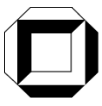
Universität Karlsruhe - Fakultät für Informatik - Bibliothek - Postfach 6980 - 76128 Karlsruhe

Software-Industrialisierung

Herausgeber:
Franz Brosch, Henning Groenda, Lucia Kapova, Klaus Krogmann, Michael Kuperberg, Anne Martens, Pierre Parrend, Ralf Reussner, Johannes Stammel

Autoren:
Emre Taspolatoglu, Anton Truong, Christian Baumgart,
Tom Beyer, Philipp Meier

Interner Bericht 2009-4



Universität Karlsruhe (TH)
Forschungsuniversität · gegründet 1825

ISSN 1432-7864

Fakultät für **Informatik**

Universität Karlsruhe (TH)
Forschungsuniversität • gegründet 1825

Software-Industrialisierung

Seminar im Wintersemester 2008-2009

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

Christian Baumgart, Tom Beyer, Franz Brosch, Henning Groenda, Lucia Kapova, Klaus Krogmann, Michael Kuperberg, Anne Martens, Philipp Meier, Pierre Parrend, Ralf Reussner, Johannes Stammel, Emre Taspolatoglu, Anton Truong

Vorwort

Die Industrialisierung der Software-Entwicklung ist ein zur Zeit sehr stark diskutiertes Thema. Es geht dabei vor allem um die Effizienzsteigerung durch die Steigerung des Standardisierungsgrades, des Automatisierungsgrades sowie eine Erhöhung der Arbeitsteilung. Dies wirkt sich einerseits auf die den Software-Systemen zu Grunde liegenden Architekturen, aber auch auf die Entwicklungsprozesse aus. So sind Service-orientierte Architekturen ein Beispiel für eine gesteigerte Standardisierung innerhalb von Software-Systemen. Es ist zu berücksichtigen, dass sich die Software-Branche von den klassischen produzierenden Industriezweigen dadurch unterscheidet, dass Software ein immaterielles Produkt ist und so ohne hohe Produktionskosten beliebig oft vervielfältigt werden kann. Trotzdem lassen sich viele Erkenntnisse aus den klassischen Industriezweigen auf die Software-Technik übertragen.

Die Inhalte dieses Berichts stammen hauptsächlich aus dem Seminar „Software-Industrialisierung“, welches sich mit der Professionalisierung der Software-Entwi- cklung und des Software-Entwurfs beschäftigte. Während die klassische Software-Entwicklung wenig strukturiert ist und weder im Bezug auf Reproduzierbarkeit oder Qualitätssicherung erhöhten Anforderungen genügt, befindet sich die Software-Entwick- lung im Rahmen der Industrialisierung in einem Wandel. Dazu zählen arbeitsteiliges Arbeiten, die Einführung von Entwicklungsprozessen mit vorhersagbaren Eigenschaften (Kosten, Zeitbedarf, ...), und in der Folge die Erstellung von Produkten mit garantierbaren Eigenschaften. Das Themenspektrum des Seminars umfasste dabei unter anderem:

- Komponentenbasierte Software-Architekturen
- Modellgetriebene Softwareentwicklung: Konzepte und Technologien
- Industrielle Softwareentwicklungsprozesse und deren Bewertung

Das Seminar wurde wie eine wissenschaftliche Konferenz organisiert: Die Einreichungen wurden in einem zweistufigen Peer-Review-Verfahren begutachtet. In der ersten Stufe wurde eine Begutachtung der studentischen Arbeiten durch Kommilitonen durchgeführt, in der zweiten Stufe eine Begutachtung durch die Betreuer. In verschiedenen *Sessions* wurden die Artikel wie bei einer *Konferenz* präsentiert. Die besten Beiträge wurden durch zwei *Best Paper Awards* ausgezeichnet. Diese gingen an Tom Beyer für seine Arbeit *Realoptionen für Entscheidungen in der Software-Entwicklung*, sowie an Philipp Meier für seine Arbeit *Assessment Methods for Software Product Lines*. Ergänzt wurden die Vorträge der Seminarteilnehmer durch zwei eingeladene Vorträge: Collin Rogowski von der 1&1 Internet AG stellte den agilen Softwareentwicklungsprozess beim Mail-Produkt GMX.COM vor. Heiko Koziolek, Wolfgang Mahnke und Michaela Saeftel von ABB referierten über das Thema Software Product Line Engineering anhand der bei ABB entwickelten Robotik-Applikationen.

Gliederung

Die Themen dieses Seminars spiegeln verschiedenen Frage- und Problemstellungen wieder, die sich bei der Software-Industrialisierung ergeben. Dieser technische Bericht gliedert sich dabei wie folgt: Die ersten beiden Artikel *Java Development Tools und Eclipse* sowie *GMF - Graphical Modeling Framework* legen das Augenmerk auf Technologien der modellgetriebenen Softwareentwicklung, sowie darunter liegende Basistechnologien. Danach setzt sich der Artikel *Performance Patterns* mit der Performanz von Softwarearchitekturen und typischen Mustern zur Performanz-Optimierung auseinander. Die generelle Bewertung von Architekturen durch Anwendung der Realoptionentheorie beschreibt der Artikel *Realoptionen für Entscheidungen in der Software-Entwicklung*. Den Abschluss des Berichts bildet eine Betrachtung verschiedener *Assessment Methods for Software Product Lines*.

Dank

Wir möchten uns an dieser Stelle bei allen Teilnehmern des Seminars für ihre engagierte Mitarbeit sehr herzlich bedanken. Ein mehrstufiger Begutachtungsprozess bestehend aus Peer–Reviews sowie Gutachten durch die Betreuer ermöglichte die Auswahl qualitativ hochwertiger Artikel. Insgesamt wurden fünf Ausarbeitungen für diesen technischen Bericht angenommen. Auf der Homepage[1] zu diesem Seminar sind daneben auch die Vortragsfolien der Seminarteilnehmer zu finden, die auf den Sessions der Konferenz des Seminars vorgestellt wurden.

Ganz besonders möchten wir uns bei Herrn Collin Rogowski von der 1&1 Internet AG sowie Herrn Heiko Koziolek, Herrn Wolfgang Mahnke und Frau Michaela Saeftel von ABB für ihre sehr interessanten Vorträge bedanken.

Karlsruhe, Februar 2009

Franz Brosch
Henning Groenda
Lucia Kapova
Klaus Krogmann
Michael Kuperberg
Anne Martens
Pierre Parrend
Ralf Reussner
Johannes Stammel

[1] http://sdqweb.ipd.uka.de/wiki/Seminar_Software-Industrialisierung_WS0809

Inhaltsverzeichnis

Software-Industrialisierung

Java Development Tools und Eclipse 1
Emre Taspolatoglu
1 Einführung und Motivation 1
2 Eclipse 1
2.1 Idee und Entwicklung 2
2.2 Architektur und Plugin-Technologie 2
2.3 Stärken 4
3 Java Development Tools 5
3.1 Eclipse und JDT 6
3.2 JDT 6
3.3 Plug-ins 8
4 Zukunft und Schlussfolgerung 16

GMF - Graphical Modeling Framework 19
Anton Truong
1 Einleitung 19
2 Grundlagen zu EMF 20
2.1 Zuordnung des Frameworks 20
2.2 Welches Problem wird gelöst? 21
3 Grundlagen zu GEF 21
3.1 Model-View-Paradigma 22
4 GMF 22
4.1 Konzepte 23
4.2 Aufgabe 25
4.3 Prozess zur Generierung eines Editors 25
4.4 Vorteile und Nachteile von GMF gegenüber GEF 30
5 Beispiel - Mindmap 30
5.1 Erstellung des Domänen- und Domänen Gen Modells 30
6 Zusammenfassung 34

Performance Patterns 37
Christian Baumgart
1 Einleitung 37
1.1 Design Patterns 37
1.2 Performance 38
1.3 Software Performance Engineering (SPE) 38
2 Allgemeine Performance Patterns 39
2.1 Lazy Load 39
2.2 Fast Path 42

2.3 First Things First . . . 43
2.4 Flex Time . . . 43
3 Distributionsorientere Patterns . . . 43
3.1 Remote Facade . . . 44
3.2 Batching . . . 45
3.3 Command Pattern . . . 45
4 Concurrency Patterns . . . 48
4.1 Pooling . . . 48
4.2 Caching . . . 48
4.3 Optimistic Offline Lock . . . 50
5 Ausblick . . . 50

Realoptionen für Entscheidungen in der Software-Entwicklung . . . 53
Tom Beyer
1 Einführung . . . 53
1.1 Bewertung von Software-Produkten . . . 53
1.2 Klassische Investitionsbewertung mit der Kapitalwert-Methode . . . 54
1.3 Wertorientierte Entwicklungsprozesse . . . 55
1.4 Optionen und Realoptionen . . . 56
2 Wertorientierte Software-Entwicklung mit Realoptionen . . . 61
2.1 Sensitivitätsanalyse der Modell-Parameter . . . 61
2.2 Entwicklungsprozesse . . . 64
2.3 Zeitpunkt der Markteinführung . . . 67
3 Fazit und Schlussfolgerung . . . 68
3.1 Grenzen des Ansatzes . . . 68
3.2 Ausblick . . . 68

Assessment Methods for Software Product Lines . . . 71
Philipp Meier
1 Introduction . . . 71
2 Architecture Assessment . . . 72
3 Established Single Product Architecture Assessment Methods . . . 74
3.1 ATAM: Architecture Trade-off Analysis Method . . . 74
3.2 SAAM: Software Architecture Analysis Method . . . 78
3.3 ARID: Active Review for Intermediate Designs . . . 78
3.4 CBAM: Cost Benefit Analysis Method . . . 78
4 Product-line-specific Issues . . . 79
4.1 Additional Requirements . . . 79
4.2 Variability . . . 79
4.3 Evaluation Time . . . 81
5 Product-line-specific Architecture Assessment Methods . . . 81
5.1 HoPLAA: Holistic Product Line Architecture Assessment method 81
5.2 Focus on Reference Architecture . . . 84
5.3 Focus on Individual Quality Attributes . . . 87
6 Conclusions . . . 93

Java Development Tools und Eclipse

Emre Taspolatoglu

Betreuer: Klaus Krogmann

Zusammenfassung In dieser Seminararbeit werden Eclipse und Java Development Tools (JDT) besprochen. Obwohl das Hauptthema JDT ist, wurde es in der Literaturrecherchephase schnell klar, dass das JDT und Eclipse zusammengehören und zumindest JDT nicht einzeln behandelt werden kann. Deswegen werden Entwicklung, Architektur und Stärken von Eclipse behandelt, bevor JDT intensiver behandelt wird, damit die Entsteheung des JDT und seine Arbeitsweise besser verstanden werden können. Im weiteren Verlauf wird daher erörtert, warum Eclipse und JDT seit ihrer Ersterscheinung so beliebt sind, was sie den Anwendern anbieten und welche Funktionen und Unterschiede sie im Vergleich zu anderen integrierten Entwicklungsumgebungen haben.

1 Einführung und Motivation

Obwohl theoretisch nur ein Kompiler und die ältesten Versionen eines Binders und Laders genügen, um Code in beliebigen Programmiersprachen zu schreiben und Programme zu erstellen, bevorzugen die Entwickler integrierte Entwicklungsumgebungen (engl. Integrated Development Environment) und die darauf basierenden Programmiersprachenwerkzeuge, um ihre Arbeit zu erleichtern. IDEs für Entwickler Systeme, mit denen sie fast ihren ganzen Tag verbringen. Die Anfordungen an die Nutzbarkeit dieser Systeme sind daher ziemlich hoch. Daher versuchen moderne IDEs ihren Anwendern und Entwicklern alle Art der Freiheit und Bedienerfreundlichkeit beim Erstellen von Programmen, Anwendungen und Plug-ins, auch auf unterschiedlichen Programmiersprachen, zu ermöglichen.

2 Eclipse

David Geer behauptet in seinem Artikel „Eclipse Becomes the Dominant Java IDE“ [Geer05], dass Eclipse im Markt der integrierten Entwicklungsumgebungen eine der erfolgreichsten ist und immer mehr Erfolg haben wird, wie viele andere Anwender. Welche Unterschiede, Vor- und Nachteile Eclipse im Vergleich zu anderen Java-IDEs wie Borlands JBuilder, Microsofts Visual Jsharp, Oracles JDeveloper oder Suns NetBeans hat, ist nicht einfach aufzuzählen. Daher erfolgt im weiteren Verlauf der Arbeit eine genaue Abgrenzung und Beleuchtung der Eclipse IDE. Diese macht deutlich, warum Eclipse unter den IDEs besonders erfolgreich ist. [Weye03,Geer05]

In den 90er Jahren wurden die Programmiersprachen immer etwa komplexer, um den Bedarf nach besseren, reibungslosen Anwendungen zu decken. Seitdem ist die Nachfrage nach einer freien, funktionalen und bedienerfreundlichen Kombination aus integrierter Entwicklungsumgebung und Programmiersprachenwerkzeug ständig größer geworden. Dazu wurde Java von Sun Microsystems im Jahr 1995 als eine objektorientierte Programmiersprache eingeführt und nacher im Jahr 2001 sind in Java geschriebene Eclipse IDE und Java Development Tools erschienen. [Weye03,Geer05]

2.1 Idee und Entwicklung

Eclipse ist nicht nur eine einfache, integrierte Entwicklungsumgebung, sondern eine Mischung aus einer normalen IDE zum erleichterten Erstellen vom Programm-Code und einer alleinstehenden Plattform, was genau die Hauptidee von Eclipse bildet. Das Ziel bei der Erstentwicklung von Eclipse war, eine offene, Betriebssystem-übergreifende Plattform als Basis für die Integration unterschiedliche Werkzeuge wie Kompiler, Debugger zu schaffen.

Die Firma Object Technology International (OTI) hat in der ersten Hälfte der 90er Jahren die ersten Schritte für die Technologie von Eclipse gemacht, die auf Java basiert. [Hou07] Das Großunternehmen IBM hat nachher im Jahr 1996 die Firma OTI gekauft. Mit diesem Eclipse Projekt wurde ab dem Jahr 1998 unter dem Dach von IBM weitergearbeitet. [Geer05] Die in Java geschriebene Technologie der internen Architektur von Eclipse war mit etwa zwei Millionen Codezeilen auf aktuellem Stand und konnte dank Erfahrungen (zum Beispiel wegen „Visual Age for Smalltalk Java“) von IBM sehr gut nach den Standarts von Object Management Group (OMG) [Obje] weiterentwickelt werden. Diese Standarts beziehen sich auf die Objekt Orientation, mit denen man heute die Modellierungs-, Programmiersprachen usw. standardisiert, um Probleme mit den Integrationen, Kooperationen usw. zu vermeiden. [MeFRD08] IBM hat im Jahr 2001 Eclipse der gemeinnützigen Gesellschaft „Eclipse Foundation“ übergeben. [Weye03] Die Eclipse Foundation hat zur Zeit etwa mehr als hundert strategische Mitglieder, inklusive einiger der größten IT-Unternehmen wie IBM, Borland, SAP, Oracle, Motorola usw. [Ecli09b] Erst unter Common Public License (CPL) lizenzierte Eclipse ist seit der Übergabe als ein Open-Source-Projekt von Eclipse Foundation getrieben und wurde unter Public License (EPL) lizenziert. Dabei soll erwähnt werden, dass EPL und GPL (GPL - General Public License) nicht gleich sind. [Eclia,Weye03]

2.2 Architektur und Plugin-Technologie

Die Basisarchitektur von Eclipse ist eine pure Komponentenarchitektur, was bei den anderen üblichen kommerziellen oder freien Entwicklungsumgebungen von anderen Anbietern (zum Beispiel NetBeans oder JBuilder) nicht der Fall ist. Die anderen Entwicklungsumgebungen haben allgemein eine einfachere monolithische Architektur, obwohl sie auch Plug-ins unterstützen können, wie zum

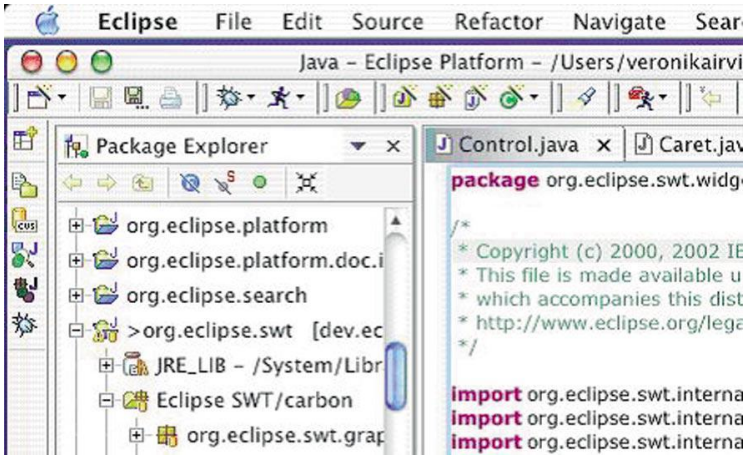


Abbildung 1. Betriebssystem-unabhängige grafische Benutzeroberfläche durch SWT [Weye03]

Beispiel NetBeans. Aber trotz der Plugin-Unterstützung haben Entwickler wegen dieser nach außen-geschlossenen, monolithischischen Architektur Probleme, die die Umgebungen nach ihren Bedürfnissen speziell anpassen wollen. Zum Einrichten durch andere Teilkomponente braucht man immer Programmierschnittstellen, die sogenannten Werkzeug-APIs (eng. „Application Programming Interface“). Die Schnittstellen sind die einzige Möglichkeit, solche IDEs zu erweitern. Sie haben auch das Ziel die größeren Einarbeitungszeiten bei der Integration zu vermeiden. Falls aber der Entwickler seine eigene Erweiterung schaffen will, dann muss er mit begrenzter Freiheit wegen der restriktiven Regeln einer API rechnen, weil bei solchen IDEs mit bestimmten monolithischen Architekturen die APIs schon vorgegeben sind. Weiteres können aber die Erweiterungen und Module von den Drittherstellern Probleme mit der Integration an das gesamte System wegen unterschiedlicher API-Regeln verursachen, was besonders von den Entwicklern unerwünscht sind, die nicht an bestimmten Firmen und ihren Plug-ins oder Erweiterungen stecken bleiben wollen. [Weye03]

Die komponenten-basierte Architektur von Eclipse beseitigt solche Probleme, die oben beschrieben sind. Die Komponenten, auf denen die Gestaltung von Eclipse basiert, sind *Plug-ins*. Diese Plug-ins sind die kleinsten Bausteine von Eclipse. Daneben gibt es die sogenannten *Features*, die aus diesen kleinsten Bausteinen bestehen. Sie sind eine Gruppe aus Plug-ins, die etwa die gleichen Dienste anbieten oder sich an ähnlichen Funktionen beteiligen. Diese Plug-ins und Features geben Eclipse die Eigenschaft „Kompositionalität“, weil sie immer zusammenarbeiten, untereinander kommunizieren und somit das System in einer Übereinstimmung weitererweitern. [Weye03]

Eclipse besteht aus diesen Plug-ins untermauert von einer Infrastruktur, nämlich Plattform Runtime, die die Kommunikation unter diesen Bausteinen ermöglicht. Wie diese Architektur aus Plug-ins zusammengebaut ist, ist leicht zu verstehen. Jeder Plug-in hat entweder einen sogenannten Extension Point, der als Schnittstelle zur Komponentenerweiterung und Zusammenbindung definiert ist, und benutzt selbst einen Extension Point eines anderen Plug-ins, oder kann beispielweise über exportierte Paketen genutzt werden. Unter Eclipse gibt es ein so genanntes „lazy loading“, bei dem der Kernel nur diejenigen Plug-ins, die nicht als „lazy“ markiert sind, aktiviert. Alle anderen Plug-ins werden erst dann geladen, wenn sie tatsächlich benutzt werden. Dadurch spart Eclipse Zeit beim Starten. Eclipse überprüft inzwischen bei jedem Neustart die Version vorliegender Plugins um beim Starten die jeweils aktuellste Version aktivieren zu können. Falls die Plug-ins Abhängigkeiten zu anderen Plugins haben, dann können Sie mit diesen mittels Extension Points und der Platform Runtime kommunizieren. So kommt am Ende eine fertige, funktionstüchtige integrierte Entwicklungsumgebung raus. [vdKK03,Eclia,Ecli09c]

Von IBM entwickeltes Standard Widget Toolkit (abgekürzt SWT), vergleichbar mit Abstract Window Toolkit (abgekürzt AWT) des JDKs, hat die Aufgabe, die native GUI, grafische Benutzeroberfläche, vom Betriebssystem zu übernehmen und auf der Eclipse Plattform zu nutzen. Als ein Beispiel könnte man die Abbildung 1 ansehen, wobei das SWT die Grafikoberfläche von MAC OS auf Eclipse nutzt. Daran sieht man schon besser, dass die Eclipse und auch JDT nicht nur intern sondern auch extern plattformunabhängig sind. [Geer05]

Eclipse ist eine Plattform, die selbst wieder aus Plug-ins besteht, ist letztendlich eine Basis zum Aufbauen von den integrierten Entwicklungsumgebungen für unterschiedliche Programmiersprachen. Dabei werden unterschiedliche Plug-ins in Features zu einer Einheit zusammengefasst. Diesen Hauptunterschied, nämlich die Flexibilität von Eclipse wegen des Aufbaus zwischen den Architekturen der herkömmlichen IDEs und Eclipse kann man an der Abbildung 2 besser erkennen. Man sieht daran, dass die erweiterbare IDEs ganz im Herzen eine vorgeschriebene Basis haben, auf die die unterschiedlichen Erweiterungen hinzugefügt werden, während Eclipse bestehend aus nur Plug-ins, die zusammen funktionieren können, zu einer funktionstüchtigen IDE gemacht wird.

2.3 Stärken

Diese Plugin-Architektur von Eclipse gibt den Entwicklern die Freiheit, indem sie die Module, Werkzeuge, Komponenten usw. von Fremdanbietern und Eclipse eigenen zusammenarbeiten lässt und diese strenge Trennung zwischen offiziellen und nicht-offiziellen Teilkomponenten bzw. Erweiterungen bei den anderen üblichen IDEs aufhebt. Damit können die Benutzer ihre Umgebung genauso anpassen wie sie sie brauchen und wollen. Da diese große Trennung bei Eclipse nicht existiert, können die Fremdanbieter Eclipse sehr gut unterstützen. Leider müssen aber die Plugin-Entwickler mit etwa höheren Einarbeitungzeiten rechnen, weil

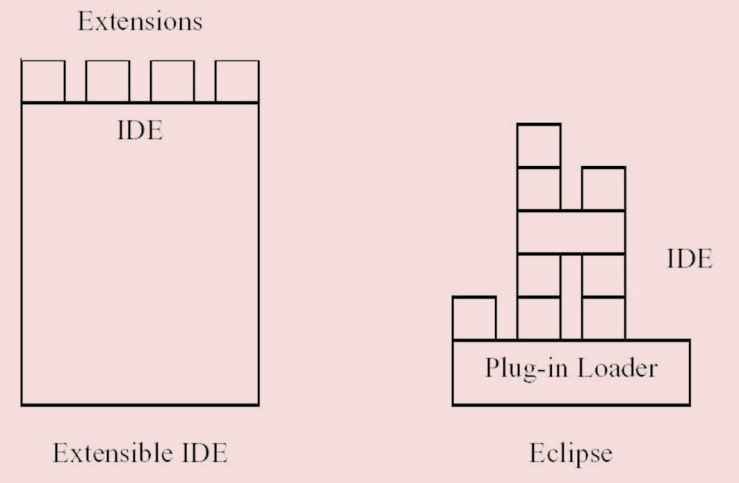


Abbildung 2. Plugins-basierte Architektur von Eclipse [Weye03]

die Plugins-basierte Architektur von Eclipse im Gegensatz zu anderen IDEs mit bestimmten Werhzeug-APIs einigermaßen komplexer ist. Dafür existiert aber bei Eclipse die Plug-in Entwicklungsumgebung, PDE (Plug-in Development Environment), und zwar als Standardteil vom Eclipse SDK (Software Development Kit). Dieses Kit versucht, die Arbeit von Plugin-Entwickler zu erleichtern. Neben der Unterstützung aus „dritter Hand“ existiert ganz normal die dauerhafte Unterstützung von der Eclipse Foundation durch ständige Erweiterungen und Verbesserungen, was eigentlich einer der wichtigsten Gründe vom Erfolg von Eclipse ist. Somit ist Eclipse eine universelle Plattform für sowohl Werkzeug- als auch Anwendung-Entwickler, indem Eclipse sich als Basisplattform für die Entwicklung der eigenen Werkzeuge anbietet. Durch die individuelle Zusammenstellung von Eclipse-Plugins und -Features zu sogenannten „Distributionen“ können nach speziellen Bedürfnissen angepasste IDEs erstellt werden. Die Grobarchitektur von Eclipse, Eclipse Plattform, die aus Plattform Runtime als Basis und den internen Plug-ins wie Workspace besteht, und Eclipse Projekt bestehend aus Eclipse Plattform eingebunden mit JDT und PDE, also Eclipse SDK, kann man an der Abbildung 3 ansehen. [MuKF06,Eclia,MeFRD08,Weye03]

3 Java Development Tools

Die Java Development Tools (JDT) sind eine Sammlung von nützlichen Plug-ins für die Entwicklung Java-basierter Programme und Anwendungen. Obwohl es immer zusammen mit Eclipse zum Gespräch kommt, besitzt JDT eine Struktur als ein einzeln stehendes Plugin-Paket. Trotzdem erweitert JDT die von Eclipse bereits entwickelte Plug-ins, indem die zugehörigen Extension Points zur Verbesserung und Anpassung an die Programmiersprache verwendet werden. JDT hat

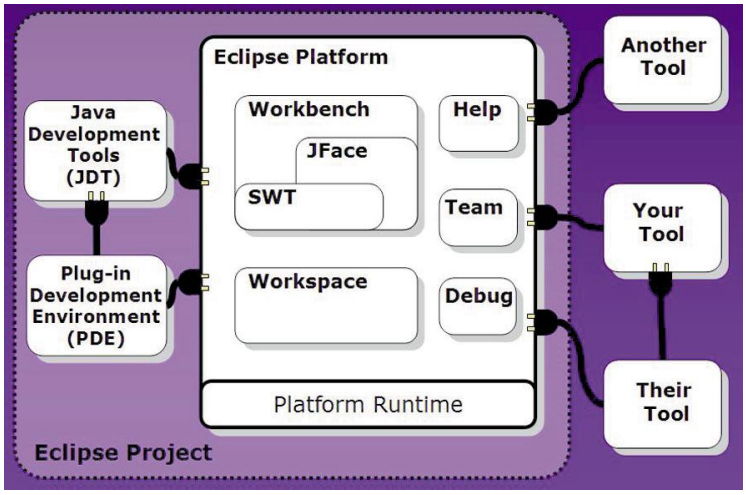


Abbildung 3. Grobarchitektur von Eclipse [Weye03]

im Inneren als Hauptbestandteile Java Core und Java UI(User Interface - Benutzeroberfläche). Das Paket besitzt noch als sogenannte Hilfskomponente Java Text, Java APT (Annotation Processing Tool) und Java Debug. [Eclib]

3.1 Eclipse und JDT

Nachdem Java im Jahr 1995 von Sun und Eclipse im Jahr 2001 von IBM mit der ersten Version 1.0 eingeführt wurden, sind Java Development Tools erschienen. JDT ist ein Open Source Projekt, das zusammen mit Eclipse von Eclipse Foundation entwickelt wurde, nachdem IBM diese Foundation zur Weiterentwicklung von Eclipse verantwortlich gemacht hat. Seitdem ist JDT standardmäßig ein Hauptbestandteil von Eclipse SDK und wird auch als ein Subprojekt von Eclipse durch Eclipse Foundation in Zusammenarbeit mit Sun weiterentwickelt. Standard Eclipse SDK steht als eine Kombination von Eclipse Basis-Plattform (auch als Rich Client Plattform bezeichnet), JDT und PDE zur Verfügung, wie in Abbildung 3 zu sehen ist. [Weye03]

3.2 JDT

Es ist tatsächlich so, dass die Implementierung von JDT auf Plugin-System von Eclipse basiert. Die zwei Hauptbestandteile von JDT sind einmal der Kompiler mit zusätzlichen Funktionen, der sogenannte JDT Core, und zweitens die grafische Benutzeroberfläche, sogenannte JDT UI. Wie diese beiden Hauptbestandteile von JDT in Eclipse bei der Entstehung integriert sind, kann man an der Abbildung 4 besser erkennen. Daran sieht man, dass bei JDT Core zum Beispiel der Java Builder sich an Builders von Workspace einstecken lässt, und bei

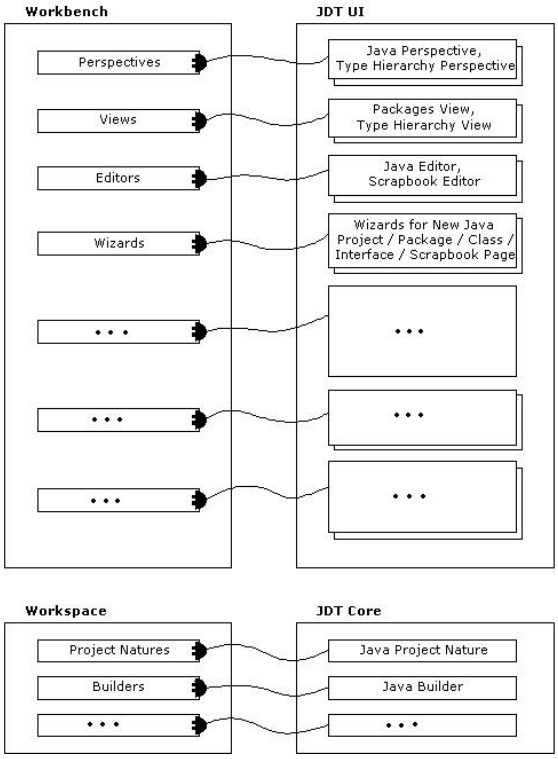


Abbildung 4. Verbindungen zwischen Kern von JDT und Eclipse Plattform [Ecli09c]

JDT UI zum Beispiel die Java Perspective sich an Perspectives, Packeges View sich an Views, Java Editor sich an Editors und Wizards for New Java Projekt an Wizards jeweils von Workbench einstecken lassen. Dabei sind Workbench und Workspace Bestandteile von Eclipse. Mit diesen Begriffen wird es in den kommenden Abschnitten etwa näher gegangen. [Ecli09c] Die anderen drei Plug-ins in dem Paket, JDT Text, JDT APT und JDT Debug, sind eigentlich als Hilfskomponente gedacht, weil sie mit ihren Funktionen die mühsame Arbeit, wie Code schreiben oder kontrollieren, der Entwickler erleichtern.

Aufgrund seiner Implementierung ist JDT mit allen Java-Versionen von 1.4 bis zu den neuesten kompatibel, also braucht man mindestens die Version 1.4 der Programmiersprache um die Werkzeuge benutzen zu können. Zur Erweiterung von JDT und ihren Komponenten steht eine dazugehörige API zur Verfügung, die sich in zwei Teilen aufteilt, nämlich intern und öffentlich/extern. Obwohl es bei der öffentlichen API mit größerer Dokumentation möglichst versucht wird, dass keine massiven Veränderungen in der Evolution von JDT auftreten, kann Eclipse Foundation bei der internen API mit leider kleinerer Dokumentation es nicht

garantieren, dass es sich nichts ändert. Dies stellt insbesondere für Komponenten-Entwickler von Eclipse JDT ein Problem, weil sie besonders die internen APIs benötigen, um JDT zu warten und auszudehnen. [Robb07]

3.3 Plug-ins

JDT unterscheidet intern fünf Arten von Plug-ins. JDT Core und JDT UI sind die Basisbestandteile, auf denen man andere nichttriviale Tools wie Aspect Java Development Tools (AJDT) aufbauen kann. AJDT ist die spezielle Version von JDT, die sich auf die aspekt-orientierte Version von Java bezieht. [Robb07] Weiteres dienen die anderen Plug-ins vom JDT wie Text, APT(Annotation Processing Tool) und Debug zur Vervollständigung und Verbesserung des ganzen Pakets für die Entwickler, indem mit unterschiedlichen Funktionen den End-Anwendern beim Erstellen des Java-Quelltexts geholfen wird. [Eclib]

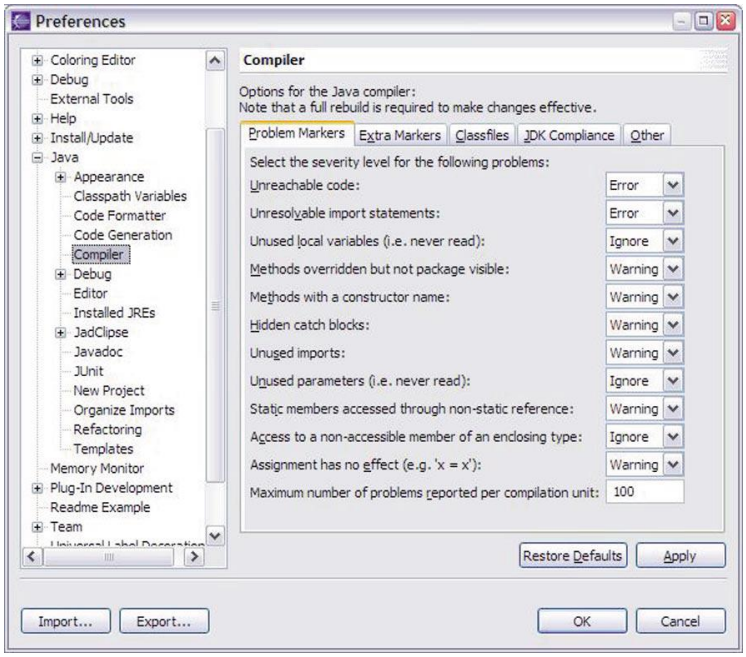


Abbildung 5. Eclipse Komplierer Einstellungsmenu [Weye03]

JDT Core. Core (auf Deutsch übersetzt „Kernel“) ist die Hauptinfrastruktur vom JDT und definiert die sich an die Basis des Systems orientierte Funktionen, welche nicht zur Benutzeroberfläche gehören. Auf der tiefsten Ebene vom Core hat ein alleinstehender inkrementeller Kompiler Platz, der die Extension

Points von den inkrementellen „Buildern“ von Eclipse in Anwendung bringt. JDT Core umfasst aktuell 1174 Klassen. Vom damaligen „Visual Age for Smalltalk Java - Compiler“ evolvierte JDT Kompiler braucht keine einzige eigene Plattform zum Funktionieren, sogar nicht Eclipse RCP. Das heißt, dass er allein den Java-Quelltext durch Befehle über Kommandozeile oder über Batch-Modus kompilieren kann, wie er es auf einer integrierten Entwicklungsumgebung mit einer Benutzerobefläche erledigt, die die Java unterstützt. Über welches Menü man den Kompiler nach eigenen Bedürfnissen einstellen kann, kann man an der Abbildung 5 anschauen. Wie man sieht, besitzt der Kompiler einen sehr großen Bereich für unterschiedliche Einstellungen wie Markierungen oder Mahnungen. [Ecli09c,Robb07] JDT Core stellt eine API zur Verfügung. Diese API ermöglicht

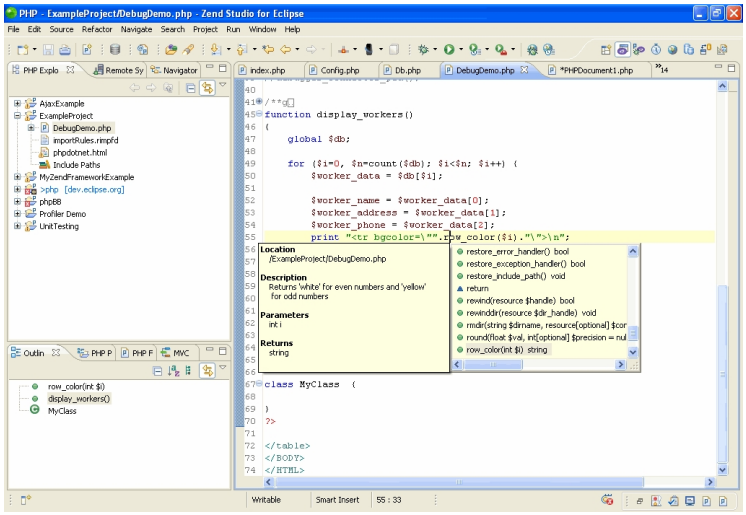


Abbildung 6. CodeAssist mit Hilfe von JDT Core und Text [Zend]

die ordentliche Navigation des abstrakten Syntaxbaums von Java, der die zentrale interne Datenstruktur eines Java Projekts definiert, indem er die Elementen wie zum Beispiel binäre Klassen, Typen, Methoden, oder Felder enthält. Dieser Kompiler unterstützt weiterhin „Code Assist und Select“, was zum Beispiel vorschlägt, welche mögliche Methoden es zu einem im Editor gegebenen Typ gibt. Diese Funktion vom JDT Core funktioniert zusammen mit Texteditor, der eigentlich zum JDT Text gehört. Wie diese Hilfsfunktion sich beim Codieren in den Texteditor integriert sind und dem Entwickler Empfehlungen gibt, kann man an der Abbildung 6 erkennen.

Mit seinem größten Vorteil, dass der Core keine Eclipse-Abhängigkeiten hat, bildet er das Zentrum vom JDT mit JDT UI, der Benutzeroberfläche. [MuKF06,Hou07]

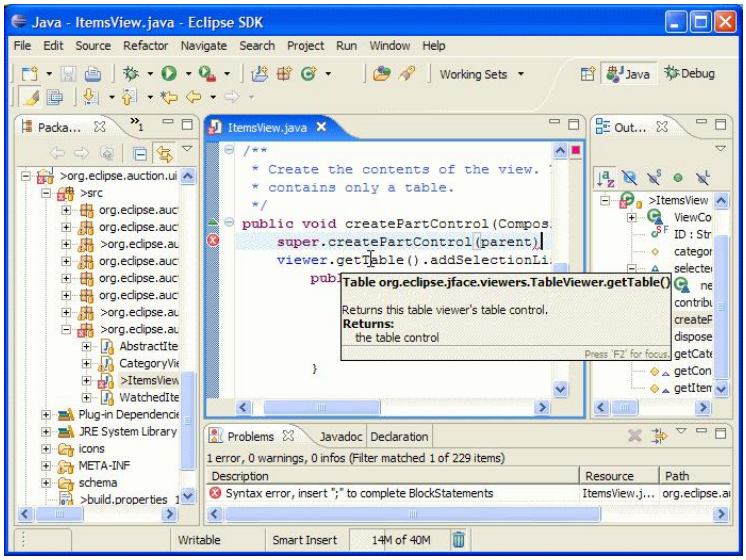


Abbildung 7. Default Eclipse JDT Perspektive auf Windows XP [Eclia]

JDT UI. UI (User Interface) ist die grafische Benutzeroberfläche des JDTs. Wie im letzten Abschnitt erklärt wurde, bauen UI und Core zusammen das Fundament von JDT. Sie übernehmen gemeinsam die Hauptfunktionen, welche man unbedingt benötigt, um mit Quelltextschreiben in den Umgebungen umzugehen, wie kompilieren oder zeigen des Quelltextes. Die grafische Benutzeroberfläche dient als Schnittstelle zur interaktiven Kommunikation zwischen dem System und End-Anwender möglichst.

Eclipse arbeitet mit *Workbenchs*. Sie können als die Implementationen der grafischen Oberfläche von Eclipse definiert werden. Eine Workbench ist aus Editoren, Views und Perspektiven zusammengesetzt. Die *Editoren* eignen sich zu den typischen Aufgaben wie Öffnen, Editieren, Speichern, Schließen usw. *Views* stellen Informationen über Objekte grafisch in den Fenstern zur Verfügung, welche derzeit benutzt werden. Alles was grafisches wird in Views dargestellt. Und *Perspektiven* fassen letztendlich Editoren und Views zu einer Einheit für unterschiedliche Anwenderaufgaben, -probleme und deren Lösungen. JDT bringt insgesamt acht standarte Perspektiven mit und gibt dem Entwickler noch die Möglichkeit eigene zu definieren. Die zwei populärsten und meist benutzten Perspektiven sind einmal die Java-Perspektive, wie man an der Abbildung 7 ansehen kann, und zweitens die Debug-Perspektive, bei der die Werkzeuge zum Debuggen in dafür geeigneten Views dargestellt werden. [MuKF06] An dieser Abbildung erkennt man an der linken Spalte den Package Explorer, in der Mitte den Java Editor, an der rechten Spalte den Outline View und im unteren Teil den Problems View. [Weye03,MuKF06]

Es gibt sowohl Gemeinsamkeiten als auch Unterschiede zwischen Editoren und Views. Beide haben das Ziel, das Codieren dem Entwickler verständlicher, leichter und vielleicht auch spannender zu machen. Der größte Unterschied ist aber, dass die Änderungen in einem Editor, die von einem Entwickler gemacht wurden, explizit gespeichert werden sollen, was in Views nicht der Fall ist. [MuKF06]

Eine der größten Rollen vom JDT UI ist seine Kooperation mit JDT Core im Bereich „Refactoring“. Refactoring ist eine der wichtigsten Funkitonen von JDT, die den Entwicklern zahlreiche Erleichterungen bereitstellt. Die sind beispielsweise Funktionen wie Umbenennen, Schieben oder Namenändern einer vorgeschriebenen Methode oder Klasse, wie man an der Abbildung 8 im Refactoring-Menu von JDT ansehen kann. JDT UI implementiert eigene Wizards. Die Vaterklassen

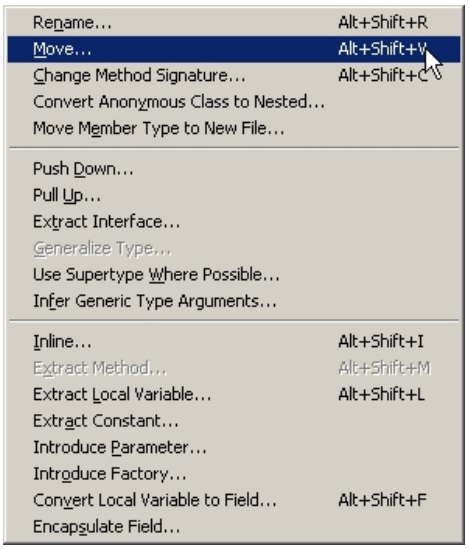


Abbildung 8. Refactoring Menu [Renj06]

und Hilfsklassen von Wizards sowie die graphischen Komponenten sind jedoch in JFace definiert. Diese Wizards dienen dabei zum Erzeugen unterschiedlicher Java-Elemente. Sie sind aber nicht speziell für das JDT-Paket, sondern können sie auch außerhalb des JDTs zahlreich viele Elemente erstellen, dort wo sie gehören, zum Beispiel beim Plug-in-Enwicklung. Ganz allgemein haben die Wizards die Funktion verschiedene Elemente auf dem aktuellen Workbench zu erstellen, zum Beispiel ein neues Java-Projekt.

JDT Text. Text unterstützt standardmäsig die Bearbeitung vom Java-Quelltext durch Texteditoren. Diese Editoren basieren von der Implementierung und Funk-

tionalität her auf den Standardeditor von Eclipse. Dieser Plug-in versorgt den Java-Editor mit unterschiedlichen Funktionen, die den Anwendern besonders bekannt sind, weil sie auf der grafischen Oberfläche auftauchen, wie in Abbildung 9 dargestellt ist. Sie haben das Ziel, die Arbeit der Entwickler beim Codeschreiben zu erleichtern und damit Zeit und Ressourcen zu sparen. Die von JDT Text definierte Texteditoren arbeiten meistens mit den anderen JDT-Plug-ins zusammen. Wie vorher erwähnt erledigen die Editoren sowohl von Eclipse als auch von

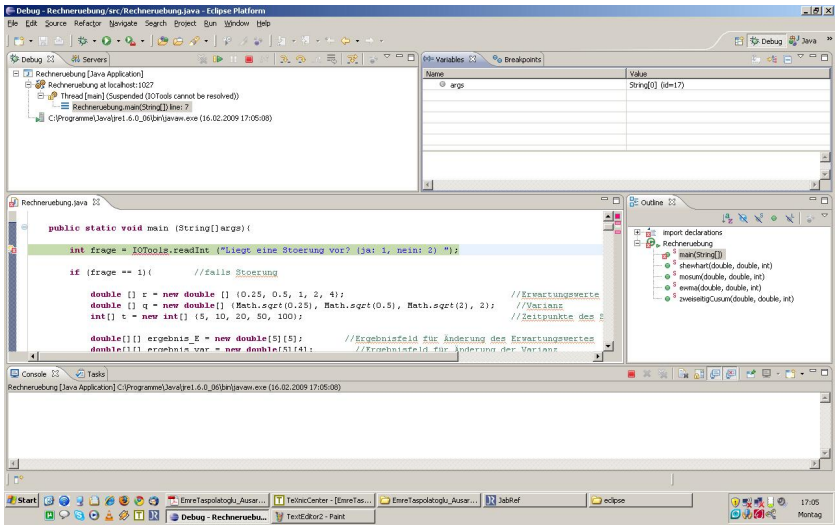


Abbildung 9. Text Editor von Eclipse JDT

JDT die einfachsten Aufgaben wie Öffnen, Speichern usw. Sie haben aber auch viele andere Funktionen. Die Fähigkeiten von dem Java-Texteditor kann man in zwei Gruppen aufteilen, basiert auf, womit diese Funktionen durchgeführt werden. Erste Gruppe enthält die einfacheren Funktionen vom Java-Editor, welche er auch allein ausarbeiten kann. Einige Beispiele dazu könnten ganz trivial die Presäntation, das Durchgehen oder Editieren vom Quelltext auf Views sein. Die Funktionen, die der Texteditor nicht allein sondern mit Hilfe von JDT Core ausarbeitet, bilden die zweite Gruppe. Ein wichtiger Beispiel dazu ist das auf Kontext basierte „Code Assist und Select“ mit JDT Core, was im Abschnitt vom JDT Core besprochen wurde. [Hou07] Einige anderer hilfreichen Funktionen vom JDT Text sind „Keyword und Syntax Coloring“, was besonders auf dem Bildschirm das Ansehen des Quelltextes und die Überprüfung der Syntax mit den eigenen Augen erleictert, QuickFix oder z. Bsp. Kodeschablonen den sozusagenen richtigen Weg dem Entwickler zeigt. Außerdem helfen sie dem JDT APT beim Definieren und Platzieren der Markierungen zu den Annotationen oder dem JDT API beim Zeigen der Javadoc-Spezifikationen. [Hou07]

JDT Text ist der Plug-in, der sich seit der Erstentwicklung des JDTs am meisten evolviert hat. Der Grund dafür könnte sein, dass die Anwender sich besonders mit den Texteditoren des Werkzeugs beschäftigen. Die Programmierer sollen mit der grafischen Umgebung und in diesem Fall mit den Editoren gut umgehen. Um das zu erleichtern, verbessern die Plug-in-Entwickler besonders die Editoren mit neuen hilfreichen Funktionen und Erweiterungen. An dieser Stelle wird den interessierten Lesern die wissenschaftliche Arbeit von Daging Hou „Studying the Evolution of the Eclipse Java Editor“[Hou07] empfohlen, welche sich besonders mit der ständigen Evolution vom Text Editor des JDTs beschäftigt. [Hou07]

```java
@Entity
@Table(name = "people")
public class Person implements Serializable {
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private int id;
    @Column(length = 250)
    private String firstName;
    @Column(length = 250)
    private String lastName;
    @Column(length = 250)
    private String email;

    public String getEmail() {
        return email;
    }

    public void setEmail(String email) {
        this.email = email;
    }

    public String getFirstName() {
        return firstName;
    }

    public void setFirstName(String firstName) {
        this.firstName = firstName;
    }

    public int getId() {
        return id;
    }

    public String getLastName() {
        return lastName;
    }

    public void setLastName(String lastName) {
        this.lastName = lastName;
    }
}
```

Abbildung 10. Ein Quelltextbeispiel zu den „Annotationen“ [hufk09]

JDT APT. Um das Bild von JDT zu vervolständigen, wird es hier der Plug-in JDT APT kurz dargestellt. Den interessierten Lesern werden die „new and noteworthy“ Notizen von der Eclipse Foundation auf der Webseite vom Eclipse

Projekt weiterempfohlen. [Eclic,Eclia]

APT steht für „Annotation Processing Tool“. Annotationen, auch als Metadaten bezeichnet, sind eine Art Code, der durch den Kompiler nicht kompiliert wird, weil er nicht zur Semantik des Quelltextes gehört. Annotationen beeinflussen trotzdem das Programm in der Laufzeit, weil die anderen Sprachtools wie Bibliotheken sie benutzen können. Ab der Version 5.0 von Java können die Anwender ihre eigenen Annotationen definieren. [Sun]

JDT APT ist die Prozessinfrastruktur zu den „Annotations“, erst aber ab der Java Version 5.0. Er ist ein Prozessor, ein Kompiler-Plug-in, der beim Kompilieren Informationen über dem Quelltext sammelt und dazu zusätzliche Typen, Informationen oder sogar Errors nach den Annotationen im Quelltext vom Entwickler erzeugt. Was noch bei APT zu beachten ist, dass dieser Plug-in nicht auf Suns Java implementiert, sondern durch ein Werkzeug-API von Eclipse entwickelt wurde, um nun intern auf Eclipse IDE zu funktionieren. Wie diese Annotationen im Quelltext aussehen und wie JDT APT mit denen umgeht kann man an der Abbildung 10 ansehen. [Eclic]

JDT Debug. Debugging ist eine der unverzichtbaren Funktionen der integrierten Entwicklungsumgebungen. Ganz allgemein kann man das Debuggen als ein System mit verschiedenen Funktionen und Methoden zur Fehlersuche im Quelltext definieren. Dabei wird der Quelltext vom Debugger beobachtet. Der Debugger markiert dann nach dem sogenannten Debug-Ausführung die mögliche Fehlerstellen und warnt den Entwickler dazu. Die Fehlersuche kann den Entwicklern meistens Schwierigkeiten verursachen, und zwar besonders dann, wenn der Quelltext und die Anwendung etwa eine große Komplexität besitzen. Dann wird es für den Entwickler mühsamer, und sogar unmöglicher den Fehler aufzuspüren. Deswegen spielen die Debugger eine sehr große Rolle für Anwender und es ist sehr wichtig, diese Funktion auf dem neusten Stand zu halten. [Daum08]

JDT Debug ist eigentlich ein Subprojekt von „Eclipse Debug Project“, der von den Eclipse Foundations Plug-in- und Systementwicklern entwickelt wird. JDT Debug basiert sich auf Plattform-Debug von Eclipse. Zum besseren Verständnis werden sie unten als zwei verschiedene Aufzählungspunkte erklärt. [Eclia] Daneben wird für die Fehler, sozusagene Bugs, wurde damals eine Lager getrieben von Eclipse Foundation aufgebaut, falls sie gefunden und gemeldet werden. Diese Lager ist seitdem noch aktiv und wird immer ständig größer mit der größen Teilnahme von allen Entwicklern aus der ganzen Welt. Mit einem interaktiven System „Bugzilla - bug tracking system“ [Ecli09a] zwischen allen Anwendern werden in diese Lager die Bugs eingesammelt und für die anderen frei zugänglich gemacht. [MeFRD08]

Plattform-Debug Plattform-Debug erzeugt durch Schnittstellen von Eclipse eine allgemeine Debugger-Basis, die unabhängig von verwendeten Programmierspra-

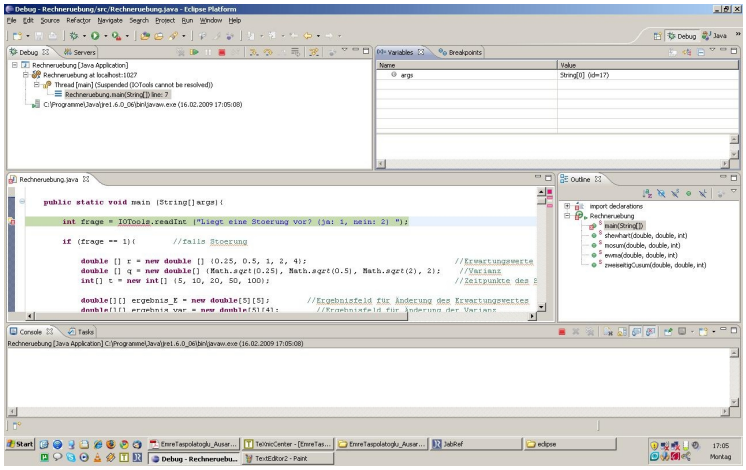


Abbildung 11. Debug-Perspektive

chen auf Eclipse IDE das Debuggen ermöglicht. Unter den definierten sprachenunabhängigen Mechanismen des Plattform-Debugs kann man zum Beispiel das Starten eines Programms, Definieren und Platzieren der Breakpoints und Erstellen der Warnmeldungen aufzählen. Der Plattform-Debug wird dem Anwender über einer sprachenunabhängigen Benutzeroberfläche angeboten. Der größte Vorteil dieses Modells ist seine Sprachenunabhängigkeit. Somit kann dieses Modell in der Regel von jedem Programmiersprachenwerkzeug als Vorbild für ihre eigene Debuggers genommen werden. Sein Nachteil ist auf der anderen Seite, dass er allein nicht debuggen kann. Er braucht auf jedem Fall einen Plug-in des verwendeten Programmiersprachenwerkzeugs, der ihm die sprachenspezifische Eigenschaften weiterleitet und aus ihm einen normalen Debugger implementiert. [Eclia,Daum08]

JDT-Debug JDT Debug dient zur Debugunterstützung der laufenden Java Anwendungen. Dieser Java Debugger wurde durch Erweiterung vom Eclipses Standard Debugger implementiert und ist mit allen auf JPDA (Java Plattform Debugger Architecture) basierte Java Virtuelle Maschinen kompatibel, aber auch abhängig vom Plattform-Debugger. Die spezielle Architektur vom JDT Debugger ermöglicht die speziellen Debugging-Funktionen wie das Erstellen der Break-Points und Exception-Levels, oder Hot-Swaps. Break-Points sind sogenannte spezielle Haltestellen für den Debugger während des Durchlaufens im Quelltext und Exception-Levels sind dabei entstandene Debug-Ebenen, wobei sich der Debugger je nachdem unterschiedlich verhaltet. Hot-Swaps machen es möglich, nachdem man bei einem Break-Point unterschiedliche Änderungen vorgenommen hat, das geänderte Programm von dieser Haltestelle weiter auszuführen. Diese spezielle Funktionen vom JDT-Debugger in der Debug-Perspektive kann man an der Abbildung 11 ansehen. [Daum08] Der Java-Debugger basiert auf dieses generi-

sches Debug-Modell durch standardisierte Debug-Events wie Suspend, Exit und Debug-Auktionen wie Resume, Terminate, Step usw. [Weye03] Debuggers wichtigste Funktion ist aber die sogenannte „Launch Configuration“. Dabei startet der Java-Debugger entweder auf Laufzeit- oder auf Debug-Modus eine Java Virtuelle Maschine, um die Überprüfungen unterschiedlicher Codes durchzuführen. Er kann sich aber auch an einer schon aktiven Java Virtuelle Maschine anschließen. Da er nur sprachensprezifisch arbeiten kann, ist der Java-Debugger nicht flexibel wie ein Plattform-Debug-Modell. [Eclia]

4 Zukunft und Schlussfolgerung

Die Zukunft von JDT ist breit und glänzend. Weil Eclipse Foundation die Weiterentwicklung von JDT ständig unterstützt und die Kooperation mit dem Projekt Eclipse erlaubt, wird JDT in Zukunft noch mehr nützliche Funktionen besitzen, die den Entwicklern die Arbeit ziemlich erleichtern werden. Die Unterstützung durch Dritthersteller steigt inzwischen auch ständig auf.

Ob aber JDT auch seine Nachteile wie Komplexität oder Systemanpassungsprobleme beibehalten wird, oder ob die unterschiedlichen Sorgen von Anwendern wie zum Beispiel Komponentenanpassungen von Drittherstellern verschwinden, kann man nicht genau vorhersehen. Letztendlich ist JDT aus der Sicht des Autors der beste Sprachenentwicklungstwerkzeug für Java zur Zeit, der auf so einer freiheitliche und benutzerfreundlichen IDE wie Eclipse basiert. Es ist auch einer der größten Vorteile von diesem System, dass es völlig kostenfrei ist. Es ist denn am Ende nicht so schwer zu behaupten, dass das Paar Eclipse und JDT zusammen den ersten Platz in den Rankings von auf Java konzentrierten integrierten Entwicklungsumgebungen haben können.

Literatur

Daum08. Berthold Daum. *Java-Entwicklung mit Eclipse 3.3 - Anwendungen, Plugin und Rich Clients.* dpunkt.verlag Heidelberg. 2008.

Eclia. Eclipse Foundation. Eclipse. Zuletzt zugegriffen am 09.01.09. http://www.eclipse.org.

Eclib. Eclipse Foundation. Eclipse Java Development Tools (JDT) Subproject. Zuletzt zugegriffen am 09.01.09. http://www.eclipse.org/jdt/.

Eclic. Eclipse Foundation. JDT APT Projekt. Zuletzt zugegriffen am 09.01.09. http://www.eclipse.org/jdt/apt/.

Ecli09a. Eclipse Foundation. Bugzilla - bug tracking system. Zuletzt zugegriffen am 09.01.09, 2009. https://bugs.eclipse.org/bugs/.

Ecli09b. Eclipse Foundation. Eclipse Foundation Memberships. Zuletzt zugegriffen am 09.01.09, 2009. http://www.eclipse.org/membership/exploreMembership.php.

Ecli09c. Eclipse Foundation. Eclipse Platform Technical Overview. Zuletzt zugegriffen am 09.01.09, 2009. http://www.eclipse.org/articles/Whitepaper-Platform-3.1/eclipse-platform-whitepaper.html.

Geer05. David Geer. Eclipse Becomes the Dominant Java IDE. *IEEE Computer*, 38(7), 2005, S. 16–18.

Hou07. Daqing Hou. Studying the Evolution of the Eclipse Java Editor. In Li-Te Cheng, Alessandro Orso und Martin P. Robillard (Hrsg.), *ETX*. ACM, 2007, S. 65–69.

hufk09. hufkens.net RIA Developer]. Connecting an AIR client with BlazeDS using Spring/Hibernate. Zuletzt zugegriffen am 09.01.09, 2009. http://www.hufkens.net/2008/05/air-blazeds-spring-hibernate/.

MeFRD08. Tom Mens, Juan Fernández-Ramil und Sylvain Degrandsart. The Evolution of Eclipse. In *ICSM*. IEEE, 2008, S. 386–395.

MuKF06. Gail C. Murphy, Mik Kersten und Leah Findlater. How Are Java Software Developers Using the Eclipse IDE? *IEEE Software*, 23(4), 2006, S. 76–83.

Obje. Object Management Group. Object Management Group. Zuletzt zugegriffen am 09.01.09. http://www.omg.org/.

Renj06. Zhou Renjian. Tutorial of J2S in Eclipse (3): Inheritance Example by User and Pet – We Need This Complexity. Zuletzt zugegriffen am 09.01.09, Jan. 15 2006. http://j2s.sourceforge.net/articles/tutorial-hello-j2s-inheritance.html.

Robb07. Robby. An Evaluation of The Eclipse Java Development Tools (JDT) as a Foundational Basis for JML Reloaded, September 13. 2007.

Sun . Sun Microsystems. Java Dokumentation. Zuletzt zugegriffen am 09.01.09. http://java.sun.com/j2se/1.5.0/docs/guide/language/annotations.html.

vdKK03. M. G. J. van den Brand, H. A. de Jong, P. Klint und A. T. Kooiker. A Language Development Environment for Eclipse. In *eclipse '03: Proceedings of the 2003 OOPSLA workshop on eclipse technology eXchange*, New York, NY, USA, 2003. ACM, S. 55–59.

Weye03. Markus Weyerhäuser. Die Programmierumgebung Eclipse. *Javaspektrum Cebit*, Band -, 2003, S. –.

Zend. Zend - The PHP Company. Zend Studio Funktionalitäten - Editor- und Dateiverwaltungsfunktionen. Zuletzt zugegriffen am 09.01.09. http://www.zend.com/de/products/studio/features.

GMF - Graphical Modeling Framework

Anton Truong

Betreuerin: Dipl. - Info. Martens Anne

Zusammenfassung Grafische Editoren für domänenspezifische Sprachen rücken immer mehr in den Vordergrund. Es gibt schon einige Möglichkeiten diese zu erzeugen. GEF bietet solch eine Möglichkeit. Der Vorteil von Graphical Editing Framework ist, dass fast für jedes Modell ein grafischer Editor erzeugbar ist. Leider ist bei GEF der Nachteil, dass viel Arbeit, in Form von Programmierung, investiert werden muss. Hier kommt das Graphical Modeling Framework ins Spiel. GMF generiert einen grafischen Editor für EMF-Modelle. Der Aufwand ist im Vergleich zu GEF sehr gering. Über viele unterstützende Werkzeuge lässt sich schnell ein grafischer Editor erstellen. Eine Modellgenerierung und zwei grafische Definitionen sind notwendig zur Erstellung des grafischen Editors. In dieser Arbeit wird es in den ersten zwei Abschnitten eine Einführung in EMF und GEF geben. Im nächsten Abschnitt wird dann GMF behandelt, da GMF Elemente auf EMF und GEF zurückgreifen.

1 Einleitung

Das GMF, kurz Graphical Modeling Framework, ist ein Plugin von Eclipse. Dieses Framework baut hauptsächlich nur auf Eclipse Modeling Framework-Modelle auf. Anders als bei den anderen Frameworks wie Graphical Editing Framework hat hier das Graphical Modeling Framework den Vorteil, dass die grafischen Elemente z.B. in einem Mindmap auch von Anfang an korrekt funktionieren, ohne jegliche Modifikationen an den Paletten oder den Menüs, da das GMF das Modell bereits kennt. In dieser Hinsicht wird es dadurch die Möglichkeit geben, schnell und einfach einen grafischen Editor basierend auf GMF zu erstellen[KEVH]. Im Verlauf der Geschichte der grafischen Editoren ist es ersichtlich, dass die Generierung von grafischen Unterstützungswerkzeugen für ein Modell, egal ob es ein EMF-Modell oder ein anderes war, mittels Graphical Editing Framework nur langsam und mühsam war bzw. ist. Die Schwierigkeiten bei der Verwendung von GEF sind das komplexe Framework zu verstehen und der redundante Code. Das GEF ist abgesehen von den Schwierigkeiten bzw. Hürden, die bei Verwendung zu bewältigen sind, ein ausgezeichnetes Framework zu Erstellung grafischer Editoren auf Basis von Modellen zum Beispiel von EMF-Modelle. Das GEF, im Sinne des Model-View-Controller (MVC) Paradigmas, erlaubt es einem die eigenen Modelle zu verwenden. In den frühen Tagen des GEFs wurden meistens selbstdefinierte Modelle verwendet, weil dies mit GEF möglich war. Dies veranlasste einem auch Instanzen bereitzustellen, die die Veränderungen am Modell mitbekommen sollen. Änderungen sind beispielsweise Struktur- und

Funktionsänderungen. Der Umstieg auf die Verwendung von EMF-Modellen, brachten auch weitere Möglichkeiten mit sich. Diese Möglichkeiten waren zum Beispiel, dass EMF die Möglichkeit anbietet die Veränderung am Modell weiterzugeben. Es gab jedoch einige technische Herausforderungen (wie verschiedene Command Stacks), um die EMF-Modelle innerhalb des GEF-Frameworks zu integrieren. Diese zögerte die Übernahme des EMF-Modells für die GEF-basierten Editoren. [Anis06]Dadurch, dass das Framework modell-agnostisch ist, ist es ein exzellentes Framework. Modell-agnostisch heißt, dass das Framework nicht unbedingt sein Modell kennt und insbesondere nicht auf EMF Modelle festgelegt ist. Daraus folgt auch mehr Freiheit bei der Generierung von grafischen Editoren unter Verwendung von GEF. [Anis06]Viele Freiheiten heißt auch viel Arbeit. Zum Beispiel ist es notwendig, viel Code von Hand zu schreiben. Der Code muss ins EditPart geschrieben werden, um den View, der mit dem Modellobjekt verbunden ist, zu instanziieren und es es müssen Codes erstellt werden, um den View anzuzeigen, der mit den Daten aus dem Modellobjekt gefüllt wird. [eclic] Im Falle von GMF ist das Schreiben von Code nicht notwendig um Repräsentationen von Modellobjekten im View anzeigen zu lassen. Das Domänenmodell wird meistens 1:1 auf die grafische Definition von GMF abgebildet[Zün]. Das Ziel, das sich das GMF gesetzt hat, ist voll funktionsfähige grafische Editoren auf Basis von EMF-Modellen zu generieren. In vielen Fällen muss kein Editorcode mehr von Hand geschrieben werden. Fluss-Diagramme, Mindmaps, UML-Diagramme oder Editoren zur Abbildung von Geschäftsprozessen sind mögliche Beispiele für solche auf GMF-basierte Editoren. Solche Sprachen, die auf die Bedürfnisse der entsprechenden Domäne angepasst sind, werden auch als Domänen-spezifische Sprachen bezeichnet[KEVH].

2 Grundlagen zu EMF

In diesem Kapitel wird es eine Einführung in das Eclipse Modeling Framework (EMF) geben. Zum besseren Verständnis wird anhand eines Beispiels das Eclipse Modeling Framework erklärt.

Was ist EMF? Eine Entwicklung einer domänenspezifische Sprache beginnt in der Regel mit der Betrachtung des Modells und geht anschließend über zu benutzeroberflächenorientierten Aufgaben. Das Eclipse Modeling Framework wurde ins Leben gerufen um die Entwicklung und Implementierung eines strukturierten Modells zu erleichtern. Das Java Framework bietet einen codeerzeugende Möglichkeit um den Schwerpunkt auf das Modell selber zu richten und nicht auf seine Implementierungsdetails. [MDGW+04]

2.1 Zuordnung des Frameworks

Das Eclipse Modeling Framework wurde als eine Meta Object Facility (MOF) - Implementierung der Object Management Group (OMG) ins Leben gerufen und hat sich zu dem was es heute ist entwickelt. EMF ist eine Erweiterung des

MOF2.0. Es ist ein Open Source Code, der die MOF2.0 Ecore Modelle erweitert und ihren Aufbau in einer Weise restrukturiert, so dass es für die Benutzer einfacher ist. Das Eclipse Modeling Framework ist ein Teil der Model Driven Architecture(MDA). Seine momentane Implementierung ist ein Teil der MDA in der Eclipse family tools. Die Idee hinter MDA ist, dass es ermöglicht den ganzen Applikationslebenszyklus zu entwickeln und zu bereitzustellen, indem es den Schwerpunkt auf das Modell setzt. Das Modell selber wird in einem Meta-Modell beschrieben. Dann wird, durch die Benutzung von Transformationen, das Modell verwendet um Softwarekomponenten zu generieren, die dann im echten System implementiert werden.

Es werden zwei Typen von Mappings definiert: Metadata Interchange, in der Dokumente wie XML, DTD und XSD generiert werden; und Metadata Interfaces, die auf Java abzielen oder auf andere Sprachen und einen IDL Code generieren. MDA ist momentan im Standardisierungsprozess bei der Object Management Group. [MDGW+04]

2.2 Welches Problem wird gelöst?

EMF kann verwendet werden um ein Modell zu beschreiben und zu bauen. Basierend auf dieser Defintion, kann ein Java Code generiert und erweitert werden, durch das Hinzufügen höherer Javacodes. Dieses implementierte Modell kann als Basis für jede Java-Applikationentwicklung benutzt werden. Mittels EMF: Beschreibung und Generierung eines Modells [MDGW+04]

Möglichkeiten zur Erstellung eines Modells In EMF kann das Modell auf drei verschiedene Wege generiert werden:

Die erste MöglichkeitDie XMI Datei wird sofort geschrieben, d.h. die Interfaces und Klassen werden selber geschrieben. Des Weiteren ist es auch möglich so eine XMI Datei aus den Werkzeugen wie Rational Rose und das Omondo EclipseUML-Plugin zu exportieren. Dies wird dann direkt in das Projekt geladen. Die letzte Möglichkeit zu Erstellung eines Modells ist das Kommentieren der Java Interface mit den Modelleigenschaften. [MDGW+04].

3 Grundlagen zu GEF

Das Graphical Editing Framework erlaubt es einem auf einfacher Weise eine grafische Repräsentation für existierende Modelle zu entwickeln. Alle grafischen Visualisierungen werden durch das Draw2D Framework ermöglicht, welches ein Standard 2D Abbildungsframework, basierend auf SWT from eclipse.org, ist [MDGW+04].

Die Editiermöglichkeiten des Graphical Editing Framework erlauben es dem Benutzer grafische Editoren für fast jedes Modell zu bauen. Mit diesen Editoren ist es weiter möglich einfache Modifikationen an den jeweiligen Modellen zu machen, wie das Ändern der Eigenschaften der jeweiligen Bestandteile oder

komplexe Operationen sowie das Ändern der Struktur des jeweiligen Modells auf verschiedene Weise zum selbem Zeitpunkt [MDGW+04].

All diese Modifikationen am jeweiligen Modell können im grafischen Editor gehandhabt werden. Dies ist durch die Verwendung sehr einfacher Funktionen wie 'Drag and drop', Kopieren, Einfügen und Aktionen aufgerufen aus Menüs oder Toolbars möglich[MDGW+04]. Außerdem stellt GEF eine Reihe von Editorfunktionen bereit, welche vom Entwickler direkt genutzt oder entsprechend erweitert werden können. Des Weiteren lassen sich mit GEF eine Vielzahl von Applikationen erstellen, wie z.B. GUI Builders, UML Diagrammeditoren (wie z.B. das OMONDO Plugin) oder XML-Editoren.

3.1 Model-View-Paradigma

Das Model-View-Controller Paradigma hat sich die Zerlegung von interaktiven Systemen in Teile als Ziel gesetzt. Diese sind soweit unabhängig voneinander. Dadurch können sie einzeln geändert bzw. ausgetauscht werden. Diese Trennung (Ausgabe, Eingabe und Verarbeitung) gibt es in der Softwarewelt schon eine Weile. Was das MVC-Paradigma macht ist die Übertragung dieser Trennung auf GUI (Graphical User Interface)-basierte Systeme. Man kann sagen, dass die drei Komponenten (Verarbeitung, Ausgabe und Eingabe) jeweils mit (Model, View und Controller) übereinstimmen. Hiermit wird der Aufbau von Anwendungen mit mehreren Betrachtungen auf die identischen Daten vereinfacht. Des Weiteren hat das MVC-Paradigma das Ziel, dass Umgestaltungen an den Views nach Möglichkeit ohne Umgestaltungen an Model und Controller vorgenommen werden können. [Kepl05]

- Das Modell repräsentiert hier die Daten einer Anwendung. Nicht nur die Verwaltung ist es für die Daten, sondern auch für alle Änderungen derselben zuständig. Die Anzahl der Verbindungen mit einem Modell kann beliebig sein.[Kepl05]

- Die Darstellung der Daten übernimmt das View-Objekt. Man beachte, dass es nur eine mögliche Darstellung des Modells von vielen ist. Wie beim Abschnitt 'Model' erwähnt, kann das View-Objekt nicht mit mehr als einem Modell verbunden sein.[Kepl05]

- Die Aktionsmöglichkeiten des Benutzers mit der Applikation wird durch den Controller festgelegt. Dieser Controller nimmt die Inputs des Benutzers entgegen. Dann folgt eine Abbildung dieser Eingaben auf Funktionen des Modells oder des Views.[Kepl05]

4 GMF

GMF steht für Eclipse Modeling Framework, kurz GMF. Das GMF stellt generiende Komponenten und eine Laufzeitstruktur dar, die für die Entwicklung

grafischer Editoren, basierend auf EMF und GEF, geeignet sind. Das Projekt zielt darauf ab, diese Komponenten und exemplarische Werkzeuge für die Auswahl der Domänenmodelle bereitzustellen, die seine Fähigkeiten illustrieren.

Aufbauend auf den beiden in den vorherigen Abschnitten eingeführten Technologien EMF und GEF, bietet GMF die Möglichkeit, grafische Editoren für EMF Modelle zu generieren. Ein UML Modellierungswerkzeug, einen Workflow-Editor etc. sind einige Beispiele für grafische Editoren. [Eclib]Im Gegensatz zu GEF, lassen sich mittels GMF für ein EMF-Modell mit viel weniger Aufwand grafische Editoren erzeugen. Das liegt daran, dass beim GEF besonders für grafische Darstellungen, verbunden mit einem Modellobjekt, Code geschrieben werden muss[eclic] Im Fall von GMF, wird im Wesentlichen das Domänenmodell auf das Graphical Definition Model abgebildet[Zün]. Das GMF bietet hier Möglichkeiten die Formen einfach auszuwählen.

Von Frederic Plante, IBM (16. Januar 2006) "Das grafische Modellierungsframework (GMF) ist ein neues Eclipse-Projekt mit dem Potential eine Meilenstein-Framework zu werden für die schnelle Entwicklung von standardisierten Eclipse grafischen Modellierungseditoren. GMF ist geteilt in zwei Hauptkomponenten: die Laufzeit und die Werkzeugbereitstellung, die verwendet wird um Editoren zu generieren, welche wiederum in der Lage sind sich auf die Laufzeit auszuwirken"[Eclib].
Im Kapitel GMF wird als erstes über die Konzepte berichtet, wie die Trennung abstrakter und konkreter Syntax. Im Anschluss werden die Aufgaben von GMF beschrieben. Hier werden die Hauptbestandteile zur Erstellung eines grafischen Editors aufgelistet und erklärt. Als weiteren Punkt werden die Vor- und Nachteile von GMF und von GEF miteinander verglichen. Dann wird kurz das Unterstützungswerkzeug, das Dashboard, anhand einer Abbildung erklärt. Schließ werden die einzelnen Schritte im allgemeinen erklärt. Es wird erklärt welche Funktion jeder einzelne Schritt hat. Die einzelnen Schritte wären, Erstellung des Ecore-Modells, grafische Defintion, Werkzeugdefinition und schließlich das Mapping.

4.1 Konzepte

Im folgenden werden die zwei Konzepte der MDA besprochen, die bei GMF eine wichtige Rolle spielen: Die Trennung von konkreter und abstrakter Syntax, sowie die Trennung von Domänen- und Diagrammmodell. Als erstes gibt es Konzepte vom GMF, die zur Trennung abstrakter und konkreter Syntax, sowohl im Editordesign als auch im generierten Editor, verwendet werden[EHMB07]. Unter konkreter Syntax versteht man die Syntax, die in der jeweiligen Sprachumgebung definiert wird. Im Falle des GMFs können es Diagramme, Graphen etc. sein. Im weiteren Verlauf dieser Arbeit wird eine Mindmap als grafischer Editor dargestellt. An diesem Beispiel kann man sehr gut den Unterschied zwischen konkreter und abstrakter Syntax sehen. Der im Beispiel erzeugte Editor, kann als konkrete Syntax einer UML-artigen (gerichtet oder ungerichtet) Darstellung bezeichnet werden[BoHe]. Auf der anderen Seite gibt es die abstrakte Syntax. Darunter kann sich die interne Darstellung von Strukturen verstehen. Diese werden jedoch in der konkreten Syntax definiert. In diesem Fall, des im

späteren Verlauf genannten Beispiels, ist die abstrakte Syntax als eine Auflistung der einzelnen Elementen dargestellt. Das Beispiel Mindmap besteht aus Topics und Subtopics. Dies wird in der unteren Abbildung als abstrakte Syntax in Form eines Klassendiagramms dargestellt. Dazu gibt es zwei konkrete Syntax: Die Baumansicht und der Editor, der später noch im Beispiel generiert werden wird. Die einzelnen Elemente in der Baumstruktur Eigenschaften wie Name des Elements[BoHe].

Syntax 2.png

Abbildung 1. Beispiel für eine abstrakte Syntax UML-Klassendiagramm

Des Weiteren ist die Trennung von Domänen- und Diagrammmodelle ein weiteres Konzept von GMF[EHMB07]. Unter Domänenmodellen, bzw. auf englisch 'Domain Model', kann man eine Beschreibung vieler Elemente sehen. Das 'Domain Model' gibt hier eine Beschreibung der eigentlichen Beziehung der Objekte im System. Objekte können physikalische Objekte und abstrakte Konzepte

sein[Comp]. Diagrammmodelle definieren Diagrammdarstellungen. Es ist auch möglich Darstellungen verschiedener Domänenmodelle zu erzeugen wie Klassendiagramme für EMF-Modelle.

4.2 Aufgabe

Es gibt drei Modelle, die das GMF benutzt um ein mit EMF-Mitteln beschriebenes, domänenspezifisches Metamodell auf konkrete Syntax bzw. entsprechenden grafischen Editor abzubilden[Kief07] In der unteren Abbildung nochmal das Dashboard, das die drei wichtigsten Elemente(gmfgraph, gmftool und gmfmap) zur Erstellung des grafischen Editors aufzeigt:

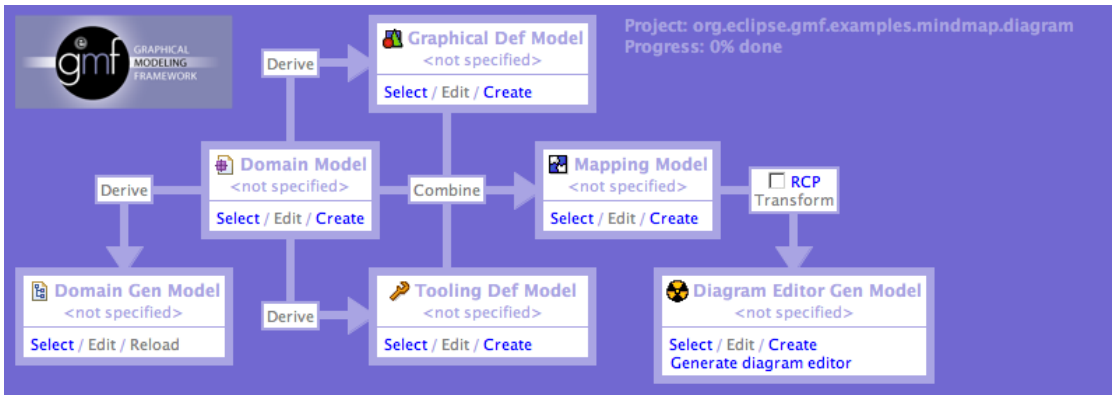


Abbildung 2. Das Dashboard

- gmfgraph-Modell: Dieses Modell beschreibt hier die grafischen Darstellungsformen des Editors. Formen sind: Ovale, Kreise, Rechtecke, Linien, visuelle Gestaltungsmittel, wie Hintergrundfarbe, Linienfarbe, Linienstärke oder Linienart und topologische Gestaltungselemente, wie Komposition oder Enthaltensein[Kief07].
- gmftool-Modell: Hier wird die Werkzeugleiste beschrieben. Die einzelne Formen können selbst definiert werden[Kief07].
- gmfmap-Modell: Dieses Modell ist hier sehr wichtig, da dieses Modell die Elemente des gmfgraph-Modell und gmftool-Modell mit den Elementen des Metamodells verbunden werden. Dadurch werden die einzelne Objekte der Werkzeugleiste zu Elementen der Modellierung definiert und Beziehung zu Instanzen des Metamodells festgelegt[Kief07].

4.3 Prozess zur Generierung eines Editors

In der obigen Abbildung ist das GMF-Dashboard dargestellt. Das Dashboard ist eine großartige Unterstützung zur Erstellung grafischer Editoren basierend

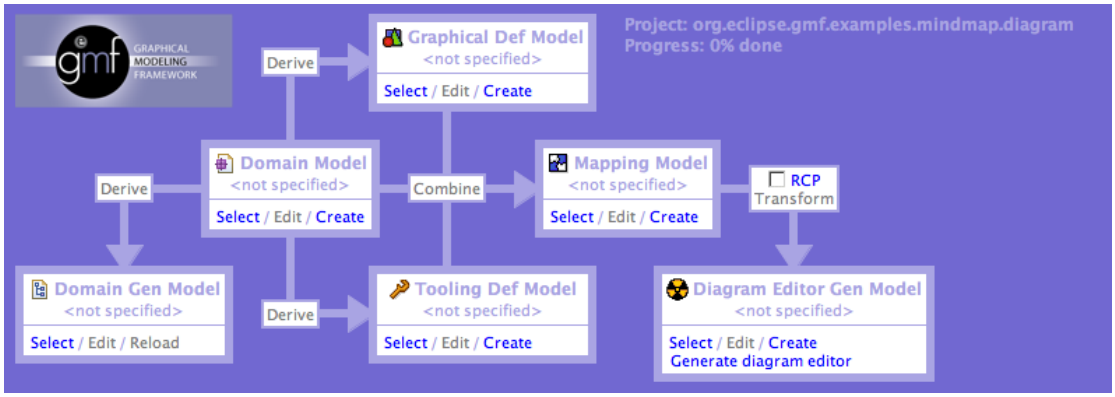


Abbildung 3. Dashboard aus Eclipse

auf EMF-Modellen. Dieses so genannte Werkzeug zeigt einem, welche Elemente noch nötig sind, die zur Erstellung des grafischen Editors notwendig sind. Außerdem gibt es eine Prozentanzeige, die den Fortschritt des Prozesses darstellt. Das Dashboard zeigt, dass der erste Schritt, die Erstellung bzw. das Laden der Ecore-Datei, ist. Dann wird die genmodel-Datei daraus erzeugt. Die ecore- und genmodel-Dateien sind EMF-Elemente. Auf diese zwei Elemente werden später die GMF Elemente gemappt. Daraus wird dann unter GMF der grafische Editor erstellt. Die genmodel-Datei beinhaltet den Kern der Generierungsfunktion von GMF. Java-Quelle können aus dieser erzeugt werden, die letztendlich das ecore-Metamodell repräsentiert. Sie kann Anweisungen für die Transformation beinhalten. Die Anweisungen werden getrennt vom Modell abgespeichert[Holt08].

Erstellung des Ecore-Models und anschließende Erstellung des Genmodels Der erste Schritt zur Erstellung eines grafischen Editors im GMF geht über das EMF. Man beginnt mit dem Metamodell der domänenspezifischen Sprache. Es beschreibt, wie die Modelle beschaffen sind(welche Elemente sie enthalten, wie diese in Beziehung stehen, evtl. weitere Gültigkeitseinschränkungen), die später im Editor erstellt und bearbeitet werden könne. Letztendlich heißt das, dass diese ecore- und genmodel-Dateien Metadaten sind, auf die schließlich die GMF Elemente gemappt werden.

Der Anfang der Erstellung des auf GMF-basierenden grafischen Editors ist ein EMF-Projekt. Das EMF-Projekt stellt diese Metadaten zur Verfügung bzw. diese werden im EMF-Projekt erzeugt. Eines ist dennoch zu beachten. Die Einstellungen im Eclipse müssen richtig sein, d.h. besonders das Compiler-Level sollte auf das entsprechenden Level gesetzt sein. Der Unterschied liegt darin, dass die Java-Codes unterschiedlich aussehen bzw. unterschiedlich programmiert werden. Hier ist ein Beispiel das den Unterschied verdeutlicht, wie die Code beim jeweiligen Compiler-Level aussehen:

Library.java (Java 1.4)

```
package org.eclipse.example.library;
import java.util.List;

/**
 * @model
 */
public interface Library
{
  /**
   * @model
   */
  String getName();

  /**
   * @model type="Writer" containment="true"
   */
  List getWriters();

  /**
   * @model type="Book" containment="true"
   */
  List getBooks();
}
```

Library.java (Java 5.0)

```
package org.eclipse.example.library;
import java.util.List;

/**
 * @model
 */
public interface Library
{
  /**
   * @model
   */
  String getName();

  /**
   * @model containment="true"
   */
  List<Writer> getWriters();

  /**
   * @model containment="true"
   */
  List<Book> getBooks();
}
```

Abbildung 4. Unterschied zwischen der Programmierung bei unterschiedlichen Compiler-Level[Ecli07]

Nachdem die ecore- und genmodel-Dateien erstellt bzw. die Metadaten erstellt wurden, hat hier Eclipse einen sehr guten Assistenten, der die Erstellung des EMF-Modells sehr vereinfacht. Der Assistent bietet wie bei der Erstellung eines neuen Projektes auch die Option ein EMF-Modell zu erstellen.

Grafische Definition In diesem Abschnitt geht es darum Formen zu erstellen, die später die einzelnen Komponenten des EMF-Modells repräsentieren. Dies ist letztendlich eine Liste von Formen, die in der gmfgraph-Datei beschrieben werden. Diese werden schließlich benutzt um die Klasse aus dem Domänenmodell im Diagramm anzuzeigen[Rich07] Über das Dashboard ist sehr schnell möglich die gmfgraph-Datei zu erstellen. Diese beinhaltet letztendlich alle Informationen bezüglich der Formen. Die nächste Abbildung zeigt einige der möglichen Formen. Es können beispielsweise Polygone, Rechtecke, Ellipsen, etc. erstellt werden(s. Abbildung 6).Des Weiteren sind in der Abbildung einige Elemente, wie Beschreibung einer Form, die mit einem bestimmten Modellobjekt in Verbindung steht(s. Abbildung 6). Durch das GMF ist es sehr einfach grafische Formen zu erstellen, die später im Diagramm dargestellt werden.

Nicht so einfach wäre der Weg über GEF. Hier wäre es notwendig gewesen für die einzelnen Formen jeweils Code zu schreiben. Jeder dieser Codes hätte die Eigenschaften des einzelnen Elements beschreiben müssen. Auch wenn dies ein mühsamer Weg ist, so bietet es viel mehr Gestaltungsmöglichkeiten und der Editor ist nicht, so wie bei GMF, auf EMF-Modelle angewiesen. In der unteren Abbildung ist ein solcher Code dargestellt. Dieser Code repräsentiert eine Person mit Vornamen und Nachnamen.[Rich07].

Figure to represent a person

```
public class PersonFigure extends Figure implements IPersonFigure{
   private Label labName;
   private Label labSurname;
   private IFigure contentPane;
   public PersonFigure(){
           setLayoutManager(new ToolbarLayout());
           labName = new Label();
           add(labName);
           labSurname = new Label();
           add(labSurname);
           contentPane = new Figure();
           contentPane.setLayoutManager(new ToolbarLayout());
           contentPane.setBorder(new TitleBorder("fruits list :"));
           add(contentPane);
   }
   public void setName(String name){
           labName.setText(name);
   }
   public void setSurname(String surname){
           labSurname.setText(surname);
   }
   public IFigure getContentPane(){
           return contentPane;
   }
}
```

Abbildung 5. Code in GEF. Repräsentation einer Person mit Vor- und Nachname[eclic]

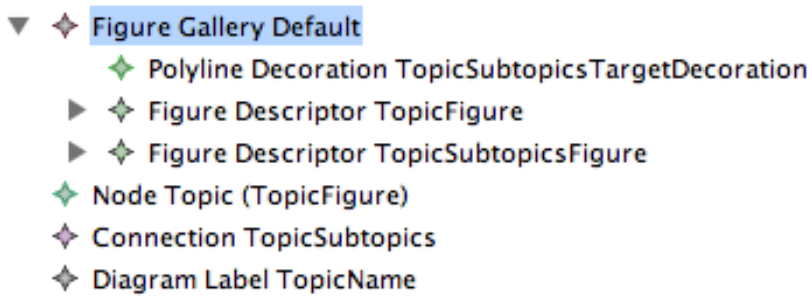


Abbildung 6. Auflistung einiger Formen in der grafischen Defintion in Eclipse

Werkzeugdefintion Im Dashboard ist jetzt erkennbar welcher Schritt der nächste ist. Er besteht letztendlich darin, die gmftool-Datei zu generieren. Wie immer ist es über das Dashboard sehr einfach diese gmftool-Datei zu erstellen. Diese Datei, wie auch die vorherigen erstellten Dateien, wird im Explorer vom Eclipse angezeigt. Wie auch bei der grafischen Definition, in der es über die Pop-Ups möglich ist verschiedene Formen zu erstellen, ist es hier in der Werkzeugdefinition nicht anders. Auch hier ist es über die Pop-Ups möglich Werkzeuge zu erstellen. Werkzeuge sind beispielsweise Paletten, Menüs, etc. [Eclib]

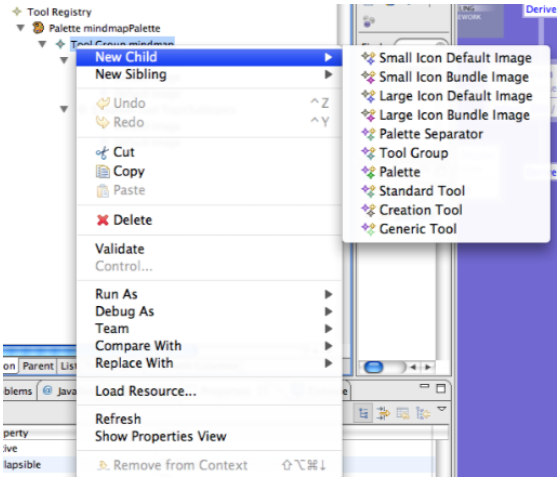


Abbildung 7. Abbildung aus Eclipse: Mehrere Auswahlmöglichkeiten der Werkzeuge

Mapping und Generierung eines Editors Bei der letzten Erstellung der GMF-Datei geht es um das Mapping. Mapping ist nichts anderes als die grafische Definition und die Werkzeugpaletten des EMF-Modells (ecore-Datei) zusammen zu bringen. Um dies zu bewerkstelligen, müssen dem Mapping-Modell die anderen Modelle bekannt gemacht werden. Hierzu werden die drei Dateien ecore, gmfgraph und gmftool benötigt. Diese Zusammenführung wird über das GMF-Dashboard ermöglicht. Sobald alle drei Dateien in das Mapping geladen sind, gibt es die Möglichkeit einige Modifikationen vorzunehmen. Diese Modifikationen sind zum Beispiel das Weglassen oder Definieren der einzelnen Elemente des Modells. Beim Definieren der einzelnen Elementen geht es hauptsächlich darum, welche Elemente (Nodes) und welche Links sind. Sind alle Konfigurationen beendet, so wird die gmfmap-Datei erzeugt. Aus dieser Datei wird schließlich über ein Pop-up das Generator-Modell erzeugt, das als gmfgen-Datei im Kontextmenü abgelegt ist.

Der letzte Schritt ist die Generierung des grafischen Editors. Im letzten Bereich des Dashboards wird nun die gmfgen-Datei ausgewählt und geladen. Nun wird das neue Plugin erzeugt, das wiederrum im Kontextmenü abgelegt wird. Dieses Plugin wird dann in in der Runtime-Workbench von Eclipse gestartet. Es ist nun benutzbar. In der neuen Runtime-Workbench gibt es die Möglichkeit das neue Plugin zu verwenden. Durch das Initialisieren der Plugin-Datei wird der eigentliche grafische Editor gestartet. Es existieren jetzt zwei konkrete Sichten. Im Falle eines UML-Diagramms wird durch das Erzeugen neuer UML-Elemente in der konkreten Syntax auch in der zweiten Sicht (Baumsicht) verändert.[KEVH]

4.4 Vorteile und Nachteile von GMF gegenüber GEF

Dies hängt davon ab, was man erreichen möchte. Für vollausgestattete grafische Editoren, die die hohen Fähigkeiten des GEFs benötigen und mit einem EMF Domänenmodell arbeiten, ist GMF eine gute Alternative. Dennoch könnte man an leichten Implementierungen, schreibgeschützen Visualisierungen, etc. interessiert sein.[Eclib] Wie in der Einführung von GMF, hat es den Vorteil einfach und schnell einen grafischen Editor für ein EMF-Modell zu generieren. Der Nachteil bei GEF ist, dass vieles selbsterstellt werden muss. Im Falle der Darstellungen von Modellobjekten im View, ist es notwendig Code dafür zu schreiben. Weiter ist es notwendig Codes in den EditPart zu schreiben, um den View, der mit dem Modellobjekten verbunden ist, zu instanziieren und es müssen Codes erstellt werden, um den View anzuzeigen, der mit den Daten aus dem Modellobjekt gefüllt wird. Der EditPart bei GEF, ist eine Instanz, die jedes Modellobjekt mit ihrem View verbindet[eclic]. Im Gegensatz dazu wird im GMF dieser Vorgang automatisiert. Es ist jedoch weiterhin möglich einige Modifikationen durchzuführen.

5 Beispiel - Mindmap

Hier wird anhand eines Beispiels, die Erstellung eines grafischen Editors deutlicher gemacht.

5.1 Erstellung des Domänen- und Domänen Gen Modells

Über das Dashboard wird die ecore-Datei geladen. Anschließend wird über den selben Weg auch die genmodel-Datei erzeugt[Eclia]. Wie der unteren Abbildung zu entnehmen ist (s. Abbildung 8), wird das Domänenmodell baumartig dargestellt. Des Weiteren erkennt man den Knoten 'Topic', der wiederrum weitere Elemente beinhaltet. Subtopics ist eine Assoziation, die angibt, dass ein Topic mit mehreren anderen Topics in einer Relation stehen kann. Diese gibt auch an, dass dieses Topic das übergeordnete Topic ist. Es können keine bis beliebig viele Topics subtopics für ein Topic sein.

Sobald die genmodel-Datei erstellt wurde, geht es nun zur grafischen Definition. Im Assistenten gibt es eine Möglichkeit eines von den Diagramm-Elementen

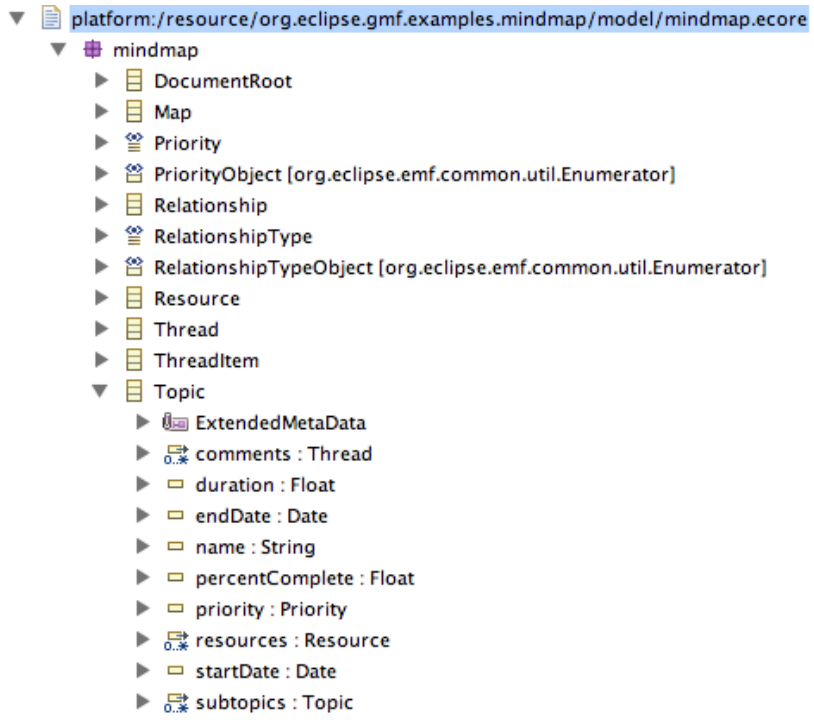


Abbildung 8. Baumstruktur des Domänenmodells

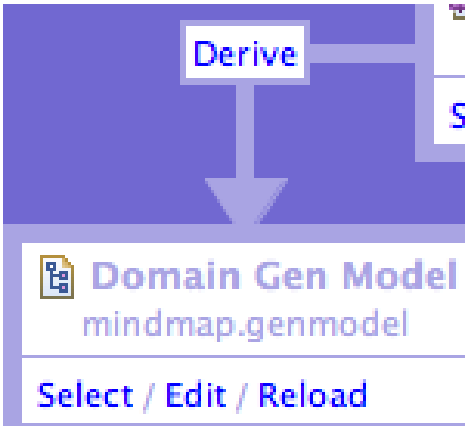


Abbildung 9. Teil vom Dashboard - Abschnitt zur Erstellung der genmodel-Datei über 'Derive'

aus dem Ecore Domaine Model auszuwählen. Wichtig ist, dass 'Map' als Diagram Element ausgewählt wurde[Eclia]. Denn 'Map' enthält alle Informationen, wie die einzelnen Mindmapelemente benutzt und miteinander verbunden werden. Auch in den nächsten Schritten der Generierung der grafischen Elemente sowie der tooling Elemente, muss beachtet werden welches Diagramm-Element ausgewählt wird, in diesem Beispiel 'Map'.

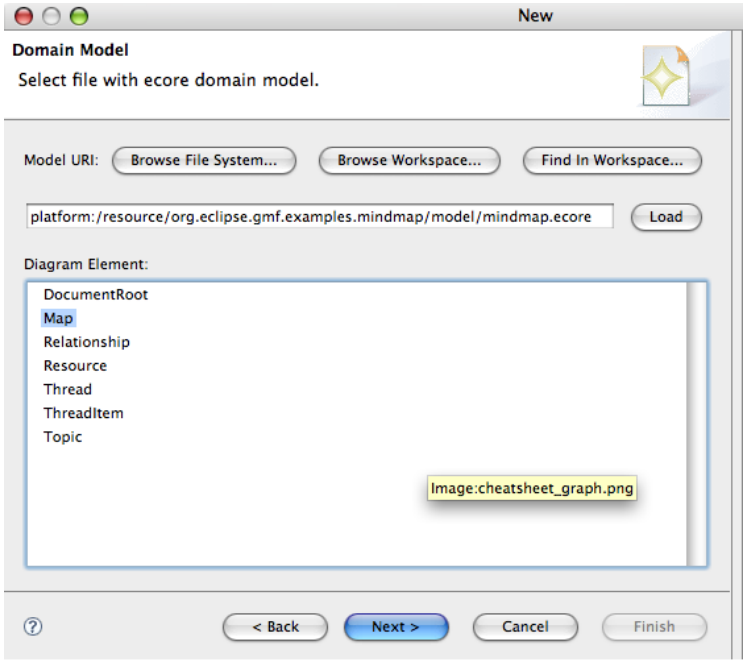


Abbildung 10. Auswahl eines Diagramm-Elements

Im nächsten Schritt gibt es die Möglichkeit die Anzahl der Elemente, die im grafischen Editor benutzbar sind, zu verändern. Hier werden minimale Elemente ausgewählt: Thema, Unterthema und Namen. Es gibt viele weitere Elemente, die dann später im grafischen Editor benutzbar sind, wie bei der Erstellung eines Thema mit anschließender Möglichkeit diesem Objekt einen Kommentar hinzuzufügen[Eclia].

Anschließend, um die ausgewählten Elemente benutzen zu können, müssen auch die geeigneten Werkzeuge verfügbar sein. In diesem Beispiel werden die Werkzeuge fürs Thema und Unterthema ausgewählt[Eclia].

Sind beide Schritte ausgeführt folgt jetzt das Mapping. Die drei Dateien ecore, gmfgraph und gmftool werden jetzt mit dem Modell zusammengeführt. Auch hier gibt es die Möglichkeit das Modell einigen Modifikationen zu unterziehen. In diesem Beispiel ist Thema ein 'Node' und das Unternehma gehört

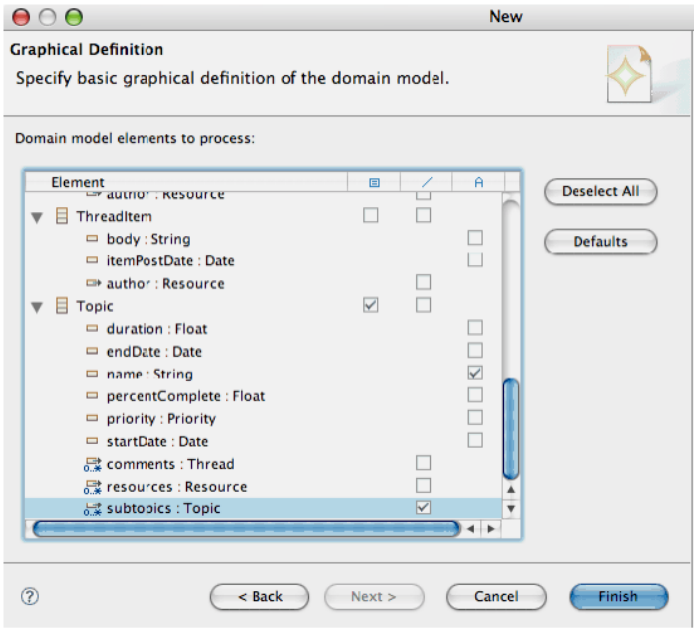


Abbildung 11. Auswahl an Werkzeugen, die später im grafischen Editor sichtbar und benutztbar sind

zur Kategorie 'Links'. Es ist auch möglich einige Elemente zu entfernen. Hier bietet der Assistent von Eclipse die Funktionen 'Remove', 'As Link' und 'As Node', mit denen die jeweiligen Elemente verschoben werden können. Nach der Erstellung der gmfmap-Datei, ist es über die Eigenschaften möglich sichtbar zu machen, welche Elemente wie verbunden werden können und welchen Kategorien sie angehören[Eclia].

Nachdem das Mapping durchgeführt wurde, geht es jetzt zur Erstellung des Generator Modells. Über ein Popup-Menü gibt es die Option 'Create Generator Model'. Diese Option erstellt die notwendige gmfgen-Datei, die im letzten Schritt zur Erstellung des Editors gebraucht wird. Im letzten Schritt im Dashboard wird nur noch die gmfgen-Datei geladen[Eclia].

Abbildung 12. Hier wird die gmfgen-Datei geladen

Schließlich wird jetzt der Diagramm-Editor generiert. Im Kontextmenü erscheint ein neuer Ordner mit der Endung 'diagram'. Über diesen neuen Ordner wird eine neue Instanz von Eclipse gestartet [Eclia].

In der neuen Workbench von Eclipse ist es nun möglich über ein neues Projekt das neue Mindmap-Plugin zu erstellen. Die Mindmap-Datei enthält das Modell selbst und kann mit dem Standardbaumeditor oder mit unserem neu generierten Editor geöffnet werden. Nach der Initialisierung dieser Datei wird schließlich der eigentliche grafische Editor dargestellt. Jetzt ist es möglich eine Mindmap zu erstellen[Eclia].

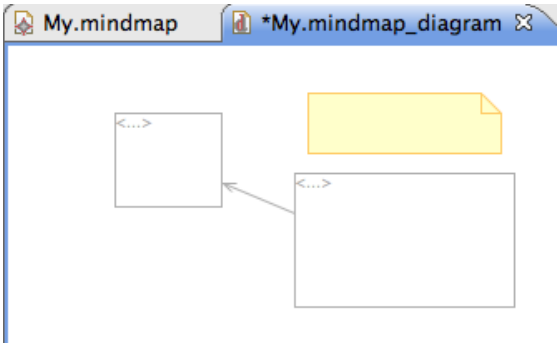


Abbildung 13. Beispielmindmap

Diese Sicht zeigt die konkrete Syntax. Wird jetzt die ursprüngliche Datei geöffnet und es wurden schon einige Elemente im Mindmap erzeugt, so werden auch genau diese Elemente baumartig dargestellt.

6 Zusammenfassung

Das Definieren der Werkzeugelemente, der grafischen Elemente und das Mapping Element sind nötig um grafische Editoren für EMF-Modelle zu erstellen. In der Werkzeugdefintion werden Elemente wie Palette oder Menüs erstellt, die später im Editor wiederzufinden sind. Im Schritt der Defintion der grafischen Elemente gibt man den einzelnen Objektelementen eine Form. Zum Beispiel ein Rechteck, das in einem Mindmap eine klare Funktion hat. Das Mapping vereinigt das Modell, die Werkzeuge und die grafischen Elemente miteinander. Die Prozentanzeige im Dashboard gibt an, wieviel man noch braucht bis zur Fertigstellung des grafischen Editors. Das GMF erspart dem Entwickler viele aufwendige Arbeitsschritte bei der Erstellung grafischer Editoren. Leider erstellt das GMF grafische Editoren nur für EMF-Modelle. Das GEF ist hier anders: Es bietet die Möglichkeit für viele andere Modelle (nicht nur für EMF-Modelle) grafische Editoren zu erstellen. Das heitßt wiederum, dass vieles nicht standardisiert ist. Viele aufwändige Schritte sind nötig - wie das Schreiben von Code - um einen grafischen

Editor zu erstellen. In meinen Augen hat GMF noch viel Potential, aber im aktuellen Entwicklungsstand ist es nur zum "Ausprobieren" da. Die Entwickler haben hier die Möglichkeit schnell einen grafischen Editor zu erstellen, um das Modell zu testen.

Literatur

Anis06. Chris Aniszczyk. Learn Eclipse GMF in 15 minutes, September 2006. http://www-128.ibm.com/developerworks/opensource/library/os-eclgmf/#N10129.

BoHe. Harold Boley und Michael Herfert. Abstrakte und konkrete Syntax. http://www.dfki.uni-kl.de~vega/relfun+/prosyn/subsection3_2_1.html.

Comp. Computertechdoc.org. The Computer Technology Documentation Project-UML Domain Model. Zuletzt zugegriffen am 28.01.2009. http://www.comptechdoc.org/independent/uml/begin/umldomainmodel.html.

Eclia. Eclipse.org. GMF-Tutorial. Zuletzt zugegriffen am 28.01.2009. http://wiki.eclipse.org/index.php/GMF_Tutorial.

Eclib. Eclipse.org. Introducing the Runtime GMF. Zuletzt zugegriffen am 28.01.2009. http://www.eclipse.org/articles/Article-Introducing-GMF/article.html.

eclic. eclipseWiki. GEF-Description. Zuletzt zugegriffen am 29. Januar 2009. http://eclipsewiki.editme.com/GefDescription.

Ecli07. Eclipse.org. *Generating an EMF Model*. Zuetzt zugegriffen am 28.01.2009, Juni 2007. http://help.eclipse.org/ganymede/index.jsp?topic=/org.eclipse.emf.doc/references/overview/EMF.Edit.html.

EHMB07. Claudia Ermel, Frank Hermann, Tony Modica und Enrico Biermann. Das Eclipse Projekt - Graphical Modeling Framework. Zuletzt zugegriffen am 28.01.2009, Mai 2007. http://tfs.cs.tu-berlin.de/vila/www_ss07/folien/16-GMF-4.pdf.

Holt08. Benjamin Holtz. Diplomarbeit über die Modellierung einer Systeminstallation. Diplomarbeit, Technische Universität Ilmenau, 20. März 2008. http://www.db-thueringen.de/servlets/DerivateServlet/Derivate-14311/Diplomarbeit.pdf.

Kepl05. Johannes Kepler. Implementierung des Model-View-Controller Paradigmas für das WeLearn-System (Web Environment for Learning). Diplomarbeit, Universität Linz, Februar 2005. http://www.fim.uni-linz.ac.at/diplomarbeiten/Diplomarbeit_stoiber/Dietmar_Stoiber_-_Model-View-Controller_Paradigma.pdf.

KEVH. Bernd Kolb, Sven Efftinge, Markus Voelter und Arno Haase. Graphical Modeling Framework. Zuletzt zugegriffen am 28.01.2009. http://www.voelter.de/data/articles/ix-gmf2.pdf.

Kief07. Johann Kiefel. Entwicklung eines auf Eclipse und GMF basierenden, graphischen Modellierungswerkzeugs fuer das Palladio Komponentenmodell. Diplomarbeit, Universitaet Karlsruhe, 2007.

MDGW+04. Bill Moore, David Dean, Anna Gerber, Gunnar Wagenknecht und Philippe Vanderheyden. *Eclipse Development using the Graphical Editing Framework and the Eclipse Modeling Framework*. Redbooks. Februar 2004.

Rich07. Jeff Richley. GMF: Beyond Wizards, November 2007. http://www.onjava.com/pub/a/onjava/2007/07/11/gmf-beyond-the-wizards.html?page=1.

Zün. Albert Zündorf. Software-Engineering-2-Konstruktion-interaktiver_(CASE)-Tools. Zuletzt zugegriffen am 28.01.2009. http://www.se.eecs.uni-kassel.de/se/fileadmin/se/courses/SE2WS0809/slides/SE_2_ToolConstructionWS0809.ppt.

Performance Patterns

Christian Baumgart

Betreuerin: Lucia Kapova

Zusammenfassung Die Performance von komplexen Software-Systemen ist ein Schlüsselaspekt für das heutige Software Engineering. Was bedeutet Performance? Wie, wann und womit kann man die Performance in einem Softwaresystem verbessern? Dieses Paper zeigt Wege auf, Performance schon frühzeitig in Entwurfsphasen zu berücksichtigen und stellt eine Auswahl von Design Patterns vor, die besonders für die Entwicklung von performance-effizienten Software-Architekturen geeignet sind.

1 Einleitung

Performance spielt heute in fast jedem komplexen Softwareprojekt eine Schlüsselrolle und hat direkten Einfluss auf Qualität, Benutzerfreundlichkeit und Zuverlässigkeit eines Systems.

Als typische nichtfunktionale Anforderung wird Performance allerdings nicht immer in Anforderungsdokumenten definiert und ist häufig impliziter Natur. Dies führt bei vielen Entwicklern zu einer Vernachlässigung von Performance-Zielen in Entwurfsphasen. Häufig wird Performance-Verbesserung auf die Zeit während oder nach der Implementierung verschoben [SmWi02]. Dieser *Fix-it-Later*-Ansatz führt allerdings oft zu hohen Kosten, da Bottlenecks durch eine schlechte (initiale) Software-Architektur begründet sein können, und durch Refactoring nur langsam und aufwendig ausgebessert werden können.

Die Patterns in dieser Ausarbeitung lassen sich auf verschiedene häufig auftauchende Probleme im Entwurf heutiger Softwareprojekte anwenden. Kapitel 1 gibt einen Einstieg in die Themen *Performance*, *Design Patterns* und *Software Performance Engineering*. In Kapitel 2 werden allgemeinere Problemstellungen, die in vielen Projekten auftauchen, behandelt. Spezielle Patterns für *verteilte Systeme* bzw. *beschränkten Zugriff* auf Ressourcen werden in den Kapiteln 3 und 4 vorgestellt.

1.1 Design Patterns

Software *Design Patterns* (Entwurfsmuster) sind Lösungsmuster für häufige Probleme der Softwareentwicklung. Sie kapseln Problemstellungen und Lösungen, die sich über längere Zeiträume als besonders nützlich erwiesen haben.

Grundsätzlich besteht ein Software Design Pattern aus 3 Teilen:

- **Name**: Ein eindeutige Name ermöglicht verschiedenen Softwareentwicklern eine eindeutige Zuordnung der Modellierung zu Design Patterns und unterstützt so die Kommunikation. Pattern-Namen in Codekommentaren oder UML-Diagrammen erleichtern das Verständnis von fremdem Code.

- **Problem und Kontext**: Beschreibung des Problems und Einordnung der Anwendbarkeit der Patterns in einen bestimmten Kontext.
- **Lösung**: Lösungsbeschreibungen für das Problem. Je nach Abstraktionsebene des Patterns werden textuelle Ausführungen oder Diagramme verwendet.

Ursprünglich wurde der Begriff des *Pattern* Ende der 70er-Jahre durch den Architekten Christopher Alexander eingeführt:

> "Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice." [Alex77]

Connie Smith et. al [SmWi02], weithin bekannt unter der *Gang of Four* (GoF), nahmen die Ansätze von Alexander auf und übertrugen sie auf das Software Engineering. Das 1994 erschienende Buch *Design Patterns: Elements of Reusable Object-Oriented Software* [SmWi02] enthielt eine Sammlung von Design Patterns und gilt bis heute als eines der einflussreichsten Standardwerke moderner Softwareentwicklung.

In den letzten Jahren sind im Software Engineering ausgehend von den GoF Kern-Patterns viele zusätzliche Patterns hinzugekommen. Diese Ausarbeitung stellt eine Auswahl von Patterns zusammen, die sich besonders gut auf performance-kritische Problemstellungen anwenden lassen.

1.2 Performance

"*Performance* ist ein Indikator der anzeigt, wie gut ein Softwaresystem oder eine Komponente seine Anforderungen für Rechtzeitigkeit erfüllt. Dabei wird Rechtzeitigkeit gemessen in *Anwortzeit* und *Durchsatz*. Die Antwortzeit ist die Zeit, die für die Beantwortung eines Requests benötigt wird. [...] Der Durchsatz eines Systems ist die Anzahl der Requests, die in einer gegebenen Zeitintervall bearbeitet werden können."[SmWi02].

Das Ziel bei der Verbesserung von Performance-QOS (*Quality of Service*) ist also die Einschätzung, Messung und Verbesserung der Rechtzeitigkeit. Prinzipiell ergeben sich zwei orthogonale Ansätze: Verkürzung des *Antwortzeitverhaltens* eines gegebenen Systems (auch als *Tuning* bekannt) oder Erweiterung der Systems(*Skalierung*).

1.3 Software Performance Engineering (SPE)

In den achtziger Jahren prägten Connie Smith et al. [SmWi02] den Begriff des *Software Performance Engineering* (SPE) und versuchten, Performance-Anforderungen in einen ganzheitlichen Software-Entwicklungs-Prozess zu integrieren.

Der SPE-Prozess umfasst dabei

- Prinzipien für die Entwicklung zeitkritischer Software

- Performance Patterns und Antipatterns(performance-ineffiziente Patterns) für performance-orientertes Design
- Techniken für die Entwicklung von Performance-Zielen und Anforderungen
- Techniken für das Sammeln von Daten für Performance-Analysen (Monitoring)
- Richtlinien für die Analysetypen für jeden Abschnitt des Entwicklungsprozesses

Software Performance Engineering ist ein eher linearer Entwicklungsprozess, der die Modellierung von Performance-Anforderungen in einem Softwareprojekt *von Anfang an* einführt und so dem *Fix-it-Later*-Problem entgehen will. Dieser Ansatz ist allerdings problematisch, da die Beurteilung von performancekritischen Bereichen eines Systems oft nicht von Anfang an möglich und aufgrund der Komplexität von großen Projekten schwer überschaubar ist. Allerdings ist die Einbindung von SPE z.B. in Planungsphasen evolutionärer bzw. inkrementeller Entwicklungsmodelle sehr gut möglich.

Abbildung 1 zeigt einen Überblick über den Gesamtprozess.

2 Allgemeine Performance Patterns

2.1 Lazy Load

Problem In objektorientierten Systemen wird der Zugriff auf externe Ressourcen üblicherweise in Klassen abstrahiert. Object-Relationship-Mapping Frameworks (ORM) etwa bilden Tabellenzeilen in Datenbanken auf (Entitäts-)Klassen und Spalten auf Klassenfelder ab. Die Felder werden dann bei der Initialisierung einer Entität mit Daten aus der Datenbank gefüllt.

Als Benutzer einer solchen Architektur ist man aber häufig garnicht an allen Feldern einer Entität interessiert. Die überflüssig initialisierten Felder verursachen überflüssigen Speicher- und Rechenzeit-Overhead.

Lösung *Lazy Load* [Fowl02] verschiebt den Zugriff auf externen Ressourcen auf einen möglichsten späten Zeitpunkt. Bei der Initialisierung einer Entität werden die Felder leer initialisiert. Der echte Zugriff auf eine externe Ressource erfolgt erst dann, wenn ein Benutzer der Entitär einen Feld-Accessor aufruft. Wird der Accessor in der Programmausführung nicht aufgerufen, findet kein (womöglich langsamer bzw. speicherintensiver) Zugriff auf die externe Ressource statt. So wird das Antwortzeitverhalten eines Systems verbessert.

Beispielimplementierung: Lazy Initialization Eine einfache konkrete Implementierung von Lazy Load ist *Lazy Initialization*: Die ressource-verwaltende Entität (hier BigDataEntity wird zunächst mit leeren Feldern initialisiert. Sobald der erste Client auf die Ressource zugreifen möchte, lädt die Entität die Ressource (lazy) und belässt sie im Speicher. [Fowl02]

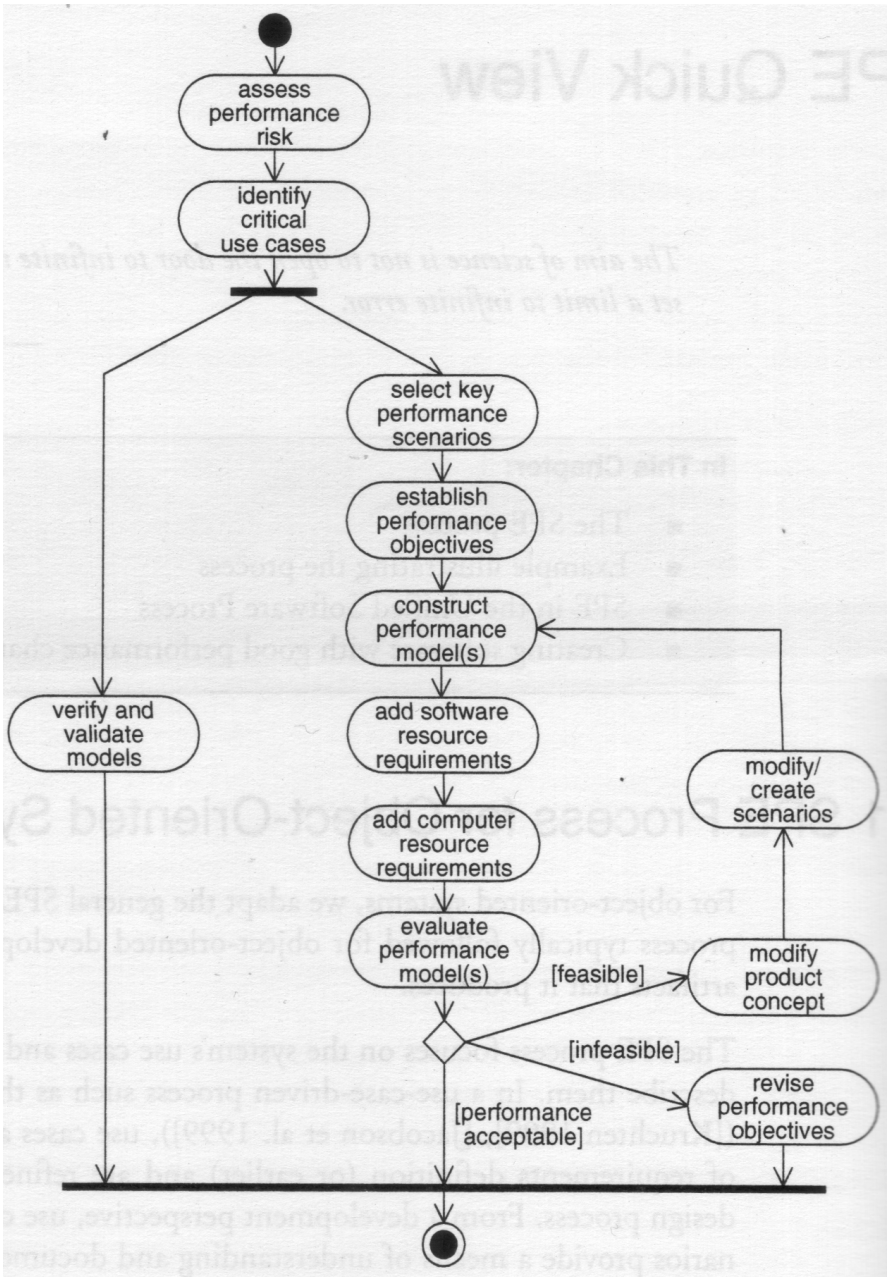


Abbildung 1. Überblick: Software Performance Engineering in objektorienten Systemen [SmWi02]

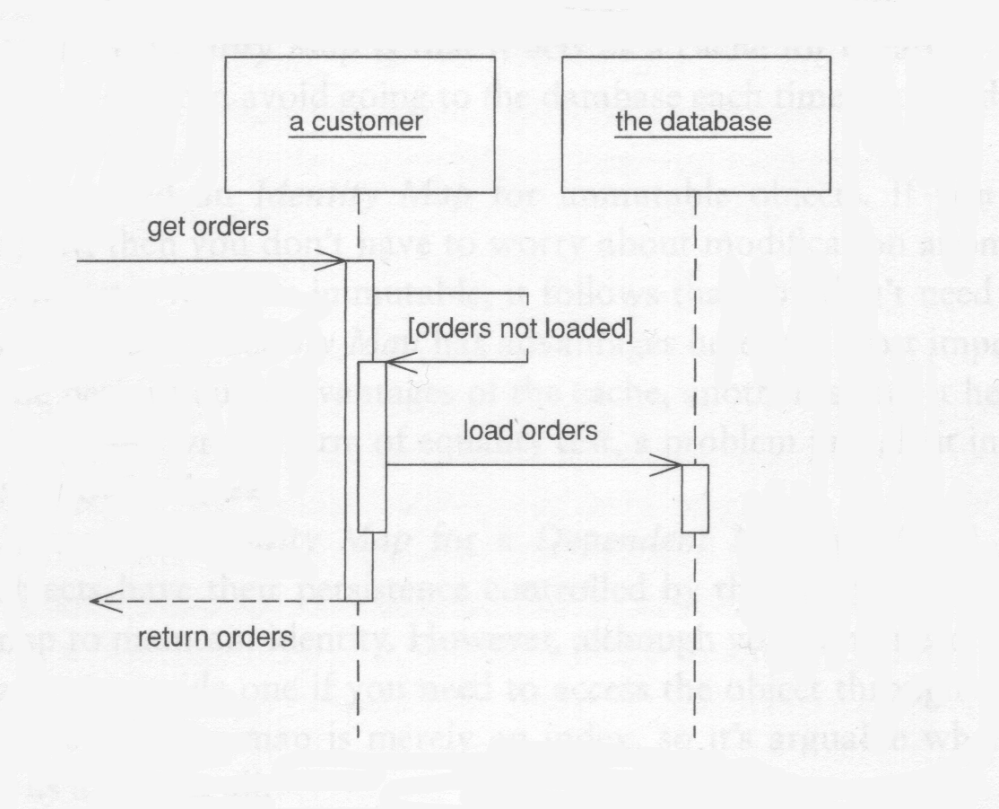


Abbildung 2. Lazy Load einer Customer-Entität aus einer Datenbank [Fowl02]

```
class BigDataEntity {
  // ...
  private Data data = null;

  public Data getData() {
    if(data == null)
      data = doSomeTimeConsumingResourceLoading();

    return data;
  }
}
```

Das zeit- und speicheraufwendige Laden der Ressource `data` wird erst dann ausgeführt, falls ein Benutzer die `getData()` Methode aufruft. Die essentielle Performanceverbesserung tritt genau dann auf, wenn die Ressource `data` in mindestens einer Programmausführung nicht benötigt wird.

2.2 Fast Path

Problem Ein komplexes Softwaresystem kann eine Vielzahl unterschiedlicher Features aufweisen. Die *80-20-Regel*(auch: Pareto-Prinzip) lässt sich auf dieses Szenario übertragen: 80 Prozent der Ausführungszeit wird durch die Ausführung von 20 Prozent der Features in Anspruch genommen. [Meye05] Letztere bilden den *dominanten Workload* [SmWi02] des Softwaresystems.

So bietet ein typischer Bankautomat beispielsweise Funktionen für Überweisungen, Daueraufträge, Kontoauszüge, Einzahlungen, Auszahlung etc. Während einige erfahrene Benutzer alle Funktionen am Bankautomat benutzen, wird man feststellen können, dass Auszahlungen von den Beträgen EUR 20, EUR 50 und EUR 100 die mit Abstand am häufigsten ausgeführte Funktion ist.

Lösung Das *Fast Path*-Pattern konzentriert sich auf den dominanten Workload eines Softwaresystems und bietet einen *schnellen Pfad* für diese dominant genutzten Features an. Es kaskadiert dabei eine Folge einzelner Funktionalitäten zu einem übergeordneten Feature und optimiert dessen Antwortzeitverhalten.

So könnte der Bankautomat aus dem oberen Beispiel Schnellwahltasten für die Auszahlung von 20, 50 und 100 Euro anbieten. Die Performanceverbesserung ergibt sich dann durch die kürzere Zeit, die für den Vorgang einer Auszahlung benötigt wird.

Fast Path lässt sich oftmals durch ist *Caching* (siehe auch Abschnitt 4.2) realisieren: Stellt man fest, dass einige Ressourcen in einem System häufiger benutzt werden als andere, könnte es sich lohnen, diese Ressources z.B. auf schnelleren Speicher zu duplizieren. Durch diesen Fast Path wird die Zugriffszeit des Gesamtsystems gesenkt und die Performance gesteigert. [SmWi02]

2.3 First Things First

Problem Bei temporären Überlastungen der Infrastruktur können die Performance-Anforderungen einiger Tasks (Arbeitseinheiten) nicht mehr oder nur deutlich verzögert erfüllt werden. Dabei haben einige Tasks eine höhere (wirtschaftliche oder funktionale) Bedeutung als andere und sollten bevorzugt ausgeführt werden. Zum Beispiel ist in einem Webshop die Verarbeitung von Bestellungsanfragen bei hoher Auslastung wirtschaftlich wichtiger die Erstellung von Reports oder Backups.

Lösung *First Things First* weist den Tasks unterschiedliche *Prioritäten* zu. Höher prioristierte Tasks werden bevorzugt bearbeitet, niedriger priorisierte können nachrangig oder gar nicht ausgeführt werden. Das Antwortzeitverhalten eines Systems wird so (für wichtige Tasks) bei Spitzenauslastungen erhöht. [SmWi02]

Typische Beispiele von *First Things First* sind Betriebssystem-Scheduler, die Prozessen unterschiedliche Prioriäten (in Unix etwa: *Niceness*) zuweisen und diesen ausgehend von der Priorität verschieden viel Rechenzeit zuweisen.

2.4 Flex Time

Problem Softwaresysteme können zu verschiedenen Zeitpunkten deutlich unterschiedliche Auslastungen aufweisen. So wird z.B. ein in Deutschland angesiedelter Webshop an Feiertagen tagsüber mehr Anfragen verarbeiten als in den Nachtzeiten an Werktagen. Dabei sind einige Arbeitseinheiten (z.B. Indizierungen oder Rundmails) weniger zeitkritisch und könnten, wenn die Infrastruktur es erlaubt, nachrangig in Zeiten von geringerer Last ausgeführt werden.

Lösung Flex Time verteilt Arbeitseinheiten zeitlich angepasst an die Workload-Charakteristik der Infrastruktur. Es verhindert damit ein Performance-Bottleneck bei Spitzenauslastungen.

In einem ersten Schritt werden Spitzenlastzeiten durch Performancemessungen (Monitoring) oder zur Laufzeit durch automatische adaptive Methoden identifiziert. Befinden sich zu diesen Zeiten nicht-zeitkritische Arbeitseinheiten in der Ausführung, werden diese Arbeitseinheiten auf eine alternative Zeit verschoben.

Bei diesem Ansatz ergibt sich allerdings das Problem, dass verschiedene getrennte Applikationen die gleichen alternativen Zeitpunkte wählen könnten und es so erneut zu Bottlenecks kommt [SmWi02]. Dies kann z.B. durch einen gemeinsamen, zentralen Workload-Scheduler verhindert werden (siehe z.B. IBM Tivoli Workload Scheduler [Tivo09]).

3 Distributionsorientere Patterns

Verteilte Systeme werden in letzter Zeit durch die Entwicklungen im Pervasive Computing/Cloud Computing oder dem Durchbruch des Internets immer

populärer. Allerdings ist das Einbetten von objektorienterten Architekturen in verteilte Systeme nicht trivial, obwohl heute eine Vielzahl von Frameworks und Bibliotheken zur Unterstützung verfügbar sind.

Übliche objektorientierte Verteilungs-Frameworks benutzen einfache Klassen und geben diese auf Wunsch im Netzwerk für entfernten Methodenaufruf frei (sog. *Distribution by Class* [Fowl02]). Dies führt bei vielen Entwicklern zu der *(Fehl-)*Annahme, man solle existierende Geschäftslogik, z.B. Datenzugriffsobjekte (*Data Access Objects*), ohne weitere Modifikation für entferne Aufrufe (*Remote*) freigeben. Bei der Programmierung könne man dann transparent auf diese entfernten Objekte zugreifen, als wären sie im lokalen Prozess. Dieser Ansatz ignoriert allerdings das signifikante Performance-Bottleneck des Netzwerks: Funktionsaufrufe auf Objekte in fremden Prozessen oder gar entfernten Computern sind um Größenordnungen langsamer als prozessinterne [Fowl02].

Die Patterns in diesem Abschnitt zeigen einige Lösungswege auf, die typischen Performance-Probleme in verteilten Systemen zu minimieren.

3.1 Remote Facade

Problem *Distribution by Class* - Frameworks, wie sie heute bei vielen Webservice- und RMI (Remote Method Invocation)-Bibliotheken in objektorientierten Architekturen verwendet werden, führen häufig dazu, dass Klassen der Geschäftslogik für den lokalen Prozess-Kontext ohne weitere Modifikation im Netzwerk freigegeben werden. Dabei sind lokale, prozessinterne Schnittstellen zu diesen Klassen meist *feingranularer* Natur, um möglich großen Funktionsumfang zu ermöglichen. In den meisten Frameworks sind einzelne, feingranulare Funktionsaufrufe allerdings mit erheblichem Overhead verbunden.

Lösung Das *Remote Facade* [Fowl02] überträgt das GoF *Facade Pattern* [EGVl95] auf verteilte Klassen.

Die prozess-interne Geschäftslogik mit feingranularen Schnittstellen wird erhalten. Für die Anbindung an verteilte Systeme wird eine explizite Menge an Klassen mit vereinfachten, grobgranularen Schnittstellen durch *Distribution by Class*-Frameworks entwickelt und freigegeben. Diese Klassen bilden die dedizierte *Remote Facade* (Außenfassade) des Systems. Durch die grobgranulare Struktur wird der Overhead durch einzelne Funktionsaufrufe reduziert. Die Kosten für die daraus folgenden redundant übertragenden Daten sind meistens vernachlässigbar.

Abbildung 3 zeigt ein Klassendiagramm für die Implementierung des Remote Facade Patterns am Beispiel eines Datenzugriffsobjekts(engl. Data Access Object, DAO) für den Zugriff auf Personendaten. Der lokale prozessinterne Zugriff erfolgt über die Klasse Person. Für entfernte Aufrufe wird das im Netzwerk freigegebene Objekt *PersonFacade* mit seiner grobgranularen Schnittstelle verwendet.

Bei verteilten Anwendungen mit verschiedenen Netzwerkanbindungen wären auch mehrfache *Remote Facade*-Schichten denkbar. So könnte man je nach Anforderung Fassaden für z.B. WAN-, LAN- oder IPC(*Inter-Process-Calls*)-Anbindungen mit unterschiedlicher Granularität entwickeln.

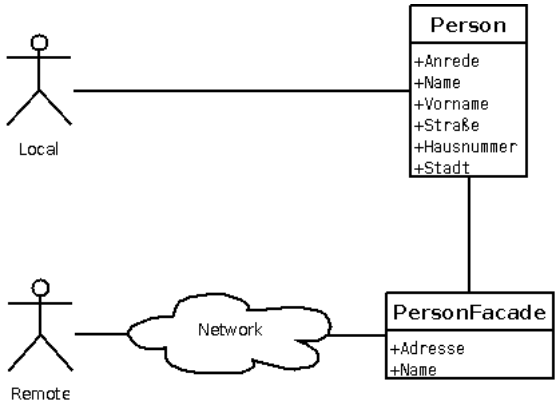


Abbildung 3. Das Remote Facade PersonFacade als Vergröberung der Klasse Person

3.2 Batching

Problem In verteilten Systemen bringt der einzelne, asynchrone Aufruf einer entfernten Arbeitseinheit zumeist einen signifikanten Overhead mit sich. Dabei haben viele dieser Aufrufe weniger strenge zeitliche Anforderungen und könnten verzögert ausgeführt werden.

Lösung *Batching* ist eine Methode, diskrete Arbeitseinheiten zusammenzufassen und gebündelt zu verarbeiten. Sie findet dann Anwendung, wenn die Summe der Overhead-Kosten für die Verarbeitung einer einzelnen Arbeitseinheit deutlich größer sind als die Kosten der gebündelten Arbeitseinheiten. Bei vielen HTTP-Webservice-Bibliotheken beispielsweise wird für jede Anfrage eine TCP-Verbindung aufgebaut und ein HTTP- bzw. SOAP-Header generiert. Sendet man mehrere Arbeitseinheiten gebündelt, reduziert sich dieser Overhead.

Durch Batching wird der Overhead von einzelnen Aufrufen minimiert, indem Arbeitseinheiten zusammengefasst werden und in einem Stapel abgeschickt werden.

Abbildung 4. Batching(oben) im Vergleich zu einzelnem Aufruf(unten) reduziert Overhead(schwarz)

3.3 Command Pattern

Problem In verteilten Softwaresystemen ist es häufig wünschenswert, Funktionen auf anderen Rechnern auszuführen. Viele objektorientierte Frameworks für verteilte Anwendungen arbeiten aber auf Klassen- bzw. Objektbasis. Während Serialisierung von Objekten, also die Umwandlung in übertragbare Bytestöme, meistens zur Verfügung steht, ist die Serialisierung von Methoden und Code häufig nicht möglich bzw. aus Gründen der Sicherheit nicht wünschenswert.

Lösung Das Command Pattern [EGVl95] trennt die Daten explizit von den Codestücken und wandelt Methoden in Klassen um. Durch die Umwandlung in Klassen können erweiterte objektorientierte Paradigmen wie Vererbung, Interfaces oder Komposition angewendet werden.

Häufig wird das Command Pattern verwendet, um *Undo*-Operationen im Userinterface elegant zu implementieren, indem zusätzlich zu der execute() -Methode eine undo() -Methode implementiert wird, die den Zustand vor der Ausführung von execute() wiederherstellt. Speichert man dann Benutzerinteraktionen als Command-Patterns z.B. in einem Stack, können durch Aufrufe von undo() Commands rückgängig gemacht werden.

Bei verteilten bei der Serialisierung und Verteilung eines werden nur die Datenfelder übertragen.

Beispielimplementierung Im folgenden Beispiel wird die Transformation einer Methode in eine Klassenstruktur demonstriert.

```
int doComplexCalculation(int paramA, String paramB) {
  // insert some complex calculation code here
  return result;
}

// Usage:
System.out.println( doComplexCalculation(100, "bar") );
```

Bei der Überführung in das Command-Pattern wird die Methode doComplexCalculations(..) in eine Klasse gekapselt. Die Parameter und Rückgabewerte werden in Instanzvariablen gespeichert. Möchte man die Berechnung auf entfernten Instanzen ausführen, werden nur diese serialisiert.

```
class ComplexCalculationCommand {
  public int paramA;
  public String paramB;
  public int result;

  void execute() {
    // insert some complex calculation code here
    result = ...;
  }
```

```
}

// Usage:
ComplexCalculationCommand complexCalculation = new ComplexCalculationComman
complexCalculation.paramA = 100;
complexCalculation.paramB = "bar";

// execute in local context
complexCalculation.execute();

// or dispatch to some remote server
someRemoteServer.execute( complexCalculation );
```

Erweiterung um Quality of Service Anforderungen Während das Kern-Command-Pattern eine rein struktuelle Transformation durchführt, sind v.a. die Möglichkeiten, die die Objektorientierung mit sich bringt, interessant. So kann man durch Vererbung oder Komposition zusätzliche Funktionalität oder Assertions hinzufügen.

Abbildung 5 zeigt ein Klassendiagramm, in der die ComplexCalculation aus dem oberen Beispiel für Performance-Monitoring-Zwecke in ein PerformanceMonitorCommand verschachtelt wird. Zur Analyse der Performance eines verteilten Systems kann man dann die erweiterten Felder wie totalProcessingTime verwenden.

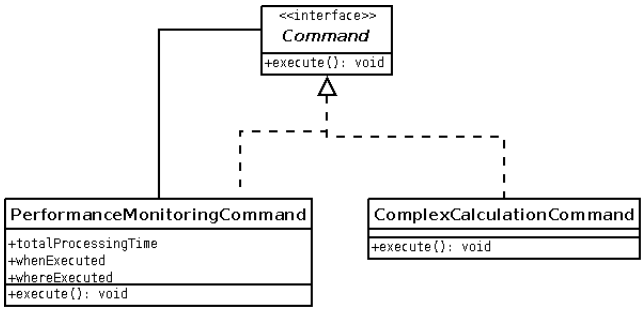


Abbildung 5. Erweiterung des Command-Patterns um Performance-QOS Monitoring Attribute

4 Concurrency Patterns

Paralleler Zugriff auf beschränkte Ressourcen und dessen Verwaltung findet sich in beinahe jedem Softwarsystem. Aus Performance-Gesichtspunkten sind dabei

Zugriffe auf Ressourcen, deren Zugriff auf einen Task zu einer Zeit beschränkt ist, das zentrale Bottleneck, da diese sequentiell abgearbeitet werden müssen. In diesem Abschnitt werden Patterns vorgestellt sich auf solche *Concurrency*-Problemstellungen anwenden lassen und die Abhängigkeiten zu Ressourcen minimieren.

4.1 Pooling

Problem Viele Softwaresysteme wie Webserver, Datenbanken oder andere Middleware weisen eine hohe Anzahl gleichartiger Anfragen auf, die parallel zeitkritisch bearbeitet werden müssen. Intern werden solche Anfragen meistens nebenläufig über *Worker*, z.B. durch Threads oder Prozesse (*Thread-per-Request* bzw. *Process-per-Request*), abgearbeitet. Die Instantiierung bzw. Deinstantiierung von Threads und Prozessen ist bei den meisten Umgebungen allerdings mit einem signifikaten Overhead verbunden.

Lösung Die Pooling Strategie vermindert den Overhead bei der Instantiierung von oft benutzten Workern, indem es eine gewisse Anzahl an Workern unabhängig von der aktuellen Auslastung in einem *Pool* vorhält. Eingehenden Anfragen werden dann einem Worker aus dem Pool zugeteilt. Hat der Worker die Bearbeitung der Anfrage abgeschlossen, wird sein Status wieder zurückgesetzt, und steht dem Pool für weitere Anfragen wieder zur Verfügung.

Die Kosten für die Instanziierung/Deinstanziierung der Worker wird durch Pooling-Mechanismen eingespart.

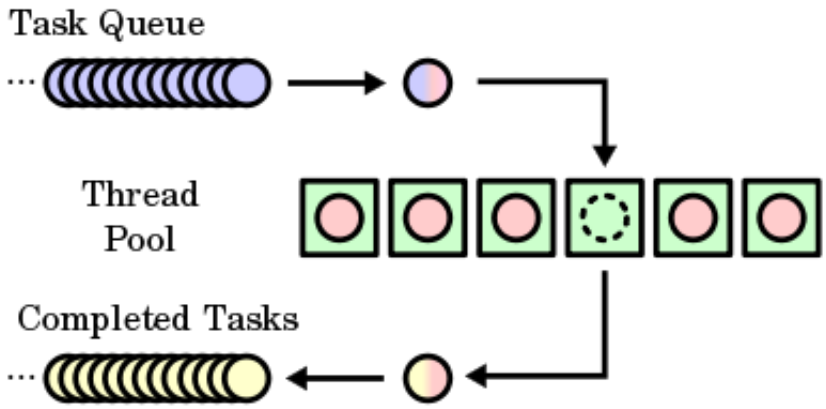


Abbildung 6. Pooling zur parallelen Verarbeitung von Tasks in einem Thread Pool [Wiki09]

Farming Auf höherer Ebene wird Pooling auch als *Farming* [MsPe04] bezeichnet. Abbildung 7 zeigt die Skalierungsmöglichkeiten, die sich durch Farming bei Webanwendungen ergeben. Die Verteilung der Client-Requests auf die Worker(Web-Server) wird durch einen zentralen *Load Balancer* vorgenommen. Steigt die Zahl der Clients dauerhaft an, erweitert man die Web-Farm einfach durch zusätzliche Web-Server und registriert diese beim Load Balancer. So wird die Skalierbarkeit des Systems erhöht.

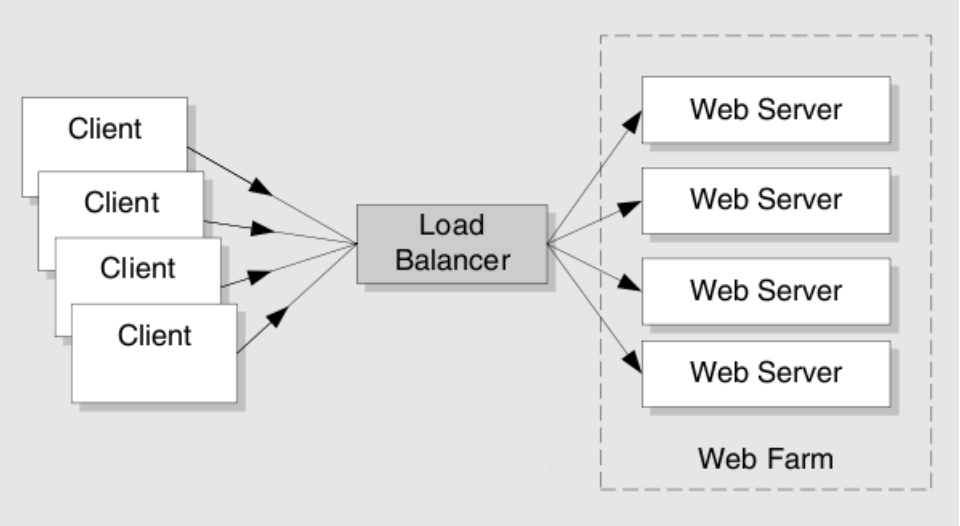


Abbildung 7. Web Farming [MsPe04]

4.2 Caching

Problem Wiederholtes laden, bearbeiten und freigeben von entfernten Ressourcen bringt im Gegensatz zu lokalem Zugriff einen Performance-Overhead mit sich. Unterliegen diese Ressourcen außerdem Beschränkungen bezüglich gleichzeitigen Zugriffs, muss bei Zugriff einer schon belegten Ressource gewartet werden, bis sie von einem andereren Benutzer wieder freigegeben wird.

Lösung Das *Caching*-Pattern hält häufig genutzte Ressourcen temporär in einem oder mehreren Zwischenspeichern(*Caches*) mit schnellem Zugriff vor. Wird die Resource nicht mehr benötigt oder ist Speicherplatz im Cache erforderlich, wird die Ressource entfernt. [KiJa04] Die Benutzung eines Caches kann aus Sicht des Benutzers transparent erfolgen.

Dies reduziert den Overhead beim Zugriff auf entfernten Ressourcen. Spielt konkurrierender Zugriff auf die Ressource eine Rolle, so können Abhängigkeiten

durch Duplizierung einer Ressource in Caches bei lesendem Zugriff vermieden werden. Allerdings müssen dann für Rückschreibzugriffe komplexere Methoden verwendet werden.

4.3 Optimistic Offline Lock

Problem Oft wird das Problem des gleichzeitigen Zugriffs gelöst, indem die Ressource beim aquirieren gesperrt und nach dem Zurückschreiben wieder freigeben wird(*Pessimistisches Locking*). Auf eine gesperrte Ressource kann während eines Locks von Dritten dann nicht mehr zugegriffen werden. Dieser Ansatz macht besonders bei Ressourcen mit einer hohen Konfliktwahrscheinlichkeit Sinn.

Bei hoher Anzahl von parallelen Lesezugriffen auf die beschränke Ressource führt eine pessimistische Locking-Strategie allerdings zu Performance-Overhead, da die Zugriffe sequentiell abgearbeitet werden müssen, selbst wenn kein Konflikt auftritt bzw. auf die Ressource nicht schreibend zugegriffen wird.

Lösung *Optimistic Offline Lock* [Fowl02] führt eine Konfliktbehandlung nur dann aus, wenn während einer Transaktion auch wirklich ein Konflikt aufgetreten ist.

Jeder beschränkten Ressource wird eine Versionsnummer zugeordnet, die sich bei jedem Schreibzugriff inkrementell erhöht. Beim aquirieren/laden einer Kopie der Ressource wird zunächst kein Lock gesetzt. Mehrere Benutzer können nun gleichzeitig auf diese Ressource zugreifen und sie evtl. bearbeiten. Nach der abschließenden Freigabe der Ressource wird nun anhand der Versionsnummer überprüft, ob in der Zwischenzeit eine Änderung der Ressource durch Dritte stattgefunden hat. Wird an dieser Stelle ein Konflikt erkannt, kann er z.B. durch Wiederholung der Transaktion oder Meldung an den Benutzer behandelt werden.

5 Ausblick

Performance Patterns sind Lösungen für viele architektonische performancekritische Problemstellungen im Softwareentwurf. Diese Ausarbeitung enthält eine kleine Auswahl von etablierten Patterns, es existieren weit mehr.

Während Performance Patterns gute Wege zur performance-orientieren Softwarearchitektur aufzeigen, sind für ganzheitliches Performance-Engineering weitere Domänen nötig:

- *Performance Monitoring*: Messung und Auswertung des Systems zur Laufzeit, um Bottlenecks zu identifizieren
- *Plattformspezifische Eigenheiten*, z.B. die Betrachtung von Systeminternas oder die Benutzung von performanten Programmiersprachen/Frameworks
- *Software-Entwicklungs-Prozess*: Auswahl von bzw. Einbettung in ein Prozess-Modell, wie z.B. Software Performance Engineering (s. Abschnitt 1.3)
- *Refactoring*: In der Praxis werden die meisten Performance-Bottlenecks erst nach Tests (Monitoring) eines lauffähigen Systems aufgedeckt. Refactoring ist die Disziplin, Patterns in eine existierende Codebasis einzubetten.

Literatur

Alex77. Silverstein M. Alexander C., Ishikawa S. *A Pattern Language.* Oxford University Press. 1977.

EGVl95. Ralph Johnson Erich Gamma, Richard Helm und John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison Wesley. 1995.

Fowl02. Martin Fowler. *Patterns of Enterprise Application Architecture.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA. 2002.

KiJa04. Michael Kircher und Prashant Jain. *Pattern-Oriented Software Architecture: Patterns for Resource Management.* John Wiley & Sons. 2004.

Meye05. Scott Meyers. *Effective C++ : 55 Specific Ways to Improve Your Programs and Designs.* Addison-Wesley Professional. third. Auflage, May 2005.

MsPe04. *Improving .NET Application Performance and Scalability (Patterns And Practices).* Microsoft Press, Redmond, WA, USA. 2004.

SmWi02. Connie U. Smith und Lloyd G. Williams. *Performance solutions: a practical guide to creating responsive, scalable software.* Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA. 2002.

Tivo09. IBM Tivoli Workload Scheduler. http://www-01.ibm.com/software/tivoli/products/scheduler , 2009. [Online; Letzter Zugriff am 02. Jan. 2009].

Wiki09. Wikipedia. Thread Pool Pattern — Wikipedia, The Free Encyclopedia. http://en.wikipedia.org/wiki/Thread_pool_pattern , 2009. [Online; Letzter Zugriff am 07. Jan. 2009].

Realoptionen für Entscheidungen in der Software-Entwicklung

Tom Beyer

Betreuer: Johannes Stammel

Zusammenfassung Diese Arbeit geht auf die Verwendung von Realoptionen zur Unterstützung von Entscheidungen in der Software-Entwicklung ein. Dazu werden zunächst die allgemeinen Konzepte dargestellt und dann anhand eines Beispieles veranschaulicht. Anschließend werden allgemeine Entscheidungsregeln abgeleitet. Zum Abschluss werden drei Vorgehensmodelle zur Software-Entwicklung (Wasserfallmodell, Spiralmodell, Extreme Programming) aus Sicht der Realoptionen-Theorie untersucht und die Ergebnisse zusammengefasst.

1 Einführung

Die heutige Software-Entwicklung verwendet eine Vielzahl von Vorgehensmodellen. Diese Modelle beschreiben wie das Entwicklungsprojekt ablaufen sollte und wann welche Produkte dabei entstehen sollten. Implizit geben sie auch Hilfestellung beim Treffen von Entwurfsentscheidungen. In den meisten Fällen müssen hierbei auch Unsicherheiten berücksichtigt werden, da sich die Anforderungen an das Software-System in Zukunft ändern können. Darum ist Flexibilität von Software immer mehr ins Zentrum der Aufmerksamkeit gerückt. Oft fällt es jedoch schwer, den Nutzen von Flexibilität zu bewerten. Dies ist jedoch notwendig um zu entscheiden, ob in Flexibilität investiert werden sollte oder nicht. Realoptionen stellen einen Ansatz dar um die Nutzenanalyse von Entwurfsentscheidungen bezüglich der Flexibilität zu erleichtern.

Zunächst werden in dieser Arbeit die allgemeinen Konzepte vorgestellt. Nach der Präsentation eines ausführlichen Beispiels werden dann Entscheidungsregeln, die sich aus der Realoptionen-Theorie ergeben, beschrieben. Den Abschluss der Arbeit bildet eine Diskussion dreier Vorgehensmodelle zur Software-Entwicklung.

1.1 Bewertung von Software-Produkten

Das Ziel der Software-Entwicklung ist nicht in erster Linie die bloße Erstellung von Software, sondern das Schaffen von Wert für den Hersteller in Form eines Software-Produkts. Es geht also nicht darum irgendein Software-Produkt zu erstellen, sondern eines, das möglichst viel Wert generiert. Erst daraus ergeben sich Konsequenzen für das Design und die Implementierung des Produkts. Die Architektur ist also kein Selbstzweck, sondern dient der Wertmaximierung. Im Folgenden wird der Wert des Software-Produkts definiert als der Marktwert,

den der Hersteller durch dieses Produkt zusätzlich erlangt. Diese Definition entstammt [SuCS99, S. 216].

Das Ziel dieses Ansatzes ist es, die Verbindung zwischen Entwurfsentscheidungen und der Wertentwicklung des Produktes explizit zu machen. Die Wertentwicklung ist dabei die Differenz von Nutzen und Kosten der Entwurfsentscheidung. In der Regel sind die Kosten deutlich einfacher zu erfassen als der Nutzen [SuCS99, S. 220]. Betrachten wir zum Beispiel die Entscheidung zwischen einer monolithischen Architektur und einer modularisierten Architektur. Üblicherweise ist die monolithische Architektur mit geringeren Entwicklungskosten, aber auch mit geringerer Flexibilität verbunden. Flexibilität meint hierbei die Eigenschaft der Architektur sich an neue Anforderungen anpassen zu lassen. Die Flexibilität zahlt sich aber natürlich nur aus, wenn auch Anpassungen in Zukunft gemacht werden. Der Nutzen ist mit Unsicherheit behaftet. Der Wert setzt sich aus sicheren und unsicheren Cashflows zusammen.

1.2 Klassische Investitionsbewertung mit der Kapitalwert-Methode

Klassischerweise wird die Kapitalwert-Methode verwendet, um den resultierenden Wert zu berechnen. Bei positivem Ergebnis ist die Investition lohnenswert, ansonsten nicht. Die Berechnung erfolgt folgendermaßen. Gegeben seien die Cashflows (Zahlungsströme) $c_{0,1}, \ldots, c_{0,k_1}, \ldots, c_{T,1}, \ldots, c_{T,k_T}$, wobei $c_{t,1}, \ldots, c_{t,k_t}$ die Cashflows in Periode t sind, der Kalkulationszinssatz i und die zugehörigen Wahrscheinlichkeiten $p_{0,1}, \ldots, p_{0,k_1}, \ldots, p_{T,1}, \ldots, p_{T,k_T}$, wobei $p_{t,j}$ die Wahrscheinlichkeit angibt, dass Cashflow $c_{t,j}$ realisiert wird. Der Kapitalwert C_0 berechnet sich dann folgendermaßen:

$$C_0 = \sum_{t=0}^{T} \sum_{j=1}^{k_t} (1+i)^{-t} p_{t,j} c_{t,j} \tag{1}$$

Natürlich muss zu einem festen Zeitpunkt t die Funktion $p_t(j) = p_{t,j}$ eine Wahrscheinlichkeitsfunktion bilden. Zum Zeitpunkt t sind im Allgemeinen verschiedene Cash Flows unterschiedlich wahrscheinlich. In der Realität wird aber nur ein Cash Flow realisiert. Dies mündet in folgender Nebenbedingung:

$$\forall t \in \{0, \ldots, T\} : \sum_{j=1}^{k_t} p_{t,j} = 1 \tag{2}$$

Allgemein ist man der Auffassung, dass der Kapitalwert nicht die Flexibilität einer Investition berücksichtigt, und somit den Wert von Investitionen mit hoher Flexibilität unterschätzt [SuCS99, S. 221]. Grund hierfür ist, dass die Kapitalwert-Methode davon ausgeht, dass nur zur Periode $t = 0$ eine Entscheidung getroffen werden muss. In späteren Perioden kann der Investor keine weiteren Entscheidungen mehr treffen. Dies ist aber beispielsweise in oben genanntem Fall nicht gegeben. Eventuell mag es sogar Sinn machen, die Entscheidung zu verschieben, um weitere Informationen über die künftigen Anforderungen an das

System zu gewinnen. Wenn ein Projekt negativ verläuft, macht vielleicht auch der Abbruch des Projektes Sinn. Dies alles wird mit der Kapitalwertmethode nicht erfasst.

In dieser Arbeit wird davon ausgegangen, dass sich der Wert eines Produktes nur aus damit verbundenen finanziellen Zu- oder Abflüssen zusammensetzt. Diese Zu- oder Abflüsse können heute oder künftig geschehen. Damit wird beispielsweise nicht die Kompetenz berücksichtigt, die ein Entwicklungsteam durch die Umsetzung hinzugewonnen hat. Diese Kompetenz könnte dazu beitragen, die Entwicklung neuer Produkte kostengünstiger durchzuführen. Es wird angenommen, dass dieser Wertzuwachs bei dem neuen Produkt berücksichtigt wird, da hier die Produktionskosten durch den Kompetenzaufbau geringer werden.

1.3 Wertorientierte Entwicklungsprozesse

Software-Entwicklung ist ein Prozess unter Unsicherheit und mit nur unvollständiger Information. Sowohl der Fortschritt der Implementierung, Änderungen der Kundenanforderungen als auch die zukünftigen Cashflows sind nur unzureichend bekannt. Zusätzlich kommt hinzu, dass Entscheidungen, die früh im Entwicklungsprozess getroffen werden, oft große Auswirkungen auf den weiteren Entwicklungsprozess haben. So werden beispielsweise Entscheidungen bezüglich der Software-Architektur oder der verwendeten Technologien früh getroffen. In diesem Stadium ist jedoch die Unsicherheit (beispielsweise über die künftigen Anforderungen) am größten. Es könnte sich als wertschöpfend erweisen, die Entscheidungen später im Prozess zu treffen [Erdo99].

Von besonderer Bedeutung ist hierbei die Bewertung von Flexibilität. Entscheidet man sich beispielsweise für eine flexiblere Architektur, so sind Änderungen der Anforderungen einfacher während des Entwicklungsprozesses umzusetzen. Sie ermöglichen es so, Aufwärts-Potenziale (*engl.* „upside potential“) zu nutzen und sich gegen Abwärts-Potenziale (*engl.* „downside potential“) abzusichern. Aufwärts-Potenziale meint hierbei die potenziellen, positiven Auswirkungen einer Entscheidung und Abwärts-Potenziale entsprechend die potenziellen, negativen Auswirkungen. Ein Beispiel hierfür ist die Adaption eines neuen Standards zum Datenaustausch. Da nicht abzusehen ist, ob sich der Standard durchsetzen wird, ermöglicht eine flexiblere Architektur auch eine spätere Implementierung des Standards (vielleicht hier schon mit der Unterstützung von Werkzeugen anderer Hersteller). Ein Negativ-Beispiel dafür ist der Konkurrenzkampf zwischen den beiden Formaten für optische Speichermedien Blu-ray und HD DVD. Durch die frühe Festlegung der Hersteller auf ihr jeweiliges Format konnten einige von ihnen ihre Entwicklungskosten nicht kompensieren und außerdem hat der Konkurrenzkampf die Marktdurchdringung der neuen Formate verlangsamt, da sich die Verbraucher im Gegensatz zu den Herstellern nicht auf ein Format festlegen wollten.

Ein weiteres Beispiel ist das Jahr-2000-Problem [SuCS99, S. 218]. Dieses betrifft Software, bei der das Jahr nur mit zwei Ziffern und nicht mit vier Ziffern gespeichert wurde. Das Problem ist hier nicht so sehr, dass vier Ziffern mehr Speicherplatz benötigen als zwei, sondern dass die Entwurfsentscheidung nicht

in einem einzigen Modul gekapselt wurde. Deshalb war es sehr aufwändig, alle Stellen zu finden, an denen Änderungen vorzunehmen waren.

Bisherige Software-Entwicklungsprozesse bedienen sich bestimmter Heuristiken oder Muster, die sich in der Praxis als nützlich erwiesen haben. Aus wissenschaftlicher Perspektive stellt sich jedoch die Frage, warum diese nützlich sind und wann diese Prinzipien gebrochen werden sollten. Auch wenn es in der Praxis meist schwer sein wird, den Wert eines Systems exakt festzustellen, ist es doch sinnvoll, über solche Fragestellungen auch aus qualitativer Sicht nachzudenken. Insbesondere auch deshalb weil zumindest die Parameter (Unsicherheiten bezüglich den Anforderungen, künftige Cashflows, etc.) bestimmt werden, auch wenn man sie nicht quantifizieren kann.

Das Ziel ist deshalb, Entscheidungen im Software-Entwicklungsprozess systematisch auf die Wertschöpfung auszurichten. Hierbei soll insbesondere die Bewertung von Flexibilität im Vordergrund stehen. Dazu wird im Folgenden auf die Verwendung von Realoptionen eingegangen. Diese helfen, Flexibilität in die Bewertung von Entscheidungen mit einzubeziehen.

1.4 Optionen und Realoptionen

Optionen entstammen der Finanzwelt. Sie sind ein Derivat, das heißt ein Vertrag auf Grundlage einer anderen Bezugsgröße (z.B. Aktien oder Anleihen) zwischen zwei Marktteilnehmern [KoOv07, S. 1 – 5]. Optionen können selber als Wertpapiere gehandelt werden. Man unterscheidet *Put-Optionen* und *Call-Optionen* [KoOv07, S. 330f]. Eine Call-Option gibt dem Käufer das Recht, den darunter liegenden Basiswert zum *Ausübungspreis* zu kaufen. Eine Put-Option gibt dem Käufer hingegen das Recht, den darunter liegenden Basiswert zum Ausübungspreis zu verkaufen. Der Ausübungspreis und der Basiswert sind durch die Option festgelegt. Beide Typen gibt es in zwei Varianten: eine *amerikanische Option* kann vom Kauf der Option bis zum *Ausübungsdatum* eingelöst werden und eine *europäische Option* kann nur am Ausübungsdatum eingelöst werden. Auch das Ausübungsdatum ist durch die Option festgelegt. Der Käufer hat nur das Recht, aber nicht die Pflicht, die Option einzulösen. Hat er die Option eingelöst, ist sie anschließend wertlos und es besteht kein Vertragsverhältnis zwischen den Partnern mehr.

Man unterscheidet Finanzoptionen, deren Basiswert finanzieller Natur ist, von Realoptionen, deren Basiswert nicht-finanzieller Natur ist [SuCS99, S. 222]. Das Konzept von Realoptionen wurde von Myers in [Myer77] vorgestellt.

Realoptionen ermöglichen es, die Flexibilität einer Software im Entwicklungsprozess zu bewerten. Investitionen in Flexibilität selber kann man sich auch als den Kauf einer Option vorstellen. Er ermöglicht es dem Käufer (in diesem Fall dem Hersteller), kostengünstiger nachträglich Änderungen am System durchzuführen, um eine bestimmte Änderung der Anforderungen umzusetzen. In der Regel ist dies mit einem höheren Aufwand zu Beginn der Entwicklung verbunden. Dies entspricht dem Preis der Option selbst. Die Frage ist nun, welcher Preis für die Option angemessen ist. Dies ist eine Frage, mit der sich die Finanzwissenschaft schon ausgiebig beschäftigt hat. Ihre Erkenntnisse helfen auch hier weiter.

Für Finanzoptionen werden häufig Bewertungsmodelle auf Basis des Nichtvorhandensein von Arbitrage-Geschäften verwendet. Arbitrage bedeutet das Ausnutzen von Preisunterschieden des gleichen Produktes auf verschiedenen Märkten. Dies kann beispielsweise der Fall sein, wenn die Aktie eines Unternehmens an verschiedenen Börsen gehandelt wird. In der Finanzwelt sind solche Arbitrage-Geschäfte kaum möglich, da sich Preisunterschiede in der Regel sehr schnell ausgleichen [SuCS99, S. 222]. Diese Tatsache wird für die Bewertung von Finanzoptionen ausgenutzt. Für viele Entscheidungen in der Software-Entwicklung sind solche Techniken aber leider nicht anwendbar, da vielen Annahmen (z.B. vollständige Information) nicht zutreffen. Im Folgenden wird daher auf die Bewertung von Realoptionen mit Ereignisbäumen eingegangen, wie sie in [SuCS99, S. 222 – 233] vorgestellt wird. Das Hauptziel ist die Bewertung von Flexibilität.

[SuCS99, S. 223f] unterscheidet verschiedene Typen von Realoptionen (siehe auch [Erdo99]):

- Optionen, die es erlauben, spätere Investitionen mit besseren Rahmenbedingungen durchzuführen
- Optionen, mit denen der Käufer das Recht hat, eine Investition zu verschieben und beispielsweise mit mehr Informationen durchzuführen
- Option, ein Projekt in frühem Stadium abzubrechen
- Option, die Produktion zu verkleinern oder zu vergrößern, wenn sich die Rahmenbedingungen ändern
- Option, die Materialien des Produktes oder das Produkt selbst zu ändern
- Zusammengesetzte Optionen, das heißt Optionen auf Optionen
- Option, eine aus mehreren Anlagen auszuwählen
- Interaktionen zwischen Realoptionen

Zur Veranschaulichung wird nun ein Beispiel behandelt. Gegeben seien eine monolithische Architektur A und eine modulare Architektur B. Architektur B ist im Gegensatz zu Architektur A schon von Beginn an darauf ausgelegt, als verteiltes System arbeiten zu können. Falls die Last ansteigt, können Komponenten einfach auf neue Server ausgelagert werden. Die derzeitige Last kann aber noch von einem Server getragen werden. Der Nutzen B_A bzw. B_B der Architekturen A bzw. B kann dann in Form von Cashflows (beispielsweise vom Kunden an den Hersteller) gemessen werden. Die Kosten C_A bzw. C_B sind dann überwiegend die Entwicklungs- und Betriebskosten. Der Wert V_A der Architektur A ist dann:

$$V_A = B_A - C_A \tag{3}$$

Für den Wert V_B ergibt sich:

$$V_B = B_B + O_B - C_B \tag{4}$$

Hierbei ist O_B der Wert der Option, in Zukunft auf ein verteiltes System umsteigen zu können. Die Kosten C_B stellen hierbei auch die Kosten für den Kauf der Option dar, da durch Implementierung der Architektur B die Option vorhanden ist.

Wenn man annimmt, dass der Nutzen beider Architekturen gleich ist ($B_A = B_B$) – dies kann beispielsweise der Fall sein, wenn der Kunde nur für den Dienst selber einen fixen, von der Last unabhängigen Betrag bezahlt – dann ist $V_B > V_A$ genau dann wenn $O_B > C_B - C_A$. Aber offensichtlich ist diese Analyse unvollständig, denn sie berücksichtigt nicht den Aufwand der dadurch entsteht, dass Architektur A an ein verteiltes System angepasst werden muss. Um diese Unsicherheiten zu modellieren werden nun Ereignisbäume vorgestellt [SuCS99, S. 229ff].

Um die Unsicherheit zu beschreiben führen wir eine Zufallsvariable X zusammen mit einem Ereignisbaum der (endlichen) Tiefe N ein. Die Tiefe N stellt dabei den ganzen Zeitraum (in Jahren, Monaten, etc.) dar, den wir abbilden möchten. Der Wurzelknoten (Knoten der Tiefe 0) des Ereignisbaums stellt die Gegenwart dar. Die Knoten der Tiefe k ($0 \leq k \leq N$) repräsentieren die Zustände der Welt zum Zeitpunkt k. Wenn Knoten v (Tiefe k) ein Vorgänger von Knoten w (Tiefe $k+1$) ist, so ist es möglich von Zustand v in den Zustand w zu gelangen (Notation: $v \to w$).

Die Zufallsvariable X ist dann eine Funktion, die jedem Knoten v eine reelle Zahl $X(v)$ zuordnet. Ein stochastischer Prozess ist dann eine Reihe von Zufallsvariablen $\{X_k\}_{k=0}^{N}$ unter der Voraussetzung, dass für jedes k ($0 \leq k \leq N$) $X_k(v) \neq 0$ genau dann wenn die Tiefe von v gleich k ist. Weiterhin beschreiben wir Übergangswahrscheinlichkeiten: $P(w|v)$ ist die Wahrscheinlichkeit von Zustand v nach Zustand w zu gelangen. $P(v)$ gibt die Wahrscheinlichkeit an, vom Wurzelknoten nach v zu gelangen. Somit gilt:

$$P(w|v) = \frac{P(w)}{P(v)} \tag{5}$$

Für die Zufallsvariable X definieren wir den Erwartungswert $E(X)$ wie folgt:

$$E(X) = \sum_{\text{Knoten } v} X(v)P(v) \tag{6}$$

Der bedingte Erwartungswert ist wie folgt definiert:

$$E(X|v) = \sum_{w:v\to w} X(w)P(w|v) \tag{7}$$

Nun wollen wir Optionen modellieren. Wir werden den Ausübungspreis mit L bezeichnen und das Ausübungsdatum mit T. Wir werden weiterhin annehmen, dass sowohl das Ausübungsdatum T als auch die Tiefe N des Ereignisbaumes mit den gleichen Zeiteinheiten gemessen werden und dass $T \leq N$ gilt. Der Preis des Basiswertes werde durch einen stochastischen Prozess $\{S_k\}$ beschrieben und die Tiefe des Ereignisbaumes entspreche dem Ausübungsdatum ($T = N$).

Nun betrachten wir den optimalen Zeitpunkt der Ausübung einer amerikanischen Call-Option. Offensichtlich ist es nicht sinnvoll, im Zustand v die Option auszuüben wenn $S(v) \leq L$ ist. Man könnte in diesem Zustand den Basiswert zu einem geringeren Preis auf dem Kapitalmarkt erwerben. Durch die Option

ist der Käufer nicht gezwungen die Option einzulösen. Wenn $S(V) > L$ gilt, ist die Entscheidung nicht so einfach. Man könnte als Käufer der Option auf einen noch höheren Preis spekulieren und damit den Profit erhöhen. Zum Zeitpunkt k können wir den Payoff (Gewinn) G_k der Option als Zufallsvariable wie folgt berechnen:

$$G_k = \max(S_k - L, 0) \tag{8}$$

Der Payoff $G(v)$ im Zustand v ist dann:

$$G(v) = \max(S(v) - L, 0) \tag{9}$$

Wir führen eine Entscheidungsregel τ ein, die Knoten auf $\{0, 1\}$ abbildet. Dabei gilt, dass $\tau(v) = 1$ genau dann, wenn die Option in Zustand v eingelöst wird. Wir bezeichnen den erwarteten Payoff einer Entscheidungsregel τ mit $V^\tau(v)$. In jedem Zustand w können wir den Payoff mit $G(w)\tau(w)$ berechnen. Dann erhalten wir für den erwarteten Payoff $V^\tau(v)$:

$$V^\tau(v) = E(G\tau|v) \tag{10}$$

Das Ziel ist nun die Bestimmung der optimalen Entscheidungsregel τ, für die $V^\tau(v)$ maximal ist. Dieses Maximum bezeichnen wir mit der Zufallsvariable V:

$$V(v) = \max_\tau V^\tau(v) \tag{11}$$

Es gilt auch $V(v) \geq \max(S(v) - L, 0)$, da die Option im Zustand v eingelöst werden könnte und der Payoff $\max(S(v) - L, 0$ realisiert werden würde.

Es kann gezeigt werden, dass für $V(v)$ gilt:

$$\tau(v) = \begin{cases} 1, & \text{wenn } \max(S(v) - L, 0) = V(v) \text{ oder } \delta(v) = N \\ 0, & \text{sonst} \end{cases} \tag{12}$$

Hierbei ist $\delta(v)$ die Tiefe des Knotens v im Entscheidungsbaum. Dies bedeutet, dass die Option eingelöst werden sollte, wenn in einem Zustand der erwartete Payoff dem Gewinn bei sofortiger Einlösung entspricht.

Es ist wichtig anzumerken, dass die Entscheidungsregel auch im Laufe des Prozesses und nicht nur am Anfang Hilfestellung geben kann, da sie beispielsweise auch nach Eintreten sehr unwahrscheinlicher Zustände berechnet werden kann. Im Gegensatz zur Kapitalwert-Methode erlaubt sie deshalb ein wesentlich dynamischeres Entscheiden, welches auch neue Informationen berücksichtigt. Sie ermöglicht nicht nur Aussagen darüber, ob überhaupt investiert werden soll, sondern auch wann.

Die Berechnung von $V_k(v)$ erfolgt mit dynamischer Programmierung. Dabei wird zunächst

$$V_N = \max(S_N - L, 0) \tag{13}$$

und dann schrittweise

$$V_k(v) = \max(\max(S_k(v) - L, 0), E(V_{k+1}|v)) \tag{14}$$

ausgenutzt.

Als Beispiel soll nun die Modularisierung eines Softwaresystems vorgestellt werden [SuCS99, S. 236ff]. Diese Modularisierung könnte in Zukunft die Wartungskosten reduzieren falls viele Änderungen an dem System vorgenommen werden sollen und wir von unserem Kunden für jede Änderung Geld erhalten. Es ist jedoch auch möglich, dass nur wenige Änderungen vorgenommen werden und die Modularisierung sich nicht gelohnt hat. Nehmen wir an, dass die Modularisierung 1600 GE kosten würde. Der Vereinfachung halber nehmen wir hier an, dass der Zinssatz 0% beträgt. In dem einen Szenario würden wir einen Cashflow von 3300 GE (wartungsintensives Szenario: viele Anforderungsänderungen) erhalten und in dem anderen würden wir nur 1100 GE (wartungsarmes Szenario) erhalten. Beide Szenarien sind gleich wahrscheinlich. Verwenden wir nun Gleichung 1 so erhalten wir den Kapitalwert:

$$C_0 = -1600 + 0,5 \cdot 1100 + 0,5 \cdot 3300 = 600 \tag{15}$$

Da der Kapitalwert positiv ist, sollte diese Investition getätigt und die Modularisierung sofort durchgeführt werden. Dabei geht man jedoch das Risiko ein, einen Verlust von $-1600+1100 = -500$ GE zu machen. Verwendet man Realoptionen gelangt man zu einem anderen Ergebnis.

In diesem Fall ist der Ausübungspreis $L = 1600$. Weiterhin ist $S_{1,1} = 1100$ (wartungsarmes Szenario) und $S_{1,2} = 3300$ (wartungsintensives Szenario). Beide Szenarien sind wiederum gleich wahrscheinlich. S_0 berechnet sich nun als der erwartete Cashflow (Kapitalwert) zu diesem Zeitpunkt:

$$S(v) = \sum_{w:v \to w} P(w|v) S_w \tag{16}$$

Man erhält also:

$$S_0 = 0,5 \cdot S_{1,1} + 0,5 \cdot S_{1,2} = 0,5 \cdot 1100 + 0,5 \cdot 3300 = 2200 \tag{17}$$

Nun können wir mithilfe der dynamischen Programmierung eine optimale Lösung finden. Dazu verwenden wir zunächst Gleichung 13:

$$V_{1,1} = \max(S_{1,1} - L, 0) = \max(1100 - 1600, 0) = 0 \tag{18}$$

$$V_{1,2} = \max(S_{1,2} - L, 0) = \max(3300 - 1600, 0) = 1700 \tag{19}$$

Damit können wir nun V_0 mit Gleichung 14 berechnen:

$$V_0 = \max(2200 - 1600,\ 0,5 \cdot 0 + 0,5 \cdot 1700) = 850 \tag{20}$$

Durch Gleichung 12 erhalten wir: $\tau(0) = 0$, da $\max(S_0 - L, 0) < V_0$ und $\delta(0) = 0 < 1$. Das bedeutet, dass wir abwarten sollten und nicht sofort in die Modularisierung des Produktes investieren sollten.

Im Allgemeinen können wir folgende Entscheidungsregel ableiten [SuCS99, S. 238]:

> Wenn zu einem bestimmten Zeitpunkt k der Wert der erwarteten künftigen Cashflows S_k mindestens um V_k größer ist als der Investitionspreis L (d.h. $S_k \geq L+V_k$), dann entscheide man sich für dieses Design, ansonsten nicht.

Wir haben gesehen, dass sich durch Realoptionen wesentlich flexiblere Entscheidungen modellieren lassen als mit der klassischen Kapitalwert-Methode [BeKa99, S. 14f]. Auch wurde durch das Beispiel gezeigt, dass nicht mehr Informationen notwendig sind um zu einer Entscheidung zu kommen und dass aber trotzdem verschiedene Ergebnisse aus den Methoden resultieren können.

2 Wertorientierte Software-Entwicklung mit Realoptionen

Nachdem die Konzepte der Realoptionen-Theorie vorgestellt und anhand eines Beispiels veranschaulicht wurden, werden nun die Konsequenzen für die Software-Entwicklung diskutiert. Zunächst wird der Einfluss der Parameter untersucht. Anschließend werden Entscheidungsheuristiken vorgeschlagen. Diese bilden dann die Grundlage für die Analyse von drei Software-Entwicklungsprozessen.

2.1 Sensitivitätsanalyse der Modell-Parameter

Die heutigen Strategien zur Software-Entwicklung basieren auf Heuristiken, die sich in der Vergangenheit bewährt haben. Es gibt keine fundierte Begründung für diese Heuristiken. Obwohl sie sich in der Vergangenheit bewährt haben (und dies wohl auch in Zukunft tun werden), ist es wünschenswert, zu verstehen warum diese Heuristiken funktionieren (und wann nicht) und ebenso auf diese Weise neue Heuristiken zu finden, die sich dann wiederum in der Praxis bewähren. Die Theorie der Realoptionen kann hier ein gutes Fundament für ein solches Vorgehen darstellen. [BaEm04] nennt vier Arten, wie Optionen zum Wert der Software beitragen können:

1. Akkumulierte Einsparungen durch Änderungen am System ohne die Architektur ändern zu müssen
2. Unterstützung der Wiederverwendung bestimmter Teile des Systems
3. Verbesserung des künftigen Wachstumspotenzials durch Reduktion der Entwicklungskosten in der Zukunft
4. Architektur stellt für Unternehmen eine Form des Eigenkapitals dar, welches einen Wettbewerbsvorteil darstellen kann

Im Folgenden werden wir uns an der Darstellung in [SuCS99, S. 244ff] orientieren, um den Einfluss der Parameter auf den Entscheidungsprozess zu untersuchen.

Einfluss des Ausübungspreises (Investitionskosten) An dem Beispiel des vorherigen Kapitels sieht man deutlich, dass der Ausübungspreis L (in diesem Fall die Investitionskosten in die Umstrukturierung) direkten Einfluss auf unsere Entscheidung haben kann. Aus Gleichung 17 sehen wir, dass eine Änderung von L keinen Einfluss auf S_0 hat. Jedoch hat sie direkten Einfluss auf $V_{1,1}$ und $V_{1,2}$ (siehe Gleichungen 18 und 19). Je höher die Investitionskosten L sind, desto geringer wird der Wert der Option an diesem Punkt. Überschreitet L gar die zu erwarteten Cashflows zu diesem Zeitpunkt, so wird die Option wertlos. Auch der Wert der Option zum Zeitpunkt 0 hängt von L ab. Dies ergibt sich aus Gleichung 20. Je größer L wird, desto mehr Wert verliert die Option. Aus Gleichung 20 erkennt man ebenso, dass sich L auf beide Argumente der max-Funktion negativ auswirkt. Sowohl der unmittelbare Gewinn durch Ausübung der Option $\max(S_0 - L, 0)$, als auch $V_{1,1}$ und $V_{1,2}$ hängen von L ab. In diesem Fall ist es so, dass wenn L kleiner als 1100 ist, eine direkte Investition zum Zeitpunkt 0 sinnvoll ist. Dies ergibt sich aus:

$$0,5 \cdot (3300 - L) = 2200 - L \tag{21}$$

Eine weitere Eigenschaft ist der nicht-lineare Zusammenhang zwischen den Parametern. Durch die max-Funktion ergibt sich eine zusammengesetzte lineare Funktion.

Das bedeutet zusammengefasst:

> Sind die Kosten der Investition hinreichend niedrig, so lohnt eine direkte Investition mehr als Abwarten. In diesem Fall sollte man sich umgehend dafür entscheiden.

Dies widerspricht zumindest teilweise der Entwurfsrichtlinie, dass Entwurfsentscheidungen so spät wie möglich getroffen werden sollten [SuCS99, S. 245]. Sind die zusätzlichen Investitionskosten eher gering, so kann diese Entscheidung sofort getroffen werden.

Einfluss der Cashflow-Varianz Ändern wir die Werte des Beispiels wie folgt ab:

$$S_{1,1} = 400 \tag{22}$$

$$S_{1,2} = 4000 \tag{23}$$

$$L = 1100 \tag{24}$$

Daraus ergibt sich:

$$S_0 = 0,5 \cdot S_{1,1} + 0,5 \cdot S_{1,2} = 2200 \tag{25}$$

Man sieht, dass sich der erwartete Gewinn nicht geändert hat. Nur das Aufwärts- und das Abwärts-Potential sind größer geworden. Die Änderung von L hat im vorherigen Beispiel bewirkt, dass gleich in der ersten Periode die Investition

erfolgen sollte (und nicht erst später, wie es sich mit $L = 1600$ ergeben hat). Nun kommt man zu folgenden Ergebnissen:

$$V_{1,1} = \max(S_{1,1} - L, 0) = 0 \tag{26}$$

$$V_{1,2} = \max(S_{1,2} - L, 0) = 4000 - 1100 = 2900 \tag{27}$$

Daraus folgt:

$$V_0 = \max(S_0 - L,\ 0,5 \cdot V_{1,1} + 0,5 \cdot V_{1,2}) = 1450 \tag{28}$$

Wegen $\max(S_0 - L, 0) < V_0$ folgt:

$$\tau(0) = 0 \tag{29}$$

Nun scheint es sinnvoller zu sein, die Investition abzuwarten und nicht das höhere Risiko einzugehen. Interessanterweise ist dies der Fall obwohl L und der erwartete Cashflow konstant geblieben sind. Das heißt, dass auch die Kapitalwert-Methode in Zustand 0 in einem positiven Kapitalwert resultiert hätte:

$$C_0 = -1100 + 0,5 \cdot 400 + 0,5 \cdot 4000 = 1100 \tag{30}$$

Daraus lässt sich folgende Heuristik ableiten:

> Je höher das Risiko der Investition ist (das heißt die Differenz der Cashflows in den verschiedenen Szenarien), desto eher ist es sinnvoll, die Investition zu verschieben und zu einem späteren Zeitpunkt eine Entscheidung zu treffen.

Einfluss der Szenario-Wahrscheinlichkeiten Betrachten wir nun den Einfluss der Wahrscheinlichkeiten, dass das jeweilige Szenario eintritt. In unserem Fall gibt es zwei Szenarien: das wartungsarme und das wartungsintensive. Bezeichnen wir die Wahrscheinlichkeit des wartungsarmen Szenarios mit p. Für unser Beispiel erhalten wir dann:

$$L = 1600 \tag{31}$$

$$S_{1,1} = 1100 \tag{32}$$

$$S_{1,2} = 3300 \tag{33}$$

$$S_0 = pS_{1,1} + (1-p)S_{1,2} = 1100p + 3300(1-p) \tag{34}$$

Für den Kapitalwert ergibt sich dann:

$$C_0 = -1600 + p \cdot 1100 + (1-p) \cdot 3300 = 1700 - 2200p \tag{35}$$

Damit gilt:

$$C_0 > 0 \Leftrightarrow p < \frac{17}{22} \tag{36}$$

Intuitiv aus oben stehender Beziehung oder mit ähnlicher Argumentation (siehe [SuCS99, S. 247] für eine detaillierte Herleitung) kann man folgende Heuristik herleiten:

> Je höher die Wahrscheinlichkeit des ungünstigen Szenarios ist, desto größer ist der Anreiz, die Entscheidung später zu treffen.

Zusammenfassung der Entscheidungsheuristiken Aus den vorherigen Abschnitten haben sich nun zusammenfassend folgende Entscheidungsheuristiken für die Software-Entwicklung ergeben:

- Wenn zu einem bestimmten Zeitpunkt k der Wert der erwarteten künftigen Cashflows S_k mindestens um den Wert der Option V_k größer ist als der Investitionspreis L (d.h. $S_k \geq L + V_k$), dann entscheide man sich für dieses Design, ansonsten nicht.
- Sind die Kosten der Investition hinreichend niedrig, so lohnt eine direkte Investition mehr als Abwarten. In diesem Fall sollte man sich umgehend dafür entscheiden.
- Je höher das Risiko der Investition ist (das heißt die Differenz der Cashflows in den verschiedenen Szenarien), desto eher ist es sinnvoll, die Investition zu verschieben und an einem späteren Zeitpunkt eine Entscheidung zu treffen.
- Je höher die Wahrscheinlichkeit des ungünstigen Szenarios ist, desto größer ist der Anreiz, die Entscheidung später zu treffen.

[SuCS99, S. 248] nennt zusätzlich:

- Je höher die Varianz der zukünftigen Kosten ist, desto größer ist der Anreiz, die Entscheidung später zu treffen.

2.2 Entwicklungsprozesse

In den letzten Jahrzehnten hat sich die Art der Software-Entwicklung stark geändert. Dies betrifft nicht nur die grundlegenden Technologien und Programmiersprachen, sondern auch die dahinterstehenden Konzepte. Ein grundlegender Trend ist beispielsweise die immer stärkere Kapselung einzelner Funktionseinheiten; zunächst auf feingranularer Ebene und dann immer gröbere Einheiten. Dies begann mit strukturierter Programmierung, Modulen, objektorientierter Programmierung und nun sind wir bei Komponenten angekommen. Eine solche Kapselung kann man auch als Kauf einer Realoption verstehen [BaEm04]. Sie erlaubt es, zu einem späteren Zeitpunkt, Änderungen mit deutlich geringerem Aufwand vorzunehmen, da (im Idealfall) nur noch eine einzige Funktionseinheit angepasst werden muss. Durch die Verschiebung der Implementierung kann flexibel auf Situationen reagiert werden, in denen eine hohe Varianz der erwarteten Cashflows gegeben ist, die Wahrscheinlichkeiten ungünstig verteilt sind oder die Implementierung der Funktionalität mit sehr hohen Kosten verbunden ist. Warum dies der Fall ist, wurde im vorherigen Kapitel mit der Theorie der Realoptionen an Praxisbeispielen erläutert.

Die Bewertung dieser Optionen erlaubt es auch, den Nutzen abzuschätzen, der durch den Kauf von Komponenten anderer Hersteller erzielt werden kann. Dies wiederum ist hilfreich für die Festlegung eines maximalen Einkaufspreises.

Ebenso kann die Umstrukturierung einer bestehenden Software als Kauf einer Option verstanden werden. Dadurch können Wartungsarbeiten in der Zukunft

schneller und damit auch kostengünstiger durchgeführt werden. Durch Realoptionen wird diese Option explizit gemacht und allein dies hilft bei der Entwicklung, da Risiken, Kosten und Nutzen erfasst und beobachtet werden, auch wenn eine Quantifizierung in der Praxis nicht immer einfach durchzuführen sein wird.

[SuCS99, S. 249] nennt einen weiteren Aspekt des Kapselungprinzips, bei dem Realoptionen helfen können. Das Kapselungsprinzip sieht vor, dass in einer Einheit die Aspekte des Systems zusammengefasst werden, die sich wahrscheinlich zusammen ändern könnten. Die Grenzen dieser Einheit sollen aber möglichst stabil bleiben [BaEm04]. Nun kann es aber auch Sinn machen, die Aspekte zu kapseln, bei denen die Wahrscheinlichkeit einer Änderung gering ist, aber bei denen ein hohes Risiko (d.h. sehr hoher Aufwand, die Änderung umzusetzen) gegeben ist. Dies kann sowohl die Struktur und die Topologie der Architektur, als auch die für die Implementierung verwendete Middleware betreffen [BaEm04].

Wasserfallmodell Das Wasserfallmodell stellt den klassischen Software-Entwicklungsprozess dar [Meye97, S. 924f]. Es verfolgt einen Gegenentwurf zum „code it now and fix it later"-Ansatz. Die folgenden Phasen laufen linear ab:

1. Machbarkeitsstudie
2. Anforderungsanalyse
3. Spezifikation
4. Grobentwurf
5. Feinentwurf
6. Implementierung
7. Validierung und Verifikation
8. Distribution

Beispielsweise kann die Entwurfs-Phase nicht beginnen solange die Spezifikations-Phase noch nicht abgeschlossen ist. Es wird also kein iterativer Ansatz verfolgt. Wenn ein Fehler in einer vorherigen Phase gemacht wurde, ist er nur sehr schwer zu beheben. Je früher der Fehler begangen wurde, desto schwerer (und aufwändiger) ist die Ausmerzung.

Aus Realoptionen-Sicht ist dieser Ansatz nicht sinnvoll. Alle Entwurfsentscheidungen werden sehr früh getroffen obwohl ein Abwarten in bestimmten Situationen angemessener wäre. Dies kann beispielsweise der Fall sein, wenn der erwartete Nutzen verschiedener Szenarien sehr weit auseinander liegt oder wenn die Implementierung extrem teuer ist. Im Wasserfallmodell werden also Risiken nicht direkt angegangen oder berücksichtigt. Außer in der ersten Phase (Machbarkeitsstudie) wird nicht gezielt in Optionen investiert, obwohl sie einen wesentlichen Wert der späteren Software darstellen könnten [SuCS99, S.251], [BaEm04]. In der Anforderungsanalyse und der Spezifikationsphase kann zwar natürlich in Optionen durch Flexibilität investiert werden, doch geschieht dies nicht gezielt, da (1.) Anforderungen sich im Laufe des Projektes nicht ändern können und daher keiner beobachteten Unsicherheit unterworfen sind und (2.) weil die Entwurfsentscheidungen sehr früh getroffen werden müssen und daher die Entscheidungen nicht aufgeschoben werden können (bis bessere Informationen zur Verfügung stehen).

Spiralmodell Das Spiralmodell ist eine Weiterentwicklung des Wasserfallmodells [Meye97, S. 925]. Im Spiralmodell werden die folgenden vier Schritte iteriert bis das Projektziel erreicht wurde:

1. Festlegungen der Ziele
2. Risikoanalyse und Beurteilung der Alternativen
3. Entwicklung und Test
4. Planung der nächsten Phase

Dabei werden bei der Risikoanalyse (2.) auch Prototypen entwickelt, Nutzerumfragen oder -experimente durchgeführt, um die Risiken zu mindern [SuCS99, S. 250]. In jeder Iteration soll man dem Projektziel ein Stück näher sein, auch wenn die Projektziele nicht mehr die gleichen Ziele sind wie am Anfang des Projektes. Durch dieses Vorgehen werden Risiken schrittweise reduziert und es wird ermöglicht (im Gegensatz zum Wasserfallmodell), Entwurfsentscheidungen später zu treffen (in der nächsten Iteration). Dadurch werden Risiken und Unsicherheiten direkt adressiert und sie können konkreter gehandhabt werden.

Das explizite Ziel eines Prozesses gemäß dem Spiralmodell ist die Wertmaximierung. Dieses steht im Einklang mit vorherigen Überlegungen, die eine stärker ökonomische und weniger technische Orientierung der Software-Entwicklung propagierten.

Optionen werden explizit durch die Entwicklung und Evaluierung von Prototypen geschaffen. Durch Prototypen kann aber auch schon früh Unsicherheit reduziert werden und zwar auf einem kontrollierten Weg. Der Hersteller wartet also nicht das Auftreten externer Ereignisse ab, sondern versucht selber die Risiken zu begrenzen. Die Kosten, die durch Entwicklung des Prototypen anfallen, sollten dann durch den zu erwarteten Informationszuwachs gerechtfertigt werden. Auch dies kann durch Optionen bewertet werden. Dies ist insbesondere der Fall wenn das Risiko der Implementierung hoch ist, d.h. ein sehr ungünstiges Szenario möglich scheint. Je höher dieses Risiko ist, desto wertvoller ist die Option. Also sollte auch mehr in den Prototypen investiert werden. Zusätzlich werden durch den iterativen Prozess regelmäßig Risikobewertungen durchgeführt und geprüft, ob die Durchführung der nächsten Phase zur Wertschöpfung beitragen kann und wenn ja, wie dies geschehen soll.

Weiterhin werden durch den iterativen Prozess die Investitionen gering gehalten solange größere Risiken vorhanden sind. Dies steht im Einklang mit den Heuristiken, die im vorherigen Kapitel aus den Überlegungen zur Realoption-Theorie abgeleitet wurden.

[SuCS99, S. 253] nennt ebenso, dass es Sinn machen kann, in ein Projekt zu investieren, obwohl der Kapitalwert negativ ist, wenn durch eine kleine Investition ein sehr hoher Return erzielt werden könnte. So kann die frühe Adaption eines neuen Standards sinnvoll sein, auch wenn sich dieser noch nicht durchgesetzt hat, aber eine schnelle Adaption möglich scheint (dies war beim Konkurrenzkampf zwischen Blu-ray und HD DVD nicht der Fall). Diese interessante Idee wurde auch von einigen Händlern in eher populärwissenschaftlicher Literatur vertreten [Tale07].

Extreme Programming Das Vorgehensmodell „Extreme Programming“ (XP) wird von Beck und Andres in [BeAn06, Kapitel 1] folgendermaßen beschrieben:

> XP is a lightweight methodology for small-to-medium-sized teams developing software in the face of vague or rapidly changing requirements. [...] XP adapts to vague or rapidly changing requirements. XP is still good for this situation, which is fortunate because requirements need to change to adapt to rapid shifts in the modern business world. However, teams have also successfully used XP where requirements don't seem volatile, like porting projects.

XP geht davon aus, dass Änderungen der Anforderungen stetig eintreten, da erst im Laufe des Projektes die Anforderungen wirklich klar werden. Darum wird auf einfache Lösungen Wert gelegt. Zusätzlich werden Änderungen stetig in die Software integriert und getestet. Dies soll aber mit einer möglichst einfachen, wenig generischen Architektur umgesetzt werden. Die Folge ist stetiges Refactoring und Änderungen an der Architektur.

Aus Realoptionen-Sicht werden im XP-Vorgehensmodell wenige Optionen geschaffen, da durch wenig generische Architekturen nicht auf Flexibilität hin entwickelt wird. Außerdem erfolgt keine regelmäßige Risiko- und Alternativenbewertung, die für die Bestimmung der Wertentwicklung notwendig wäre. Positiv ist anzumerken, dass durch stetige Interaktion mit dem Kunden und der Implementierung von Prototypen aktiv die Suche nach neuen Informationen betrieben wird. Diese Informationen könnten dann für die Wertevaluierung und Planung des Projektes genutzt werden.

Diese negative Bewertung trifft jedoch nicht auf alle flexiblen Vorgehensmodelle zu. [Hutt06] und [RaDa08] beispielsweise argumentieren im Allgemeinen für den Einsatz von flexiblen Vorgehensmodellen aus Options-Perspektive. Insbesondere Agile Methoden nutzen Realoptionen gezielt aus und erlauben einen besseren Umgang mit der Unsicherheit der Anforderungen, insbesondere weil es möglich ist, Entwurfsentscheidungen später zu treffen [MaMa07].

2.3 Zeitpunkt der Markteinführung

Eine weitere Frage bei der Software-Entwicklung ist, ob das Produkt auf den Markt gebracht werden sollte, auch wenn es noch nicht vollständig implementiert wurde, und wenn ja, wann. Die Entwicklungszyklen sind immer kürzer geworden und Produkte werden immer „unfertiger“ auf den Markt gebracht (im „beta“-Stadium). Gemäß der Optionen-Theorie wird am meisten Wert erzeugt, wenn das Produkt konkurrenzlos auf den Markt kommt. Wenn ein Konkurrenzprodukt schon auf dem Markt ist, kann nur ein Teil des bestehenden Marktes abgeschöpft werden, aber nicht der ganze Markt [SuCS99, S. 255]. Dies heißt natürlich nicht, dass man fehlerhafte oder schlechte Produkte auf den Markt bringen sollte. Aber eine Reduktion des Funktionsumfangs zugunsten eines früheren Markteintritts kann durchaus berechtigt sein.

3 Fazit und Schlussfolgerung

Wir haben gesehen, dass durch die Realoptionen-Theorie einige Einsichten in den Entscheidungsprozess in der Software-Entwicklung gewonnen werden können. Sie kann genutzt werden, um den Entwicklungsprozess stärker wertorientiert auszurichten. Insbesondere stellt sie eine Verbesserung im Vergleich zur Kapitalwert-Methode da, weil sie wesentlich flexiblere Entscheidungsprozesse ermöglicht, die beispielsweise auch die Varianz der Cashflows verschiedener Szenarien berücksichtigt.

Folgende Heuristiken für die Software-Entwicklung konnten durch die Realoptionen-Theorie begründet werden:

- Je größer die Cashflow-Varianz der Szenarien, desto größer der Wert der Option und desto später sollte die Option eingelöst werden
- Je größer die Investition (Ausübungspreis), desto geringer der Wert der Option und desto später sollte die Option eingelöst werden
- Je größer die Wahrscheinlichkeit des ungünstigen Szenarios, desto geringer der Wert der Option und desto später sollte die Option eingelöst werden
- Je früher der Markteintritt erfolgt, desto größer sind die zu erwarteten Cashflows

3.1 Grenzen des Ansatzes

Obwohl der Realoptionen-Ansatz sehr nützlich bei der Entscheidungsfindung ist, ergeben sich in der Praxis doch Probleme [SuCS99, S. 257 – 259]. Zunächst ist die Verwendung von Arbitrage-basierenden Bewertungsmethoden oft nicht möglich, da die Werte nicht frei gehandelt werden. Die daher notwendige Schätzung der künftigen Cashflows ist eine weitere Schwierigkeit. In der Praxis ist es nicht so einfach, diese anzugeben und ihnen Wahrscheinlichkeiten zuzuordnen. Es ist jedoch anzumerken, dass durch das alleinige Nachdenken über die Parameter das Entscheidungsproblem besser verstanden werden kann. Außerdem können verschiedene Parameterkombinationen berechnet werden, um die Sensitivität dieser festzustellen. Oft dürfte sich dann schon eine Entscheidung als den anderen überlegen abzeichnen.

Weiterhin ist die Definition eines Entscheidungsproblems an sich in der Praxis schwierig, weil die Entscheidungen meist sehr eng verwoben sind und Abhängigkeiten bestehen. Es ist daher schwierig ein, eine einzige Entscheidung so zu isolieren, dass sie mit dem Realoptionen-Ansatz betrachtet werden kann. Andererseits betont die Realoptionen-Theorie den Wert von Flexibilität und versucht ihn zumindest zu quantifizieren.

3.2 Ausblick

Wir haben gesehen, dass sich Realoptionen nutzen lassen um bestehende oder neue Entwicklungsheuristiken zu untersuchen. Um die Praxistauglichkeit dieses Ansatzes zu untersuchen ist es sicher notwendig, es bei einem konkreten Projekt

zu untersuchen. Hierfür ist sicher auch eine Unterstützung durch Werkzeuge sinnvoll. Zusätzlich ist auch die Schulung der Teammitglieder notwendig. Ansonsten kann der Realoptionen-Ansatz nicht seine volle Wirkung entfalten.

Realoptionen sind daher sicher ein guter Weg hin zu einem „Software-Entwurf als Investition“ [BoSu00], insbesondere weil sie es ermöglichen den Wert eines Software-Produkts zu quantifizieren.

Literatur

BaEm04. Rami Bahsoon und Wolfgang Emmerich. Evaluating Architectural Stability with Real Options Theory. *Software Maintenance, IEEE International Conference on*, Band 0, 2004, S. 443–447.

BeAn06. Kent Beck und Cynthia Andres. *Extreme programming explained.* Addison-Wesley. 2. Auflage, 2006.

BeKa99. Michel Benaroch und Robert J. Kauffman. A Case for Using Real Options Pricing Analysis to Evaluate Information Technology Project Investment. *Info. Sys. Research*, 10(1), 1999, S. 70–86.

BoSu00. Barry W. Boehm und Kevin J. Sullivan. Software economics: a roadmap. In *The Future of Software Engineering, 22nd International Conference on Software Engineering.* ACM Press, 2000, S. 319–343.

Erdo99. Hakan Erdogmus. Valuation of complex options in software development. In *First Workshop on Economics-Driven Software Engineering Research*, 1999.

Hutt06. Harald Hutter. Anwendung des Realoptionenansatzes bei Softwareentwicklungsprojekten mit flexiblen Vorgehensmodellen, Juni 2006. Seminararbeit zur LV Informationswirtschaft VI an der Universität Wien.

KoOv07. Robert W. Kolb und James A. Overdahl. *Futures, options, and swaps.* Blackwell. 5. Auflage, 2007.

MaMa07. Chris Matts und Olav Maassen. Real Options Underlie Agile Practices, InfoQ, June 2007.

Meye97. Bertrand Meyer. *Object-oriented software construction.* Prentice Hall. 2. Auflage, 1997.

Myer77. Stewart C. Myers. Determinants of corporate borrowing. Working papers 875-76, Massachusetts Institute of Technology (MIT), Sloan School of Management, 1977.

RaDa08. Zornitza Racheva und Maya Daneva. Using measurements to support real-option thinking in agile software development. In *APOS '08: Proceedings of the 2008 international workshop on Scrutinizing agile practices or shoot-out at the agile corral*, New York, NY, USA, 2008. ACM, S. 15–18.

SuCS99. Kevin Sullivan, Prasad Chalasani und Vibha Sazawal. Software Design as an Investment Activity: A Real Options Perspective. In *University of Virginia Department of Computer Science*, 1999.

Tale07. Nassim Taleb. *The Black Swan.* Random House. 2007.

Assessment Methods for Software Product Lines

Philipp Meier

Advisor: Henning Groenda

Abstract Architecture is at the center of reuse within a software product line. The quality of the reference architecture directly affects the quality of all products in the product line and is thus one of the most critical issues in product line engineering. Architecture assessment is an essential tool for ensuring a quality architectural design both for current and for future requirements. This paper shows the key concepts and requirements of software product line architecture assessment and provides an overview over the available methods and their applicabilities.

1 Introduction

In any complex software system, the architecture is of central importance, as it has a big impact on extra-functional properties of the finished system. A *software product line* is defined by Clements and Northrop as

> "a set of software-intensive systems sharing a common, managed set of features that satisfy the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way." [CN01, p. 5]

The central goal of a software product line is the efficient reuse of common entities and components. As this can best be achieved on an architectural level, the role of the architecture is especially important in this context. A *reference architecture* captures the commonalities and the variability between the individual product architectures. The quality of the reference architecture directly influences the quality of the product line as a whole. As new products are added to the family, their architectures are derived from the reference architecture, inheriting its quality and deficiencies.

Architecture assessment, or *architecture evaluation*, is "the systematic examination of the extend to which an architecture fulfills requirements" [ES05]. Apart from ensuring the quality of an architecture directly, evaluation benefits stakeholder communication and generally supports the decision making process [ES05]. The architectural documentation is also improved, e.g. as risks and important design tradeoffs are recorded in the process. Evaluation is equally important during the design of a new architecture as it is during the evolution of existing architectures. The artifacts and information available during the evaluation process differ depending on the situation.

Evaluating an architecture without a specialized method is possible and may even be practical for smaller projects. Using a specialized method, however,

has important benefits. A good method can ensure that the right questions are asked early [KKC00] and that all necessary information is captured and stored for later use. Furthermore, a mature method captures the experiences of a large number of projects, reducing the risk of making mistakes twice.

In the field of architecture assessment a number of such proven methods have emerged, each targeting a different set of questions or priorities. Product lines introduce new requirements for architecture assessment and new methods have emerged to deal with them. However, which of the new methods prove to be most successful in practice is still unclear.

The contribution of this paper is (i) to show the key concepts and requirements of software product line architecture assessment and (ii) to provide an overview over the available methods and their applicabilities.

This paper is structured as follows. Section 2 introduces the underlying concepts of architecture assessment. Section 3 describes a selection of well-established *assessment methods for single product architectures*, normally referred to as *software architecture assessment methods*. The focus is on the most widely used and proven method ATAM (Section 3.1 on page 74). In section 4 product-line-specific concepts and issues are explained, followed by an overview of existing *product-line-specific architecture assessment methods* in section 5. The focus is on the recent method HoPLAA (Section 5.1 on page 81), as it represents an holistic approach, integrating the assessment process of the reference and the individual product architectures. The other methods are grouped into those, which focus more on the reference architecture (Section 5.2 on page 84), and those, that focus more on a certain set of quality attributes (Section 5.3 on page 87). For each method, the purpose, existing underlying concepts, specific concepts, steps, output, and organizational and process management are pointed out. Items for which not enough information was available are omitted. Section 6 summarizes the paper.

2 Architecture Assessment

Before introducing software architecture assessment, it is helpful to look at the issues that are actually addressed by an architecture. According to [vdLSR07], the primary elements, or concerns of an architecture are:

architecturally significant requirements The requirements that significantly affect the architecture. This is a subset of all system requirements and is kept as part of the architecture documentation to provide detail on which basis the architecture was designed.

conceptual architecture The key concepts of the architecture. They capture the problem domain and explain the domain elements and their relations in an implementation independent manner.

structure The decomposition of the system into its major components and their dependencies and relationships.

texture Rules and guidelines that govern the design and evolution of the architecture. They help to preserve the architecture's core concepts

and to solve problems dealing with how to interpret the architecture documentation.

It is important to keep in mind which of these concerns any of the architectural artifacts addresses.

Furthermore, it is helpful to understand a number of underlying concepts of architecture assessment. The central concept most assessment methods focus on is that of a quality attribute. *Quality attributes* are the extra-functional properties of a system (e.g. performance, security, reliability). Quality attributes can be categorized into operational and non-operational quality attributes. *Operational quality attributes* are those that can be observed during execution [ES05]. A related concept is that of a quality attribute characterization. A *quality attribute characterization* captures the knowledge about a quality attribute in compact form. In [KKC00, p. 9ff], a separation of each characterization into three sections is proposed:

- **external stimuli**: Events relevant to the quality attribute that cause the system to react (e.g. a user action) or that make changes to the system necessary (e.g. a feature request).
- **responses**: Measurable or observable reactions to the events listed in the external stimuli section (e.g. mean time between failures, throughput).
- **architectural decisions**: Architectural components, connectors and their properties that are known to significantly influence the reactions in the responses section (e.g. a type of scheduling policy, level of redundancy of a component).

It is desirable to first build a set of quality attribute characterizations before assessing an architecture, as it forces the stakeholders to concretize otherwise rather abstract quality requirements. A good starting point for this task are quality attribute characterizations from previous projects or from the research community (see Appendix A of [KKC00]), that can then be tailored to suit the project.

An area of an architecture in which the design potentially determines a quality attribute is called *sensitivity point*. If more than one quality attribute is affected, it is called a *tradeoff point*, because the design might not equally satisfy all affected quality attributes and a compromise might have to be made [KKC00], [OM07].

Architecture evaluation can be divided into qualitative and quantitative evaluation [ES05]. *Qualitative evaluation* is a subjective approach based on questioning techniques and typically involves scenarios. *Quantitative evaluation* is based on concrete measurements on the actual software or on representative models. Most assessment methods are of qualitative nature. One reason is that it is usually costly to model a system for a quantitative evaluation. Another is that architecture evaluation is often applied during the design and requirements phases where no measurable system is available. Furthermore, a qualitative approach requires less technical knowledge and it is is easier to involve all stakeholders to express their requirements towards the system.

A *scenario* is a "short statement describing an interaction of one of the stakeholders with the system" [KKC00, p. 13]. The maximum duration within which the interaction must complete is also included in the scenario. The goal is to describe the interaction as concrete as possible and to allow anyone to verify the scenario without deeper understanding of the problem domain (i.e. what is considered "normal"). After [KKC00], scenarios can be categorized into three types:

- **use case scenarios**: They describe a user interaction with the finished and running system. Use case scenarios typically relate to operational quality attributes.
- **growth scenarios**: They describe typical future changes to the system. Growth scenarios typically relate to non-operational quality attributes.
- **exploratory scenarios**: They describe situations that stress the system. They help to identify the boundaries and limits of a design and to identify implicit assumptions.

By finding scenarios for all categories it is ensured, that the system is tested from many angles, increasing the chances to expose potentially risky design decisions.

3 Established Single Product Architecture Assessment Methods

In this section a small selection of established architecture assessment methods are described. ATAM is covered in detail, SAAM, ARID and CBAM are described only briefly.

3.1 ATAM: Architecture Trade-off Analysis Method

ATAM [KKC00] is developed by the Software Engineering Institute (SEI) at the Carnegie Mellon University. It "has evolved from SAAM (Section 3.2 on page 78) by extending it to the analysis of multiple quality attributes" and it "has the best documented record of successful usage in practice" [OM07, p. 310]. For its success, the HoPLAA method (Section 5.1 on page 81) is based on ATAM and thus ATAM is covered in detail as well. ATAM evaluates if the design decisions satisfy the corresponding requirements. Risks are identified, as well as sensitivity and tradeoff points (Section 2 on page 72). It is used to guide further development by improving architectural awareness and documentation quality. Requirement statements about the architecture and design are developed and improved. According to [KKC00] ATAM has several key strengths. It can be applied early in development during the requirements and design phase. It is inexpensive and quick, if the architecture is well documented or modeled. It does not limit its applicability to one or two quality attributes, but covers all important attributes and trends.

purpose "The purpose of the ATAM is to assess the consequences of architectural decisions in the light of quality attribute requirements" [KKC00, p. 2]. In essence it reveals, which design decisions affect which quality attributes.

existing underlying concepts ATAM is based on the concepts of quality attribute characterization, scenarios (Section 2 on page 72), and architectural styles and patterns.

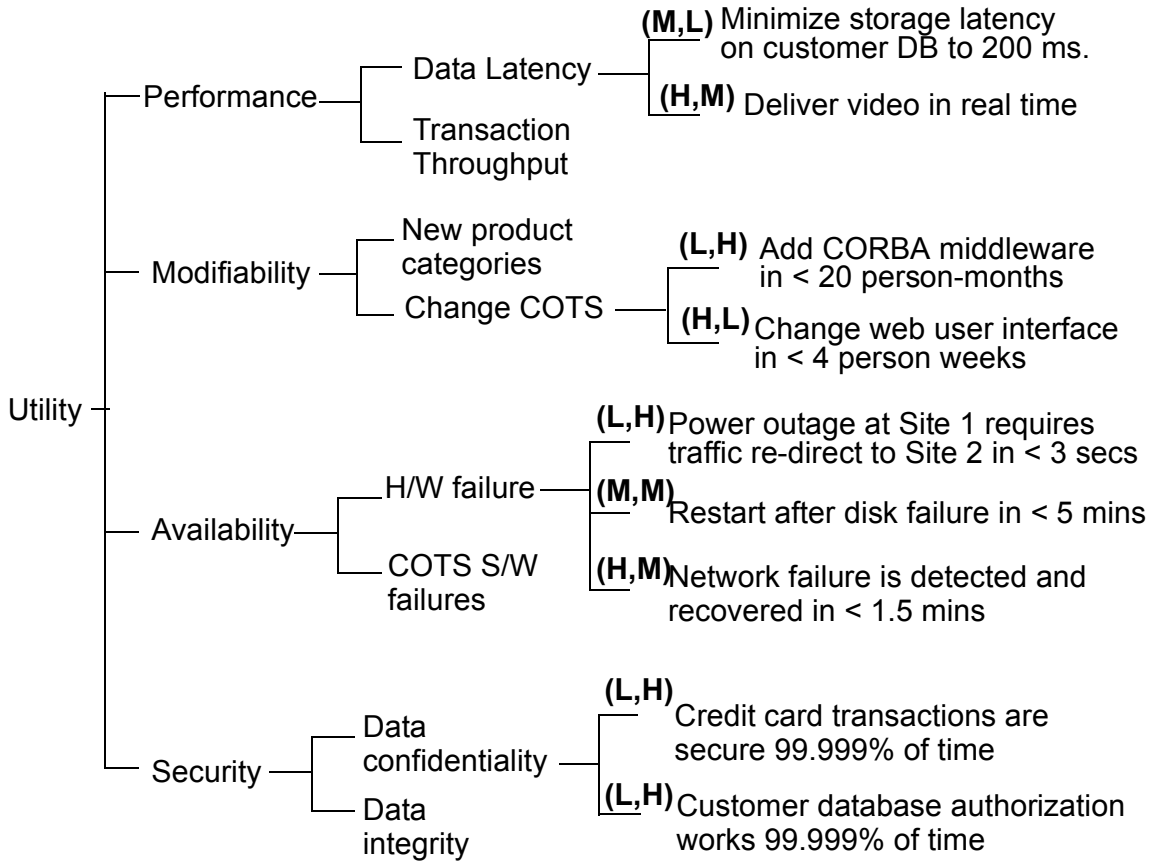
specific concepts ATAM introduces several interesting concepts. One is that of *utility trees*. They "provide a top-down mechanism for directly and efficiently translating the business drivers of a system into quality attribute scenarios" [KKC00, p. 16]. They are used in a smaller group context, usually architects and project leaders, and represent a structural and prioritized list of scenarios. Instead of prioritizing the rather abstract quality attributes like performance and modifiability directly, each quality attribute is broken down into sub-factors that determine how well a quality attribute is satisfied. These sub-factors are further broken down into concrete scenarios, each with two ratings attached. One for the importance of the scenario and one for the expected difficulty of achieving the required level of quality. The ratings are usually only coarse (e.g. high, medium, low), as a finer rating reduces the repeatability of the results. The utility tree guides and structures further analysis as more stakeholders are involved. Figure 1 on page 76 shows an example utility tree.

To gather a more complete set of scenarios, *scenario brainstorming* [KKC00, p. 18] is used. During a group meeting of all stakeholders, scenarios are gathered through brainstorming and later prioritized through voting. In a final step, the utility tree is validated and extended by including newly discovered scenarios and comparing the purpose and priority of existing scenarios.

The idea of concretizing quality attributes through sub-factors and scenarios is also applied to the architectural patterns that are documented or uncovered during the evaluation. The effect of a pattern on a particular quality attribute is analyzed in detail by developing scenarios or by using a suitable modeling technique. Such a pair of architectural pattern and detailed quality attribute analysis is called *attribute-based architectural style*, or ABAS [KKC00, pp. 19/20].

steps The ATAM can be divided into nine distinct steps. As presented in [KKC00, p. 25ff], they can be modified to suite different team sizes or different amounts of available information about the architecture. They also do not need to be executed strictly in order.

1. **Present the ATAM**: In this step the evaluation team presents the ATAM to the stakeholders. The goal is to clarify for everyone exactly how information will be gathered, what it will be used for, and who will record it.
2. **Present business drivers**: In this step the project manager presents the system from a business perspective. This includes the system's most important functional requirements, constraints, business goals and context, major stakeholders, as well as the architectural drivers. The goal is for all participants to understand the system under evaluation.



L: low, M: medium, H: high
Scenario parameters: (i) importance for the success of the system and
(ii) expected difficulty of achievement

Figure 1. Example utility tree [KKC00, p. 17]

3. **Present architecture**: In this step the lead architect presents the architecture in more detail, involving the technical constraints, interactions with surrounding systems and known architectural approaches used to meet quality attribute requirements. Also it is specified, what information needs to be collected before starting the evaluation.
4. **Identify architectural approaches**: In this step the architect identifies architectural patterns, especially those that are inherent in the architecture design, but not yet part of its documentation. The analysis team records the identified patterns.
5. **Generate quality attribute utility tree**: In this step the architecture team, manager and customer representatives build the utility tree. The goal is to identify the most critical quality attributes and corresponding characterization scenarios to focus and guide the limited evaluation effort.
6. **Analyze architectural approaches**: In this step the whole evaluation team tries to associate the highest priority quality attribute requirements of the utility tree with the architectural patterns that are relevant to achieving them. During the process, any discovered risks, non-risks, sensitivity and tradeoff points are recorded. The analysis in this step is not complete, but matches the available information.
7. **Brainstorm and prioritize scenarios**: In this step all stakeholders brainstorm for scenarios of all three kinds (use case, growth and exploratory scenarios) (Section 2 on page 72). In a second step the scenarios are prioritized through voting. The goal is to probe the system from all angles and to discover any inconsistencies in the requirements. Inconsistencies between the utility tree and the new prioritized list of scenarios are to be explained or corrected.
8. **Analyze architectural approaches**: This is a testing step. The highest ranked scenarios from the brainstorming session are mapped to matching architectural patterns. If no satisfactory match is found, a reiteration starting at step four is necessary, as the previous steps were lacking.
9. **Present results**: In this step the ATAM outputs are finalized and a summary is presented to all stakeholders.

output The ATAM output artifacts are the documentation of the architectural patterns, a prioritized set of scenarios, a set of attribute-based questions, the utility tree and lists of all discovered risks, non-risks, sensitivity and tradeoff points.

organizational and process management Typically the ATAM is carried out in two phases [KKC00, p. 39ff]. In the first phase, the architect team and a small subset of important stakeholders carry out all ATAM steps up to step six, focusing on understanding the system and identifying missing information to perform a full analysis. Before proceeding with the second phase involving all stakeholders, the first group gathers all missing information and ensures sufficient architecture documentation. This may take several weeks. The second phase restarts at step one carrying through to

the end. Although the actual agenda may differ depending on the size of the project and the size of the evaluation team, one full day for phase one and two full days for phase two should be considered.

[KKC00, p. 45ff] includes a detailed sample evaluation of BCS, a battlefield control system. It can easily be followed, as the description is focused around the method, rather than around the actual system that was evaluated. It is less a critical evaluation of ATAM, but rather serves to show what a complete ATAM evaluation might look like in practice.

[OM07, p. 310] states that "ATAM has a well-documented record of successful use in a wide range of applications, including automotive systems, financial services software, simulation systems, industrial applications, web-based systems" and others, referencing the corresponding papers.

More information can be found in [CKK01, p. 43ff].

3.2 SAAM: Software Architecture Analysis Method

SAAM [KBWA94] is a scenario based architecture assessment method developed at the Software Engineering Institute (SEI). Even though newer and more advanced methods exist, many of them are based on it or borrow ideas from it. Its purpose is to analyze one or more software architectures regarding one quality attribute at a time. In [KBWA94] the method is explained briefly and documented mostly through three example case studies in which user interface architectures are evaluated for modifiability. More information can be found in [CKK01, p. 211ff]. SAAM consists of five steps [KBWA94, p. 82]: characterize a functional partition for the domain, map this functional partition onto the architecture's structural decomposition, choose a set of quality attributes with which to assess the architecture, choose a set of scenarios testing the desired quality attributes, and evaluate the degree to which each architecture provides support for each task.

3.3 ARID: Active Review for Intermediate Designs

ARID [Cle00] is an assessment method developed at the Software Engineering Institute (SEI) that focuses on the quality of the early design of the individual architecture components. It is complementary to ATAM (Section 3.1 on page 74), working at a finer granularity. It ensures that the design sufficiently supports the implementation. For this, active design reviews (ADR) and scenarios are employed. The method consists of two phases, each containing several steps. In [Cle00] ARID is explained in detail and the method description is accompanied by practical observations and improvement ideas.

3.4 CBAM: Cost Benefit Analysis Method

CBAM [KAK02] is another assessment method complementary to ATAM (Section 3.1 on page 74). It was developed at the Software Engineering Institute

(SEI) and focuses on economic and management aspects of an architecture. It depends on the ATAM output and supports the stakeholders in choosing the right architectural decisions elicited through ATAM in the perspective of cost, benefit, schedule and risk. It uses scenarios and consists of two iterations of several steps each, which are explained in detail. [KAK02] includes a case study of a NASA ECS project about processing of earth observation data collected by satellites. More information of integrating ATAM and CBAM can be found in [NBC+03].

4 Product-line-specific Issues

In this section issues of architecture assessment that come with product line engineering are introduced. They include new requirements, variability, and new possibilities in evaluation time.

4.1 Additional Requirements

As opposed to single product evaluation, architecture evaluation in a product line context has to incorporate a number of new requirements. Most notably not one but a number of dependent architectures have to be dealt with. By introducing a reference architecture from which the individual product architectures are derived, both their commonalities and their variability are captured. Figure 2 on page 80 outlines the relation between the reference architecture and the product architectures, and classifies requirement attributes in this new context. They are classified into product line quality attributes, domain-relevant attributes, and functional requirements or common behavior [ES05]. *Product line quality attributes* are those attributes that allow the reference architecture to be the basis for current and future products in the domain. They define the reference architecture's variability and include modifiability, configurability, and related quality attributes. *Domain-relevant attributes* are quality attributes that are especially important for the whole domain and thus need to be addressed in the reference architecture. Depending on the domain, they could for example include performance, security, and reliability. *Functional requirements or common behavior* represent behavior common to all or most of the products. The set of assets implementing this behavior specific to the domain is called the *domain platform* (see domain engineering [vdLSR07, p. 49ff]).

The impact of this situation on architecture evaluation is that product architecture decisions can invalidate reference architecture quality attribute requirements, and vice versa, complicating the assessment. Independent assessment of the reference architecture and the family's product architectures potentially misses these connections and tradeoffs [OM07]. Furthermore, scenario-based methods get complicated as the number of scenarios is heavily increased [OM07]. However, due to their close relationship, assessment effort spent on the reference architecture can potentially be reused for product architecture assessment and amortized over the whole product family [OM07].

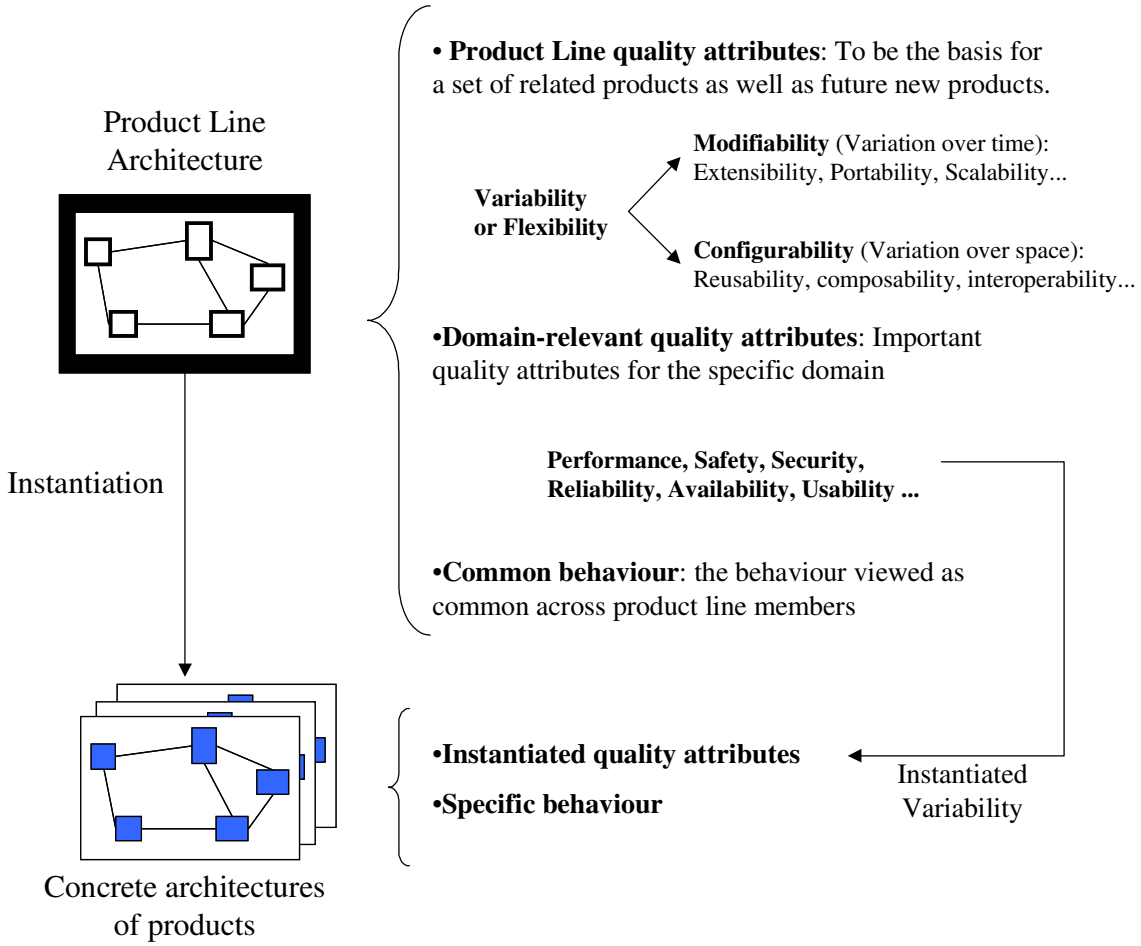


Figure 2. Classification of Product Line Requirements [ES05]

4.2 Variability

The reference architecture supports both commonalities and variation among the product architectures. The variability enabled through the reference architecture should be made explicit and documented as precise as possible. Points in the architecture or decisions allowing optionality or the selection between a number of variants are called *variation points*. The set of variation points and their variants can be formalized in a *variation model* [vdLSR07, p. 40]. Basic techniques to implement variability in an architecture include adaption, replacement and extension. [vdLSR07, pp. 40-42]. A sensitivity point (Section 2 on page 72) in the reference architecture that contains at least one variation point is called an *evolvability point* [OM07, p. 314]. As the variant of an evolvability point chosen for a product architecture critically influences the quality guarantees of the reference architecture, this decision should not be made solely in a product context. One solution is to provide guidelines for the evolvability points, describing potential conflicts and allowing the architect to make a good design decision without having to understand the complete reference architecture [OM07, p. 314].

4.3 Evaluation Time

Architecture evaluation can take place at different times and phases in the development cycle. Both the reference architecture and the product architectures can be evaluated. Single product architecture evaluation normally takes place during design and during evolution. Product line engineering additionally introduces new situations at which an architecture assessment makes sense [ES05]. Prior to instantiating a new product line, the existing product architectures can be assessed to identify which architecture fits best as a basis for the reference architecture to be introduced. Whenever a new product is derived from the reference architecture and added to the product line, an evaluation can discover which variants best match the product's quality requirements. Finally, an architecture evaluation can be used to keep the reference architecture, which is evolved through domain engineering, and the product architectures in sync. It is up to the architects to choose the best time to execute the evaluation, as evaluating the architecture at all possible times is likely to be too costly.

5 Product-line-specific Architecture Assessment Methods

In this section a number of product line architecture assessment methods are described. HoPLAA is covered in detail. The remainder of the methods are explained in less detail and are grouped into those that focus more on the reference architecture and those that focus more on a specific set of quality attributes.

5.1 HoPLAA: Holistic Product Line Architecture Assessment method

HoPLAA [OM07] is an adaption of ATAM (Section 3.1 on page 74), designed to cope with the additional complexities introduced by product line engineering. Instead of applying a single product architecture assessment method to the reference architecture and each product architecture, HoPLAA integrates the assessment of the whole product line. The reference architecture is assessed first and its output is tailored to support the individual product architecture evaluations to follow.

purpose Like ATAM, the purpose of HoPLAA is to identify and document how architecture design decisions affect quality attributes, and to determine, to which degree they satisfy the requirements. HoPLAA also identifies risks and tradeoffs between architectures, especially between the reference architecture and each of the product architectures.

existing underlying concepts HoPLAA is based on the concepts of scenarios, quality attribute characterizations and the utility trees introduced through ATAM.

specific concepts HoPLAA extends the utility tree concept by adding a rating of generality, one of mandatory, alternative or optional, to the attached scenarios. This helps to identify which of the often numerous scenarios are truly relevant for the reference architecture, without limiting the stakeholders during scenario elicitation.

steps The HoPLAA method is composed of two stages. In the first stage, only the reference architecture is evaluated. In the second stage, the individual product architecture follow. Phase I consists of the following steps [OM07, pp. 314/315]:

1. **Present the HoPLAA Stage I**: The evaluation team presents the HoPLAA stages and steps to the stakeholders, focusing on stage I.
2. **Present the product line architectural drivers**: In this step the project manager presents the product line from a business perspective. This includes the product line scope definition and the commonalities and variabilities between the individual products of the product line.
3. **Present the product line architecture**: The architect team presents the reference architecture in more detail.
4. **Identify architectural approaches**: The evaluation team identifies architectural patterns used in the reference architecture. The patterns are documented, but analyzed later.
5. **Generate, classify, brainstorm, and prioritize quality attribute scenarios**: In this stage, the extended utility tree is built. Scenarios are elicited by the stakeholders through brainstorming and classified into the utility tree. The goal is to identify scenarios of high generality, either scenarios common to all products in the family and the scenarios that capture family-wide quality attribute requirements. All other scenarios are added to the utility tree, but are analyzed in stage II. The gathered scenarios are prioritized by the sum of all three ratings assigning

an index value between one and tree for each. In a second iteration, more scenarios are elicited from the top-ranking scenarios, and the most generic scenarios are prioritized by voting.

6. **Analyze architectural approaches/generic scenarios**: In this step, the high-priority generic scenarios from the previous step are analyzed. During the process, architectural risks, non-risks, sensitivity points, tradeoff points and evolvability points are identified and recorded. For each evolvability point, an evolvability guideline is developed and recorded.
7. **Present results**: A stage I report is prepared containing the list of architectural approaches, utility tree, generic scenarios, product-specific scenarios, risks and non-risks of the reference architecture, sensitivity points, tradeoff points, evolvability points, evolvability guidelines and any risks endangering the product line mission.

Phase II consists of the steps [OM07, pp. 315/316]:

1. **Present the HoPLAA Stage II**: Recap the HoPLAA steps focusing on stage II.
2. **Present the architectural drivers**: Provide an overview of the reference architecture and present the driving requirements of the product architecture under evaluation. Provide a description of how this product varies from the reference architecture in respect to functional and quality goals.
3. **Present the product architecture**: Present the product architecture, emphasizing areas that have recently been enhanced through the realization of variation points.
4. **Identify architectural approaches**: In this step the architect team identifies architectural patterns used in the product architecture. If a variation point is realized using a different pattern than in the reference architecture, the rationale behind the decision is recorded as well. The stakeholders identify and document the variation points that have been realized as variants in this product architecture.
5. **Generate, brainstorm and prioritize quality attribute scenarios**: In this step, the stakeholders create a list of scenarios relevant to this product architecture. First, relevant scenarios are extracted from the utility tree of stage I, then more product-specific scenarios are elicited through brainstorming and prioritized by voting. Using the list of variants, the evolvability points realized in the product architecture are identified and relevant mandatory quality attributes are extracted. The results are either recorded by extending the utility tree from stage I or by creating a new utility tree special to this product architecture.
6. **Analyze architectural approaches**: In this step the architect team need to analyze the two classes of scenarios gathered in the previous step. It is analyzed, if the generic quality attribute scenarios continue to be satisfied by the employed variants. The remainder of the scenarios is analyzed and lists of architectural risks, non-risks, sensitivity points and tradeoff points are assembled and recorded.

7. **Present results**: A report is prepared, containing the same elements as the final stage I report, except evolvability points and evolvability guidelines.

It is unclear why the ATAM steps five and six, serving to probe the architecture documentation for missing information, have been dropped. However, it is noted, that "certain preliminary and post-evaluation activities [...] are recommended to set the stage for, and conclude the activities of this method" [OM07, p. 316]. This could mean that other quality assurance methods ensure a satisfactory level of architecture and requirements documentation prior to starting the evaluation.

output The output of HoPLAA is a superset of that of ATAM. The reference architecture evaluation output of phase I is extended to contain a list of evolvability points (Section 4.2 on page 79) and corresponding evolvability guidelines. Phase II output corresponds to the ATAM output.

organizational and process management While evaluating the reference architecture will take more time using HoPLAA compared to ATAM, the proceeding product architecture evaluations reuse information of phase I and the overhead is amortized over the whole product family. Depending on the size of the evaluation team, phase II could be parallelized.

The method description in [OM07] is followed by a case study about btLine, a product in the electronic payment network domain, in which the advantage of assessing inter-architectural trade-offs is shown.

5.2 Focus on Reference Architecture

This subsection covers the product line architecture assessment methods that focus more on the reference architecture of a product line.

D-SAAM: Distributed-SAAM [GvDvD05] is a reference architecture evaluation method based on SAAM (Section 3.2 on page 78). SAAM is adapted to reduce organizational efforts at an increase in evaluation team preparation efforts. D-SAAM is explained through an example case study, also elaborating on the implications of analyzing a reference architecture.

purpose The purpose of D-SAAM is to analyze a reference architecture regarding a single quality attribute. In the example case study maintainability is analyzed.

specific concepts D-SAAM introduces the concept of *stakeholder sessions*. Instead of eliciting the scenarios in a group with all stakeholders present, each stakeholder is interviewed in an individual session focusing on the area of his expertise. While the evaluation team needs to prepare each of the sessions in more detail, the organizational impact of the evaluation process is reduced. A similar concept is also employed by Riva and Rosso (Section 5.3 on page 89).

D-SAAM also extends the concept of direct and indirect scenarios introduced by SAAM to accommodate the generality of a reference architecture. Direct scenarios are further divided into scenarios directly supported by the reference architecture, and those that are supported by implementing existing variation points, called floating scenarios. To improve documentation, a cookbook is introduced, guiding the implementation of variation points to achieve specific floating scenarios.

organizational and process management Due to the reduced organizational effort, a typical evaluation can be executed in half a day, instead of the two days required for SAAM.

The method paper [GvDvD05] contains a case study of RACE, a reference architecture in the copier machines embedded domain.

PuLSE-DSSA: Product Line Software Engineering - Domain-Specific Software Architecture introduced in [DFK98] and documented in [ABFG00] is an iterative method to create a reference architecture. The method is listed here as each iteration includes an evaluation step. It is integrated with and depends on the artifacts of the PuLSE methodology [DFK98, p. 28].

purpose The purpose of PuLSE-DSSA is the "systematic and iterative development of reference architectures for software product lines" [DFK98, p. 1] and the integration of architecture evaluation with architecture creation [ABFG00, p. V].

existing underlying concepts PuLSE-DSSA is heavily based on scenarios (Section 2 on page 72).

specific concepts To reduce efforts in subsequent evaluations, scenarios are classified into generic and property-related scenarios. *Generic scenarios* depend on the domain and cover the architectural significant functional requirements of the product line. *Property-related scenarios* are independent of the domain and can be reused between evaluations. They cover quality aspects like coupling, performance and extensibility [DFK98, p. 26]. Furthermore, during evaluation, traceability information linking scenarios to components and connections of the reference architecture is maintained.

In [OM07] it is stated that the method's applicability is limited due to the PuLSE platform dependence and for the lack of tradeoff analysis, as evaluation criteria are iteratively defined per scenario. During each iteration possible tradeoffs are resolved using the scenario priorities, rather than documented and analyzed explicitly. In [DFK98, p. 28] it is mentioned that the method is used on two running projects, but no case studies or references are supplied.

AQA: Architecture Quality Analysis [MND02] is a qualitative technique loosely based on SAAM (Section 3.2 on page 78). It is embedded in QUADA, a top-down, deductive process for designing a reference architecture and initiating a product line in a revolutionary (not evolutionary) fashion. AQA can be

applied both to a reference architecture during requirements analysis and after it has been designed. Therefore, the method is separated into a conceptual and concrete part.

purpose The purpose of AQA is to assess a reference architecture regarding its quality attributes. The conceptual part of AQA focuses on commonality and variability and on increasing domain knowledge.

existing underlying concepts Being a qualitative technique, AQA is based on scenarios (Section 2 on page 72).

specific concepts The analysis steps make use of a set of architectural views generated during the QAD design steps. They include structural, behavior and deployment views. During the conceptual analysis phase a knowledge base is developed capturing the domain requirements and quality attribute models, which are similar to quality attribute characterizations (Section 2 on page 72). During the concrete analysis, a *customer value analysis* is performed to prioritize quality attributes and the corresponding scenarios.

[MND02] includes an example case study in the domain of middleware services, focusing however on the design part of QUADA.

Gannod and Lutz's approach [GL00] is a method that includes both manual and semi-automatic analysis of an existing reference architecture, focusing on modifiability and the commonalities within the product line.

purpose The purpose of the method is "to perform architectural analysis on an existing product line architecture [...] by both manual and tool-supported methods" [GL00, p. 548].

existing underlying concepts The manual analysis is based on scenarios (Section 2 on page 72).

specific concepts To allow tool-supported analysis of the reference architecture, it first needs to be adequately formalized using a form of Architectural Description Language (ADL). In the paper a variety of ADLs are used to model the structure and connections of the architecture and the behavior common to all product instances. Model checking is then used to further analyze this formalized behavior.

steps The method includes three essential steps:

1. **Architecture recovery, discovery, and specification**: The goal of this step is "to familiarize the analyst with the problem domain and implemented solutions" [GL00, p. 550]. All necessary information for manual analysis in step two is gathered and formalized to prepare the tool-supported analysis in step three.
2. **Manual architectural analysis**: This step is similar to SAAM (Section 3.2 on page 78). Scenarios for all categories of modifiability (i.e. extensibility, deletion, portability and restructuring) are elicited and used to evaluate their impact and the fitness of the reference architecture.

3. **Tool assisted architectural analysis**: The common behavior formalized in step one is analyzed using tools matching the used ADLs. This step is not product-line-specific, but the expensive investments in such a detailed analysis are amortized over all products of the product line.

The method is explained and evaluated through a case study in the domain of spaceborne telescopes [GL00, p. 553ff].

SACAM: Software Architecture Comparison Analysis Method [SBV03] focuses on comparing a set of existing architectures in regard to a set of business goals. The generated documentation further supports the variability analysis of product line initiation.

purpose The purpose of SACAM is "to provide a rationale for an architecture selection process by comparing the fitness of architecture candidates for envisioned systems" [SBV03, p. 1].

existing underlying concepts SACAM is based on quality attribute scenarios, which are generated using the scenario generation process also used in ATAM (Section 3.1 on page 74).

specific concepts As the architectures being compared were developed apart, their documentation usually shows a high level of heterogeneity. To allow for a reasonable comparison, a set of comparison criteria are first determined from the business goals. Indicators are elicited that concretize how well an architecture supports one or more of the criteria. Architectural documentation standards and reconstruction techniques are then employed to extract the relevant architectural views that allow to decide on the indicators. Metrics can also be used (e.g. at code level).

output The output of SACAM includes the scores and corresponding reasoning for each architecture and a set of generated artifacts, such as architectural documentation.

organizational and process management The method is typically executed over a two week period. Three of the days require the attendance of most or all of the stakeholders.

[SBV03] includes an example case study in which two architectures from different domains are compared that deal with a door opening mechanism.

5.3 Focus on Individual Quality Attributes

This subsection covers the product line architecture assessment methods that focus more on a specific set of quality attributes.

FAAM: Family-Architecture Analysis Method [Dol02] is an iterative method to evaluate interoperability and extensibility of a reference architecture during design time. It extends SAAM (Section 3.2 on page 78). The method description is very detailed and for each step and sub-steps practical techniques are provided.

purpose The purpose of FAAM is to assess information-system family architectures, focusing on active involvement of stakeholders and family-relevant system qualities [Dol02, p. 10]. The described version of FAAM only evaluates interoperability and extensibility.

specific concepts FAAM is a stakeholder centric method. This means that the stakeholders are not merely interviewed to elicit a set of scenarios, but that the stakeholders are made responsible to create and maintain their requirements documentation.

steps FAAM contains seven steps, each containing a number of sub-steps [Dol02, p. 108ff].

1. **Define assessment**: The scope and intention of the evaluation is determined and recorded.
2. **Prepare system-quality requirements**: The stakeholders create requirements documentation suitable for the assessment.
3. **Prepare architecture**: The architects prepare the required architectural documentation.
4. **Review/refine artifacts**: Refine and correct the artifacts of steps two and three taking into consideration any other constraints on the evaluation process (e.g. organizational or business constraints).
5. **Assess architecture conformance**: Evaluate the fitness of the architecture in regards to the elicited requirements. Employs SAAM techniques (Section 3.2 on page 78).
6. **Report results and proposals**: Report the results and capture lessons learned for evaluation process improvement.
7. **Facilitate architecture assessment**: This step is performed throughout the whole evaluation and supports the other activities.

organizational and process management FAAM is meant to be organically integrated into the development process as a normal development activity. Each iteration should take about a week, not all stakeholders being equally occupied.

[Dol02] includes two case studies. One in the medical domain and one in the planning domain (p. 147ff).

Maccari's approach [Mac02] is a qualitative method focusing on the evolution of a reference architecture during design. In [Mac02] it is described briefly, but explained through two industrial case studies.

purpose The purpose of the method is "assessing the capability of a software product family architecture to adapt to evolution" [Mac02, p. 591].

existing underlying concepts The method is based on scenarios (Section 2 on page 72).

steps The method is composed of five steps:

1. **Create initial scenarios**: The stakeholders gather a set of evolution scenarios. This can be done offline in a distributed fashion (see Riva and Rosso's approach (Section 5.3 on page 89)).

2. **Refine scenarios**: The scenarios are further refined or corrected through peer review or a group meeting.
3. **Rank scenarios**: The scenarios are prioritized to focus the final evaluation meeting efforts.
4. **Evaluate**: The chief architect presents the architecture to the assembled stakeholders, who then evaluate the architecture starting with the highest priority scenarios. All issues are recorded in detail.
5. **Report**: The assessment coordinator compiles a report containing a list and description of all issues, defects and shortcomings, as well as any process improvement ideas.

organizational and process management Based on experiences during the two case studies, the evaluation requires about 15 working days for the evaluation facilitators and two to five working days for each of the other stakeholders spread over an approximately three months period. This sums up to roughly three person months.

[Mac02] includes two industrial case studies in the mobile telephone software and mobile networking domain.

Riva and Rosso's approach [RR03] is an adaption of Maccari's approach (Section 5.3 on page 88) to deal with a larger project size and much larger number of involved stakeholders. It is evolution-related and focuses on identifying defects and shortcomings in the reference architecture, without performing a concluding detailed analysis (e.g. tradeoff analysis). The method is experience-based and explained through an industrial case study.

purpose The purpose of the method is to identify defects and shortcomings of a product line reference architecture, especially for cases, in which a great number of stakeholders are involved and a face to face meeting might not be possible.

specific concepts Like D-SAAM (Section 5.2 on page 84), Riva and Rosso's method involves the evaluation team performing individual *stakeholder interviews* during requirements and scenario elicitation. This replaces a large group meeting and collective brainstorming. As a great number of stakeholders are involved, each being an expert at different levels of abstraction, and because each interview preparation includes results of previous interviews, the evaluation team has to choose carefully the order in which to interview the stakeholders. Riva and Rosso propose grouping the stakeholders and performing the interviews in the order of the groups: (a) those responsible for requirements engineering, (b) experts for parts of the product family or a key quality attribute and (c) those responsible for development and maintenance of the reference architecture.

To optimize follow-up work after the assessment is complete, an action point table is created during evaluation. An *action point* consists of a business problem, problem, possible solutions and their tradeoffs, components affected and "real action". It provides business justification and practical guidance

for fixing a problem and the "real action" provides the management with enough information to directly initiate follow-up activities.

output The method output consists primarily of a prioritized list of the technical and organizational issues that were encountered during the evaluation. In addition, the action point table and further requirements documentation is produced.

[RR03] includes a case study of a fairly large subset of Nokia's mobile terminal product family software.

SBA: Scenario-Based Architecting [AHI+04] is an assessment method focusing on variability in product families. It builds on a set of architectural views at different levels of abstraction to formalize scenarios and to elicit new scenarios by exploring the variability space.

purpose The purpose of SBA is to "identify and quantify the potential benefits of the different architectural variability points" [AHI+04, p. 284].

existing underlying concepts SBA is based on scenarios and quality attribute characterizations (Section 2 on page 72).

specific concepts Instead of relying on human readable text to specify a scenario, SBA allows to formalize scenarios through the use of *variation models*. Their semantics are similar to that of a regular feature diagram, but they are created for the application, functional, conceptual and realization architectural views capturing different levels of abstraction. The lower-level variation models contain higher-level entities and describe their sub-options. Maintaining these relations is called multi-view variation modeling. The benefit is that commonalities and differences between scenarios, which now formally represent a fixed configuration of the variation models, are explicit and that new scenarios can be elicited by selecting different configuration options.

During analysis the concept of a *determining* and an *assessment view* is used. For each quality attribute the highest level view determining the quality attribute and the lowest level view allowing a sound reasoning are selected. Scenarios are then analyzed starting in the determining view and using the variation model relations to find the affected entities in the assessment view.

[AHI+04] includes a case study of Cathlab, a solution in the medical equipment domain.

COSVAM: The COVAMOF Software Variability Assessment Method [DNBS04] is an iterative method to assess the variability within a software product line during evolution. The method can be applied to, but is not limited to the reference architecture.

purpose The purpose of COSVAM is to "provide a technique for variability assessment in the context of evolution" [DNBS04, p. 1].

existing underlying concepts The concept of a variation point (Section 4.2 on page 79) is central to COSVAM and scenarios (Section 2 on page 72) are used to specify product feature requirements.

steps COSVAM consists of five main steps, each with a number of sub-steps. If information is missing or incorrect in one step, a reiteration starts at the corresponding previous step.

1. **Identify assessment goal**: In this step the assessment scope is identified and the method process is tailored to produce the required assessment results and to fit time and availability constraints. It consists of the sub-steps: initiation, define assessment outcome, select scope, and identify members of the assessment team.
2. **Specify provided variability**: The purpose of this step is to specify the variability provided by the product line domain platform, making variation points and dependencies first-class entities and capturing relations across life cycle phases. The following sub-steps are executed: identify and obtain information sources, identify variation points, unify set of variation points, identify variants, and identify dependencies.
3. **Specify required variability**: In this step the variability required by the set of target products is specified as a delta to the provided variability documentation. The step is split into the sub-steps: identify and obtain information sources, construct product scenarios, and construct specification.
4. **Variability evaluation**: The goal of this step is to identify how well the required variability is supported by the provided variability and which changes would be necessary to provide it. The following sub-steps are executed: identify direct mismatches, identify indirect mismatches, cluster and prioritize mismatches, devise a set of possible solutions for each impact analysis set, and determine the impact of possible solutions.
5. **Interpretation**: In this final step conclusions are drawn from the evaluation, following the steps: identify relevant business goals and constraints, identify the pros and cons of different solutions, and select solutions.

[DNBS04] includes a case study in the domain of intelligent traffic systems.

Wijnstra's approach [Wij04] is less an assessment method and more a collection of ideas and experiences of how to assess the handling of variability in a software product family and its architectural views. The ideas are explained through a case study.

purpose The goal is to assess the variability approach of an existing product family.

specific concepts To identify and document the current and expected future situation of the product line, the BAPO (Business, Architecture, Process, and Organization) reasoning framework and CAFCR *architectural views* are

used. BAPO ensures that the right questions are asked to cover the architecture from all important angles. CAFCR provides architectural views that form the basis of a formal variation model that is recommended for larger systems. To extract variation points, *semi-automatic source code analysis* is proposed identifying the positions in the code where functionality is provided depending on external configuration. To improve maintainability during product line evolution, variability checks on code level should not depend on concrete variants, but on the features offered by the variants. Dependencies of generic to specific functionality should be removed or separated through a capability broker and the use of feature interfaces. Also, feature-specific functionality should be grouped as much as possible.

[Wij04] contains a case study in the medical equipment domain.

Korhonen's approach [KM04] is a collection of experiences gained during the evaluation of a single pilot product architecture for adaptability. This was the case as a new product line was to be initiated from that pilot project. The assessment is scenario based and loosely based on ATAM (Section 3.1 on page 74).

purpose The goal is to evaluate to which degree the pilot project architecture can be used as a reference architecture for a new product line.

existing underlying concepts The assessment is based on scenarios (Section 2 on page 72).

specific concepts Due to timing constraints, the scenarios are generated individually by all stakeholders and later combined in a meeting, loosening up the schedule of the evaluation and freeing the evaluation team of any preparatory work, which would be necessary to conclude interviews. It is noted, that the stakeholders were already fairly familiar with both the domain and with the system being evaluated.

organizational and process management The evaluation was executed in an agile fashion by a team of about 10 over the course of a week. Two half-day meetings, one in the beginning of the evaluation and one in the end were held with all participants.

[KM04] contains a case study in the domain of mobile machines.

Service Utilization metrics [vdHDM03] are a set of structural complexity metrics that evaluate the structural soundness of a reference architecture both during design and during evolution. They are independent of the employed architecture description language (ADL) and are tailored to provide meaning in spite of the optionality and variability inherent in software product lines.

purpose The purpose of the metrics is to assess a reference architecture in terms of its structural soundness. They provide no absolute value, but allow comparison of related architectures or product configurations, highlighting possibly problematic components.

existing underlying concepts The Service Utilization metrics are based on the idea of fan-in/fan-out metrics and require that the architecture description contains information about the individual components and about their usage (utilization) relations (e.g. provided and required public interfaces). The use of a formalized ADL allows producing the required information through a simple mapping to provided and required services. Furthermore, if the variability is formalized as well, this opens up the possibility to automate the evaluation process and enhance it through heuristics that are only possible by evaluating a great number of product configurations.

specific concepts The Service Utilization metrics take the fan-in/fan-out idea and introduce two central metrics. The *Provided Service Utilization* (PSU) and the *Required Service Utilization* (RSU). Each is the percentage of a component's available services that are actually used in the configuration under evaluation. Building the ratio over the sum of all components' PSUs and RSUs leads to *Compound PSU* (CPSU) and *Compound RSU* (CRSU) for a complete architecture. From the relative differences between these metrics and from their patterns of variation between different product configurations, potential weak-spots in the architecture structure can be derived and serve as an entry point for further manual analysis.

[vdHDM03] includes two academic and one industrial case study of a library system.

Rahman's metrics [Rah04] are a set of component based metrics that are used to evaluate the reusability and modularity of a software product line reference architecture. They use the Provided Service Utilization (PSU) of the Service Utilization metrics (Section 5.3 on page 92) and share their underlying concepts and their dependence on component and service specification.

purpose The purpose of the metrics is to assess a reference architecture regarding reusability and modularity.

existing underlying concepts Rahman's metrics build on the foundations of Service Utilization metrics (Section 5.3 on page 92) and are based on the same underlying concepts of components and their service requirements.

specific concepts Modularity is evaluated through a graph-based metric, measuring by how much the module call-graph diverts from a pure tree structure. Reusability is evaluated by aggregating a number of existing component based metrics. They include metrics measuring the observability, customizability, interface complexity, self completeness and maturity of individual components.

In interesting concept is to calculate the metrics for a comparably small subset of the architecture and evaluate if the chosen subset is representative for the attributes of the whole system using the *Wilcoxon Signed Rank Test*. This can save considerable amounts of work, especially if the metrics cannot be calculated automatically.

[Rah04] includes an academic case study of a library system.

6 Conclusions

Architecture is at the center of reuse within a software product line. The quality of the reference architecture directly affects the quality of all products in the product line. It is thus one of the most critical assets in product line engineering. Architecture assessment is an essential tool for ensuring a quality architectural design both for current and for future requirements. Because product line architecture assessment requirements are a superset of those for single architecture assessment, central concepts for architecture evaluation in general and a small selection of the most established single architecture evaluation methods are introduced before continuing with the product-line-specific issues and methods.

It is important to remember that no single method fits all practical purposes. Many methods exist that focus on specific aspects of an architecture or a specific set of available artifacts, neglecting other important parts of the equation. Therefore it is often necessary to adapt a method for the situation at hand, known as tailoring. Another option is to employ a combination of methods and to aggregate and merge their results. [ES05] provides a set of evaluation criteria that help with deciding which method to use. Also a number of product line assessment methods are already classified accordingly. Case studies on how to evaluate methods for a project at hand can be found in [DN00].

References

ABFG00. Michalis Anastasopoulos, Joachim Bayer, Oliver Flege, and Cristina Gacek. A process for product line architecture creation and evaluation. PuLSE-DSSA - version 2.0. Technical report, Fraunhofer IESE (Germany), 2000.

AHI+04. Pierre America, Dieter Hammer, Mugurel T. Ionita, Henk Obbink, and Eelco Rommes. Scenario-based decision making for architectural variability in product families. *Software Product Lines*, pages 284–303, 2004.

CKK01. Paul Clements, Rick Kazman, and Mark Klein. *Evaluating Software Architectures: Methods and Case Studies*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.

Cle00. Paul Clements. Active reviews for intermediate designs. Technical report, CMU/SEI, 2000.

CN01. Paul Clements and Linda Northrop. *Software product lines: practices and patterns*, volume 0201703327. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.

DFK98. Jean-Marc DeBaud, Oliver Flege, and Peter Knauber. PuLSE-DSSA—a method for the development of software reference architectures. In *ISAW '98: Proceedings of the third international workshop on Software architecture*, pages 25–28, New York, NY, USA, 1998. ACM.

DN00. Liliana Dobrica and Eila Niemelä. A strategy for analyzing product line software architectures. Technical report, VTT Publications, 2000.

DNBS04. Sybren. Deelstra, Jos Nijhuis, Jan Bosch, and Marco Sinnema. The COVAMOF software variability assessment method (COSVAM). *2nd Groningen Workshop on Software Variability Management*, 2004.

Dol02. Thomas J. Dolan. *Architecture Assessment of Information-Systems Families*. PhD thesis, Department of Technology Management, Eindhoven University of Technology, 2002.

ES05. Leire Etxeberria and Goiuria Sagardui. Product-line architecture: New issues for evaluation. *Software Product Lines*, pages 174–185, 2005.

GL00. Gerald C. Gannod and Robyn R. Lutz. An approach to architectural analysis of product lines. In *ICSE '00: Proceedings of the 22nd international conference on Software engineering*, pages 548–557, New York, NY, USA, 2000. ACM.

GvDvD05. Bas Graaf, Hylke van Dijk, and Arie van Deursen. Evaluating an embedded software reference architecture - industrial experience report. *Ninth European Conference on Software Maintenance and Reengineering CSMR*, pages 354–363, 2005.

KAK02. Rick Kazman, Jai Asundi, and Mark Klein. Making architecture design decisions: An economic approach. Technical report, CMU/SEI, 2002.

KBWA94. Rick Kazman, Len Bass, Mike Webb, and Gregory Abowd. SAAM: a method for analyzing the properties of software architectures. In *ICSE '94: Proceedings of the 16th international conference on Software engineering*, pages 81–90, Los Alamitos, CA, USA, 1994. IEEE Computer Society Press.

KKC00. Rick Kazman, Mark Klein, and Paul Clements. ATAM: Method for architecture evaluation. Technical report, CMU/SEI, 2000.

KM04. Mika Korhonen and Tommi Mikkonen. Assessing systems adaptability to a product family. *Journal of Systems Architecture*, 50(7):383 – 392, 2004. Adaptable System/Software Architectures.

Mac02. Alessandro Maccari. Experiences in assessing product family software architecture for evolution. In *ICSE '02: Proceedings of the 24th International Conference on Software Engineering*, pages 585–592, New York, NY, USA, 2002. ACM.

MND02. Mari Matinlassi, Eila Niemelä, and Liliana Dobrica. Quality-driven architecture design and quality analysis method: A revolutionary initiation approach to a product line architecture. Technical report, VTT Publications, 2002.

NBC+03. Robert L. Nord, Mario R. Barbacci, Paul Clements, Rick Kazman, Mark Klein, Liam O'Brien, and James E. Tomayko. Integrating the architecture tradeoff analysis method (ATAM) with the cost benefit analysis method (CBAM). Technical report, CMU/SEI, 2003.

OM07. Femi G. Olumofin and Vojislav B. Mišić. A holistic architecture assessment method for software product lines. *Inf. Softw. Technol.*, 49(4):309–323, 2007.

Rah04. Asim Rahman. Metrics for the structural assessment of product line architecture. Master's thesis, Blekinge Institute of Technology, 2004.

RR03. Claudio Riva and Christian Del Rosso. Experiences with software product family evolution. In *IWPSE '03: Proceedings of the 6th International Workshop on Principles of Software Evolution*, page 161, Washington, DC, USA, 2003. IEEE Computer Society.

SBV03. Christoph Stoermer, Felix Bachmann, and Chris Verhoef. SACAM: The software architecture comparison analysis method. Technical report, CMU/SEI, 2003.

vdHDM03. André van der Hoek, Ebru Dincel, and Nenad Medvidovic. Using service utilization metrics to assess the structure of product line architectures. In

METRICS '03: Proceedings of the 9th International Symposium on Software Metrics, page 298, Washington, DC, USA, 2003. IEEE Computer Society.

vdLSR07. Frank J. van der Linden, Klaus Schmid, and Eelco Rommes. *Software Product Lines in Action: The Best Industrial Practice in Product Line Engineering*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2007.

Wij04. Jan G. Wijnstra. Evolving a product family in a changing context. *Software Product-Family Engineering*, pages 111–128, 2004.