `<<PAhost>>` ausgestattet. Innerhalb der Stereotypen sind eine Reihe von Attributen möglich, die den Stereotyp genauer dbeschreiben. Eine kleine Liste aus [9] gibt die wichtigsten Attribute an:

Attribute	**Erläuterung**	**UML Tag**
res[a]	Ressource, auf der a ausgeführt wird	PAhost
demand[a]	Dauer der Ausführung von a	PAdemand
p[t]	Wahrscheinlichkeit einer bestimmte Verzweigung	PAprob
rate[r]	Die Rate, mit der Action a aufgerufen wird	PArate
arrivalrate[W]	Parameter λ aus den ersten Kapiteln	PAoccurence
res[a]	Medium über welches die Nachticht geschickt wird	PAextop

In Abbildung 10 wurde ausgiebig Gebrauch vom Stereotyp `<<PAstep>>` gemacht. Die gebräuchlichsten Parameter sind PAdemand, um die Zeit anzugeben, die diese Action benötigt und PAextop, falls es sich um Actions handelt, welche eine Nachricht über ein Medium wie z.B. Netzewerk schicken. Mit PAextop konnte angegeben werden, um was für ein Medium es sich handelte.
Im Sequenzdiagramm ist es nicht möglich, das Profil so detailliert zu verwenden wie im Aktivitätsdiagramm, da eine Abgrenzung der Befehlsflüsse in Actions nicht vorhanden ist.

3.5 XMI

Damit es überhaupt möglich ist, im weiteren Prozess die Diagramme so umzuformen, dass sie in LQNs übergehen, benötigt man ein Datenformat, welches von den betreffenden Umwandlungsmethoden verstanden wird. In Kapitel 3.6.2 wird XSLT beschrieben, eine Sprache, mit der man XML Dokumente umwandeln kann. XSLT benötigt XML Dokumente als Input, der Weg der in dieser Ausarbeitung und z.B. in [7] genommen wurde, um aus den Diagrammen eine XML Datei zu erzeugen, geht über XMI. XMI definiert z.B. über DTDs wie die Diagramme in XML auszusehen haben. XMI ist in dieser Beziehung leider noch nicht eindeutig. Es gibt viele Interpretationen darüber, wie ein in XML umgewandeltes Diagramm auszusehen hat, so dass nicht jeder UML-Modeler für jedes XSLT-Umwandlungsprogramm in Frage kommt.

Der XMI Standard enthält mehrere DTD-Produktionsregeln, um UML-Metamodelle in DTDs umzuformen, zudem XML-Dokument-Regeln, um aus den DTDs wieder UML-Metamodelle zu erzeugen, Entwurfsmuster für DTDs und bestehende Standard-DTDs für UML. XMI wird in [10] noch näher betrachtet. Aus Platzgründen verzichten wir hier auf die Darstellung eines Beispiels, da selbst das kleinste Beispiel bereits eine Seite füllt.

3.6 Graph- bzw. Baum-Umwandlungsmethoden

3.6.1 PROGRES

Programmed Graph Rewriting System (PROGRES) wurde in [5] verwendet und auch von anderen Gruppen dazu benutzt, Aktivitätsdiagramme in LQNs umzuwandeln. Ein Nachteil an PROGRES gegenüber XSLT besteht darin, dass die Aktivitätsdiagramme erstmal in ein von PROGRES lesbares Format umgewandelt werden müssen, während sie (vorausgesetzt, der UML-Modeler verwendet XMI, um die Diagramme zu exportieren) für XSLT bereits in der nötigen Form vorliegen.

PROGRES ist, wie der Name schon sagt, ein System, mit dem man einen Graphen in einen anderen Graphen umwandeln kann. Ein PROGRES Graph besteht aus verschiedenartigen Knoten und Kanten. Zur Umwandlung verwendet PROGRES Produktionen und Transaktionen. Produktionen bestehen aus einer linken und einer rechten Seite. Die linke Seite stellt den Teil dar, welcher im Ausgangsgraphen gesucht wird. Wird er gefunden, so werden die Kommandos, welche auf der rechten Seite stehen für die Umwandlung angewendet. Die Kommandos können alles mögliche mit dem Graphen machen, neue Knoten erstellen, Knoten löschen oder auch nur Attribute ändern. Transaktionen dienen dazu Produktionen zusammenzufassen. Produktionen und Transaktionen können mit Parametern aufgerufen werden, in den Produktionen können selbstdefinierte und importierte Funktionen verwendet werden, welche für Berechnungen der Werte herangezogen werden können.

3.6.2 XSLT

Extensible Stylesheet Language (XSL) wurde als Umwandlungssprache für XML Dokumente eingeführt, um diese anschaulich darstellen zu können. XSLT ist eine Erweiterung und quasi eine Programmiersprache in XML Form, mit der man nichts anderes

machen kann, als XML Dokumente in beliebige andere Dokumenttypen zu transformieren. XSLT Dokumente, sind wie gesagt XML Dokumente und müssen als solche eine XML-Deklaration, ein Root Element beinhalten und wohlgeformt sein, wie in [11] gefordert.

Der Umwandlungsprozess geschieht folgendermaßen: Die XSLT Software, welche den Umwandlungsprozess durchführt, traversiert den XML-Baum des umzuwandelnden Dokuments Knoten für Knoten und sucht nach Mustern, welche in der betreffenden XSLT-Datei aufgeführt sind. Wird ein Muster erkannt, so werden die betreffenden Unterknoten so umgewandelt, wie es die XSLT-Datei beschreibt. Das Resultat kann alles mögliche sein: Ein neues XML-Dokument, HTML oder Java-Code. In unserem Fall wird aus einer XML-Datei, welche Definitionen für Sequenzdiagramme enthält, ein Dokument, welches die Sequenzdiagramme umgewandelt in Aktivitätsdiagramme enthält. Und im zweiten Schritt, aus Dokumenten, welche Aktivitäts und Deployment-Diagramme enthalten, ein Dokument, welches das LQN in einer Form enthält, welches von bestehenden LQN-Solvern akzeptiert wird. Wie in [6] beschrieben.
Wir werden nun erklären, wie XSLT aufgebaut ist und im nächsten Schritt, wie damit Aktivität-Diagramme in LQNs umgewandelt werden können.

XSLT arbeitet mit Templates. Ein Template kann einen eindeutigen Namen haben (falls man in anderen Templates darauf zurückgreifen will) und benötigt ein `match=''''` Attribut, welches den Knoten angibt, auf welchen das Template angewendet werden soll.
Das `match` Attribut spezifiziert ein Muster, welches z.B. folgende Werte haben kann: Es kann ein Elementname oder eine Id sein, dann wird das Template auf jeden Knoten mit dieser Id angewendet. Es kann spezifizieren, dass nur Elemente, dessen Eltern einen bestimmten Typ haben umgewandelt werden sollen etc., die Liste ist noch viel länger.

Der XSLT-Parser traversiert jeden Knoten des Quelldokuments und ersetzt ihn bei Übereinstimmung mit dem Template. Innerhalb des Templates muss also angegeben werden, was und wie ersetzt werden soll. Dafür stehen wiederum andere Hilfsmittel zur Verfügung: Soll ein Knoten durch einen anderen ersetzt werden, muss er im Template mit dem Tag
`<xsl:element name=''elementname''>....</..>` erzeugt werden. Innerhalb dieses erzeugten Knotens können Attribute mit
`<xsl:attribute name=''attributename''> value </..>` erzeugt werden. Zudem kann man noch beliebigen Text, Kommentare etc. einfügen, dieser wird dann auch mit ausgegeben.

XSLT bietet allerdings noch mehr. Wir haben oben erwähnt, dass XSLT eher einer Programmiersprache gleicht. Dies wird anhand der weiteren Konstrukte deutlich. Neben dem schon erwähnten, bietet XSLT die Möglichkeit, Variablen zu definieren und Schleifen und Kontrollabfragen zu formulieren. Für diese Konstrukte sind Elemente wie `<xsl:for-each ...`, `<xsl:when..` (für if) `<xsl:otherwise..` (für else) vorhanden. Variablen werden mit `<xsl:variable name=''name''>Wert</..>` definiert. In den Templates können sie mit `$name` angesprochen werden.

3.7 Die Umwandlung

Ein Algorithmus für den Prozess ist in [6](p.230) vorgestellt worden:

1. Generierung der LQN Modell Struktur
   a) Finden von LQN Tasks aus Kollaborationsdiagrammen
   b) Finden von LQN Prozessoren aus Deployment-Diagrammen
2. Generierung von Entries, Phasen und Activites aus Sequenz und Aktivitätsdiagrammen
   a) Transformation von Sequenz in Aktivitätsdiagramme
   b) Transformation der Aktivitätsdiagramme
      i. Nachrichten zwischen Aktivitäten werden als Nachrichten zwischen Tasks abgebildet
      ii. Unterteilung des Aktivitätsdiagramms in Subgraphen, welche Entries, Phasen und Activites darstellen. Erstellung der zugehörigen LQN Elemente.
3. Durchlaufen den LQN-Baumes und Anzeige der textuellen Beschreibung des Modells.

3.7.1 Sequenzdiagramm zu Aktivitätsdiagramm

Falls sich unter den UML Diagrammen Sequenzdiagramme befinden, so müssen diese erst in Aktivitätsdiagramme umgewandelt werden, da für die endgültige Umwandlung in LQNs nur Aktivitäts und Deployment-Diagramme verwendet werden.

Wie in Abbildung 6 erkennbar, ist es auch in Sequenzdiagrammen möglich, zwischen asynchronen, synchronen und weitergeleiteten Nachrichten zu unterscheiden. Die Unterscheidung zwischen synchron und asynchron ist dabei besonders einfach für das menschliche Auge, da es dafür zwei unterschiedliche Pfeile gibt.
Eine synchrone Nachricht wird so abgebildet, dass das aufrufende Objekt A im Aktivitätsdiagramm vom letzen Knoten eine Kante zum aufgerufenen Objekt B bekommt, welche in einem And-Join endet, falls im Sequenzdiagramm die Nachricht in einem Block mündet. Mündet sie am Anfang eines Blocks, so führt lediglich eine Kante zu einer Action, welche neu erstellt wird. Es muss nun nur noch festgestellt werden, wann das Objekt B im Sequenzdiagramm eine Antwort sendet. An der betreffenden Stelle im Aktivitätsdiagramm kommt ein And-Join, falls das Objekt B nach der Antwort noch Operationen durchführt, ansonsten einfach eine Kante zum aufrufenden Objekt A. Diese Durchführung wird in [5] (Seite.105ff) beschrieben.

Bei einer asynchronen Nachricht, entfällt die (synchrone) Antwort von B und statt dass eine direkte Kante von A nach B gezogen wird, wird vorher ein And-Fork eingebaut, so dass A seine Ausführung noch weiter laufen lassen kann.

Weitergeleitete Nachrichten werden wie synchrone Nachrichten behandelt, mit dem Unterschied, dass es keine Antwort gibt von B gibt, sondern von einem weiteren Objekt C.

Erweitert man die Syntax des Sequenzdiagramms, um alternative Nachrichtenflüsse

wiederspiegeln zu können, kann man im Aktivitätsdiagramm zusätzlich zu And-Joins und -Forks auch Or-Joins und Forks darstellen. Das Stereotypen-Attribut `PAprob` kann dann dazu genutzt werden, die Wahrscheinlichkeit eines alternativen Wegs anzugeben.

3.7.2 Deployment-Diagramm zu LQN

Deployment-Diagramme sind das Grundgerüst für das LQN. Mit ihnen allein lässt sich schon fast (bis auf die Entries abgesehen) die Abbildung 4 erzeugen. `<<PAhost>>` werden auf Prozessoren gemappt. `<<PAressource>>`, welche mit Rechtecken dargestellt werden (Artefakte) auf Tasks und `<<PARessource>>`, welche auf Quadern dargestellt werden, als Verbindungen zwischen den Tasks, welche dann Verwendung bei der Umwandlung des Aktivitätsdiagramms finden. Die genaue Vorgehensweise ist in [8] beschrieben.

3.7.3 Aktivitätsdiagramm zu LQN

Aus dieser Umwandlung werden die meisten Daten für das zu erzeugende LQN gewonnen. Wie im Algorithmus dargestellt, wird das Diagramm erstmal dazu hinzugezogen, task-übergreifende Nachrichten zu bestimmen. Die Tasks wurden bereits aus dem Deployment-Diagramm heraus identifiziert.

Nachdem dies geschehen ist, wird das Diagramm in Subgraphen zerlegt. Aufeinanderfolgende Actions werden zu LQN-Activites zusammengefasst und einzelne Teile einer Nachricht zu Phasen umgewandelt, wie in Abbildung 9 zu sehen ist.
Die Generierung des LQN läuft nun folgendermaßen ab. Siehe dazu [6](p.233):
Jeder Task bekommt anfangs ein Entry eingebaut. Neue Entries werden nur hinzugefügt, falls im Aktivitätsdiagramm eine neue Nachricht an das dem Task zugehörige Objekt geschickt wird. Ist die neue Nachricht schon mal geschickt worden, so wird lediglich ein Zähler für Wiederholungen hochgezählt.

Alle Entries starten in Phase 1, wenn die Nachricht zurückgeschickt wird, werden alle nachfolgenden Actions zur Phase 2 zugeordnet.
LQN-Aktivitäten werden nur benötigt, falls im Aktivitätsdiagramm And- und Or-Forks auftreten. Diese können allein mit Phasen nicht mehr beschrieben werden. Forks und Joins aus dem Aktivitätsdiagramm können direkt so übernommen werden. Falls es ein Fork gibt, bei dem die Nachricht eine Swimlane überschreitet, muss eine zusätzliche Aktivität in das ausgehende Entry eingebaut werden, welches jedoch keinen Zeitverbrauch hat (PAdemand=0). Das Zeitverhaltenen einzelner Phasen wird durch das Aufsummieren aller Actions bestimmt.
Befolgt man diese Umwandlungen, führt dies zu dem in Abbildung 11 dargestellten Graph.

3.8 LQN-Solver

Die Lösung dieses LQN ist mit Methoden möglich, welche in [4] vorgestellt werden. Es gibt sogenannte LQN-Solver, die diese Aufgabe übernehmen.

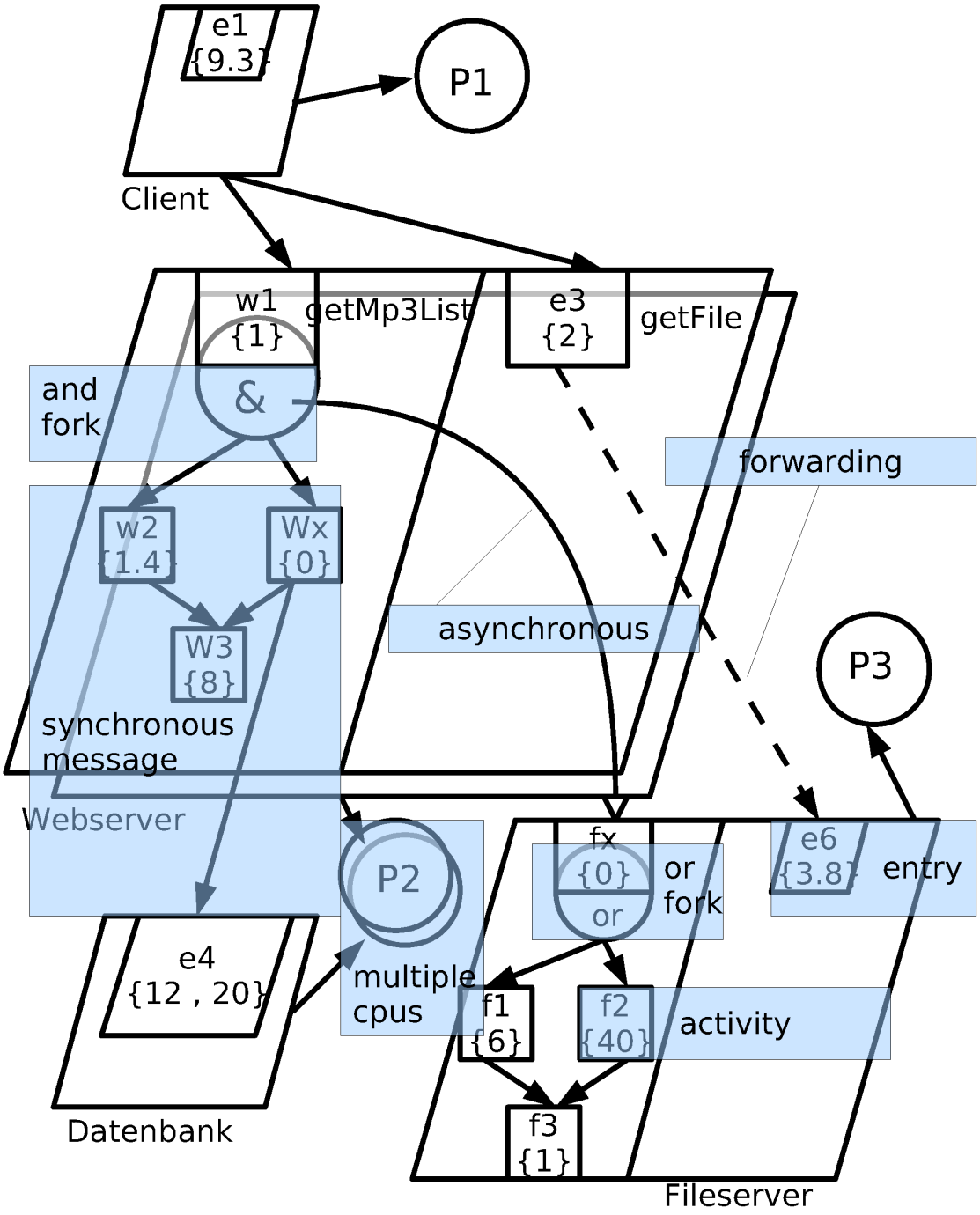


Abbildung 11: LQN mit Aktivitäten

4 Ausblick auf zukünftige Forschungen

Als weitere Ziele nennt z.B. [6] die Erweiterung der Transformation um mehr Design-Patterns und die Möglichkeit mehr als ein Aktivitätsdiagramm einzubeziehen. Zudem befassen sich die Autoren ausschließlich mit LQNs und würden gerne weitere Warteschlangenmodelle simulieren können.
Die Autoren von [9] arbeiten dagegen schon seit einiger Zeit mit QNs und würden gerne ihre Arbeiten auf LQNs übertragen. Das große Ziel ist es, dem Benutzer die Möglichkeit zu geben, das Performanz-Modell auszuwählen, mit welchem man die Systeme am besten abbilden kann.
Zudem werden derzeit nur relativ wenige der zahlreichen UML-Diagramme für die Umwandlung in LQNs oder QNs verwendet. Nach Möglichkeit soll dies auch erweitert werden, um die Abbildungen noch realistischer gestalten zu können.

5 Fazit

Performanz-Evaluation ist eine wichtige Methode, um Softwareprojekte sicherer zu einem erfolgreichen Abschluss zu führen. Sie spart Geld und schafft Vertrauen in das eigene Produkt. Performanz-Engpässe, welche früh in der Entwicklungsphase erkannt und behoben werden, können sich nicht mehr auf spätere Entwicklungsphasen auswirken und das Projekt zu Fall bringen. Zu Beginn eines Projekts stehen meist nur Architekturdaten und noch kein ausführbarer Code zur Verfügung. Diese Architekturdaten werden heutzutage üblicherweise mit der UML beschrieben.

Die Artikel [5], [6], [7], [8], [9] und viele andere beschreiben Methoden, wie man diese UML Diagramme in Warteschlangennetze überführen kann. Warteschlangennetze eignen sich dafür, den Lauf eines Prozesses durch Hard- und Softwarekomponenten, welche z.B. CPU, Hauptspeicher, Festplatte oder auch Webserver, Datenbanken etc. darstellen können, zu simulieren. Durch Betrachtung des Verhaltens des Systems über lange Zeit mit hoher Prozessdichte, ist es möglich Engpässe aufzudecken. In dieser Ausarbeitung wird besonders auf die Umwandlung der UML-Diagramme in Warteschlangennetze eingegangen. Es wurde ein Weg vorgestellt, welcher XMI zur Speicherung der Daten und XSLT zur Graphentransformation benutzt.

Der eigentliche Umwandlungsalgorithmus benutzt Kollaborations- und Deployment-Diagramme um die Grobstruktur der mehrschichtigen Warteschlangennetze (LQNs) zu generieren und Sequenz- und Aktivitätsdiagramme um dem LQN Aktivitäten und Entries hinzuzufügen. Die Forschung an Warteschlangennetzen ist mehrere Jahre alt, so dass es mittlerweile viele Warteschlangenmodelle, Transformationsmethoden und Programme zur Simulation dieser gibt. Ein Problem besteht noch darin, dass Inkompatibilitäten zwischen den verschiedenen Schritten der Umformung gibt, so dass nicht alle UML-Modellbildungsprogramme und LQN-Solver miteinander arbeiten können.

Abbildungsverzeichnis

Literatur

[1] C. U. Smith and L. G. Williams, *Performance Solutions: A Practical Guide to Creating Responsive, Scalable Software*. Addison-Wesley, 2002.

[2] K. Neumann and M. Morlock, *Operations Research*, 2nd ed. Hanser, 2002.

[3] P. C. Karl Grill, Klaus Berger, "Warteschlangentheorie," 1999.

[4] E. Ghazi, "Performanz Modellierung mit Queueing Networks," in *Seminar modellgetriebene Softwareentwicklung - Architekturen, Muster und Eclipse-basierte MDA*, Karlsruhe, Germany, 2006.

[5] H. Amer, "Automatic transformation of UML software specification into LQN performance models using graph grammar techniques," 2001.

[6] G. P. Gu and D. C. Petriu, "XSLT transformation from UML models to LQN performance models," in *WOSP '02: Proceedings of the 3rd international workshop on Software and performance*. New York, NY, USA: ACM Press, 2002, pp. 227–234.

[7] ——, "From UML to LQN by XML algebra-based model transformations," in *WOSP '05: Proceedings of the 5th international workshop on Software and performance*. New York, NY, USA: ACM Press, 2005, pp. 99–110.

[8] A. D'Ambrogio, "A model transformation framework for the automated building of performance models from UML models," in *WOSP '05: Proceedings of the 5th international workshop on Software and performance*. New York, NY, USA: ACM Press, 2005, pp. 75–86.

[9] S. Balsamo and M. Marzolla, "Performance evaluation of UML software architectures with multiclass Queueing Network models," in *WOSP '05: Proceedings of the 5th international workshop on Software and performance*. New York, NY, USA: ACM Press, 2005, pp. 37–42.

[10] Oasis-Open, "Cover pages: Xml metadata interchange (xmi)," 2006, zuletzt konsultiert: 15. 6. 2006. [Online]. Available: http://www.oasis-open.org/cover/xmi.html

[11] Jan Winkler, "Htmlworld - xslt," 2006, zuletzt konsultiert: 17. 6. 2006. [Online]. Available: http://www.html-world.de/program/xslt_ov.php

Teil IV.

Performanz

Performance-Modellierung mit Queuing Networks

Aboubakr Achraf El Ghazi

Betreuer: Michael Kuperberg

Zusammenfassung

Der Erfolg eines Softwareprojekts ist in erster Stelle davon abhängig, wie gut die im Pflichtenheft von den Kunden definierten Ziele bzw. Bedingungen eingehalten werden. Der Grad der Erfüllung dieser Bedingungen definiert die Qualität des Softwaresystems. Die Softwarequalität kann man in eine funktionale und nichtfunktionale Aspekte unterteilen, wobei zu den letzteren auch Performance zählt. Für bestimmte funktionale Aspekte wurden in der letzten Zeit Lösungen wie z.B. das KeY Projekt für Beweis der Korrektheit [Uni06] präsentiert. Für die Performance wurde in der Vergangenheit dagegen stark auf heuristische und simulationsbasierte Lösungen gesetzt. Heute ist durch die UML sowie andere Modellbeschreibungssprachen die Möglichkeiten gegeben, den Softwareentwurf genauer zu beschreiben. Dadurch ist man in der Lage, mit formalen Methoden die Performance-Qualität des Softwareentwurf zu berechnen, um noch auf der Entwurfsebene die nötigen Korrekturen vorzunehmen. Ein sehr geeignetes Modell für die Performancemodellierung ist durch die Warteschlangenthorie gegeben. Wir werden in dieser Arbeit die Theorie der Warteschlangen und Warteschlagennetzen erläutern sowie die mathematischen Grundlagen, die für das Lösen solcher Modelle herangezogen werden kennenlernen. Dann werden wir kurz darstellen, welche Methoden in aktueller Forschung benutzt werden, um den Softwareentwurf durch ein Warteschlangenmodell abzubilden. Anschließend werden wir die grundlegenden Verfahren, die ein Warteschlangen-Modell lösen, präsentieren.

1 Einleitung

Leistungsbewertung hat bei den heutigen Softwaresystemen aufgrund der ständig steigenden Komplexität eine immer wichtigere Bedeutung erlangt und sollte deswegen bereits während der Entwurfsphase berücksichtigt werden. Da Messungen während der Entwurfs- und Entwicklungsphase häufig nicht möglich sind, müssen Modellbildungstechniken angewendet werden. Dabei ist zu beachten, dass das Erstellen dieser Modelle sowie ihre Analyse nicht aufwendiger als die Simulation auf einer prototypischen Hardware sein sollte.

2 Grundlagen

Am Anfang steht eine kurze Einführung in die Grundlagen und Begriffe der Wahrscheinlichkeitstheorie, bei der wir uns an [Bol89] halten. Wir werden uns allerdings auf die Begriffe beschränken, die wir weiter unten verwenden werden. Weitergehende Einzelheiten können z.B. bei [All90] oder [Fel68] nachgelesen werden.

2.1 Wahrscheinlichkeitstheoretische Grundlagen

Eine Zufallsvariable ist eine Funktion, die das Ergebnis eines zufallsbedingten Vorgangs ausdrückt. Man unterscheidet zwischen diskreten Zufallsvariablen und kontinuierlichen Zufallsvariablen.

Diskrete Zufallsvariablen Eine Zufallsvariable, die nur diskrete Werte annehmen kann, bezeichnet man als diskrete Zufallsvariable.
Beschrieben wird die diskrete Zufallsvariable durch die möglichen Werte, die sie annehmen kann und die Wahrscheinlichkeit für diese Werte. Die Menge dieser Wahrscheinlichkeiten nennt man Wahrscheinlichkeitsverteilung oder kurz Verteilung der Zufallsvariablen. Sind üblicherweise die möglichen Werte einer Zufallsvariable X die nichtnegativen ganzen Zahlen, dann ist die Wahrscheinlichkeitsverteilung durch die Wahrscheinlichkeitsfunktion

$$p_k = P(X = k) \quad \text{für } k = 0, 1, 2... \tag{1}$$

gegeben. Es muss gelten:

$$P(X = k) \geq 0, \sum_k P(X = k) = 1. \tag{2}$$

Aus der Wahrscheinlichkeitsfunktion einer diskreten Verteilung können weitere wichtige Parameter abgeleitet werden:

Der Erwartungswert

$$\bar{X} = E[x] = \sum_k k \cdot P(X = k) \tag{3}$$

Die Funktion einer Zufallsvariable ist wieder eine Zufallsvariable mit dem Erwartungswert

$$E[f(x)] = \sum_k f(k) \cdot P(X = k) \tag{4}$$

Auf diese Weise kann man weitere nützliche Zufallsvariablen definieren.

Das n-tes MOMENT

$$\bar{X^n} = E[X^n] = \sum_k k^n \cdot P(X = k) \tag{5}$$

ist der Erwartungswert der n-ten Potenz von x.

Das n-tes ZENTRALES MOMENT

$$\overline{(X - \bar{X})^n} = E[(X - E[X])^n] = \sum_k (k - \bar{X})^n \cdot P(X = k) \tag{6}$$

ist der Erwartungswert der n-ten Potenz der Differenz zwischen X und dem Mittelwert von X.

Das zweite ZENTRALE MOMENT bezeichnet man als die Varianz von X

$$\sigma_X^2 = var(X) = \overline{(X - \bar{X})^2} \tag{7}$$

σ_X heißt Standardabweichung.

Der VARIATIONSKOEFFIZIENT ist die normierte Standardabweichung

$$c_X = \frac{\sigma_X}{\bar{X}} \tag{8}$$

Kontinuierliche Zufallsvariablen Eine Zufallsvariable X, die alle Werte im Intervall $[a, b]$ annehmen kann, d.h. $-\infty \leq a < b \leq \infty$, bezeichnet man als KONTINUIERLICHE ZUFALLSVARIABLE. Sie wird beschrieben durch die dazugehörige VERTEILUNGSFUNKTION

$$F_X(x) = P(X \leq x) \tag{9}$$

die für alle Werte x aus der Wertemenge von X die Wahrscheinlichkeit dafür angibt, dass der Wert der Zufallsvariablen X kleiner oder gleich einem betrachteten x ist.
Statt der Verteilungsfunktion kann auch die DICHTEFUNKTION $f_X(x)$ verwendet werden,

$$f_X(x) = \frac{dF_X(x)}{dx} \quad \text{mit } f_X(x) \geq 0 \text{ für alle x}, \quad \int_{-\infty}^{+\infty} f_X(x)\mathrm{d}x = 1 \tag{10}$$

Die Dichtefunktion einer kontinuierlichen Zufallsvariablen entspricht der Wahrscheinlichkeitsfunktion einer diskreten Zufallsvariablen. Die Formeln für den Mittelwert und die höheren Momente einer kontinuierlichen Zufallsvariablen erhält man deshalb aus den Formeln für diskrete Zufallsvariablen z.B.

$$\bar{X} = E[X] = \int_{-\infty}^{+\infty} x \cdot f_X(x)\mathrm{d}x \quad \text{und}$$
$$E[g(x)] = \int_{-\infty}^{+\infty} g(x) \cdot f_X(x)\mathrm{d}x$$

Exponentialverteilung Die Exponentialverteilung ist die wichtigste Verteilung in der Warteschlangentheorie. Viele Ankunfts und Bedienprozesse lassen sich damit exakt oder näherungsweise beschreiben. Ausserdem ist sie die einzige kontinuierliche Verteilung, welche die MARKOV-EIGENSCHAFT (siehe 2.3) der Gedächtnislosigkeit besitzt. Die Verteilungsfunktion einer exponentiell verteilten Zufallsvariablen X ist gegeben durch

$$F_X(x) = 1 - exp\left(-\frac{x}{X}\right); \quad x \geq 0, \tag{11}$$

$$\text{Dichtefunktion:} \quad f_X(x) = \lambda e^{-\lambda x}, \quad x > 0 \tag{12}$$

$$\text{Mittelwert:} \quad \bar{X} = \frac{1}{\lambda}, \tag{13}$$

$$\text{Varianz:} \quad var(X) = \frac{1}{\lambda^2}. \tag{14}$$

2.2 Warteschlangennetze

Elementare Wartesysteme Ein elementares Wartesystem (Abb. 1) besteht aus einer Warteschlange sowie einer oder mehrerer identischer BEDIENEINHEITEN, in denen Aufträge bedient werden. Solch ein Wartesystem wird in der Warteschlangentheorie auch BEDIENSTATION oder KNOTEN genannt.

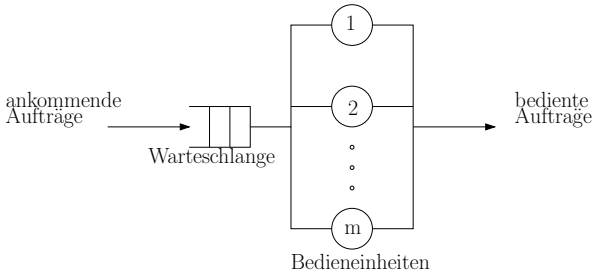


Abbildung 1: Bedienstation mit m Bedieneinheiten

Eine einzelne Bedieneinheit eines Wartesystems kann immer nur einen Auftrag gleichzeitig bedienen. Sie befindet sich deshalb stets in einem der beiden Zustände belegt oder frei. Wird ein Auftrag in einer Bedieneinheit erledigt und diese somit frei, dann wird aus der gemeinsamen Warteschange ein neuer Auftrag entsprechend einer WARTESCHLANGENDISZIPLIN ausgewählt, mit dessen Bedienung daraufhin begonnen wird.
Ein Wartesystem ist weiterhin charakterisiert durch die ZWISCHENANKUNFTZEIT t_a zwischen nacheinander ankommenden Aufträgen und durch die BEDIENZEIT t_b für einzelne Aufträge. Da der Zugang der Aufträge genau wie deren Abfertigung in der Regel zufällig erfolgt, sind die Zwischenankunftszeit und die Bedienzeit Zufallsgrößen und damit durch ihre Verteilung gegeben.
Die ANKUNFTRATE bzw. die BEDIENRATE lassen sich wie folgt berechnen:

$$\lambda = \frac{1}{\bar{t_a}}$$

$$\mu = \frac{1}{\bar{t_b}}$$

Einheitliche Beschreibung elementarer Wartesysteme Zur einheitlichen Beschreibung elementarer Wartesysteme hat sich die KENDALL'SCHE NOTATION durchgesetzt:

A/B/m-Warteslangendiziplin

Mit A die Verteilung der Zwischenankuftszeiten und B die Verteilung der Bedienzeiten des Wartesystems, wahrend m die Anzahl der identischen Bedieneinheiten angibt ($m \geq 1$).
Für A und B werden fogenden Symbole verwendet:

M	Exponentialverteilung
E_k	Erlang-Verteilung mit k Phasen
H_k	Hyperexponentialverteilung mit k Phasen

Die Warteschlangendisziplin legt fest, welcher Auftrag aus der Warteschlange als nächstes zur Bedienung ansteht.
Die wichtigsten Warteschlangendisziplin und ihre Abkurzung werden aufgezählt:

FCFS	First-Come-First-Served
LCFS	Last-Come-First-Served
SIRO	Service-In-Random-Order, d.h. die Auswahl erfogt zufällig
RR	Round Robin, dh. ist die Bedienung eines Auftrags nach einer fest vorgegebenen Zeitscheibe noch nicht beendet, so wird der Auftrag verdrängt und wieder in der Warteschlange eingereiht, die nach FCFS abgearbeitet wird

Redet man über eine Warteschlange ohne weiteren Spezifikationen, meint man eine Warteschlange der Sorte M/M/1-FCFS.

Leistungsgrößen
ZUSTANDSWAHRSCHEINLICHKEIT $p(k)$:
p(k) ist die Wahrscheilichkeit, dass es sich k Aufträge im Wartesystem befinden. Ist eine wichtige Leistungsgröße denn hieraus lassen sich die Mittelwerte aller anderen interessanten Leistungsgrößen ableiten.

AUSLASTUNG ρ:
Bei Wartesystemen mit einer Bedieneinheit gibt die Auslastung ρ den Bruchteil der Gesamtzeit an, an der die Bedieneinheit aktiv ist an. Sie berechnet sich zu:

$$\begin{aligned} \rho &= \frac{\frac{1}{t_b}}{\frac{1}{t_a}} \\ &= \frac{\text{Ankunftsrate}}{\text{Bedienrate}} \\ &= \frac{\lambda}{\mu} \end{aligned} \tag{15}$$

Bei mehreren Bedieneinheiten gibt die Auslastung auch den mittleren Anteil der aktiven Bedieneinheiten an. Da in diesem Fall $m \cdot \mu$ die Gesamtbedienrate ist gilt:

$$\rho = \frac{\lambda}{m \cdot \mu} \tag{16}$$

Mit Hilfe von ρ kann die sogennante GLEICHGEWICHTSBEDINGUNG für Netze im statischen Gleichgewicht bzw. im GLEICHGEWICHTSZUSTAND formuliert werden. Für ein System im statistischen Gleichgewicht muss gelten:

$$\rho < 1 \tag{17}$$

DURCHSATZ λ:
Der Durchsatz eines elementaren Wartesystem ist definiert als die mittlere Anzahl von Aufträgen, die pro Zeiteinheit fertig bedient werden, also die Abgangsrate. Da im statistischen Gleichgewicht die Abgangsrate eines Wartesystems gleich der Ankunftsrate λ dieses Wartesystems ist, berechnet sich der Durchsatz wie folgt:

$$\lambda = m \cdot \rho \cdot \mu \tag{18}$$

Da man im allgemein nur an Wartesystemen im statistischen Gleichgewicht interessiert ist, betrachet man den Durchsatz und die Ankunftsrate als das Selbe und bezeichnet beide mit λ.

ANTWORTZEIT t:
Die Gesamtheit der Zeit, die ein Auftrag im Wartesystem verbringt.

WARTEZEIT w:
Die Zeit, wie lange ein Auftrag in der Warteschlangen warten muss bis seine Bearbeitung beginnt.

WARTESCHLANGENLÄNGE Q:
Die Anzahl der Aufträge in der Warteschlange.

ANZAHL DER AUFTRÄGE IM WARTESYSTEM bzw. FÜLLUNG k:
Der Mittelwert $\bar{k}$ der Anzahl der Aufträge läßt sich ebenso wie die mittlere Wateschlangenläge $\bar{Q}$ über DAS GESETZ VON LITTLE eines der wichtigsten Gesetze der Warteschlangentheorie berechnen.

$$\bar{k} = \lambda \cdot \bar{t} \tag{19}$$

$$\bar{Q} = \lambda \cdot \bar{w} \tag{20}$$

$$\text{mit} \quad \bar{k} = \sum_{k=1}^{\infty} k \cdot p(k), \quad \bar{Q} = \sum_{k=1}^{\infty} Q \cdot p(Q) \tag{21}$$

Offene Warteschlangennetze Ein Warteschlangennetz ist OFFEN, wenn Aufträge von außerhalb des Netzes ankommen und Aufträge dieses Netz auch verlassen können.

Geschlossene Warteschlangennetze Ein Warteschlangennetz ist GESCHLOSSEN, wenn keine externen Auftragsankünfte und abgänge möglich sind.

Formale Beschreibung von Warteschlangennetzen Zur formalen Beschreibung von Warteschlangennetzen verwenden wir die nachfolgenden Bezeichnungen.

N	gibt die Anzahl der Knoten des Netzes an
K	bezeichnet bei geschlossenen Netzen die konstante Anzahl der Auträge im Netz
$(k_1, k_2, \ldots, k_m)$	ist der Zustand des Warteschlangennetzes, wobei
k_i	die Anzahl der Aufträge im i-ten Knoten angibt. Für geschlossene Netze gilt: $\sum_{i=1}^{N} k_i = K$
m_i	ist die Anzahl der parallelen Bedieneinheiten des i-ten Knoten ($m_i \geq 1$)
μ_i	ist die mittlere Bedienrate von Aufträgen im Knoten i
$\frac{1}{\mu_i}$	ist die mittlere Bedienzeit eines Auftags im Knoten i
p_{ij}	ist die Wahrscheinlichkeit, dass ein in Knoten i fertiggestellter Auftrag zum Knoten j überwechselt
p_{0j}	die Wahrscheinlichkeit (bei offenen Netzen), dass ein von außen kommender Auftrag zuerst den Knoten j betritt
p_{i0}	die Wahrscheinlichkeit, dass ein Auftrag nach Abfertigung durch Knoten i das Netz anschließend verläßt, gleichwertig mit $1 - \sum_{j=1}^{N} p_{ij}$
λ_{0i}	ist die mittlere Ankunftsrate von außen beim i-ten Knoten, und
λ_i	ist die mittlere Ankunftsrate von Auftägen bei Knoten i
e_i	ist die mittlere Anzahl der Besuche eines Auftrags beim i-ten Knoten, auch bekannt als BESUCHSHÄUFIGKEIT bzw. ANKUNFTSRATE, es gilt $e_i = \frac{\lambda_i}{\lambda}$ wobei λ den Gesamtdurchsatz des Netzes bezeichnet

Zur Berechnung der MITTLEREN ANKUNFTSRATEN λ_i bei den Knoten $i = 1, \ldots, N$ müssen im offenen Netz die Ankünfte von außerhalb des Netzes und die Ankünfte von allen internen Knoten summiert werden. Da im statischen Gleichgewicht die mittlere

Ankunftsrate an einem Knoten gleich der mittleren Abgangsrate diesem Knoten ist, erhalten wir:

$$\lambda_i = \lambda_{0i} + \sum_{j=1}^{N} \lambda_j p_{ji} \quad \text{für } i = 1, \ldots, N \tag{22}$$

Dieses Gleichungssystem bekommt für geschlossene Netze die Form:

$$\lambda_i = \sum_{j=1}^{N} \lambda_j p_{ji} \quad \text{für } i = 1, \ldots, N \tag{23}$$

Die Besuchshäufigkeiten können auch unmittelbar aus den Übergangswahrscheinlichkeiten bestimmt werden.
Für offene Netze gilt, da $\lambda_{0i} = \lambda \cdot p_{0i}$ ist:

$$e_i = p_{0i} + \sum_{i=1}^{N} e_j p_{ji} \quad \text{für } i = 1, \ldots, N \tag{24}$$

Für geschlossene Netze gilt:

$$e_i = \sum_{i=1}^{N} e_j p_{ji} \quad \text{für } i = 1, \ldots, N \tag{25}$$

Leistungsgrößen von Warteschlangennetzen Im Bezug zu bereits definierten Leistungsgrößen für elementare Warteschlangen wollen wir die Definitionen hier für Warteschlangennetze erweiteren.

ZUSTANDSWAHRSCHEINLICHKEIT $p(k_1, \ldots, k_N)$ des Netzes:
Es gilt stets die Normalisierungsbedingung, dass

$$\sum_{\sum_{j=1}^{N} k_j = K} p(k_1, \ldots, k_N) = 1, \text{für} \quad (0 \leq k_i \leq K) \tag{26}$$

RANDWAHRSCHEINLICHKEITEN $p_i(k)$ im Knoten-i:

$$p_i(k) = \sum_{\sum_{j=1 und k_i = k}^{N} k_j = K} p(k_1, \ldots, k_N) \tag{27}$$

AUSLASTUNG ρ_i im Knoten-i:
Für einen Single-Server-Knoten[1] gilt:

$$\rho_i = \sum_{k=i}^{\infty} p_i(k), \text{und} \tag{28}$$

$$\rho_i = 1 - p_i(0) \tag{29}$$

[1] Eine elementare Warteschlange mit $m = 1$

Für einen Multiple-Server-Knoten[2] gilt:

$$\begin{aligned}\rho_i &= \frac{1}{m_i}\sum_{k=0}^{\infty} min(m_i,k)p_i(k) \\ &= 1 - \sum_{k=0}^{m_i-1}\frac{m_i-k}{m_i}\cdot p_i(k)\end{aligned} \tag{30}$$

DURCHSATZ λ_i im Knoten-i (im Falle konstanter Bedienraten):

$$\lambda_i = m_i \cdot \rho_i \cdot \mu_i \tag{31}$$

MITTLERE ANZAHL VON AUFTRÄGEN $\bar{k}_i$ im i-ten Knoten:

$$\bar{k}_i = \sum_{k=1}^{\infty} k \cdot p_i(k) \tag{32}$$

MITTLERE WARTESCHLANGENLÄGE $\bar{Q}_i$ im i-ten Knoten mit $m_i = 1$:

$$\bar{Q}_i = \sum_{k=1}^{\infty}(k-1)\cdot p_i(k) \tag{33}$$

MITTLERE ANTWORTZEIT $\bar{t}_i$ im i-ten Knoten:

$$\bar{t}_i = \frac{\bar{k}_i}{\lambda_i} \tag{34}$$

2.3 Markov-Prozesse

Markov-Prozesse sind sehr leistungsfähige Hilfsmittel zur Modellierung und Leistungsbewertung von Softwaresystemen.
Wir werden zunächst die wichtigste Eigenschaft von speziellen Markov-Prozesse, den sogenanten Markov-Ketten erklären und zeigen, wie das Verhalten von Warteschlangenmodellen mit ihrer Hilfe präzise beschrieben werden kann und anschließend zeigen, wie durch die Analyse von Markov-Ketten, die Zustandswahrscheinlichkeiten und hieraus die Leistungsgrößen eines Warteschlangenmodells bestimmt werden können.

Markov-Ketten Die formale Definition für zeitkontinuierliche Markov-Ketten lautet: Ein stochastischer Prozess $\{X(t), t \geq 0\}$ bildet eine zeitkontinuierliche Markov-Kette, wenn für alle $n \in \mathbb{N}$, alle $x_k \in Z$ und alle $\{t_1, t_2, \ldots, t_{n+1}\}$ mit $t_1 \leq t_2 \ldots \leq t_{n+1}$ gilt:

$$\begin{aligned}P(X(t_{n+1}) = x_{n+1} | X(t_1) = x_1, X(t_2) = x_2, \ldots, X(t_n) = x_n) \\ = P(X(t_{n+1}) = x_{n+1} | X(t_n) = x_n)\end{aligned} \tag{35}$$

Betrachten wir nun die als MARKOV-EIGENSCHAFT bekannte Gl. (35). Der Ausdruck auf der rechten Seite dieser Gleichung bezeichnet die ÜBERGANGSWAHRSCHEINLICHKEIT der Markov-Kette, wofür wir jetzt schreiben,

$$p_{ij}(s,t) = P(X(t) = j | X(s) = i), t > s. \tag{36}$$

[2]Eine elementare Warteschlange mit $m \geq 1$

mit $p_{ij}(s,t)$ ist die bedingte Wahrscheinlichkeit dafür, dass sich der Prozess zur Zeit t im Zustand j befindet, wenn er zur Zeit s im Zustand i ist. Hängen die bedingten Wahrscheinlichkeiten nicht vom Zeitpunkt t ab, sondern nur von der Zeitdifferenz $\tau = t - s$, so wird die Markov-Kette HOMOGEN genannt. Gl. (36) kann in diesem Fall durch Weglassen von t vereinfacht geschrieben werden:

$$p_{ij}(\tau) = P(X(s+\tau) = j | X(s) = i).$$

Außerdem gilt:

$$p_{ij} \geq 0,\ i,j \in Z,$$

$$\sum_{j \in Z} p_{ij}(\tau) = 1,\ i \in Z,$$

$$p_{ij}(\tau) = \sum_{k \in Z} p_{ik}(\tau - \theta) p_{kj}(\theta),\ i,j \in Z. \tag{37}$$

Die Gl. (37) ist als CHAPMAN-KOLMOGOROV-GLEICHUNG für homogene Markov-Ketten. Zur Erläuterung dieser Gleichung betrachten wir Folgende Überlegung. Geht ein Prozess in der Zeit τ vom Zustand i in einen anderen Zustand j über, so muss er sich nach Ablauf des Zeitintervalls $\tau - \theta$ in irgendeinem Zwischenzustand k befinden. Ausgehend von diesem Zwischenzustand gelangt er dann in der Zeit θ zum Endzustand j. Da sich der Prozeß nach Ablauf des Zeitintervalls $\tau - \theta$ in irgendeinem von vielen möglichen Zwischenzuständen befinden kann, erhält man durch Summation über sämtliche möglichen Zwischenzustände k alle Prozessverläufe, die in der Zeit τ von i nach j führen.

Unser Ziel, bei der Untersuchung von Markov-Prozessen, ist die Bestimmung der ZUSTANDSWAHRSCHEINLICHKEITEN $p_j = P(X(t) = j)$, diese geben an, mit welcher Wahrscheinlichkeit sich der Prozess zur Zeit t im Zustand j befindet. Für einen gegebenen Anfangsbedingungen $p_i(0) = P(X(0) = i)$ gilt:

$$p_j(t) = \sum_{i \in Z} p_{ij}(t) p_i(0),\ j \in Z. \tag{38}$$

Somit ist eine Markov-Kette durch die Wahrscheinlichkeiten $p_i(0)$ für die Anfangszustände und durch die Übergangswahrscheinlichkeiten vollständig definiert. Im allgemein kann die Gl. (37) für die Berechnung der $p_{ij}(t)$ benutzt werden. Da dies im allgemeinen schwierig ist. Werden wir eine Reihe von Eigenschaften, die eine spezielle klasse von Markov-Ketten beschreiben, definieren und anschließend für diese Klasse eine vereinfachte und praktische Gleichung zur Berechnung von die Zustandswahrscheinlichkeiten angeben.

Der STATIONÄRE ZUSTAND stellt sich genau dann ein, wenn der Prozess hinreichend lange Zeit gelaufen ist bis sämtliche Einschwingvorgänge abgeklungen sind. die Wahrscheinlichkeit für einen bestimmten Prozesszustand ist dann unabhängig vom Anfangszustand des Prozesses und ändert sich auch nicht mehr mit der Zeit. Man sagt der Prozess befindet sich im STATISTISCHEN GLEICHGEWICHT.

Eine Markov-Kette heißt IRREDUZIBEL, wenn jeder Zustand der Kette ausgehend von jedem anderen Zustand der Kette erreicht werden kann, d.h. $p_{ij}(t) > 0$ für alle $i,j \in Z$.

Ein Zustand einer Markov-Kette heißt TRANSIENT, wenn der Prozess nach Verlassen dieses Zustandes mit einer Wahrscheinlichkeit größer als Null nie wieder in diesen Zustand zurückkehrt.

Ein Zustand einer Markov-Kette heißt REKURRENT, wenn der Prozess nach Verlassen dieses Zustandes mit einer Wahrscheinlichkeit größer als Null mit Wahrscheinlichkeit Eins in diesen Zustand zurückkehrt.
Die nach Verlassen dieses Zustandes bis zur ersten Rückkehr verstrichene Zeit heißt REKURRENZZEIT.

Ein Zustand einer Markov-Kette heißt REKURRENT NICHTNULL bzw. REKURRENT NULL, wenn die mittlere Rekurrenzzeit für diesen Zustand endlich bzw. unendlich ist.

Mann kann zeigen [Bol89], dass ein stationärer Zustand existiert, wenn alle Zustände der Markov-Kette rekurrent nichtnull sind. In diesem Fall werden die Zustände der Markov-Kette, wie auch die Markov-Kette selbst ERGODISCH genannt.
Für eine ergodische Markov-Kette und unter Hinzunahme der Normalisierungsbedingung $\sum_{i \in Z} p_i = 1$ ergibt sich die eindeutige Lösung für die stationären Zustandswahrscheinlichkeiten mit Gl. (37, 38) aus dem folgenden linearen Gleichungssystem

$$\vec{p} \cdot Q = \vec{0} \tag{39}$$

mit dem Zeilenvektor $\vec{p} = (p_0, p_1, \ldots)$, als dem Lösungsvektor für die Gleichgewichtszustandswahrscheinlichkeiten und Q als der GENERATORMATRIX bzw. ÜBERGANGSRATENMATRIX,

$Q = \begin{pmatrix} q_{11} & q_{12} & q_{13} & \cdots \\ q_{21} & q_{22} & q_{23} & \cdots \\ q_{31} & \cdots & \cdots & \cdots \\ \vdots & \vdots & \vdots & \vdots \end{pmatrix}$, wobei sich die Diagonalelemente dieser Matrix ergeben aus $q_{ii} = 1 - \sum_{j \neq i} q_{ij}$, also mit anderen Wörteren, die Wahrscheinlichkeit, dass keinen Übergang aus Knoten i Stattfindet.

Im Folgenden wollen wir demonstrieren, wie aus einem Warteschlangenmodell die zu grundlegende Markov-Kette und damit die so genannte GLOBALE GLEICHGEWICHTSGLEICHUNGEN Gl. (39) ermittelt wird.
Betrachte das geschlossene Warteschlangennetz aus Abb. (2) in diesem Netz befinden sich zwei Knoten (N=2) und drei Aufträge (K=3). Die Bedienzeiten sind exponentiell verteilt mit den Mittelwerten $1/\mu_1 = 5$sec und $1/\mu_2$sec. Die Warteschlangendisziplin bei beiden Knoten ist FCFS.

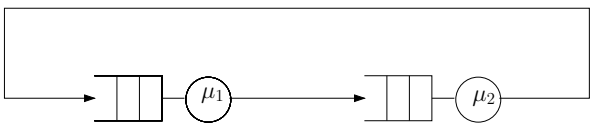


Abbildung 2: Ein geschlossenes Netz

Der Zustandsraum der Markov-Kette, die das Verhalten des Beispielnetzes beschreibt, ergibt sich aus den möglichen Zuständen des Netzes:

$\{(3,0),(2,1),(1,2),(0,3)\}$.
Zustand (k_1, k_2) gibt an, daß sich k_1 Aufträge im Knoten 1 und k_2 im Knoten 2 befinden. $p(k_1, k_2)$ bezeichnet die Wahrscheinlichkeit für diesen Zustand im Gleichgewicht.
Zur Bestimmung der Übergangsraten für den einzelnen Zustande betrachten wir z.B. den Zustand (1,2). Ein Übergang von (1,2) in den Zustand (0,3) findet genau dann statt, wenn ein Auftrag im knoten 1 fertig bedient wird; die dazugehörige Rate ist μ_1. Somit ist μ_1 die Übergangsrate vom Zustand (1,2) in den Zustand (0,3). Entsprechend ist dann μ_2 die Übergangsrate vom Zustand (1,2) nach Zustand (2,1).
Ein nützliches Hilfsmittel zur Beschreibung von Markov-Ketten sind ZUSTANDSÜBERGANGSDIAGRAMME. In Abb. (3) ist das Zustandsübergangsdiagramm für das Beispielnetz gegeben.
Mit Hilfe dieses Diagramms können die globalen Gleichgewichtsgleichungen einfach formuliert werden. Die globalen Gleichgewichtsgleichungen für das Beispiel lauten,

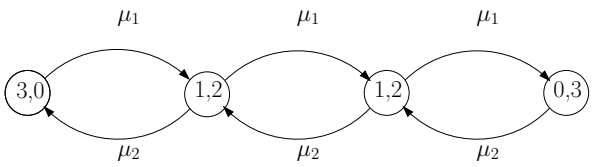


Abbildung 3: Zustandsübergangsdiagramm

$$\begin{aligned} p(3,0)\mu_1 &= p(2,1)\mu_2, \\ p(2,1)(\mu_1+\mu_2) &= p(3,0)\mu_1 + p(1,2)\mu_2, \\ p(1,2)(\mu_1+\mu_2) &= p(2,1)\mu_1 + p(0,3)\mu_2, \\ p(0,3)\mu_2 &= p(1,2)\mu_2. \end{aligned}$$

Mit der Generatormatrix

$$Q = \begin{pmatrix} -\mu_1 & \mu_1 & 0 & 0 \\ \mu_2 & -(\mu_1+\mu_2) & \mu_1 & 0 \\ 0 & \mu_2 & -(\mu_1+\mu_2) & \mu_1 \\ 0 & 0 & \mu_2 & -\mu_2 \end{pmatrix}$$

und dem Lösungsvektor $\vec{p} = (p(3,0), p(2,1), p(1,2), p(0,3))$ ergibt sich die Matrixgleichung $\vec{p} \cdot Q = \vec{0}$.

3 Von UML-Beschreibung zum Warteschlangenmodell in der Forschung

Nachfolgend wird eine Übersicht über die verschiedenen Ansätze präsentiert, wie aus einer Softwareentwurfsbeschreibung (meistens einer auf UML basierenden Beschreibung) das entsprechende Warteschlangenmodell erstellt werden kann. Wir werden auch die verschiedenen Sprachen, Algebren und Werkzeuge, die dafür benutzt wurden, kurz erwähnen.

In [BM05] wird aus den UML-Diagrammen (Use Case Diagram, Activity Diagram und Deployment Diagram mit der UML-Erweiterung Profile for Schedulability, Performance and Time Specification (PST)) die Softwarekomponenten zu vordefinierten Klassen von elementaren Warteschlangen zugeordnet und anschließend aus diesen Knoten das Warteschlagennetz für den Softwareentwurf zusammengestellt.

In [BBS02] wurde zunächst AEmilia eingeführt, eine auf Stochastic Process Algebras (SPAs) basierte Beschreibungssprache für Softwarearchitektur. Dann wurde im Gegensatz zu der üblichen Vorgehensweise eine Entwurfsbeschreibung erstellt, die nicht auf UML basiert, sondern es wurde AEmilia benutzt. Anschliessend wurde aus der AEmilia Beschreibung neben die Erkennung von Entwurfsfehlern das Warteschlangenmodell syntaktisch erstellt.

In [WW04] wird wieder auf eine nicht auf UML basierende Beschreibung des Softwareentwurfs gesetzt. Aus einer XML-basierten Beschreibungsprache namens Component-Based Modeling Language (CBML) wird das Warteschlangenmodell erstellt.

4 Ansätze zur Analyse von Warteschlangennetzen

Hier werden wir verschiedene Ansätze, wie ein Queuing-Network analysiert werden kann und wie Rückschlüsse auf Ebene der Softwarearchitektur gewonnen werden können, betrachten, wobei das Hauptinteresse in dieser Arbeit auf dem ersten Aspekt liegt.

4.1 Numerische Analyseverfahren

Eine Möglichkeit, die Leistungsgrößen eines Warteschlangenmodells zu bestimmen, besteht darin, das System der globalen Gleichgewichtsgleichungen wie in Gl. (39) mit Hilfe numerischer Verfahren zu lösen.
Numerische Analyseverfahren besitzen außer der Forderung nach Endlichkeit des Zustandsraums, die für geschlossene Modelle stets erfüllt ist, keinerlei weitere Einschränkungen.

Iterative numerische Methode Für diese Methode werden die Zustandswahrscheinlichkeiten eines zu analysierenden Warteschlangenmodells auf iterativen Weg, ausgehend von einem beliebig gewählten Anfangsvektor $\vec{p}^{(0)}$, ermittelt [WR66].
Hierzu wandeln wir zunächst die Matrixgleichung $\vec{p} \cdot Q = \vec{0}$ durch Einführung eines Skalars Δ um in $\vec{p} \cdot Q \cdot \Delta = \vec{0}$. Nach Addition von $\vec{p}$ auf beiden Seiten ergibt sich daraus $\vec{p} \cdot Q \cdot \Delta + \vec{p} = \vec{p}$ und mit I der Einheitmatrix ist schließlich $\vec{p} \cdot (Q \cdot \Delta + I) = \vec{p}$.
Das Iterationsschema zur Berechnung von $\vec{p}$ kann damit durch die Gleichung

$$\vec{p}^{(j+1)} = \vec{p}^{(j)} \cdot (Q \cdot \Delta + I) \tag{40}$$

angegeben werden.
Die Iteration wird so lange durchgeführt, bis sich die Ergebnisse beim n-ten Iterationsschritt um weniger als ein vorher festgelegtes ϵ von denen beim (n-1)-ten Iterationsschritt unterscheiden.

Zur Sicherstellung der Konvergenz des Verfahrens wird der Skalar Δ so gewählt, dass das größte Element von $Q \cdot \Delta < 1$ ist. In [Ste78] wird als geeigneter Wert $\Delta = \frac{1}{max|q_{ij}|}$ vorgeschlagen und in [WR66] $\Delta = \frac{0.99}{max|q_{ij}|}$.
Im Folgenden werden wir das geschlossene Netzwerk aus Abbildung 2 betrachten und demonstrieren, wie mit Hilfe der iterativen numerischen Methode die Leistungsgrößen bestimmt werden können.
Für die Generatormatrix Q erhalten wir nach Einsetzen der Werte $\mu_1 = 0.2$ und $\mu_2 = 0.4$:

$$Q = \begin{pmatrix} -0.2 & 0.2 & 0 & 0 \\ 0.4 & -0.6 & 0.2 & 0 \\ 0 & 0.4 & -0.6 & 0.2 \\ 0 & 0 & 0.4 & -0.4 \end{pmatrix}$$

Das betragsmäßig größte Diagonalelement ist $q_{22} = q_{33} = -0.6$, so dass wir wie in [Ste78] für den Skalar Δ wählen:

$$\Delta = \frac{1}{max|q_{ii}|} = \frac{1}{0.6} = 1.6667 \tag{41}$$

Damit ergibt sich die invariante Matrix $(Q \cdots \Delta + I)$ zu:

$$(Q \cdots \Delta + I) = \begin{pmatrix} 0.6667 & 0.3333 & 0 & 0 \\ 0.6667 & 0 & 0.3333 & 0 \\ 0 & 0.6667 & 0 & 0.3333 \\ 0 & 0 & 0.6667 & 0.3333 \end{pmatrix}$$

Der Anfangsvektor $\vec{p}^{(0)} = (p(3,0), p(2,1), p(1,2), p(3,0))^{(0)}$ kann willkürlich festgelegt werden, wobei jedoch die Normalisierungsbedingung $p(3,0)+p(2,1)+p(1,2)+p(0,3) = 1$ berücksichtigt werden muss.
Mit z.B. $\vec{p}^{(0)} = (0.65, 0.35, 0, 0)$ und $\epsilon = 0.003$ erhalten wir den in 1 gezeigten Iterationsverlauf.

Iteration	$p(3,0)$	$p(2,1)$	$p(1,2)$	$p(0,3)$
1	0.6667	0.2166	0.1167	0
2	0.5889	0.3000	0.0722	0.0389
3	0.5926	0.2444	0.1259	0.0371
4	0.5580	0.2815	0.1062	0.0543
5	0.5597	0.2568	0.1300	0.0535
6	0.5443	0.2733	0.1213	0.0612
7	0.5450	0.2623	0.1319	0.0608
8	0.5382	0.2696	0.1280	0.0642
9	0.5385	0.2647	0.1327	0.0641
10	0.5355	0.2680	0.1309	0.0656
11	0.5356	0.2658	0.1330	0.0655

Tabelle 1: Iterationsverlauf der iterativen numerischen Methode

Aus diesen Resultaten lassen sich leicht alle weiteren wichtigen Leistungsgrößen des Netzes berechnen:

a Randwahrscheinlichkeiten Gl. (27):

$$p_1(0) = p_2(3) = p(0,3) = 0.0667,$$
$$p_1(1) = p_2(2) = p(1,2) = 0.1333,$$
$$p_1(2) = p_2(1) = p(2,1) = 0.2667,$$
$$p_1(3) = p_2(0) = p(3,0) = 0.5333.$$

b Auslastungen Gl. (29):

$$\rho_1 = 1 - p_1(0) = 0.9333, \quad \rho_2 = 1 - p_2(0) = 0.4667.$$

c Durchsatz Gl. (31):

$$\lambda_1 = \rho_1\mu_1 = 0.8666 \approx 0.1867 = \lambda = \lambda_2 = \rho_1\mu_2 = 0.18668 \approx 0.1867.$$

d Mittlere Anzahl von Aufträgen Gl. (32):

$$\bar{k}_1 = \sum_{k=1}^{3} kp_1(k) = 2.2667, \quad \bar{k}_2 = \sum_{k=1}^{3} = 0.7333.$$

e Mittlere Antwortzeiten der Aufträge Gl. (34):

$$\bar{t}_1 = \frac{\bar{k_1}}{\lambda_1} = 12.1439, \quad bart_2 = \frac{\bar{k_2}}{\lambda_2} = 3.9286$$

Diese Methode wurde in [VK99] zur Lösung des Warteschlangenmodells verwendet und anschließend mit dem simulationsbasierten Verfahren (GPSS) verglichen.

4.2 Numerische Analyseverfahren für spezielle Warteschlangennetzen

Im Gegensatz zur vorigen Methode, die einen relativ uneingeschränkten Anwendungsbereich besitzt, werden wir im folgenden auf speziele Eigenschaften von Warteschlangennetzen eingehen und für solche Netze werden wir einige effiziente Verfahren, die eine exakte Bestimmung der Leistungsgrößen unter Umgehung der globalen Gleichgewichtsgleichungen ermöglichen, vorstellen.

4.2.1 Produktformlösungen

Der Begriff Produktform bzw. Prokuktformlösung wurde von [Jac57, GN67] eingeführt. Als wichtigstes Ergebnis für die Warteschlangendisziplin wird gezeigt, dass sich die Lösungen für die Gleichgewichtszustandswahrscheinlichkeiten für Netze mit dieser Eigenschaft aus Faktoren zusammensetzen, die die Zustände der einzelnen Knoten beschreiben. Diese Lösungen werden als Prokuktformlösungen bezeichnet.

Eine notwendige und hinreichende Bedingung für die Existenz von Produktformlösungen, ist die
Local-Balance Eigenschaft:
Ein Netzwerk ist im lokalen Gleichgewicht genau dann, wenn die Rate mit der ein Zustand des Netzes aufgrund des Abgangs eines Auftrags aus einem Knoten verlassen wird, gleich derjenigen Rate ist, mit der Zustand aufgrund des Übergangs eines Auftrags in diesen Knoten wieder erreicht werden kann.

Es gibt noch andere charakteristische Eigenschaften, die auf ein Warteschlangennetz mit Produktformlösungen zutreffen:

M→M Eigenschaft:
Eine Bedienstation besitzt genau dann diese Eigenschaft, wenn sie einen Poissonschen Ankunftsprozeß in einen Poissonschen Abgangsprozeß transformiert.

Station-Balance Eigenschaft:
Eine Warteschlangendisziplin erfüllt die Station-Balance-Eigenschaft, wenn die Rate, mit der Aufträge in einer Position der Warteschlange bedient werden, proportional zur Wahrscheinlichkeit ist, dass ein Auftrag diese Position betritt. Mit anderen Worten, man unterteilt die Warteschlange eines Knotens in einzelne Positionen und setzt die Raten, mit denen diese Positionen betreten bzw. verlassen werden, gleich.

4.2.2 Jackson-Methode für offener Warteschlangennetze mit Produktformlösungen

Jackson [Jac57, Jac63] hat mit der nach ihm benannten Methode für eine Klasse von Warteschlagennetzen eine Produktformlösungen errechnen können. Die von Jackson untersuchten Warteschlangennetze erfüllen folgende Bedingungen:

- Im Netz befindet sich nur eine einzige Auftragsklasse.[3]
- Die Gesamtanzahl der Aufträge im Netz ist nicht beschränkt.
- Jeder der N Knoten des Netzes kann Ankünfte von außen bekommen mit exponentiell verteilten Zwischenankunftzeiten. Abgänge von Aufträgen aus dem Netz sind bei jedem Knoten möglich.
- Sämtliche Bedienzeiten der Aufträge in den einzelnen Knoten sind exponentiell verteilt.
- Die Warteschlangendisziplin ist bei allen Knoten FCFS.
- Der i-te Knoten besteht aus $m_i \geq 1$ identischen Bedieneinheiten mit den Bedienraten $\mu_i,\ i = 1, \ldots, N$. Die Bedienraten können, ebenso wie die Ankunftsraten λ_{0i}, von der Anzahl k_i der Aufträge im jeweiligen Knoten abhängen. Man spricht dann auch von lastabhängigen Bedienraten bzw. lastabhängigen Ankunftsraten.

Für Netze in dieser Klasse gilt der von Jackson in [Jac63] bewiesene Satz.

Satz 4.1 (Satz von Jackson) *Wenn für alle Knoten $i = 1, \ldots, N$ im offenen Netz die Stabilitätsbedingung Gl. (17) $\rho_i < 1$ erfüllt ist, wobei sich die Ankuftsraten λ_i aus Gl. (31) ergeben, dann ist die Wahrscheinlichkeit für den Gleichgewichtszustand des Netzes durch das Produkt der Zustandswahrscheinlicheiten (Randwahrcheinlichkeiten) der einzelnen Knoten gegeben,*

$$p(k_1, k_2, \ldots, k_N) = p_1(k_1) \cdot p_2(k_2) \cdot \ldots \cdot p_N(k_N) \tag{42}$$

[3] In dem gesamten Artikel betrachten wir ausschließlich Netze mit nur einer einzigen Auftragsklasse

Die einzelnen Knoten des Netzes können somit als voneinander unabhängige elementare M/M/m-FCFS Wartesysteme mit der Ankunftsrate λ_i uns der Bedienrate μ_i betrachtet werden.
Mit Folgendem geben wir den Algorithmus der Jackson-Methode an:

Schritt 1: Berechne für alle Knoten $i = 1, \ldots, N$ des offenen Netzes die Ankunftsraten λ_i durch lösen des Gleichungssystems, das aus Gl. (22) für alle Knoten des Netzes gewonnen wird.

Schritt 2: Betrachte jeden Knoten i als elementares Wartesystem. Überprüfe die Stabilitätsbedingung Gl. (17) und bestimme die Zustandswahrscheinlichkeiten und die Leistungsgrößen des Knotens mit den im Grundlagen Kapitel angegebenen Formeln.

Schritt 3: Berechne mit Gl. (42) aus den Einzellösungen die Zustandswahrscheinlichkeiten für das gesamte Netz.

4.2.3 Mittelwertanalyse (MVA) für geschlossener Warteschlangennetze mit Produktformlösungen

Die Mittelwertanalyse (MVA) wurde von Reiser und Lavenberg [RL80] zur exakten Analyse geschlossener Warteschlangennetze mit Produktformlösungen entwickelt.
Man beschränkt sich bei dieser Methode darauf, unter ausschließlicher Verwendung von drei Gleichungen, die Mittelverte von Antwortzeit, Durchsatz und Anzahl der Aufträge in den Knoten zu berechnen.
Die folgenden zwei Gesetze bilden die Grundlage der Mittelwetanalyse:

- Das Gesetz von Little, das im Grundlagen Kapitel eingeführt wurde: $\bar{k} = \lambda \cdot \bar{t}$
- Das Theorem über die Verteilung beim Ankunftszeitpunkt, kurz ANKUNFTSTHEOREM. Dieses besagt, dass für geschlossene Produktformnetze hiernach die Wahrscheinlichkeit, dass ein Auftrag bei Ankunft an Knoten i den Netzwerkzustand $(k_1, \ldots, k_i, \ldots, k_N)$ vorfindet, gleich der Gleichgewichtszustandswahrscheinlichkeit $p(k_1, \ldots, k_i - 1, \ldots, k_N)$ des Netzes mit einem Auftrag weniger ist.

Die fundamentale Gleichung der Mittelwertanalyse stellt einen Zusammenhang her zwischen der mittleren Antwortzeit eines Auftrags im i-ten Knoten und der mittleren Anzahl von Aufträgen in diesem Knoten bei einem Auftrag weniger im Netz:

$$\bar{t}_i(K) = \frac{1}{\mu_i \cdot m_i} \left[1 + \bar{k}_i \cdot (K-1) + \sum_{j=0}^{m_i-2} (m_i - j - 1) \cdot p_i(j|k-1) \right]. \tag{43}$$

Hierbei bezeichnet $p_i(j|k)$ die Wahrscheinlichkeit dafür, dass im i-ten Knoten j Aufträge bedient werden unter der Bedingung, es befinden sich k Aufträge im Netz. Es gilt:

$$p_i(j|k) = \frac{e_i \cdot \lambda(k)}{\mu_i \cdot j} p_i(j-1|k-1) \quad \text{für } j = 1, \ldots, m_i - 1 \tag{44}$$

$$p_i(0|k) = 1 - \frac{1}{m_i} \left[\frac{e_i}{\mu_i} \lambda(k) + \sum_{j=1}^{m_i-1} \cdot p_i(j|k) \right] \tag{45}$$

Für ein Wartesystem mit $m_i = \infty$, da in diesem Fall kein Auftrag warten muss, gilt:

$$\bar{t}_i = \frac{1}{\mu_i}. \tag{46}$$

Neben dieser Gleichung brauchen wir zur vollständigen Beschreibung der Mittelwertanalyse noch zwei weitere Gleichungen, die sich aus dem Gesetz von Little ableiten lassen.
Die erste dient zur Bestimmung des Gesamtdurchsatzes des Netzes:

$$\lambda(K) = \frac{K}{\sum_{i=1}^{N} e_i \cdot \bar{t}_i(K)}, \tag{47}$$

die zweite zur Bestimmung der mittleren Anzahl von Aufträgen im i-ten Knoten:

$$\bar{k}_i(K) = \lambda(K) \cdot \bar{t}_i(K) \cdot e_i. \tag{48}$$

Damit kann der Algorithmus der Mittelwertanalyse für geschlossene Produktformnetze wie folgt beschrieben werden:

Schritt 1: Initialisierung:
Für $i = 1, \ldots, N$ und $j = 1, \ldots, (m_i - 1)$:
$\bar{k}_i(0) = 0, \quad p_i(0|0) = 1, \quad p_i(j|0) = 1.$

Schritt 2: Iteration über die Anzahl der Aufträge $k = 1, \ldots, K$.

Schritt 2.1: Berechne für $i = 1, \ldots, N$ die mittlere Antwortzeit eines Auftrags im i-ten Knoten:

$$\bar{t}_i(k) = \begin{cases} \frac{1}{\mu_i}\left[1 + \bar{k}_i(k-1)\right] & \text{falls} \quad m_i = 1 \\ \frac{1}{\mu_i \cdot m_i}\left[1 + \bar{k}_i \cdot (K-1) + \sum_{j=0}^{m_i-2}(m_i - j - 1) \cdot p_i(j|k-1)\right] & \text{falls} \quad m_i > 1 \\ \frac{1}{\mu_i} & \text{falls} \quad m_i = \infty \end{cases}$$

wobei die bedingten Wahrscheinlichkeiten aus Gl. (44, 45) bestimmt werden.

Schritt 2.2: Berechne den Durchsatz:

$$\lambda(k) = \frac{k}{\sum_{i=1}^{N} e_i \cdot \bar{t}_i(k)}. \tag{49}$$

Die Durchsätze einzelner Knoten ergeben sich aus:

$$\lambda_i = \lambda(k) \cdot e_i, \tag{50}$$

wobei die e_i mit Gl. (25) ermittelt werden.

Schritt 2.3: Berechne für $i = 1, \ldots, N$ die mittlere Anzahl von Aufträgen im i-ten Knoten:

$$\bar{k}_i(k) = \lambda(k) \cdot \bar{t}_i(k) \cdot e_i. \tag{51}$$

Die Iteration bricht ab, wenn K, die Gesamtzahl der Aufträge im Netz, erreicht ist.

Diese Methode wurde in [BIMR98] zur Lösung vom Warteschlangenmodell in ein System integriert, das eine UML Beschreibung des Softwareentwurfs in ein Warteschlangemodell in Produktform transformiert und anschließend mit der (MVA) Methode löst. Allerdings war der Schwerpunkt dieser Arbeit die Transformation.

4.3 Approximative Analyse von Produktformnetzen

Wir haben uns bei der im vorigen Kapitel eingeführten Methoden zur exakten Berechnung der Leistungsgrößen für Produktform-Warteschlangennetzen auf Netze mit einer einzigen Auftragklasse beschränkt. Die Methode könnte zwar durch Erweiterung zur exakten Berechnung der Leistungsgrößen von Produktform-Warteschlangennetzen mit verschiedenen Auftragklassen erweitert werden, aber dann würde ihr Bedarf an Rechenzeit und Speicherplatz exponentiell mit der Anzahl der Auftragsklassen im Netz wachsen. Da man zudem häufig nur daran interessiert ist, sehr schnell Ergebnisse zu erhalten, wobei Näherungslösungen durchaus genügen, wurden approximative Analysemethoden entwickelt.

Summationsmethode Das Konzept der Summationsmethode basiert darauf, jeden einzelnen Knoten eines zu untersuchenden Warteschlangennetzes, durch den Zusammenhang zwischen der mittleren Anzahl von Aufträgen im Knoten und dem Durchsatz durch diesen Knoten zu charakterisieren. Dieser Zusammenhang kann funktional wie folgt dargestellt werden:

$$\bar{k}_i = f_i(\lambda_i) \tag{52}$$

Die Funktion f_i hat für Knoten mit lastunabhängigen Bedienraten folgende Eigenschaften:

1. Wegen $0 \leq \rho_i \leq$ ergibt sich der Definitionsbereich zu $0 \leq \lambda_i \leq m_i\mu_i$ und für $m = 0$ gilt $0 < \lambda_i \leq K \cdot \mu_i$ da $\bar{k}_i = \frac{\lambda_i}{\mu_i}$ und $\bar{k}_i \leq K$.
2. $f_i(\lambda_i) \leq f_i(\lambda_i) + \Delta\lambda_i$ für $\Delta\lambda_i$, d.h. f_i ist monoton steigend.
3. $f_i(0) = 0$

Zur Analyse von Produktformnetzen wurden in [BFS87] folgende Formeln vorgeschlagen:

$$\bar{k}_i = \begin{cases} \frac{\rho_i}{1-\frac{K-1}{K}\rho_i} & \text{für } m_i = 1 \\ m_i\rho_i + \frac{\rho_i}{1-\frac{K-m_i-1}{K-m_i}\rho_i} \cdot P_{m_i} & \text{für } m_i > 1 \\ \frac{\lambda_i}{\mu_i} & \text{für } m_i = \infty \end{cases} \tag{53}$$

wobei ρ_i sich aus Gl. (31) berechnen lässt und die Wartewahrscheinlichkeit P_{m_i} wie in [Bol83] wie folgt sich schätzen lässt:

$$P_{m_i} = \begin{cases} \frac{\rho_i^{m_i}+\rho_i}{2} & \text{für } \rho_i > 0.7 \\ \rho_i^{\frac{m_i+1}{2}} & \text{für } \rho_i < 0.7 \end{cases} \tag{54}$$

Wir können jetzt durch Summation über alle Knoten des Netzes aus den einzelnen Funktionsgleichungen die Systemgleichung für das gesamte Netz gewinnen:

$$\sum_{i=1}^{N} \bar{k}_i = \sum_{i=1}^{N} f_i(\lambda_i) = K. \tag{55}$$

Hieraus läßt sich mit der Beziehung $\lambda_i = \lambda \cdot e_i$ eine Gleichung zur Bestimmung des Gesamtdurchsatzes λ des Netzes angeben:

$$\sum_{i=1}^{N} f_i(\lambda \cdot e_i) = g(\lambda) = K \tag{56}$$

Mit Hilfe dieser Gleichung und wegen der Monotonie von $f_i(\lambda_i)$ kann ein Algorithmus wie folgt beschrieben werden:

Schritt 1: Initialisierung:
Wähle $\lambda_u = 0$ als untere Grenze für den Durchsatz und $\lambda_o = \min_i\{\frac{\mu_i m_i}{e_i}\}$ als obere Durchsatzgrenze. Für Knoten mit $m_i = \infty$ muss m_i durch K ersetzt werden.

Schritt 2: Intervallschachtelung zur Bestimmung von λ:

Schritt 2.1: Setze $\lambda = \frac{\lambda_u + \lambda_o}{2}$.

Schritt 2.2: Bestimme $g(\lambda) = \sum_{i=1}^{N} f_i(\lambda \cdot e_i)$, wobei die einzelnen Funktionen $f_i(\lambda \cdot e_i)$ aus Gl. (53) unter Berücksichtigung von Gl. (52) ermittelt werden.

Schritt 2.3: Ist $K - \epsilon \leq g(\lambda) \leq K + \epsilon$, dann endet das Verfahren. ϵ gibt einen geeigneten Toleranzbereich an.
Ist $g(\lambda) > K + \epsilon$, dann setze $\lambda_o = \lambda$ und gehe zurück zu Schritt 2.1
Ist $g(\lambda) < K - \epsilon$, dann setze $\lambda_u = \lambda$ und gehe zurück zu Schritt 2.1

Abbildungsverzeichnis

Literatur

[All90] Arnold O. Allen. *Probability, statistics, and queueing theory.* Acad. Press, 2. ed. edition, 1990.

[BBS02] Simonetta Balsamo, Marco Bernardo, and Marta Simeoni. Combining stochastic process algebras and queueing networks for software architecture analysis. In *Workshop on Software and Performance*, pages 190–202, 2002.

[BFS87] G. Bolch, G. Fleischmann, and R. Schreppei. 1987.

[BIMR98] S. Balsamo, P. Inverardi, C. Mangano, and F. Russo. Performance evaluation of a software architecture: A case study, 1998.

[BM05] Simonetta Balsamo and Moreno Marzolla. Performance evaluation of UML software architectures with multiclass queueing network models. In *WOSP*, pages 37–42. ACM, 2005.

[Bol83] G. Bolch. Approximation von Leistungsgrössen Symmetrischer Mehrprozessorsysteme. (German) [Performance measure approximation for symmetric multiprocessor systems]. *Computing*, 31(4):305–315, 1983.

[Bol89] Gunter Bolch. *Leistungsbewertung von Rechensystemen mittels analytischer Warteschlangenmodelle.* Studienzentrum für Sehgeschädigte, 1989.

[Fel68] W. Feller. Wiley, 3. ed., rev. print. edition, 1968.

[GN67] W. J. Gordon and G. F. Newell. Closed queueing systems with exponential servers. *Oper. Res.*, 15:254–265, 1967.

[Jac57] J. R. Jackson. Networks of waiting lines. *Operations Research*, 5, August 1957.

[Jac63] J. R. Jackson. Jobshop-like queueing systems. *Management Science*, 10(1):131–142, 1963.

[RL80] Reiser and Lavenberg. Mean value analysis of closed multichain queuing networks. *JACM: Journal of the ACM*, 27, 1980.

[Ste78] W. J. Stewart. A comparison of numerical techniques in markov modeling. *Communications of the ACM*, 21(2):144–152, 1978.

[VK99] V. Vlassov and A. Kraynikov. A queuing model of a multi-threaded architecture: A case study. *Lecture Notes in Computer Science*, 1662:306–??, 1999.

[WR66] V.L. Wallace and R.S. Rosenberg. Rqa-1, the recursive queue analyzer. 1966. Technical Report No. 2, Systems Engineering Laboratory, University of Michigan, Ann Arbor.

[WW04] Xiuping Wu and C. Murray Woodside. Performance modeling from software components. In Jozo J. Dujmovic, Virgílio A. F. Almeida, and Doug Lea, editors, *WOSP*, pages 290–301. ACM, 2004.

UML-Profile für Qualitätsanforderungen: SPT, QoS/FT, MARTE

Roman Sinawski

Betreuer: Heiko Koziolek

Zusammenfassung

Die Unified Modeling Language (UML) und darauf basierenden Verfahren können helfen, die wachsende Komplexität von Software zu beherrschen. Durch die Wahl einer passenden Modellierungsmethode können Problemstellungen sowohl erkannt als auch aus verschiedenen Blickwinkeln betrachtet werden. Der Entwickler kann dabei auf unterschiedliche Darstellungs- und Ausdrucksformen zurückgreifen. Darüber hinaus besteht die Möglichkeit, eigene Methoden zu entwickeln und die UML an spezielle Szenarien anzupassen. Eine Möglichkeit hierzu stellen die sogenannten UML-Profile dar. Diese Arbeit beschäftigt sich mit dreien solcher Profile, welche speziell die Modellierung von Qualitätsaspekten in UML ermöglichen. Neben einer kurzen Vorstellung der UML und ihrer Erweiterungskonzepte sowie einer Beleuchtung des Begriffes der Qualität, werden die Profile SPT, QoS/FT und MARTE vorgestellt.

1 Einleitung

Modellierungssprachen ermöglichen die Betrachtung eines Softwareproduktes bzw. eines Systems von einer höheren Ebene heraus und führen einen Abstraktionsgrad ein, der die Übersicht erhöht. Anforderungen werden nicht lediglich in natürlicher Sprache festgehalten, sondern in einem anwendungsorientierten Modell, das konzeptionelle und formale Aspekte ausdrückt.

Einleitend wird die UML als eine der bekanntesten Modellierungssprachen vorgestellt. Zudem soll der Begriff „Qualität“ näher beleuchtet und eine Sensibilisierung für damit einhergehende Fragen und Anforderungen geschaffen werden. In Kapitel 2 wird auf Konzepte zur individuellen, szenarioorientierten Erweiterung der UML näher eingegangen, wobei speziell „UML-Profile“ betrachtet werden. Die Kapitel 3 bis 5 stellen die UML-Profile SPT, QoS/FT sowie MARTE vor.

Soweit nicht anders angegeben, beziehen sich alle Angaben auf die aktuelle Version 2.0 der UML [13].

2 Gundlagen

2.1 Geschichte und Motivation der UML

Die Zeit der „Methodenkriege“ in der objektorientierten Softwareentwicklung ist vielen Entwicklern negativ in Erinnerung geblieben. Die bereits seit Mitte der 70er Jahre aufkommenden objektorientierten Programmiersprachen brachten eine Vielzahl unterschiedlichster Ansätze für Analyse und Entwurf hervor. 1994 waren mehr als 50 solcher Methoden bekannt, die jedoch eher Spezialwerkzeuge als allgemeine Hilfsmittel waren. Es bestand durchweg das Problem, eine angemessene und einheitliche Modellierungssprache zu finden, die allen Anforderungen gerecht wurde.

Diese Schwierigkeiten führten zu einer neuen Generation von Modellierungsmethoden, zu deren Zielen vor allem auch die Vollständigkeit gehörte. Bekannte Vertreter sind vor allem die Methoden von Grady Booch, James Rumbaugh (OMT)[12] und Ivar Jacobson (OOSE)[9]. All diese Methoden ermöglichten eine mehr oder weniger vollständige Betrachtung des Softwareproduktes.

Die beginnende Vermischung der Ideen dieser Methoden gab Anlaß, über eine Fusionierung nachzudenken. Es war offensichtlich, daß der objektorientierte Markt in mehrfacher Hinsicht von einer gemeinsamen Linie profitieren konnte. Entwicklern würde ein einheitliches Werkzeug gegeben, das frei von überflüssigen und verwirrenden Unterschieden der einzelnen Teilmethoden war. In gleicher Weise konnten sich Hersteller von Modellierungssoftware aufgrund klar gesteckter Randbedingungen auf die Schaffung sinnvoller und allgemein einsetzbarer Funktionen konzentrieren. Zusammen mit dem hinzugekommenen Ivar Jacobson nannten Booch und Rumbaugh (auch bekannt als die „3 Amigos“, [3]) die folgenden Ziele für die Vereinheitlichung ihrer Methoden:

1. Die Möglichkeit, Systeme von der ersten Idee bis zum ausführbaren Ergebnis mit objektorientierten Methoden modellieren zu können.

2. Klärung von Skalierungsfragen für große Systeme.

3. Eine sowohl von Menschen als auch von Rechnern nutzbare Sprache.

Das Ergebnis ist heute eine Sprache zur Spezifikation, Konstruktion, Visualisierung und Dokumentation von Modellen für Softwaresysteme. Ihre Modellierungsprimitiven bestehen aus den Bausteinen

- Dinge (Abbildungen von Gegenständen aus der realen Welt, repräsentiert durch *Klassen*, *Schnittstellen* etc.)
- Beziehungen (verknüpfen interagierende Dinge in Form von *Abhängigkeiten*, *Assoziationen*, *Generalisierungen* etc.)
- Diagramme (Sammlungen von zusammengehörenden Dingen und Beziehungen, bspw. *Klassendiagramme*, *Objektdiagramme* etc.)

Der Einsatzbereich ist sehr breit und beinhaltet u.a. zeitkritische, konkurrierende und eingebettete Systeme, um nur wenige zu nennen. Die UML ist eine Sprache, die die Modellierung und Notation unterstützt. Sie ist jedoch bewußt keine Methode, sondern bietet die Grundlage für eine solche.

2.2 Der Qualitätsbegriff

„Qualität“ ist ein weit gefaßter Begriff und adressiert je nach Betrachtungsstandpunkt unterschiedliche Anforderungen mit einer spezifischen Detailtiefe. Mögliche Eigenschaften, die als Qualität bezeichnet werden könnten, sind

- Benutzbarkeit
- Merkmale und Charakteristiken eines Produktes oder einer Dienstleistung, die die Erfüllung getroffener Vereinbarungen sicherstellen
- Übereinstimmung mit zuvor festgelegten Anforderungen oder Bedingungen, die für eine korrekte Funktion nötig sind

Dabei beschreibt der letzte Punkt am besten, wie der in dieser Arbeit verwendete Qualitätsbegriff zu verstehen ist. Qualitätsanforderungen im Bereich der Modellierung und Spezifikation einer Softwarearchitektur sind in erster Linie sogenannte nicht-funktionale Eigenschaften wie Performance, Zuverlässigkeit, Verfügbarkeit usw. [7]. Qualitätsanforderungen werden zum einen aus Gründen der Funktionserfüllung und Vorhersagbarkeit von Software erhoben. Zum anderen stellen sie aber auch den Komfort eines Programmes sicher. Je nach Anwendung und Anwender besitzt die eine oder andere Seite mehr Gewicht. Hochkritische Echtzeitsysteme, wie sie bspw. in Flugzeugen oder PKWs vorhanden sind, stellen höchste Ansprüche an die Vorhersagbarkeit von Funktionsaufrufen und Prozeduren. Dagegen bemerkt ein PC-Nutzer eine unterschiedliche Bootdauer seines Rechners selten, wohl aber eine Verzögerung beim Klick auf eine Schaltfläche.

Um Qualitätsanforderungen ausdrücken zu können, bedarf es einer geeigneten Sprache. Sie muß alle notwendigen Mittel zur Präzisierung und eindeutigen Bestimmung der erwarteten sowie gebotenen Eigenschaften bereitstellen. Die in dieser Arbeit beschriebenen Profile nutzen die Modellierungssprache UML und erweitern diese in der Art, daß diese Anforderungen erfüllt werden können.

3 Erweiterungskonzepte der UML

Bereits relativ früh war klar, daß die UML trotz ihres Umfangs nicht alle Problembereiche würde abdecken können. Allein die große Zahl vorhandener Anforderungen auf allen denkbaren Teilgebieten würde die UML an den Rand der Unüberschaubarkeit bringen. Künftige Ansprüche durch aufkommende Technologien müssen erst gar nicht erwähnt werden. Ziel war und ist es, die Sprache fortlaufend zu entwickeln, jedoch nicht, alle Notationen und Semantiken neu aufkommender Systeme in ihren Wortschatz aufzunehmen. Aus diesem Grund schaffte man die Möglichkeit, die UML kontrolliert erweitern und an individuelle Szenarien anpassen zu können.

Im Folgenden werden die grundlegenden Erweiterungsmechanismen sowie darauf aufsetzende Verfahren vorgestellt.

3.1 Elementare Erweiterungen

Die UML kennt drei elementare Möglichkeiten der Erweiterung:

- *Stereotypen* (engl. stereotypes)
- *Eigenschaftswerte* (engl. tagged values)
- *Einschränkungen* (eng. constraints)

Jede dieser drei ermöglicht es, die Sprache kontrolliert zu erweitern. Die Betonung liegt dabei auf kontrolliert, denn der eigentliche Zweck, nämlich der der Kommunikation von Informationen, steht nach wie vor im Vordergrund.
Stereotypen versehen existierende Sprachkonzepte mit einer individuellen Semantik (Abb.1). Es entstehen somit neue Elemente, die von bereits vorhandenen abgeleitet sind. Ihre Bedeutung ist jedoch für das jeweilige Szenario spezifisch, für das sie geschaffen wurden.

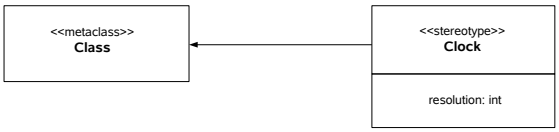


Abbildung 1: Der Stereotyp „Clock“ erweitert das Element „Class“

Eigenschaftswerte ermöglichen es, Modellelementen gewisse Eigenschaften zuzuordnen. Ähnlich wie Attribute etwas über eine Klasse aussagen, kann mit Eigenschaftswerten die Semantik eines Elementes detailliert werden. Heutige Werkzeuge sind in der Lage, UML-Modelle automatisch in Code-Fragmente und Vorlagen umzuwandeln. Mit Eigenschaftswerten kann dieser Vorgang der Codegenerierung direkt beeinflußt werden. Notiert werden Eigenschaftswerte normalerweise durch ein in geschweiften Klammern stehendes {Schlüsselwort} (siehe Abb.2). Es sind jedoch mehrere Varianten verbreitet, die sich von Werkzeug zu Werkzeug unterscheiden können.

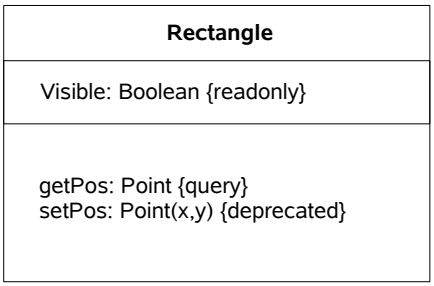


Abbildung 2: Eigenschaftswerte werden in geschweiften Klammern notiert

Einschränkungen stellen eine gewisse Integrität sicher. Sie sind Ausdrücke, die die Inhalte, Zustände oder die Semantik von Modellelementen auf einen zulässigen Bereich festlegen und stets erfüllt sein müssen. Boolesch auswertbare Einschränkungen werden auch *assertions* genannt. Die Notation kann frei formuliert oder aber in Form einer Beschreibung mittels OCL (*Object Constraint Language*) erfolgen. Letztere Möglichkeit macht es einem Werkzeug erst möglich, die Einschränkungen bei der Codegenerierung zu berücksichtigen.

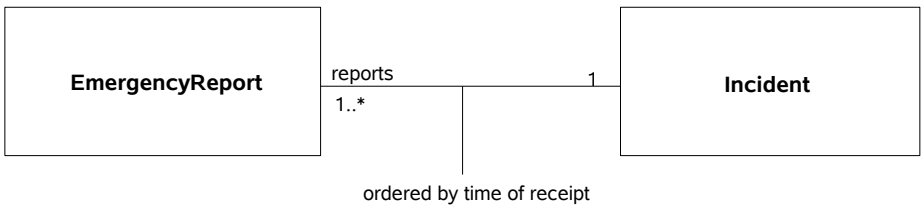


Abbildung 3: Die Notation von Einschränkungen

3.2 UML-Profile

Die im letzten Abschnitt aufgeführten Erweiterungsmechanismen *Stereotypen*, *Eigenschaftswerte* und *Einschränkungen* lassen sich in einer beliebigen Kombination zu einem sogenannten UML-Profil zusammenfassen. Somit ist es möglich, den Sprachumfang der UML als ein Ganzes zu erweitern bzw. an spezifische Einsatzbedingungen oder Domänen anzupassen. Ein UML-Profil ist somit ein Paket, das die elementaren Erweiterungsmechanismen bündelt mit dem Ziel, die Syntax und die Semantik der Modellierungssprache zu definieren bzw. zu manipulieren. Es läßt sich damit das Metamodell der UML spezialisieren und eine anwendungsspezifische Interpretierbarkeit schaffen. Beispielsweise nutzt die Model Driven Architecture (MDA) [14] Profile und spezielle Transformationsregeln zur Umwandlung von Modellen auf gegebene Plattformen (Platform Independent Model - Platform Specific Model). Ebenso lassen sich aber auch unterschiedliche Modellierungstools durch ein gemeinsames UML-Profil einheitlich verwenden – parallel zu den grundlegenden Elementen, welche in der Standard-

UML definiert sind.

Die UML2-Dokumentation führt u.a. folgende möglichen Gründe für Anpassungen des Metamodells auf:

- Berücksichtigung der speziellen Terminologie einer Plattform oder Domäne (bspw. EJB oder JavaBeans)
- Hinzufügung von Semantik, wo diese nicht spezifiziert ist (z.B. der Umgang mit Prioritäten bei empfangenen Nachrichten)
- Spezifizieren des Vorgehens bei Modelltransformationen (siehe MDA-Beispiel oben)
- Bereitstellen einer erweiterten Notation für bereits vorhandene Elemente (z.B. möchte man statt gewöhnlichen Knoten spezielle Symbole einführen, die Computer in einem Netzwerk darstellen)

Im Folgenden soll an einem einfachen Beispiel (siehe hierzu auch [6]) der Aufbau eines UML-Profils veranschaulicht werden. Angenommen, wir möchten unseren UML-Modellen zwei weitere Elemente hinzufügen: Gewichte und Farben. Desweiteren möchten wir diesen Elementen gewisse Eigenschaften und Restriktionen zuordnen. Beispielsweise sollen die Elemente ihre aktuelle Farbe und ihr Gewicht beinhalten, und gefärbte Assoziationen sollen nur zwischen Klassen gleicher Farbe möglich sein. Der Einfachheit halber sollen *Gewicht* und *Farbe* nur auf Klassen und Assoziationen angewendet werden können.

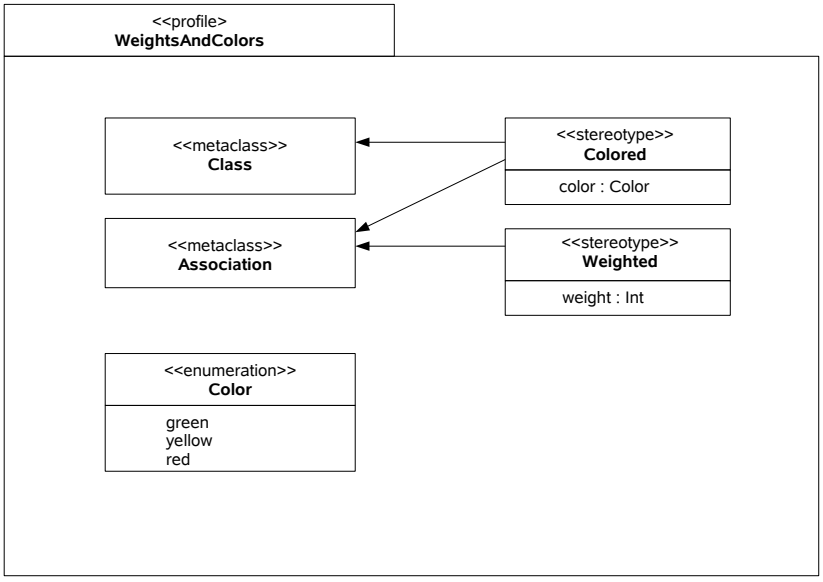


Abbildung 4: Das UML-Profil „WeightsAndColors“

Wie in Abb. 4 zu sehen ist, ist unser Profil in einem Paket namens *WeightsAndColors* untergebracht. Zur Kennzeichnung, daß es sich um ein Profil handelt, wird über

dem Paketnamen der Stereotyp <<*profile*>> angegeben. Metamodell-Elemente werden durch den Stereotyp <<*metaclass*>> gekennzeichnet. Dies sind in unserem Fall Klassen und Assoziationen, beides UML-Standardelemente. Sie werden in das Paket aufgenommen, um die Abhängigkeiten und Wirkungsweisen unserer neuen Elemente festzulegen. Unsere neuen Elemente *Colored* und *Weighted* werden als Stereotypen gekennzeichnet, indem man sie mit der Bezeichnung <<*stereotype*>> markiert. Erweiterungen werden durch Pfeile mit einer gefüllten Spitze gekennzeichnet. Das Element *Colored* soll sowohl auf Klassen als auch auf Assoziationen anwendbar sein, *Weighted* dagegen nur auf Assoziationen. Um *Weighted* und *Colored* gewisse Eigenshaften zuzuordnen, kommen die *TaggedValues* ins Spiel. Wie bereits beschrieben, können damit Elementen bestimmte Eigenschaften zugeordnet werden. Unser neues Element *Colored* soll die Eigenschaft Farbe beinhalten, *Weighted* bekommt ein Gewicht. Dies erreichen wir durch die Angabe der Kombination Wert : Typ. Da wir einen neuen Typ *color* einführen, definieren wir diesen separat und kennzeichnen ihn mit dem Stereotypen <<*enumeration*>>.

Die geforderten Einschränkungen können in jeder Sprache – die menschliche eingeschlossen – definiert werden. Es wäre z.B. möglich, explizit hinzuschreiben: „Assoziationen, die mit einer Farbe versehen werden sollen, dürfen nur dann eine Farbe bekommen, falls sie zwei Klassen mit der selben Farbe verbinden“. Um jedoch eine Automatisierung zu ermöglichen, bespielsweise zur Codegenerierung, ist es sinnvoller, dies in einer Form wie OCL zu tun:

```
context UML:InfrastructureLibrary::Core::Constructs::Association
inv: self.isStereotyped(‘‘Colored‘‘) implies
self.connection forAll(isStereotyped(‘‘Colored‘‘) implies
color=self.color
```

Dieses einfache Bespiel zeigt, daß die Erstellung von UML-Profilen relativ einfach ist. Es wird jedoch empfohlen, sorgfältig abzuwägen, ob ein neues Profil tatsächlich sinnvoll ist und ob nicht bereits UML-Elemente definiert sind, die dieselbe Aufgabe erfüllen. In jedem Fall sollte nicht vergessen werden, daß Profile ein Mittel zur kontrollierten Erweiterung darstellen, ohne dabei die ursprüngliche Semantik vorhandener UML-Elemente zu ändern. Aus diesem Grunde werden UML-Profile auch als *leichtgewichtiger* (lightweight) Erweiterungsmechanismus bezeichnet.

3.3 Meta-Modell

Sind die Mittel, die ein UML-Profil zur Verfügung stellt, nicht ausreichend oder soll ein grundlegend anderes Prinzip einer Modellierungssprache realisiert werden, kann man einen sehr intuitiven Weg gehen: man kreiert einfach eine neue Sprache. Diese Möglichkeit wurde bei der Konzipierung von UML bedacht und bewußt ermöglicht. Es wurde mit dem Meta Object Facility (MOF) eine Sprache speziell für die Erstellung objektorientierter Modellierungssprachen, oder anders gesagt von Metamodellen, geschaffen. Die UML selbst wurde der Semantik der MOF folgend konzipiert. Ein weiteres prominentes Beispiel ist das *Common Warehouse Metamodel* (CWM), das zugleich die UML als Notationssprache nutzt. Es existiert somit ein Quasi-Standard zur Schaffung neuer Modellierungssprachen. Dieser Weg wird zurecht auch als *schwergewichtig* (heavyweight) bezeichnet, soll aber in dieser Arbeit nicht näher beleuchtet werden.

4 UML-Profil : SPT

Das SPT-Profil (Schedulability, Performance and Time), auch genannt RTP (RealTime UML-Profile), ist die erste offizielle Initiative, die UML um Aspekte zu erweitern, die insbesondere der Entwicklung von Echtzeitsystemen dienen. Es wurde das Ziel verfolgt, einen einheitlichen und standardisierten Weg zur Modellierung von sowohl Zeit- als auch Nebenläufigkeitsanforderungen zu schaffen.

Im Mittelpunkt stehen die Analyse von Nebenläufigkeit und Leistung eines Softwaresystems. Dabei wird kein spezieller Weg favorisiert, sondern ein gemeinsames Basissystem zur Verfügung gestellt, das alle Methoden zuläßt. Wichtig ist, daß die Flexibilität zur individuellen Erweiterung erhalten bleibt. Der Gedanke dahinter ist, Modelle durch beschreibende und spezialisierende Anmerkungen in der Art zu erweitern, daß diese durch Werkzeuge automatisch verarbeitet werden können. Dies können u.a. Ablaufplaner oder Performance-Analysetools sein. Es lassen sich somit analytische Verfahren anwenden, die die Modelle auf spezielle Kriterien hin untersuchen. Beispiele hiefür sind *rate monotonic analysis* (RMA) oder auch *deadline monotonic analysis* (DMA). Auch der umgekehrte Weg, nämlich daß die Werkzeuge konfigurierte Modelle erstellen, wird durch die SPT-Notation explizit ermöglicht. Die Analyse der Modelle wird jedoch keinesfalls alleine dem Computer überlassen. Eine manuelle Interpretation durch den Entwickler selbst ist selbstverständlich möglich und auch einfach.

Der Aufbau des SPT-Profils erlaubt dem Benutzer, je nach Bedarf, alles oder nur Teile daraus zu verwenden. Dazu ist das Profil in die drei Hauptpakete *GeneralResourceModeling Framework* (GRM), *AnalysisModels* und *InfrastructureModels* unterteilt (siehe Abb.5), wobei jedes davon wiederum Subprofile bzw. weitere Pakete enthält. Zur besseren Zuordnung werden alle Elemente mit einem Präfix notiert, welches den Ursprung der Definition angibt. Beispielsweise besitzen Elemente aus dem Paket *GRM Framework* das Präfix GRM. Die folgenden Abschnitte stellen die einzelnen Pakete näher vor und demonstrieren die Verwendung des SPT-Profils an einem einfachen Beispiel.

4.1 General Resource Modeling Framework

Das GRM definiert Basiselemente von Echtzeitsystemen, die von allen anderen Teilen des Profils benutzt werden. Wie Abb. 5 zeigt, teilen sich die Basiselemente auf drei Subprofile auf. *RTResourceModeling* definiert den Begriff der *Ressource* und die dazugehörigen quantitativen Merkmale (QoS, Quality of Service). Eine Ressource nimmt eine zentrale Rolle in Echtzeitsystemen ein. Sie besitzt eine endliche, klar abgegrenzte Menge von Eigenschaften wie z.B. Durchsatz, Verfügbarkeit oder Zugriffsart. Alle weiteren Elemente des Profils nehmen in einer spezifischen Art Bezug darauf. *RTConcurrency* erweitert das bereits im UML-Metamodell vorhandene Konzept der Nebenläufigkeit. Im Mittelpunkt steht hier die sogenannte *ConcurrentUnit*, eine Ressource mit der Fähigkeit, Nachrichten bzw. Signale nebenläufig auszusenden. Sie kann eine unbegrenzte Anzahl von Teilressourcen (sog. ResourceServiceInstances) besitzen und somit das Konzept von Prozessen bzw. Tasks realisieren. *RTTimeModeling* definiert das Konzept der Zeit im SPT-Profil. Es umfaßt die Zeit an sich, die Dauer sowie Zeitquellen. Letzteres können beispielsweise (a-)periodische Zeitgeber oder einfache Uhren mit einem Zeitsignal sein.

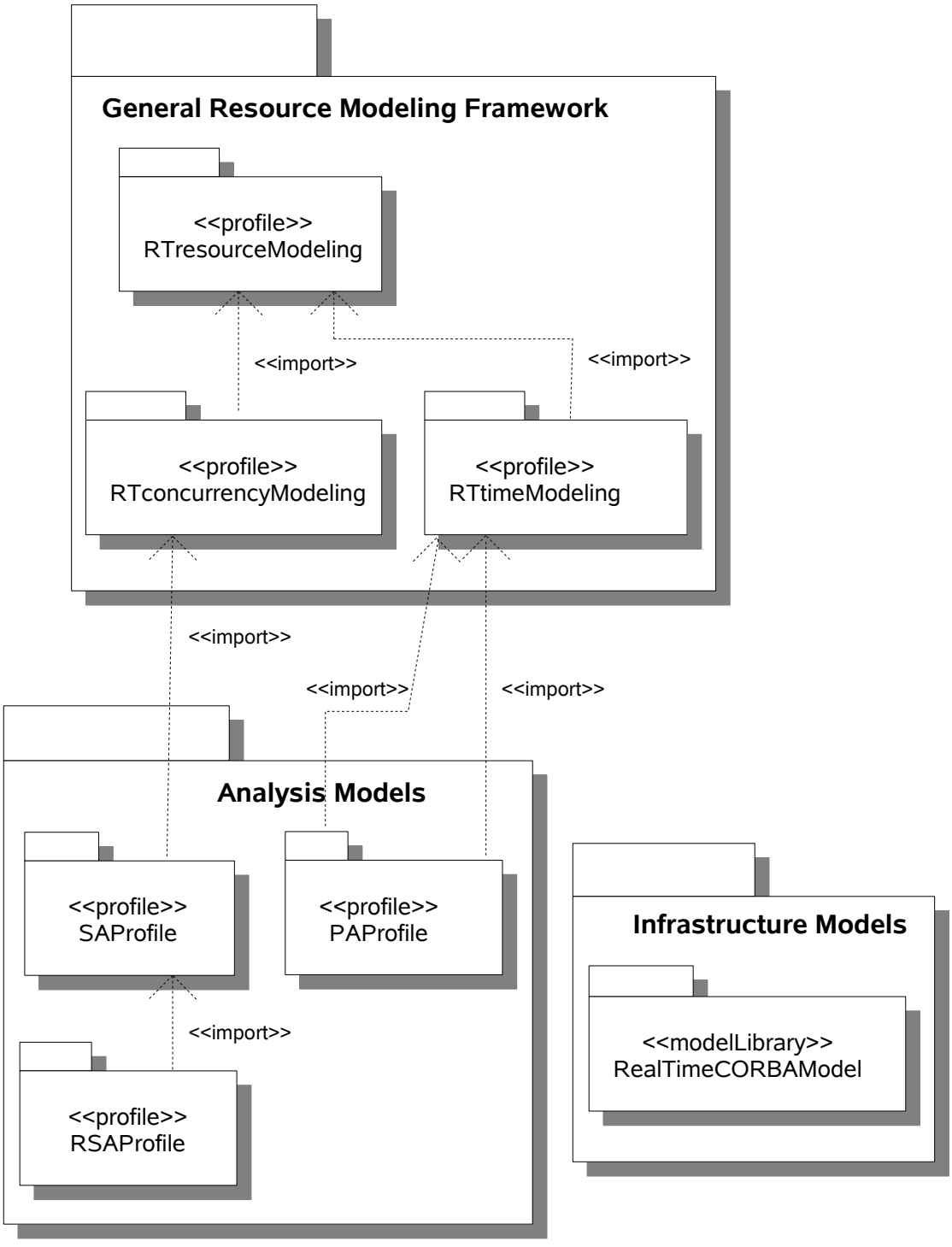


Abbildung 5: Der Aufbau des SPT-Profils

4.2 AnalysisModels

Im *AnalysisModels* - Paket werden unterschiedliche analytische Methoden definiert, jeweils aufgeteilt auf die Unterpakete *SAProfile*, *PAProfile* und *RSAProfile*. Wie anfangs erwähnt, können diese Unterprofile unabhängig von einander den jeweiligen Anforderungen entsprechend verwendet werden.

SAProfile (Schedulability Analysis Profile) und *PAProfile* (Performance Analysis Profile) stellen Erweiterungen für die Ablaufplanung bzw. Leistungsanalyse zur Verfügung. Beide benutzen Elemente aus anderen Teilen des SPT-Profils und erweitern sie zu speziellen Klassen mit spezifischen Eigenschaften. Exemplarisch sei *SAction* (schedulable action) betrachtet. Diese Klasse ist abgeleitet von *TimedAction* aus dem *TimedEvents* Paket im Subprofil *RTTimeModeling*. Sie stellt eine Aktion im Ablaufplan dar, die wiederum selbst Aktionen beinhalten kann. Neben den von *TimedAction* geerbten besitzt sie viele weitere Eigenschaften, die für die Ablaufplanung wichtig sind. Während im *SAProfile* die Konzepte aus dem *GRM* zu Stereotypen und Eigenschaften für Ablaufanalyseverfahren kombiniert werden, geschieht dies im *PAProfile* in gleicher Weise für die Leistungsanalyse. *PAProfile* ermöglicht die Angabe von quantitativen Eigenschaften, die das ganze zu modellierende System umfassen. Beide Subprofile unterscheiden sich unter anderem darin, daß sie unterschiedliche Ziele verfolgen. *SAProfile* dient der Darstellung von zeitlichen und logischen Anforderungen für die Ablaufplanung, dagegen stellt *PAProfile* Mittel zur Quantifizierung der Leistungsfähigkeit eines Systems im Sinne von Auslastung, Verbrauch oder auch Ausführungszeit zur Verfügung.

Das Subprofil *RSAProfile* bezieht sich auf *SAProfile*, genauer gesagt, es ist eine Spezialisierung dessen. Das Präfix *RSA* steht für *RealTimeCORBA Schedulability Analysis*. Im nächsten Abschnitt wird das dritte Hauptpaket *InfrastructureModels* vorgestellt. Darin enthalten ist eine Spezifikation des *RealTimeCORBA*-Models, worauf *RSAProfile* Bezug nimmt. Es besitzt auf dieses Model spezialisierte Eigenschaften, die der Ablaufplanungsanalyse von *RealTimeCORBA* dienen.

4.3 InfrastructureModels

Das Paket *InfrastructureModels* ist für die Aufnahme von (künftigen) Modellen verschiedener Infrastrukturen vorgesehen. Bereits beim *Request for Proposals* (RFP) für das SPT-Profil wurde verlangt, ein Modell der *CORBA*-Infrastuktur einzuplanen. So ist in der offiziellen SPT-Spezifikation das *RealTimeCORBAModel* bereits im Paket *InfrastructureModels* enthalten. Es war geplant, daß weitere Modelle verschiedener Hersteller folgen.

4.4 Anwendungsbeispiel

Das folgende Beispiel demonstriert die Anwendung des SPT-Profiles. Wir beschränken uns dabei auf das Subprofil *SAProfile*.

Abb. 6 zeigt den Ausschnitt eines Systems, welches fünf aktive Komponenten besitzt (gekennzeichnet durch einen doppelten Rand). Aktiv bedeutet, daß die Komponente asynchron und unabhängig von anderen Aktivitäten Signale (auch genannt „Stimuli“) aussenden und empfangen kann. Dementsprechend werden sie als Tasks bzw. Prozesse betrachtet. Die Erweiterungen aus dem *SAProfile* ermöglichen unterschiedliche Analysemöglichkeiten des Ablaufverhaltens. Soll beispielsweise geprüft werden, ob eine gülti-

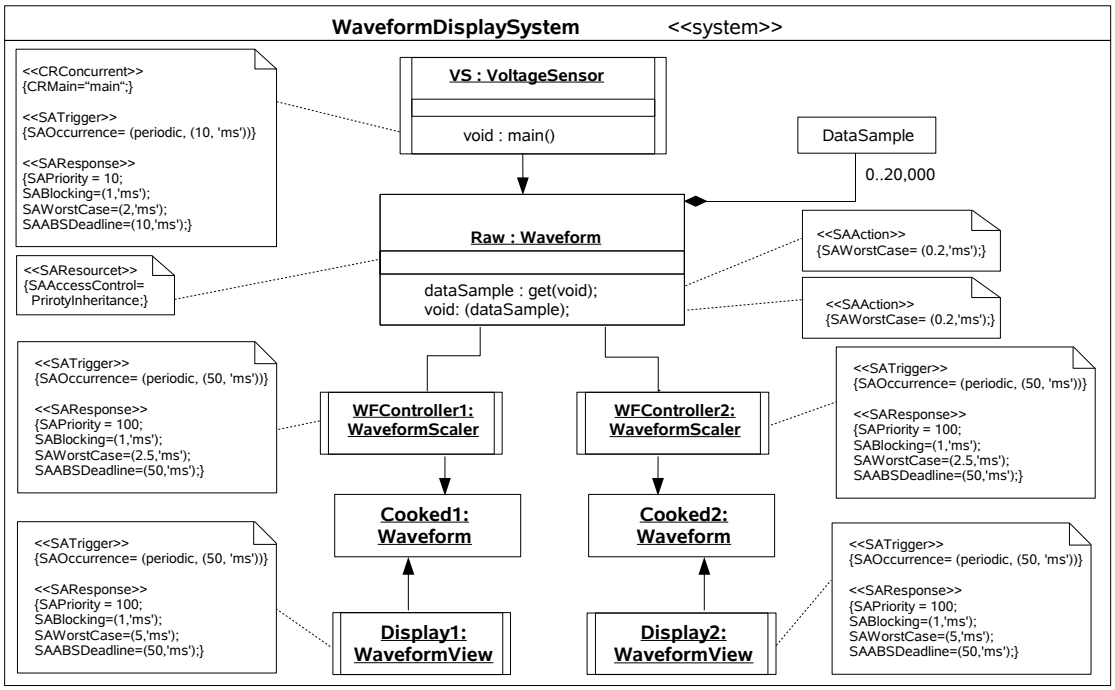


Abbildung 6: Beispielanwendung der Subprofils SAProfile

ge Einplanung dieser Tasks mittels *RMA*-Algorithmus möglich ist, betrachtet man die hierfür relevanten Eigenschaften Bearbeitungsdauer, Periode und die Anzahl einzuplanender Tasks. *SAtrigger* ist ein Stereotyp, der auf Nachrichten und Stimuli angewendet wird und (a-)periodisch wiederkehrende Aktionen kennzeichnet. Die Komponenten *VoltageSensor*, *WaveformScaler* (zweimal) und *WaveformView* (ebenfalls zweimal) besitzen also die Eigenschaft, periodisch aktiv zu werden mit den Perioden 10 ms, 50 ms und 20 ms. Die Bearbeitungsdauer bzw. die Zeit, die ein Task für seine Aktion in Anspruch nimmt, läßt sich mit der Eigenschaft *SAworstCase* ausdrücken. Dies entspricht der größten anzunehmenden Ausführungsdauer. Mit diesen Informationen und der bekannten Anzahl von Tasks läßt sich nun mittels der vereinfachten RMA-Formel (Prioritätenvergabe abhängig von der Häufigkeit des Auftretens) die Frage klären, ob eine Einplanung möglich ist.

$$\sum \frac{C_i}{T_i} \leq Utilization(n) = n(2^{\frac{1}{n}} - 1)$$

C_i und T_i stehen für die Ausführungsdauer und die Periode, n für die Anzahl und i für den jeweils betrahchteten Task. Nach Einsetzen in die Formel ergibt sich durch

$$\frac{2}{10} + \frac{2.5}{50} + \frac{2.5}{50} + \frac{5}{20} + \frac{5}{20} = 0.8 \leq 5(2^{\frac{1}{5}} - 1) = 0.74$$

ein Widerspruch, womit eine gültige Einplanung mittels RMA nicht möglich ist.

4.5 Ausblick

Die Arbeiten am SPT-Profil begannen im Jahre 1999 und fanden 2003 mit der Verabschiedung der Spezifikation durch die OMG ihren vorläufigen Abschluß. Zu diesem Zeitpunkt war noch die UML 1.x der offizielle Standard, weshalb dieses Profil noch nicht die Möglichkeiten der Version 2.0 nutzt. Mittlerweile wird an einer neuen Version gearbeitet, wobei jedoch weniger eine Aktualisierung beabsichtigt wird als viel mehr ein neues Profil, das das vorhandene ersetzen soll. Ein entsprechender *Request for Proposals* (RFP) und mittlerweilse auch erste Vorschläge dazu sind bereits veröffentlicht. Dieses sogenannte „MARTE"-Profil wird in Kapitel 5 beschrieben.

5 UML-Profil : QoS/FT

Quality of Service (QoS) wird in der Softwareentwicklung allgemein verstanden als die Güte und die damit verbundene Fähigkeit eines Dienstes, bestimmte Anforderungen zu erfüllen. Diese domänenspezifischen bzw. generellen Anforderungen sind in der Regel nicht-funktional. Hierbei steht das „Wie" im Vordergrund, d.h. die Art und Weise, wie ein Dienst seine ihm zugeordnete Funktion erfüllt und mit welchen Randbedingungen. Die Betrachtung von *QoS* verfeinert den Begriff der nicht-funktionalen Anforderung und führt quantitative sowie qualitative Merkmale ein. Ein Merkmal ist quantitativ, wenn es direkt meßbar bzw. zählbar und durch einen numerischen Wert darstellbar ist. Hingegen bedeutet qualitativ das Gegenteil, sprich nicht explizit numerisch ausdrückbar sondern vielmehr unterscheidbar bezogen auf spezifische Eigenschaften. *Fault-Tolerance* (FT) referenziert Lösungen zur Erhöhung der Zuverlässigkeit eines Software-Systems. Dabei lassen sich Unterscheidungen der Art *maximal zulässige Fehlerzahl*, *Vorgehen beim Auftreten eines Fehlers* oder auch *Behebung eines Fehlers* treffen.

5.1 Allgemeine Betrachtungen

Für die formale Definition qualitätsbezogener Anforderungen eines Systems bedarf es neben einer geeigneten Sprache eines Modells, in dem die Wechselwirkungen der Spezifikationen festgehalten werden. Hierzu müssen relevante Kriterien und geeignete Elemente identifiziert werden, mit deren Hilfe eine zufriedenstellende Beschreibung möglich ist. Zunächst lassen sich zwei Arten von Adressaten QoS-bezogener Rahmenbedingungen unterscheiden: zum einen sind es die *Anwender* eines Systems (hierzu zählen auch Clients in einem ClientServer-Verbund), zum anderen sind es die zur Verfügung stehenden *Ressourcen*. Beide stellen gewisse Ansprüche und Anforderungen, die zu erfüllen sind. Die Funktion

$$F(qi, r) \rightarrow qo$$

stellt diese Benutzeransprüche und die verfügbaren Ressourcen in Beziehung und ermöglicht die Beurteilung des Erfüllungsgrades. Hierbei bezeichnen qi (input) die Qualitätsattribute externer Komponenten (bspw. des Benutzers), qo (output) die erzielten Werte und r die Ressourcen. Es wird also der Kombination von gewissen externen Eingangsmerkmalen und der zur Verfügung stehenden Ressourcen ein Wert zugeordnet,

der etwas über die betrachtete Komponente aussagt und deren Charakterisierung erlaubt. Bspw. könnte der durchschnittlichen Tipprate von Benutzern und der bekannten CPU-Leistung eine Antwortzeit für einen automatischen Rechtschreibprüfer zugeordnet werden, der im Hintergrund läuft. Der erzielte Wert könnte somit etwas über die praktische Einsatzfähigkeit aussagen, wobei die Interpretation im entsprechenden Betrachtungskontext erfolgt.

Eine Spezifikationssprache für QoS enthält sogenannte *constructors*, mit deren Hilfe sich qualitätsbezogene Anforderungen quantifizieren lassen. Beispiele hierfür sind Elemente zur Identifizierung von Ressourcen nutzenden Komponenten, sogenannte *resource-consuming components* (RCC) sowie Elemente zur Erfassung funktionaler Maße, sogenannte *QoS-aware specification functions* (QASF). Für die Definition von Qualitätsanforderungen ist es sinnvoll, unterschiedliche Abstraktionsebenen zu betrachten. Hierzu wird in Anlehung an die ISO-Norm [8] eine Unterscheidung zwischen QoS für die Architektur einer Anwendung und deren Analyse getroffen. Für letztere werden die bereits erwähnten QASFs verwendet, wohingegen bei der Betrachtung der Archtiektur die RCCs zum Einsatz kommen.

5.2 Das QoS-Framework Metamodell

Eine Art Infrastruktur für die QoS-Modellierung wird durch das sogenannte *QoS Framework* geschaffen. Es stellt ein Metamodell dar, das nach außen hin eine robuste Hülle bietet und nach innen eine Kategorisierung und Spezialisierung erlaubt. Die Kategorisierung umfaßt dabei domänenspezifische Erweiterungen sowie die Unterscheidung zwischen statischen und dynamischen Qualitäsanforderungen. Das *QoS Framework* faßt die Metamodelle der oben besprochenen *QoS Characteristics*, *QoS Constraints* und *QoS Levels of Execution* jeweils in einem Paket zusammen.

Allgemein betrachtet stellt die QoS-Spezifikationssprache Unterstützung für folgende Elemente zur Verfügung:

- Charakteristiken (*QoS Characteristics*) : quantifizierbare Aspekte, die unabhängig von ihrer Realisierung sind und durch Eigenschaften auf niedrigeren Ebenen spezialisiert werden. Charakteristiken lassen sich im Hinblick auf besondere Domänen gruppieren. Ein Beispiel ist die Verfügbarkeit (*Availability*). Sie kann je nach Anwendung durch eine einfache stochastische Verteilung oder aber durch komplexere Metriken wie *MTTF* (*mean time to failure*) oder *MTTR* (*mean time to repair*) beschrieben werden.

- Einschränkungen (*QoS Constraints*) : alle Arten von Beschränkungen und zulässigen Wertemengen. Sie haben direkten Einfluß auf die Charakteristiken, indem sie diese charakterisieren.

- Ebenen der Ausführung (*QoS Levels of Execution*) : Ausfühungsmodi. QASFs und RCCs können unterschiedliche (oft diskrete) Arten der Funktion haben. Ein Audiosignal kann z.B. in den Qualitäten niedrig, normal und hoch abgespielt werden. Für die Kontrolle der zulässigen Ausführungspfade speziell beim Wechsel der Ausführungsmodi spielt zudem das *QoS Adaption and Monitoring* eine wichtige Rolle. Es ermöglicht auch die Erkennung von Nichteinhaltungen von Einschränkungen und spezifiziert notwendige Aktionen in einem solchen Falle.

Das *QoS Framework* spezialisiert diese Elemente durch eine weitere Gliederung: QoS Characteristics wird unterteilt in

- *QoS Dimension* - die Quantifizierung einer Charakteristik läßt sich in mehrere Dimensionen aufteilen.
- *QoS Dimension Slot* Repräsentation des konkreten Wertes einer *QoS Dimension*.
- *QoS Category* - ermöglicht die Gruppierung von *QoS Characteristics*.
- *QoS Value* - spezifische, nicht mit *QoS Characteristics* oder *QoS Context* quantifizierbare Werte.
- *QoS Context* - beschreibt den Kontext einer QoS-Eigenschaft wenn mehr als eine *QoS Characteristic* benutzt wird.

QoS Constraints teilt sich auf in

- *QoS Required* - geforderte Eigenschaften, wobei sowohl Mindest - als auch Maximalkriterien dargestellt werden können.
- *QoS Offered* - die Menge und Art der angebotenen Dienst-Qualitäten einer Komponente.
- *QoS Contract* - eine Übereinkunft über die geforderten und die angebotenen Dienst-Qualitäten.

QoS Level enthält das Element *QoS Transition*, welches die erlaubten Transitionen zwischen den unterschiedlichen Ausführungsmodi beschreibt.

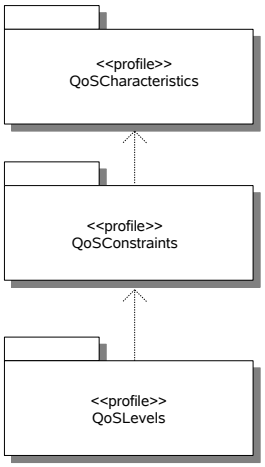


Abbildung 7: Der Aufbau des QoS-Profils

5.3 QoS und UML

Das QoS-Profil hat dieselbe Struktur wie das zugrunde liegende *QoS-Framework*. Die Erweiterungen, die es zur Verfügung stellt, sind in den drei Subprofilen *QoSCharacteristics*, *QoSConstraints* und *QoSLevels* enthalten. Ihre Definition erfolgt wie oben gesehen bereits im *QoS-Framework*. Die Subprofile überführen diese Definitionen in eine UML-konforme Form und machen sie für diese Modellierungssprache nutzbar. Wie in Abb. 7 zu sehen ist, bilden die drei Subprofile eine Hierarchie, die in gleicher Form im *QoS-Framework* existiert.

Das UML-Pofil QoS/FT ermöglicht die Spezifikation nicht-funktionaler Anforderungen von Software-Systemen und unterstützt dabei unterschiedliche Analyseverfahren. Es ist somit möglich, jedes UML-Modell mit QoS/FT- Erweiterungen bestimmter Domänen anzupassen. Dazu werden wie beim SPT-Profil Stereotypen, Eigenschaftswerte und Einschränkungen verwendet. Das Vorgehen bei der Erweiterung eines Models mithilfe des QoS-Profils erfolgt dabei in drei Schritten:

- Zuerst wird die für die Aufgabe benötigte Menge an spezifischen QoS-Merkmalen aus dem sogenannten *QoS-Catalog* zusammengetragen. Mit „Aufgabe“ ist hierbei das zu bearbeitende Modell samt der verwendeten Analyseverfahren im Kontext der Zieldomäne gemeint. Der *QoS-Catalog* ist eine Sammlung allgemeiner Charakteristiken und Kategorien, die nicht domänenspezifisch sind. Beispiele hierfür sind u.a. *Performance*, *Dependability* oder *Efficiency*. Die Charakteristiken sind Schablonen-Klassen (*template classes*) mit freien Parametern und noch ohne feste Verbindungen. In Abb. 8 a) wird exemplarisch sowohl eine bereits vorhandene (*turn arround*) als auch eine individuell erweiterte *QoS Characteristic* (*alarm-latency*, abgeleitet von *latency*) verwendet.

- Im zweiten Schritt wird das *Quality-Model* definiert. Dazu werden die Parameter der im ersten Schritt definierten Charakteristiken gesetzt und Bindungen zwischen den Template-Classes erstellt. Abb. 8 b) zeigt eine konkrete *QoS Characteristic*, die aus *turn-arround* hervorgeht und eine Bindung an die Einheit „ms“ bekommt.

- Schließlich folgt die bekannte Erweiterung der Modelle mit den Erweiterungselementen Stereotyp, Eigenschaftswert und Einschränkung aus dem QoS-Profil. Im Beispiel in Abb. 8 c) wird der Stereotyp *<<QoSRequired >>* auf eine Komponente „AS: Automation System“ angewendet. Er wurde u.a. durch die im zweiten Schritt erstellte *QoS Characteristic* „PA4ASturn-arround“ spezifiziert und legt nun fest, daß die Komponente eine Antwortzeit von maximal 15 ms haben darf.

6 UML-Profil : MARTE

Das *UML-Profile for Modeling and Analysis of Real Time and Embedded Systems* (MARTE) soll die Nachfolge des Profils SPT antreten. Zum Zeitpunkt der Erstellung dieser Seminararbeit liegen neben dem *Request for Proposals* (RFP) auch Vorschläge für die konkrete Gestaltung vor [4][5]. Eine offizielle Verabschiedung steht jedoch noch aus. Die folgenden Betrachtungen zum MARTE-Profil stützen sich auf öffentliche Diskussionen sowie eingereichte Vorschläge zum RFP.

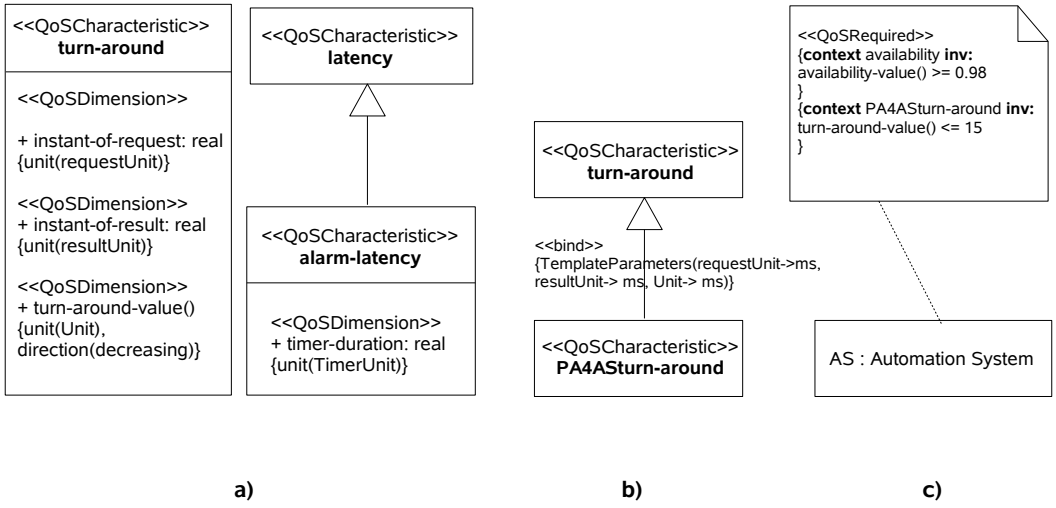


Abbildung 8: Beispiel einer QoS-Anwendung

Einleitend sollen nochmals die in den vorhergehenden Kapiteln beschriebenen Profile SPT und QoS/FT betrachtet und dabei Defizite aufgeführt werden. Damit soll die Motivation für ein neues UML-Profil begründet und eine (nicht vollständige) Übersicht über damit einhergehende Anforderungen gegeben werden.

6.1 Motivation

Die UML bietet an sich wenig bzw. nicht sehr weit gehende Unterstützung für die Modellierung von QoS-Anforderungen. Eine Lösung hierfür stellen die bereits kennengelernten UML-Profile dar. Für QoS-Anforderungen sind dies in erster Linie das SPT- sowie das QoS/FT-Profil. SPT beschäftigt sich speziell mit Echtzeitsystemen und deren Belangen. QoS/FT ist weniger auf eine Domäne beschränkt und spricht allgemein die Modellierung unterschiedlicher Arten von QoS-Merkmalen an. Beide Profile haben vor ihrer offiziellen Verabschiedung eine lange und sorgfältige Evaluierungsphase durchlaufen, in der ihre Eignung für die angestrebten Bereiche sichergestellt wurde. Jedoch kamen – wie bei den meisten geistigen Produkten – mit der Zeit Kritikpunkte auf, die sowohl auf Erkenntnisse durch den praktischen Einsatz als auch auf Erfahrungen mit aufkommenden neuen Technologien zurückzuführen sind. Eine Gegenüberstellung beider Profile findet sich in [2]. Verbesserungsvorschläge für das SPT-Profil besonders in Hinblick auf Echtzeit-Anwendungen werden in [10] erläutert. An dieser Stelle sollen einige Defizite vor allem des SPT-Profils aufgeführt werden.

- *Arten der Analyse* : SPT erlaubt die Modellierung von Anforderungen sowohl die Ablaufplanung als auch die Leistungsanalyse betreffend. Was jedoch fehlt und wünschenswert wäre, ist die Möglichkeit, Abhängigkeiten z.B. zwischen Signalen oder Ausführungszeiten auszudrücken. Dies sollte in einem Subprofil realisiert werden und ähnlich *AnalysisModels* auf die Elemente aus *GeneralResourceModels* zugreifen.

- *Vorgehen bei der Annotierung von Modellen* : Im Gegensatz zum QoS-Profil, dessen Anwendung einen dreistufigen und relativ aufwendigen Vorgang darstellt, ist das SPT-Profil leicht und schnell anzuwenden. Wünschenswert wäre jedoch die Möglichkeit, benutzerdefiniere Metriken einführen zu können. Diese müßten sich auf Basischarakteristiken oder -attribute stützen, die wie im QoS-Profil auf vorhergehenden Vereinbarungen basieren.

- *Möglichkeiten der Annotierung* : Ein von mehreren Autoren aufgeführter Kritikpunkt beim SPT-Profil ist die Beschränkung auf Sequenz-, Aktivitäts- und Kommunikationsdiagramme bei der Performanceanalyse. SPT benutzt hierfür das Konzept der *Scenario*s, welche eine Aneinanderreihung einzelner Zeitschritte (*Steps*) sind und eben nur die genannten Diagrammtypen erlauben. Eine Alternative hierzu wäre die Spezifizierung des Verhaltens eines Systems mittels Zustandsdiagrammen. Dieser Ansatz der „object-life“ basierten Analyse wird in [11] erläutert. Ebenfalls wünschenswert wäre die Unterstützung von Verzögerungsmaßen für die Ereignisverarbeitung sowie die Möglichkeit, die Größe von Nachrichten angeben zu können. Beides ist mit dem SPT-Profil nicht oder nur bedingt darstellbar.

- *Kompatibilität* : Wie bereits erwähnt, liegt dem SPT-Profil die UML 1.x zugrunde, Neuerungen der aktuellen Version 2.0 sind also nicht enthalten bzw. nicht nutzbar. Ein Grund ist etwa die neue *superstructure* der neuen UML-Version, die nicht abwärtskompatibel ist

- *Echtzeit-Modellierung* : Die Annotaionen von Echtzeit-Aspekten erfolgt mit Stereotypen und Eigenschaftswerten, die im ganzen Modell verstreut sein können. Die Folge ist oft eine erhebliche Einschränkung der Übersichtlichkeit. Insbesondere größere Systeme, die im Verhältnis zum Gesamtumfang einen relativ kleinen (aber wichtigen) Echtzeit-Teil besitzen, werden für Menschen schnell unüberschaubar. Aber auch Werkzeuge, die mit diesen Modellen arbeiten, können aufgrund fehlender Unterstützung aller Stereotypen und Eigenschaftswerte ihre Aufgabe oft nicht vollständig erfüllen. Sinnvoll wäre die Auslagerung aller Echtzeit-Aspekte in eine seperate Sicht. Hierzu sollten in gleicher Weise, wie es bei der Repräsentation von Instanzen durch Klassen und Objekte geschieht, geeignete Elemente speziell für die Echtzeitmodellierung eingeführt werden.

Abb. 9 zeigt eine tabellarische Gegenüberstellung beider Profile mit der Betrachtung spezieller Aspekte (siehe auch [4]). Zusammenfassend kann man sagen, daß das SPT-Profil zwar einfach anzuwenden ist, es ihm aber an Flexibilität fehlt. Seine Struktur erlaubt keine benutzerdefinierten QoS-Eigenschaften, um unterschiedliche Analysemethoden zu unterstützen. Dagegen bietet das QoS-Profil diese Möglichkeit u.a. aufgrund seiner Bibliotheksstruktur. Diese Flexibilität geht jedoch mit einem relativ komplexen Anwendungsprozess einher.

6.2 Ziele von MARTE

Die Notwendigkeit, das SPT-Profil zu überarbeiten, wird u.a. durch die Weiterentwicklung der UML begründet. Aber auch neue aufkommende Technologien erheben

Anforderung	SPT-Profil	QoS-Profil
Annotierungsprozess	leicht-gewichtig	schwer-gewichtig
Unterstützung für benutzerdefinierte Maße	Nein (Maße sind vordefiniert)	Ja (explizite Unterstützung)
Typ für Zeitwerte	RTtimeValue	Nein
Benutzerdefinierte Verzögerungsmaße zwischen beliebigen Ereignissen	Nein	Nein
Quantitative Variablen und unabhängige globale Parameter	Ja (Teil der TVL-Sprache)	Nein
Ausdrücke zur Definierung quantitativer Eigenschaften	Ja (Teil der TVL-Sprache)	Nein
Ausdrücke zur Definierung von Einschränkungen	beschränkt	Ja (voller Umfang von OCL)

Abbildung 9: Vergleich der Profile SPT und QoS/FT

den berechtigten Anspruch, in dieser äußerst relevanten Modellierungssprache Berücksichtigung zu finden. Seit Jahren wird die Forderung der Entwickler von Echtzeit- und Eingebetten Systemen (*Real-Time and Embedded Systems*, RTES) nach einer standardisierten Methode, die den gesamten Entwicklungszyklus eines Systems auf der Ebene der Modellierung begleitet, immer lauter. Man entschied sich, dem in einem neuen UML-Profil nachzukommen. Das Profil SPT umfaßt bereits die wichtigen Aspekte Ablaufplanung, Leistung und Zeit. Eine denkbare Erweiterung um lediglich noch nicht enthaltene Eigenschaften, die für die Modellierung von RTES notwendig sind, wurde abgelehnt. Gründe hierfür sind u.a. die zu hohe Komplexität eines solchen Vorhabens, insbesondere auch im Hinblick auf strukturelle Defizite dieses Profils.

Es herrscht ein starker Trend weg von Quelltext-basierten Spezifikationen und Beschreibungen eines Softwaresystems hin zu leistungsfähigen Modellen. Durch die Abstraktion wird eine bisher nicht dagewesene Mobilität, Flexibilität sowie Unabhängigkeit erreicht. Die Vorteile liegen auf der Hand. Unterstützt wird dieser Trend durch Technologien wie die *Model Driven Architecture* (MDA) der OMG. Im Bereich der Eingebetten Systeme verlagert sich das Gewicht weg von der Hardwarerealisierung von Funktionen hin zu Softwarelösungen. Nach wie vor bilden aber Hard- und Software eine Einheit, an die idealerweise mit ganzheitlichen Methoden herangegangen wird. Für diese Herangehensweise ist heute noch keine zufriedenstellende Lösung verfügbar. Zu den Ansätzen zählt auch das UML-Profil MARTE. Es verfolgt das Ziel, die Modellierungsanforderungen von RTES - Systemen in einem einzigen UML-Profil zusammenzufassen.

Zu den angestrebten Zielen von MARTE zählen :

- Modellierung von sowohl Software- als auch Hardwareaspekten
- Modellierung im Stil der MDA

- Konsistenz mit dem QoS/FT-Profil
- Neben Echtzeit-Aspekten auch andere QoS-Eigenschaften wie Leistungs- oder Speicherverbrauch
- Modellierung von eingebetteten, reaktiven, und rechenlastigen Systemen
- Analyse und Modellierung komponentenbasierter Systeme
- Modellierung von asynchronen/kausalen, synchronen/getakteten sowie echtzeitfähigen Systemen

6.3 Das MARTE-Framework

Die Punkte im letzten Abschnitt einschließend, sind die Hauptziele von MARTE das Definieren von Zeitstrukturen, Nebenläufigkeits- und Kommunikationsmodellen, das Vereinen von Kontrollfluß und rechenintensiven Datenpfaden sowie das Modellieren von architektonischen Plattformen und die Unterstützung der Y-Chart-Methode von Gajski und Kuhn [1]. All diese Ziele teilen elementare Belange von Zeit, Nebenläufigkeit und Ressourcen, weshalb sie in einem einzigen Profil vereint werden sollen. Das *Request for Proposals* zu MARTE impliziert dazu eine Struktur für ein zugrunde liegendes Framework. Wie in Abb. 10 zu sehen ist, gibt es eine Unterteilung in die drei Sub-Profile

- *Time and Concurrent Recources* (TCR),
- *Schedulability and Performance Analysis* (SPA) sowie
- *Real-Time and Embedded Modeling* (RTEM).

TCR beinhaltet die Basiskonstrukte aus dem *GRM*-Paket des SPT-Profils, nämlich Zeit, Nebenläufigkeit und allgemeine Ressourcen. Es soll Basiselemente für eine Infrastruktur zur Modellierung von Zeit (logisch, diskret, physisch, echt) und Nebenläufigkeits-/Kommunikationsmodellen zur Verfügung stellen. *SPA* stellt Erweiterungen bereit, mit denen sich nicht-funktionale Anforderungen in der Leistungsanalyse und Ablaufplanung untersuchen lassen. Wie bereits beim SPT-Profil, können somit Werkzeuge automatisch Analyseaufgaben übernehmen. Schließlich beinhaltet *RTEM* Eigenschaften für die Modellierung unterschiedlicher Arten von Eingebetteten Systemen. Dazu zählen u.a. Systeme mit sowohl Software- als auch Hardwareteilen auf der selben Modellierungssebene und generell alle, die sich durch die QoS-Charakteristiken *Speicherbedarf* oder *Leistungsaufnahme* beschreiben lassen. In diesem Sub-Profil kommen die Elemente aus den beiden anderen Profilen zusammen und ermöglichen so die Definition von Verhaltensmodellen, die sich über mehrere Hierarchien erstrecken. Hierdurch wird eine unabhängige High-Level-Modellierung von architektonischen Plattformen für verteilte Systeme und deren intelligente Verbindungen ermöglicht.

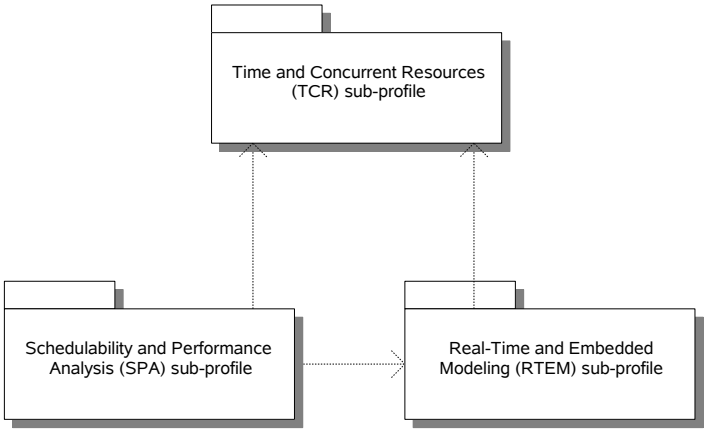


Abbildung 10: Der Aufbau des MARTE-Profils

7 Fazit

In dieser Arbeit wurden die drei UML-Profile *Schedulability, Performance and Time* (SPT), *Quality of Service and Fault Tolerance* (QoS/FT) sowie *Modeling of Real-Time and Embedded Systems* (MARTE) vorgestellt. Zunächst wurde auf die UML mit ihrem heutigen Stellenwert eingegangen und gezeigt, welche Möglichkeiten zur Erweiterung existieren. UML-Profile basieren auf grundlegenden Erweiterungsmechanismen, mit denen sich die UML an bestimmte Domänen anpassen läßt. Hierzu werden Stereotypen, Eigenschaftswerte und Einschränkungen verwendet. Das Konzept der UML-Profile wurde näher erläutert und an einem Beispiel die einfache Anwendung demonstriert. Nach diesen einführenden Grundlagen wurde auf die genannten UML-Profile eingegangen. Das SPT-Profil erlaubt die Modellierung von Aspekten die Ablaufplanung und Leistung eines Systems betreffend. An einem Beispiel wurde gezeigt, wie man mit den Erweiterungen dieses Profils bspw. Fragen der Ablaufplanung beantworten kann. Das QoS/FT-Profil bietet allgemeinere Eigenschaften zur Modellierung von QoS-Anforderungen, ist also nicht auf die Bereiche von SPT beschränkt. Diesem Profil liegt ein Qualitätsmodell zugrunde, das auf die Definition domänenspezifischer Qualitätseigenschaften angewendet werden kann. Abschließend wurde das noch nicht offiziell verabschiedete MARTE-Profil vorgestellt. Es soll das SPT-Profil ersetzen, indem es dessen Anwendungsgebiete erweitert. Dabei wird speziell die Modellierung von Echtzeit- und Eingebetten Systemen (RTES) unterstützt. Damit soll dem Entwickler eine möglichst vollständige Methode zur Erfassung aller RTES-Anforderungen in die Hand gegeben werden.

Abbildungsverzeichnis

Literatur

[1] T. Beierlein, *Taschenbuch Mikroprozessortechnik*, 3rd ed. Fachbuchverl. Leipzig im Carl-Hanser-Verl., 2004.

[2] S. Bernardi and D. Petriu, "Comparing two UML Profiles for Non-functional Requirement Annotations: the SPT and QoS Profiles," *SVERTS, Lisbone, Portugal, October*, 2004. [Online]. Available: http://www-verimag.imag.fr/EVENTS/2004/SVERTS/FINAL/p01.pdf

[3] G. Booch, J. Rumbaugh, and I. Jacobson, *Das UML-Benutzerhandbuch*, 1st ed. Bonn: Addison-Wesley, 1999.

[4] H. Espinoza, H. Dubois, S. Gérard, J. Medina, D. Petriu, and M. Woodside, "Annotating UML Models with Non-Functional Properties for Quantitative Analysis." [Online]. Available: http://www.sce.carleton.ca/faculty/petriu/papers/MARTES-LNCS-CR.pdf

[5] H. Espinoza, H. Dubois, J. Medina, and S. Gérard, "A General Structure for the Analysis Framework of the UML MARTE Profile." [Online]. Available: http://www.martes.org/prev/2005/PAPERS/8.pdf

[6] L. Fuentes and A. Vallecillo, "An introduction to UML profiles," *UPGRADE, The European Journal for the Informatics Professional*, vol. 5, no. 2, pp. 5–13, 2004.

[7] International Organization for Standardization , "Software engineering – Product quality." [Online]. Available: http://www.iso.org

[8] International Organization for Standardization, "CD15935 Information Technology: Open Distributed Processing - Reference Model - Quality of Service." [Online]. Available: http://www.iso.org

[9] I. Jacobson, *Object-Oriented Software Engineering: A Use Case Driven Approach*. Addison Wesley Longman Publishing Co., Inc. Redwood City, CA, USA, 2004.

[10] J. Medina, M. Harbour, and J. Drake, "The UML Profile for Schedulability, Performance and Time in the Schedulability Analysis and Modeling of Real-Time Distributed Systems," *SIVOES-SPT Workshop. Toronto (Canada). May*, 2004. [Online]. Available: http://www.martes.org/prev/2005/PAPERS/8.pdf

[11] J. Merseguer and J. Campos, "Exploring roles for the UML diagrams in software performance engineering," *Proceedings of the 2003 International Conference on Software Engineering Research and Practice (SERP03)(Las Vegas, Nevada, USA), CSREA Press, June*, pp. 43–47, 2003. [Online]. Available: http://webdiis.unizar.es/CRPetri/papers/jcampos/03_MC_SERP.pdf

[12] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen, *Object-oriented modeling and design*. Prentice-Hall, Inc. Upper Saddle River, NJ, USA, 1991.

[13] The Object Management Group , "Unified Modeling Language." [Online]. Available: http://www.uml.org/

[14] The Object Management Group, “OMG Model Driven Architecture.” [Online]. Available: http://www.omg.org/mda/