Gesellschaft für Informatik (GI)

publishes this series in order to make available to a broad public recent findings in informatics (i.e. computer science and information systems), to document conferences that are organized in cooperation with GI and to publish the annual GI Award dissertation.

Broken down into the fields of
• Seminar
• Proceedings
• Dissertations
• Thematics
current topics are dealt with from the fields of research and development, teaching and further training in theory and practice. The Editorial Committee uses an intensive review process in order to ensure the high level of the contributions.

The volumes are published in German or English.

Information: http://www.gi-ev.de/service/publikationen/lni/

ISSN 1617-5468
ISBN 978-3-88579-237-6

This volume contains papers from the Software Engineering 2009 conference held in Kaiserslautern from 2 to 6 March 2009. The topics covered in the papers range from software requirements, technologies, and development strategies to reports discussing industrial project experience.

Peter Liggesmeyer, Gregor Engels, Jürgen Münch, Jörg Dörr, Norman Riegel (Hrsg.): Software Engineering 2009

143

GI-Edition

Lecture Notes in Informatics

Peter Liggesmeyer, Gregor Engels, Jürgen Münch, Jörg Dörr, Norman Riegel (Hrsg.)

Software Engineering 2009

02.–06. März 2009
Kaiserslautern

Proceedings

Peter Liggesmeyer, Gregor Engels, Jürgen Münch, Jörg Dörr,
Norman Riegel (Hrsg.)

Software Engineering 2009

Fachtagung des GI-Fachbereichs Softwaretechnik

02.-06.03.2009
in Kaiserslautern

Gesellschaft für Informatik e.V. (GI)

Lecture Notes in Informatics (LNI) - Proceedings
Series of the Gesellschaft für Informatik (GI)

Volume P-143

ISBN 978-3-88579-237-6
ISSN 1617-5468

Volume Editors

Prof. Dr.-Ing. Peter Liggesmeyer
TU Kaiserslautern und
Fraunhofer-Institut für Experimentelles Software Engineering
Fraunhofer-Platz 1
67663 Kaiserslautern, Germany
Email: peter.liggesmeyer@iese.fraunhofer.de

Prof. Dr. Gregor Engels
Universität Paderborn
Fakultät für Elektrotechnik, Informatik und Mathematik
Institut für Informatik
Warburger Str. 100
33098 Paderborn
Email: engels@upb.de

Dr. Jürgen Münch
Fraunhofer-Institut für Experimentelles Software Engineering
Fraunhofer-Platz 1
67663 Kaiserslautern, Germany
Email: juergen.muench@iese.fraunhofer.de

Jörg Dörr
Fraunhofer-Institut für Experimentelles Software Engineering
Fraunhofer-Platz 1
67663 Kaiserslautern, Germany
Email: joerg.doerr@iese.fraunhofer.de

Norman Riegel
Fraunhofer-Institut für Experimentelles Software Engineering
Fraunhofer-Platz 1
67663 Kaiserslautern, Germany
Email: norman.riegel@iese.fraunhofer.de

Series Editorial Board

Heinrich C. Mayr, Universität Klagenfurt, Austria (Chairman, mayr@ifit.uni-klu.ac.at)
Hinrich Bonin, Leuphana-Universität Lüneburg, Germany
Dieter Fellner, Technische Universität Darmstadt, Germany
Ulrich Flegel, SAP Research, Germany
Ulrich Frank, Universität Duisburg-Essen, Germany
Johann-Christoph Freytag, Humboldt-Universität Berlin, Germany
Ulrich Furbach, Universität Koblenz, Germany

Michael Goedicke, Universität Duisburg-Essen
Ralf Hofestädt, Universität Bielefeld
Michael Koch, Universität der Bundeswehr, München, Germany
Axel Lehmann, Universität der Bundeswehr München, Germany
Ernst W. Mayr, Technische Universität München, Germany
Sigrid Schubert, Universität Siegen, Germany
Martin Warnke, Leuphana-Universität Lüneburg, Germany

Dissertations
Dorothea Wagner, Universität Karlsruhe, Germany
Seminars
Reinhard Wilhelm, Universität des Saarlandes, Germany
Thematics
Andreas Oberweis, Universität Karlsruhe (TH)

© Gesellschaft für Informatik, Bonn 2009
printed by Köllen Druck+Verlag GmbH, Bonn

Willkommen zur SE 2009 in Kaiserslautern!

Die Tagung Software Engineering 2009 ist die fünfte Veranstaltung einer inzwischen etablierten Reihe von Fachtagungen, deren Ziel die Zusammenführung und Stärkung der deutschsprachigen Softwaretechnik ist. Die SE 2009 bietet ein Forum zum intensiven Austausch über praktische Erfahrungen, wissenschaftliche Erkenntnisse sowie zukünftige Herausforderungen bei der Entwicklung von Softwareprodukten bzw. softwareintensiven Systemen. Sie richtet sich gleichermaßen an Teilnehmer aus Industrie und Wissenschaft.

Die Software-Engineering-Tagungsreihe wird vom Fachbereich Softwaretechnik der Gesellschaft für Informatik e.V. getragen. Die Software Engineering 2009 wird vom Lehrstuhl Software Engineering Dependability der Technischen Universität Kaiserslautern und dem Fraunhofer-Institut für Experimentelles Software Engineering (IESE) veranstaltet.

Die SE 2009 bietet im Hauptprogramm begutachtete Forschungsarbeiten und eingeladene wissenschaftliche Vorträge. Von den 35 Einreichungen für das technisch-wissenschaftliche Programm wurden 10 Vollbeiträge und 5 Kurzbeiträge akzeptiert. Darüber hinaus werden in begutachteten und eingeladenen Praxisvorträgen am Industrietag aktuelle Problemstellungen, Lösungsansätze und gewonnene Erfahrungen präsentiert und zur Diskussion gestellt. Abgerundet wird das Programm durch eine Podiumsdiskussion zum diesjährigen Schwerpunktthema der Konferenz „Garantierte Qualität – Anspruch industrieller Software-Entwicklung“. Vor dem Hauptprogramm der Konferenz finden sieben ganztägige und ein halbtägiger Workshop, sowie acht halbtägige Tutorien zu aktuellen, innovativen und praxisrelevanten Themen im Software Engineering statt.

Die Durchführung der Tagung Software Engineering 2009 wäre ohne die Mitwirkung vieler engagierter Personen nicht möglich gewesen. Ich bedanke mich insbesondere bei meinem Kollegen, Herrn Gregor Engels, für die Organisation des Industrietags und bei Dr. Jürgen Münch für die Koordination des Tutorial- und Workshop-Programms. Jörg Dörr sei für seinen unermüdlichen Einsatz rund um die Organisation der Tagung gedankt. Und nicht zuletzt möchte ich mich bei Norman Riegel insbesondere für die Arbeiten am Konferenzsystem sowie bei meiner Sekretärin Simone Rockenmeyer für ihre Bereitschaft, alle Belange der Tagung zu unterstützen, recht herzlich bedanken.

Ich wünsche Ihnen viele neue Einsichten und Erkenntnisse, anregende Diskussionen und nicht zuletzt viel Spaß!

Kaiserslautern, im März 2009

Peter Liggesmeyer

Tagungsleitung
Peter Liggesmeyer, TU Kaiserslautern, Fraunhofer IESE

Leitung Industrietag
Gregor Engels, Universität Paderborn

Leitung Workshops und Tutorials
Jürgen Münch, Fraunhofer IESE

Tagungsorganisation
Jörg Dörr, Fraunhofer IESE
Norman Riegel, Fraunhofer IESE
Simone Rockenmeyer, Fraunhofer IESE

Programmkomitee
Klaus Beetz, Siemens AG
Manfred Broy, Technische Universität München
Bernd Brügge, Technische Universität München
Jürgen Ebert, Universität Koblenz-Landau
Gregor Engels, Universität Paderborn
Michael Goedicke, Universität Duisburg-Essen
Klaus Grimm, Daimler AG
Volker Gruhn, Universität Leipzig
Wilhelm Hasselbring, Christian-Albrechts-Universität zu Kiel
Lutz Heuser, SAP AG
Stefan Jähnichen, Technische Universität Berlin
Matthias Jarke, RWTH Aachen
Udo Kelter, Universität Siegen
Claus Lewerentz, Brandenburgische Technische Universität Cottbus
Horst Lichter, RWTH Aachen
Oliver Mäckel, Siemens AG
Florian Matthes, Technische Universität München
Manfred Nagl, RWTH Aachen
Barbara Paech, Universität Heidelberg
Klaus Pohl, Universität Duisburg-Essen
Ralf Reussner, Universität Karlsruhe (TH)
Eric Sax, MBtech Group
Wilhelm Schäfer, Universität Paderborn
Andy Schürr, Technische Universität Darmstadt
Walter Tichy, Universität Karlsruhe (TH)
Markus Voß, Capgemini sd&m AG
Mario Winter, Fachhochschule Köln
Andreas Zeller, Universität des Saarlandes
Heinz Züllighoven, Universität Hamburg
Albert Zündorf, Universität Kassel

Offizieller Veranstalter
Fachbereich Softwaretechnik der Gesellschaft für Informatik (GI)

Mitveranstalter
Fraunhofer-Institut für Experimentelles Software Engineering
Technische Universität Kaiserslautern

Unterstützt wird die Tagung zudem von:
Schweizer Informatik Gesellschaft (SI)
Österreichische Computer Gesellschaft (OCG)

Sponsoren

SE 2009 Platinsponsor

SIEMENS

SE 2009 Goldsponsor

BOSCH
Invented for life

SE 2009 Silbersponsoren

JOHN DEERE

Dräger

MBtech
Mercedes-Benz technology

werum
SOFTWARE & SYSTEMS

SE 2009 Bronzesponsoren

Software Technologie
Initiative Kaiserslautern e.V.

oose.
Innovative Informatik

Inhaltsverzeichnis

Eingeladene wissenschaftliche Vorträge

High Impact Requirements Engineering in Software Ecosystems
Mathias Jarke 3

Know Risk – Know Fun
Oliver Mäckel 5

Präsentationen des Industrietags

Testen von Elektronik in der Automobilindustrie – von der ausführbaren Spezifikation bis zum automatisierten Test im Fahrzeug
Eric Sax (eingeladener Vortrag) 9

Aufbau und Betrieb eines Testcenters
Volker Asmus, Jens Banach 11

Mehr sehen mit sira – Mit einem Blick IT-Projekte durchleuchten
Hendrik Bülte, Oliver Mäckel 13

Automatisierter, flexibler Modultest in der Zielumgebung
Thomas Eißenlöffel, Sebastian Moslener 15

Quasar Analytics: Aktive Qualitätssteuerung in großen und verteilten Softwareprojekten
Alex Hofmann 17

Projektvorhersage – die nichtlineare Zukunft
Markus Manleitner 19

Global Sourcing und die Industrialisierung der Software-Entwicklung
Robert Martignoni 21

Stopping (and reversing) the architectural erosion of software systems. An industrial case study
Bernhard Merkle 23

360° Quality: Functional Quality is not enough
Frank Simon 25

Wirtschaftlich und wirksam: Entwicklungsprozesse auf Basis des Eclipse Process Frameworks
Rainer Singvogel 27

Management großer Projekte – ein modellbasierter Ansatz
Dehla Sokenou 29

Modellgetriebene Entwicklung eingebetteter Systeme mit Eclipse
Axel Terfloth, Lothar Wendehals 31

Software Architects and Testers - Collaboration Required!
Peter Zimmerer 33

Forschungsarbeiten

Modellierung

A Comprehensive Survey of UML Compliance in Current Modelling Tools
Holger Eichelberger, Yilmaz Eldogan, Klaus Schmid 39

Embedding Behavioral Models into Object-Oriented Source Code
Moritz Balz, Michael Striewe, Michael Goedicke 51

Kontextsensitive Dialogmodellierung
Jürgen Rückert, Barbara Paech 63

Modellierung und Qualitätssicherung von UML-Modellen der Geschäftslogik von Informationssystemen
Tobias Brückmann, Volker Gruhn 75

Tracing Quality-Related Design Decisions in a Category-Driven Software Architecture
Stephan Bode, Matthias Riebisch 87

Plattform & Produktlinien

Vergleich und Integration von Komposition und Annotation zur Implementierung von Produktlinien
Sven Apel, Christian Kästner, Christian Lengauer 101

Wiederverwendbare Generatorkomponenten in heterogenen Projektumfeldern
Norbert Klein, Robert Nikonowicz 113

Qualitätssicherung

Using Template Analysis as Background Reading Technique for Requirements Elicitation
Sebastian Stein, Yves Lauer, Marwane El Kharbili *127*

Testfokusauswahl im Integrationstestprozess
Lars Borner, Barbara Paech *139*

The vital few and trivial many: An empirical analysis of the Pareto Distribution of defects
Timea Illes-Seifert, Barbara Paech *151*

Kurzbeiträge

Transaktionsorientiertes Mischen von Modellen
Maik Schmidt, Sven Wenzel, Udo Kelter *165*

Eine Taxonomie für Aufgabenmodelle
Gerrit Meixner, Daniel Görlich *171*

Automating Product Derivation in Software Product Line Engineering
Goetz Botterweck, Kwanwoo Lee, Steffen Thiel *177*

Die Entwicklung einer gemeinsamen IT-Plattform im Bereich Logistik
Thomas Hering, André Ludwig, Bogdan Franczyk *183*

Ein Stichprobenverfahren zur Abschätzung der Fehlerzahl
Karl-Heinz Wichert, Melanie Späth *189*

Workshops

Human-Computer Interaction und Visualisierung (HCIV 2009)
Achim Ebert, Peter Dannenmann *197*

Modellgetriebene Softwarearchitektur – Evolution, Integration und Migration (MSEIM 2009)
Michael Goedicke, Maritta Heisel, Sascha Hunold, Stefan Kühne, Matthias Riebisch, Niels Streekmann *199*

Patterns in Enterprise Architecture Management (PEAM 2009)
Florian Matthes, Alexander Ernst *201*

Produkt-Variabilität im gesamten Lebenszyklus (PVLZ 2009)
Klaus Marquardt, Dietmar Schütz, Markus Völter *203*

Requirements Engineering und Business Process Management –
Konvergenz, Synonym oder doch so wie gehabt? (REBPM 2009)
Kurt Schneider, Jörg Dörr, Sebastian Stein, Sebastian Adam, Daniel Lübke *205*

Software ENgineering within Social software Environments (SENSE 2009)
Ralf Klamma, Volker Wulf, Matthias Jarke, Anna Glukhova *207*

Collaboration and Knowledge Sharing in
Software Development Teams (SOFTEAM 2009)
Hans-Jörg Happel, Steffen Lohmann, Walid Maalej *209*

Software-Qualitätsmodellierung und -bewertung (SQMB 2009)
Stefan Wagner, Manfred Broy, Florian Deißenböck,
Jürgen Münch, Peter Liggesmeyer *211*

Eingeladene wissenschaftliche Vorträge

High Impact Requirements Engineering in Software Ecosystems

Matthias Jarke

Informatik 5 (Informationssysteme), RWTH Aachen, and
Fraunhofer FIT
Ahornstr. 55
52074 Aachen
jarke@dbis.rwth-aachen.de

Abstract: Requirements engineering constitutes the bridge between the usage world and the development world of software-intensive systems. Its responsibility has thus often been described as to ensure building the right system, complementing other facets of system development which focus on building the system in the right manner.

Many of the RE methods proposed in the 1980's and 1990's have by now found their way into wide practical usage, even though some of even the most basic wisdoms still often get neglected when it comes to very large and political systems. However, beyond these issues, the environment in which systems operate has changed since the turn of the century. Systems are no longer developed from scratch but evolve from earlier ones, are embedded in a broader landscape or ecology of existing systems, and grow further in complexity while changing much faster than before. Embedding of systems in our sense means both the interdisciplinary linkage to other engineering disciplines, and the technical and social networking of systems e.g. in organizational information systems and internet communities.

The presentation will give an overview of some of these changes and the related challenges to requirements engineering research derived from them. It will then specifically focus on the role of software platforms as part of, and contributor to system ecologies. The discussion will address both the viewpoint of the platform owners and developers, and the viewpoint of the communities using the platforms.

Know Risk – Know Fun

Oliver Mäckel

Siemens AG
Corporate Technology
CT PP 6
Otto-Hahn-Ring 6
81739 München
oliver.maeckel@siemens.com

Abstract: Die aktuellen Veränderungen in der Finanzindustrie lösen nicht nur in der Finanzbranche weitreichende Veränderungen aus. Es wird stärker als bisher die Aufgabe der Unternehmen sein, dass die richtigen Mittel zum richtigen Zeitpunkt effizient eingesetzt werden. Denn die hohen finanziellen Belastungen beim Scheitern von IT-Projekten und Software-Entwicklungsprojekten können Unternehmen bis an den Rand der wirtschaftlichen Bedrohung führen.

Eine „Garantierte Qualität" muss daher der höchste Anspruch industrieller Software-Entwicklung sein und bleiben. Die Erreichung einer „Garantierten Qualität" stellt dabei aber immer noch eine große Herausforderung dar. Bisher wird typischerweise eine umfassende Behandlung von Qualität durch Qualitätsmodelle und darauf aufbauenden Bewertungsmethoden erwartet.

Eine „Garantierte Qualität" im Rahmen einer effizienten Abwicklung von IT-Projekten oder Software-Entwicklungsprojekten zu erreichen, wird allerdings nicht durch die Auswahl des richtigen Qualitätsmodells bestimmt. Der Erfolg hängt vielmehr davon ab, wie professionell die möglichen Projekt-Risiken und Projekt-Unsicherheiten identifiziert, bewertet und gemanagt werden.

Strukturiertes und pro-aktives Risikomanagement wird in der heutigen Zeit für jedes Unternehmen überlebensnotwendig. Gerade in der aktuellen weltwirtschaftlichen Situation wird sich zeigen, wer Risiken und Unsicherheiten erfolgreich identifizieren und managen kann.

Oliver Mäckel, seit 1997 als Safety- und Risikomanager und Gutachter sowohl für softwareintensive Systeme als auch für Großanlagen unterwegs, hinterfragt in seinem Vortrag die gängigen Qualitätsmodelle und Risikoanalysestandards. Ferner werden in seinem Vortrag zudem Möglichkeiten und Szenarien diskutiert, wie man sich den Herausforderungen effektiv stellen kann.

Präsentationen des Industrietags

Testen von Elektronik in der Automobilindustrie - von der ausführbaren Spezifikation bis zum automatisierten Test im Fahrzeug

Eric Sax

Test-Engineering
MB-Technology GmbH
Kolumbusstr. 2
71063 Sindelfingen
eric.sax@mbtech-group.com

Zusammenfassung

Auffälligkeiten elektronischer Steuergeräte im Fahrzeug sorgten in der jüngeren Vergangenheit für Diskussionen gerade zwischen Automobilherstellern und Zulieferern.

Als Reaktion auf diese Herausforderungen wurden Qualitätssicherungsmaßnahmen bei Automobilherstellern und Lieferanten mit dem Ziel verstärkt,

- bereits zu einem frühen Zeitpunkt Qualitätsprobleme zu erkennen und
- Mittel und Methoden zur Fehlerbeseitigung bereit zu stellen.

Testen, als Baustein der Qualitätssicherung, rückt dabei immer stärker in den Fokus dieser Maßnahmen. Insbesondere auf der Seite des Equipments ist zu beobachten, dass bereits heute ausgereifte Werkzeuge zum Testen von Elektrik und Elektronik zum Einsatz kommen. Dazu gehören beispielsweise:

- Der Test von modellbasiert entwickelter Software bereits auf Modell-Ebene (z.B. Model-, bzw. Software-in-the-Loop).
- Der Test von einzelnen Steuergeräten in einer Laborumgebung (z.B. Hardware-in-the-Loop einzelner Steuergeräte aus Sicht des OEM)
- Der Test mehrerer Steuergeräte im Verbund (z.B. Hardware-in-the-Loop Prüfständen auf Integrationsebene)
- Der Test von E&E-Komponenten direkt in Fahrzeug-Prototypen.

Neben dieser Werkzeug orientierten Sicht spielen darüber hinaus durchgängige Prozesse über Werkzeug- und Firmengrenzen hinweg eine immer wichtigere Rolle.

Daher wurden im Automobilbereich Referenzprozesse eingeführt, deren korrekte Anwendung regelmäßig überprüft wird (z.B. SPICE[1]).

Testen und die dazugehörigen Prozesse werden heute aber häufig nur als Fragmente des gesamten Entwicklungsprozesses und nicht in durchgängiger Form betrachtet. Da der Umfang der Testaufgaben im Bereich Elektrik und Elektronik aber stetig wächst, ist diese punktuelle Sicht kaum noch geeignet heutige Anforderungen an die Steuergeräte-Qualität zu erfüllen. Diese Herausforderung trifft in erster Linie die Lieferanten, die den Großteil der Steuergeräte-Entwicklung abdecken, während das Zusammenspiel der einzelnen Steuergeräte im Verbund die Integrationsaufgabe des Automobilherstellers darstellt.

Als Reaktion auf diese Situation suchte die Automobilindustrie Möglichkeiten Testprozesse zu bewerten und gezielt zu verbessern.

Basierend auf diese bereits existierenden Prozessmodellen (z.B. „TPI© automotive“) und der Erfahrung im Bereich des Tests von Steuergeräten im Automobilbereich sind neue Vorgehensmodelle entstanden (z.B. PROVEtech:TP5), die explizit das Testen von Steuergeräten in der Automobilindustrie adressieren.

Besonderes Augenmerk gilt dabei immer den arbeitsteiligen Test-Aktivitäten aller Beteiligten wie:

- Aufstellen einer Test-Strategie
- Durchführung der Test-Planung
- Definition der Test-Fälle
- Durchführung der Test-Implementierung
- Verwaltung von Abweichungen

Am Ende kann nur ein klares Vorgehensmodell ergänzt um ein eindeutiges Rollenverständnis die Qualität der Steuergeräte und Steuergeräteverbünde im Fahrzeug über den Lebenszyklus hinweg sicherstellen. Wenn dann noch das passende Equipment in der richtigen Phase des Lebenszyklus zum Einsatz kommt, werden wir weiter auf Elektronik als Innovationstreiber im Auto setzen können.

[1] Vgl. ISO 15504

Aufbau und Betrieb eines Testcenters

Volker Asmus, Jens Banach
Fourth Project Consulting GmbH
Goethestr. 48
D-38440 Wolfsburg
volker.asmus@fourthproject.de, jens.banach@fourthproject.de

Abstract: Das Durchführen automatisierter toolgestützter Tests ist nicht nur aus wirtschaftlichen Gründen sinnvoll, sondern auch aus Qualitätsgründen geboten. In diesem Beitrag werden wirtschaftliche Kriterien benannt, die zur Bewertung herangezogen werden können, wenn es um die Entscheidung über den Umfang von Testmethoden geht.

Einsatzbereiche für das Testen

Das Testen von Software gewinnt im Rahmen der Softwareentwicklung eine immer größer werdende Bedeutung. Testen hat sich in allen Vorgehensmodellen der Softwareentwicklung als integraler Bestandteil etabliert. Im sogenannten V-Modell startet der Testprozess nach der Implementierung mit dem Unit Test und endet mit der Abnahme der Software durch den Auftraggeber. Im Rahmen der modellbasierten Entwicklung zeichnen sich Tendenzen ab, dass Testaktivitäten bereits im linken Ast zum Einsatz kommen, das V-Modell sich somit zum W-Modell erweitert [Sc07][1].

Ziel

Unabhängig vom Einsatzbereich besteht das Ziel des Testens darin, (dem Kunden) zu belegen, dass die an das System gestellten Anforderungen erfüllt sind. Die Unterschrift des Auftraggebers unter dem Abnahmedokument gehört für den Projektleiter zu den wichtigsten Augenblicken während der Projektlaufzeit. Mit dem Abnahmedokument kann das Rechnungswesen des Auftragnehmers die Faktura für die letzte Teilzahlung erstellen. Auf Kundenseite kann die Applikation nach diesem Zeitpunkt an die Betriebsorganisation übergeben werden.

Problem

Hat sich damit aber das Testen während des weiteren Betriebs erledigt? In den meisten Fällen ja - leider. Anschließend wird fleißig weiterentwickelt, Bug fixes und Releases einem kurzen Test per Hand unterzogen und danach freigegeben. Getestet werden lediglich noch die beseitigten Fehler bzw. die neu eingeführten Funktionen. Das Nachprogrammieren neuer Funktionen führt zum sogenannten „Code Smell“ und meint damit, dass die Wartbarkeit des Quelltextes schwieriger wird. Die Einführung neuer

[1] Ina Schieferdecker: „Modell-Based Testing with the focus on embedded Software“ 12./19. November 2007 (http://www.ets.tu-berlin.de/lehre/WS0809/qes/download/MBT_ES.pdf)

Funktionalitäten wie z.B. die Erweiterung des Rollen und Rechte-Systems können zu Performance Einbußen führen. In komplexen Systemumgebungen treten Folgefehler in Teilen der Systemumgebung auf, die nicht getestet wurden.

Lösung

Bei Systemumgebungen mit erheblichem Einfluss auf den geschäftlichen Erfolg haben große Unternehmungen damit begonnen, das automatisierte Testen im Release-management während der Betriebsphase einzuführen. Ausgeführt werden diese Test-reihen durch Testinstanzen, die den gesamten Testprozess abdecken und organisatorisch vom Projekt getrennt sind, um ihre Aufgaben unabhängig, aber in Abstimmung mit dem Projekt, durchführen zu können. Man spricht dann von „Testcentern“ oder auch von einer „Testfactory“.
Getestet werden mindestens die funktionalen Anforderungen, ergänzt durch Lasttests. Worin besteht nun aber der wirtschaftliche Nutzen? Aufgrund der Häufigkeit und des Wiederholungscharakters ist das Dokumentieren und Automatisieren von Testdurch-läufen bei Regressionstests wirtschaftlich sinnvoll. Ferner führen etablierte, an „Best Practises“ orientierte Prozesse eines Testcenters zu Prozessoptimierungen.

Vorteile der Testautomatisierung

Im Vergleich zu manuell durchgeführten Regressionstests ist der Aufwand bei automati-sierten Tests in der Regel auf lange Sicht deutlich geringer. Außer dem Anpassen der vorhandenen Testskripte sind eventuell die Testdaten zu aktualisieren. In Abhängigkeit von der eingesetzten Testsoftware, liefert diese nach dem Testdurchlauf im Testprotokoll auch einen Hinweis auf die Fehlerursache. Dies wiederum führt zu Einsparungen in der anschließenden Fehlerbeseitigung. Darüber hinaus werden Einsparungen durch Fehler-vermeidung realisiert, wobei ein Testcenter entsprechend seiner Aufgabenstellung hier wirksam tätig werden kann. Fehler die übersehen und die produktiv gesetzt werden, verursachen im schlimmsten Fall einen Systemausfall. All zu häufig müssen bereits er-fasste Daten nachträglich manuell auf der Datenbank geändert oder ergänzt werden. Dies sind Kosten, die durch die Betriebsorganisation zu tragen sind und darüber hinaus zu Unzufriedenheit bei den Benutzern führt.
Die Automatisierung von Tests ermöglicht dem Testmitarbeiter den Testabdeckungsgrad zu erhöhen. Dies spielt bei größeren Systemumgebungen mit einer großen Anzahl an Schnittstellen, beteiligten Systemen und Funktionen eine große Rolle. Die Komplexität solcher Systeme ist eigentlich nur noch mit einer vernünftig ausgestatteten Testum-gebung beherrschbar.

Aufwand

Der Aufwand für die Einführung eines Testcenters besteht im Aufbau einer Test-umgebung für die zu testende Systemumgebung und beginnt zunächst bei der Investition in Soft- und Hardware. Um ein möglichst objektives Ergebnis der Testdurchläufe zu erhalten, ist zu empfehlen, die Testsoftware auf einer separaten Hardware zu installieren. Abschließend sind Mitarbeiter zu qualifizieren, die die Testsoftware bedienen und die Testprotokolle zu interpretieren wissen.
Die Kosten für den Betrieb eines Testcenters bestehen neben den Basiskosten für Infrastruktur und Personal im Wesentlichen in der Pflege der Testskripte sowie dem Aktualisieren der Testdaten.

Mehr sehen mit sira – Mit einem Blick IT-Projekte durchleuchten

Hendrik Bülte, Oliver Mäckel

Siemens AG
CT PP 6
Otto-Hahn-Ring 6
81739 München
hendrik.buelte@siemens.com
oliver.maeckel@siemens.com

Risiken zu erkennen, zu vermeiden und zu bewältigen, ist wichtiger Bestandteil von IT-Projekten. Eine bewusste Auseinandersetzung mit einem pro-aktiven Risikomanagement ist daher unausweichlich. Um die Risiken bei steigender Komplexität zu bewältigen, ist eine übergreifende Betrachtung des gesamten IT-Projektes notwendig. Darüber hinaus sind auch nicht monetär bewertbare Risiken, sogenannte Unsicherheiten, festzuhalten. Oftmals überwiegen diese Unsicherheiten die quantitativen Risiken und können so den Projekterfolg torpedieren. Für ein ganzheitliches Management der Risiken und Unsicherheiten müssen diese „mit einem Blick" wahrgenommen werden können. Ein wesentlicher Nachteil der standardmäßig angewandten Risikoanalysen liegt darin, dass entweder eine technische oder eine wirtschaftliche Risikobewertung vorgenommen wird. Außerdem werden Unsicherheiten zumeist überhaupt nicht erfasst und der zusätzliche Nutzen einer geeigneten ganzheitlichen Visualisierung wird darüber hinaus nicht ausreichend eingesetzt. Der im Folgenden erläuterte Ansatz zur Risikoanalyse, genannt sira, bietet die Möglichkeit, die technische Risikoanalyse mit der Wirtschaftlichkeitsbetrachtung zu kombinieren. Neben dem Risikobewusstsein-Check liegt eine weitere Stärke von sira in einer umfassenden Visualisierung aller Einzelrisiken als auch des Risikoportfolios des Gesamtprojektes, wobei neben den Risiken auch die Unsicherheiten in die Betrachtung einfließen können. Die umfassende Visualisierung, besonders als Vorstellung für das Top-Management geeignet, ermöglicht eine Betrachtung der Entwicklung des Risikoportfolios innerhalb eines Projektes und/oder als Vergleich zu anderen Projekten (Abb. 1.1).

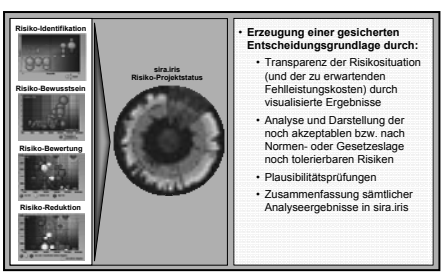


Abb. 1.1: Visualisierung der sira Risikoanalyse

Der PMI-Standard unterteilt Risikomanagement in vier Phasen. In der ersten Phase der Risikoanalyse, der Risikoidentifizierung, die iterativ durchzuführen ist, sollten alle für das Projekt relevanten Risiken erkannt und deren Eigenschaften dokumentiert werden. Dabei liegt der Schwerpunkt auf der Erfassung von zukünftigen und derzeit noch nicht bekannten Risikopotenzialen. Für sira, die über den PMI-Standard hinausgeht, ist es wichtig ein Ziel bzw. einen Bezugspunkt für die Analyse, wie beispielsweise einen Auslieferungstermin für eine Software, festzulegen. Workshops und Interviews mit unterschiedlichen Gruppenzusammensetzungen aller am Projekt beteiligten Personengruppen sowie Experten aus verschiedenen Geschäftseinheiten dienen dazu, die individuellen Erfahrungen und Kompetenzen der Mitarbeiter abzurufen und Risikopotenziale zu identifizieren. Eine weitere Voraussetzung für eine erfolgreiche Identifikation von Risiken ist die bereits in der Vorbereitung der Risikoanalyse auf Grundlage von Softwareanforderungen oder vertraglicher Themen erarbeitete hierarchische Themenliste. Diese Themenliste, bei Softwareentwicklungsprojekten bspw. nach Features oder Funktionen gegliedert, ist elementarer Bestandteil für die weitere Risikoauswertung.
Nach der Risikoidentifizierung mit der Aufnahme der genauen Beschreibung sowie der Eigenschaften der Risikosituation ist die Risikobewertung ein weiterer essentieller Prozessschritt. Um ein Risiko zu bewerten, ist es notwendig, den erwarteten Verlust im Falle eines Eintretens zu bestimmen. Das heißt, die Experten bestimmen in den Gesprächen die Höhe des möglichen Schadens im Eintrittsfall. Dafür müssen für jedes Risiko die Eintrittswahrscheinlichkeit und das finanzielle Schadensausmaß bei Eintritt von den Experten, die eine realistische Einschätzung der Risiken liefern sollen, bewertet werden. Hiermit erfolgt eine finanzielle Gesamtbewertung beziehungsweise die quantitative Risikobewertung vor Durchführung potenzieller risikoreduzierender Maßnahmen. Können Risiken nicht quantifiziert werden, wird eine Unsicherheitsbewertung des Risikos nach Handhabbarkeit, Vorhersagbarkeit und Schaden jeweils mit der Bewertung gering, mittel oder hoch vorgenommen.
Die Bestimmung, ob ein Risiko vorliegt oder nicht, und dessen Bewertung sind in diesem Prozess stark abhängig von der Subjektivität der beurteilenden Personen. Daher ist eine weitere wichtige Beurteilungsgröße eines jeden Risikos die qualitative Risikobewertung durch die Experten, die einer intuitiven Einschätzung des Risikos entspricht. Ist diese Bewertung zusammen mit der quantitativen Risikobewertung erfolgt, kann anschließend ein Risikobewusstsein-Check durchgeführt werden. Für den Risikobewusstsein-Check gilt, dass ein proportionaler Zusammenhang zwischen intuitiver Einschätzung und quantitativer Risikobewertung bestehen sollte.
In der dritten Phase von sira, der Risikobewältigung, werden Maßnahmen definiert und detailliert besprochen, um Risiken aktiv entgegenzuwirken. Maßnahmen entsprechen dabei einer oder mehreren Handlungsalternativen, die umgesetzt werden sollten, um ein gesetztes Ziel zu erreichen. Im hier betrachteten Fall werden Maßnahmen mit dem Ziel der Risikoreduktion umgesetzt. Nachdem entschieden wurde, welche Maßnahmen im Rahmen des Maßnahmencontrollings durchgeführt werden sollten, ist im letzten Schritt von sira deren Durchführung zu planen und zu überwachen.
Ein einmaliges Durchlaufen der vier Phasen ist auf Grund der heutigen Komplexität und einer sich schnell verändernden Projektumgebung ungenügend, um ein pro-aktives Risikomanagement zu betreiben. Die Überprüfung mittels sira sollte eine kontinuierliche Überwachung und Steuerung der gesamten Prozessschritte, das Monitoring, beinhalten.

Automatisierter, flexibler Modultest in der Zielumgebung

Thomas Eißenlöffel, Sebastian Moslener

MBtech Group
Kolumbusstr. 2
71063 Sindelfingen
info@mbtech-group.com

Abstract: Wie lassen sich verschiedene, am Markt verfügbare Werkzeuge zu einer flexiblen, kostengünstigen Lösung integrieren, um Kfz-Steuergeräte-Applikationen automatisiert zu testen? Die Testdurchführung soll sowohl auf einer Entwicklungsplattform als auch auf einer Zielprozessor-Plattform mit einer beliebig großen Anzahl von Tests erfolgen. Zusätzlich soll die Codeabdeckung der Gesamttests bestimmt werden.

Die Vorteile des hier aufgezeigten Konzepts werden hinsichtlich Qualitätsverbesserungen und Verkürzung der Testzeiten erläutert. Die Auswahl der Werkzeuge wird begründet und ihre Integration in verschiedene Testumgebungen beschrieben sowie die jeweiligen Grenzen und Alternativen aufgezeigt. Anhand eines realen Entwicklungsprojekts wird der Nutzen einer solchen Werkzeugkette dargestellt und von den Erfahrungen der Benutzer berichtet. Ein Ausblick auf mögliche zukünftige Erweiterungen schließt das Thema ab.

Aufbau der Toolkette

Zur Erfüllung der unterschiedlichen QS-Anforderungen bei der Softwareentwicklung, sowie der Einhaltung der Sicherheitsnormen bei der Steuergeräteentwicklung im Automobilbereich, wurden folgende Anforderungen an eine „Testsuite“ definiert:

- Plattform-Unabhängigkeit und hohe Flexibilität
- Einfache Test-Spezifikation und -Protokollerstellung
- Anforderungsverfolgung
- Einfache Handhabung
- Verarbeitung großer sowie dynamische Erzeugung komplexer Testvektoren
- Ermittlung der Testabdeckung C1/C2
- Einfache Integration verschiedener, am Markt verfügbarer Werkzeuge (Kosten)

Die von MBtech entwickelte Testsuite deckt diese Anforderungen ab und integriert verschiedene, am Markt verfügbare Werkzeuge. Bei der Auswahl wurde besonders darauf geachtet, am Markt eingeführte und verbreitete wie auch leicht integrierbare Werkzeuge zu verwenden, die sich zeit- und kostensparend in unterschiedlichen Kombinationen zu verschiedenen Testumgebungen zusammenstellen lassen.

Werkzeugkette / Entwicklungsumgebung

Die Werkzeugkette besteht aus Programmen zum Testmanagement, zur Codeabdeckungsanalyse und zur Generierung der Dokumentation, die je nach Testanforderung oder Projektspezifikation austauschbar und anpassbar sind. Je nach Bedarf lässt sich die Toolkette mit zusätzlichen Anwendungen ergänzen.

Die Entwicklungsplattform dient zum Test von Software-Modulen während der Entwicklung am Arbeitsplatz. Als Basis wird typischerweise ein PC unter Microsoft Windows eingesetzt. Diese Plattform besteht aus C-Compiler/Linker, Generierung Spezifikation, Unit-Test Managementtool und Coverage Analyzer.

Die Zielprozessor-Plattform HCS12 dient zum Test von SW-Modulen am Ende der Implementierungsphase unter Berücksichtigung der Eigenschaften der Zielplattform. Hierzu wird neben einem PC der Zielprozessor (z. B. ein Evaluierungsboard) benötigt und folgende Werkzeuge verwendet: Target - C-Compiler, Unit-Test Managementtool, Generierung Dokumentation, Coverage Analyser und Kommunikation Target ⇔ PC.

Die modellbasierte Entwicklungsplattform der MBtech, genannt PROVEtech:SD, dient der Prüfung der Anwendungssoftware in einer virtuellen Zielumgebung (z. B. Fahrzeug) und besteht aus Testumgebung, Testmanagement, C-Compiler/Linker und Coverage Analyzer.

Testergebnisse

Der Einsatz der Werkzeugkette auf einer Entwicklungsplattform ermöglicht die schnelle Verifikation von Software-Änderungen. Die Ausführung des Tests auf dem Zielprozessor erlaubt die Validierung des Objekt-Codes und damit u. a. das Auffinden und Umgehen von Cross-Compiler-Fehlern. Die Codeabdeckung lässt sich sowohl auf der Entwicklungsplattform als auch auf der Zielprozessorplattform ermitteln. Dieser Ansatz spart Zeit bei kleinen Code-Änderungen, da er sich auf einen einzigen Test auf der Zielprozessorplattform beschränkt.

Ausblick

Für nicht sicherheitskritische Anwendungen, bei denen eine Testüberdeckungsanalyse der Kategorie C0 ausreichend ist, kann ein Open Source-Werkzeug eingesetzt werden. Das Kommunikationskonzept der Zielprozessor-Plattform ist neben CAN-Bus auch auf weitere Kommunikationskanäle, wie z. B. FlexRay, übertragbar. Eine Erweiterung der Werkzeugkette um Tools zur Bestimmung der Testfälle ist angedacht.

Quasar Analytics: Aktive Qualitätssteuerung in großen und verteilten Softwareprojekten

Alex Hofmann

Capgemini sd&m Research
Capgemini sd&m AG
Carl-Wery-Str. 42
D-81739 München
alexander.hofmann@capgemini-sdm.de

1 Steigende Anforderungen an die Softwareentwicklung

Hoher Termin- und Budgetdruck, kürzere Entwicklungszeiten bei gleichzeitig komplexer werdenden Softwaresystemen, die zudem oft noch verteilt entwickelt werden: Die Anforderungen an die Entwicklung individueller Softwarelösungen steigen. Während auf die Optimierung von Prozessen in Entwicklungsprojekten bereits großer Wert gelegt wird, vernachlässigen viele Unternehmen die Produktqualität. Dabei macht sich eine schlechte Software-Qualität schnell bemerkbar: 60 bis 70 Prozent der IT-Budgets werden derzeit in Wartung und Weiterentwicklung investiert. Kosten und Risiken können aber reduziert werden, wenn die Produktqualität bereits im Entwicklungsprozess gesteuert wird. Dafür liefert Quasar Analytics Methodik und Werkzeuge: Qualitätseigenschaften von Software lassen sich eindeutig planen und Sollabweichungen werden früh erkannt. Software-Dienstleister können so ihre Projekte im vorgegebenen Budget- und Terminrahmen abwickeln und die Hauptanforderung der IT-Entscheider erfüllen: schnell stabile Systeme zu liefern.

Ziel von Quasar Analytics ist es, die Qualität der Software während des gesamten Entwicklungsprozesses anhand definierter Kriterien zu steuern. Diese Vorgehensweise geht deutlich über die heute zumeist betriebene Prüfung von Codequalität (mittels Findbugs, Checkstyle, CPD, JavaNCSS usw.) hinaus. Die gesamthafte Planung und Steuerung von Software-Qualität wird heute nur selten betrieben – auch weil dafür geeignete Werkzeuge und Standardisierungen fehlen.

2 Quasar Analytics als Gesamtansatz

Den Kern der Methodik bildet das sogenannte Qualitätsmodell, mit dem die Qualität der Software anhand von ganz konkreten Eigenschaften und Kennzahlen bewertet wird. Hierzu werden Sensoren in die Entwicklungsumgebungen integriert, die permanent und bereits sehr früh im Entwicklungsprozess Werte für die definierten Kennzahlen auf der Ebene von Komponenten messen. Die Werte werden in ein Software-Cockpit eingespeist. Darüber hat der Software-Architekt jederzeit Zugriff auf den aktuellen Qualitätsstand. Sollabweichungen werden schnell erkannt. Die Kennzahlen des Qualitätsmodells – das sogenannte „Blutbild der Software" – sind damit ein Frühwarnsystem für Qualitätsprobleme und ein aktives Steuerungsinstrument für anspruchsvolle Softwareprojekte.

Auf der Qualitätssteuerung über das Qualitätsmodell setzt die zweite Stufe auf: Quality-Gates – definierte Prüfpunkte vor kritischen Projektphasen, anhand derer die inhaltliche Reife einer Phase überprüft wird. Das erste Quality-Gate findet bereits vor Abgabe des Angebots statt und leitet zur Spezifikation der Software über. Ein weiteres Gate befindet sich im ersten Drittel der Spezifikationsphase, um wesentliche fachliche Entwurfsentscheidungen zu überprüfen. Vor einer Realisierung in der Breite befindet sich ein Gate, das die Konstruktionsentscheidungen noch einmal im Abgleich mit den Anforderungen des Kunden spiegelt und die Einhaltung von standardisierten Entwurfsprinzipien prüft. Die Prüfungen werden von externen, nicht am Projekt beteiligten Experten durchgeführt, um eine größtmögliche Objektivität zu gewährleisten. Im Rahmen dieser Qualitätssicherung werden die im Qualitätsmodell gewonnenen Kennzahlen noch einmal explizit anhand vorgegebener Sollwerte beurteilt und durch weitere Messungen ergänzt.

Die dritte Stufe – definierte Test- und Abnahmeverfahren auf Komponentenebene – ergänzt die Qualitätskontrolle. Hierzu gehören während der gesamten Entwicklungsphase funktionale und nichtfunktionale Testläufe, die beispielsweise die fachliche Korrektheit sowie Performance und Stabilität des Systems prüfen. Eine kontinuierliche Integration prüft die Zusammenarbeit der einzelnen Komponenten.

Literaturverzeichnis

[BB+78] B. W. Boehm, J. R. Brown, H. Kaspar, M. Lipow, G. McLeod, M. Merritt: Characteristics of Software Quality, North Holland, 1978.

[BW84] V.R. Basili, D.M. Weiss: A Methodology for Collecting Valid Software Engineering Data, IEEE Transaction on Software Engineering, SE-10, No.6, S.728-738.

[BEJ06] R. Buschermöhle, H. Eekhoff, B. Josko: Erfolgs- und Misserfolgsfaktoren bei der Durchführung von Hard- und Software-Entwicklungsprojekten in Deutschland, BIS Verlag der Carl von Ossietzky Universität Oldenburg, 2006.

[Fe96] N. E. Fenton, S. L. Pfleeger: Software Metrics – A Rigorous and Practical Approach, International Thomson Publishing Company, second Edition, 1996.

[Gr92] R. B. Grady: Practical software metrics for project management and process improvement, Prentice Hall, 1992

Projektvorhersage - die nichtlineare Zukunft

Markus Manleitner

Product Release
Dräger medical
Moislinger Allee 53-55
23542 Lübeck
Markus.Manleitner@draeger.com

1 Problem

In jedem Projekt gibt es Kennzahlen. Oft entsteht der Wunsch aus den jüngsten Trends eine Prognose für die Zukunft abzuleiten und sehr schnell wird durch die letzten Werte eine Gerade gelegt und weit in die Zukunft verlängert. Doch die Zukunft ist nichtlinear und deshalb führen viele leichtfertig erstellte Vorhersagen mit Regressionsgraden in die Irre. Zum einen, weil die Kennzahl gar nicht geeignet ist, zum anderen weil nicht alle Kennzahlen linear verteilt sind. Neben der Güte des Modells ist für den Anwender die Güte der Vorhersage viel entscheidender, aber kennen Sie diese Werte für Ihre Prognosen?

2. Modellierung

In rund 20 Projekten wurden die Fehlerfinderaten (Abb.1 links) und die kumulierten Summen pro Monat (Abb.1 rechts) aufgetragen, sowie eine Auswahl von möglichen Modellfunktionen, die den s-förmigen Verlauf der Daten unterstützen.

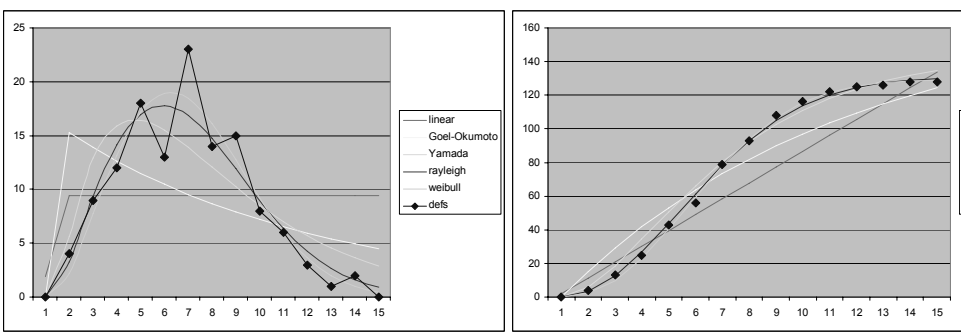


Abbildung 1 Fehler pro Monat und kumulierte Werte

Die Weibullverteilung erreicht durch drei Formparameter die beste Anpassung an die realen Datenpunkte (siehe Abb.2), dieser zusätzliche Freiheitsgrad reduziert jedoch in gleichem Maß die Güte der Vorhersage bei unvollständigen Datenreihen. Da der Formfaktor ohnehin bei allen Projekten nahe 2.0 lag haben wir uns daher für die Modellierung mit der Rayleigh-Verteilung entschieden und mit nur zwei Freiheitsraden eine gute Modellierung erreicht.

	linear		Goel-Okumoto		Yamada		Rayleigh		Weibull
corr	0,9648	corr	0,9837	corr	0,9953	corr	0,9992	corr	0,9996
error	4290	error	2462	error	430	error	56	error	41

Abbildung 2 Güte der Modelle

Die horizontale Skalierung der Modellfunktion erfolgt über $\mathbf{a=1/(2*T_{max}*T_{max})}$, dem Zeitpunkt, an dem die Verteilungsfunktion $\mathbf{v(t) = 2adte^{-at\hat{}2}}$ ihr absolutes Maximum erreicht. Die Güte der Abschätzung erfolgt über den Pearson-Korrelationsfaktor. Er verändert sich nicht mit der vertikalen Skalierung, womit eine Unabhängigkeit der beiden Parameter erreicht wird. Die vertikale Skalierung erfolgt über den Parameter d, der dem Maximalwert für V(t) für t -> unendlich entspricht. Die Güte der Abschätzung wird über die Summe der Fehlerquadrate ermittelt.

Für den Verwender ist weniger die Güte der Modellierung von Interesse, als die Güte der Vorhersage. Nach dem Wendepunkt können Genauigkeiten zwischen 0,1 und 5% für eine 6-Monatsprognose erreicht werden. Der Trend der beiden Formparameter a und d stellt ein wichtiges Indiz für die Güte der Vorhersage dar. Verlässliche Prognosen können also nur aus einem kontinuierlichen Metrikprogramm heraus erstellt und beurteilt werden.

3. Ausblick

Die S-Kurven lassen sich neben der Vorhersage auch für eine Reifegradbestimmung des Produktes verwenden Durch eine Normierung der Achsen lassen sich Projekte zum miteinander vergleichen, wenn die Prozesse dafür stabil und gleichartig genug sind.

Rayleigh-Verteilungen lassen sich auch auf Projektbudgets anwenden. [DL02] Die Anwendung erstreckt sich auf Plausibilitätsprüfung von Budgetplanungen, Vorhersage von zu erwartenden Ausgaben und Budgetüberschreitungen.

Literaturverzeichnis

[Kan02] Stephen H. Kan: Metrics and models in software quality engineering
[Lyu96] Michael R. Lyu: Handbook of software reliability engineering
[DL02] David Lee: Norden-Rayleigh Analysis: A useful tool for EVM in development projects

Global Sourcing und die Industrialisierung der Software-Entwicklung

Robert Martignoni

robert.martignoni@pagoges.de

1 Ausgangssituation und Ziel

Die Industrialisierung der Software-Entwicklung hat in den letzten Jahren an Bedeutung gewonnen. Zusätzlich ist Global Sourcing als Form des globalen Ressourcen-Managements nicht mehr aus der Software-Entwicklung wegzudenken. Die Verteilung der Entwicklung auf global verteilte Teams stellt neue Herausforderungen an die Vorbereitung, das Management und die Durchführung von Software-Entwicklungsprojekten. Existierende Entwicklungsprozessmodelle reflektieren diese Herausforderungen nur unzureichende.

Der Vortrag beschreibt einen möglichen Ansatz zur Optimierung von global verteilten Software-Entwicklungsprojekten. Dieser Ansatz basiert auf Grundpfeilern der Industrialisierung der Software-Entwicklung.

2 Lösungsansatz

Die Industrialisierung der Software-Entwicklung ist durch den Grad der Automatisierung, Standardisierung und Steuerungsmöglichkeiten geprägt.

Automatisierung Die Automatisierung dient primär der direkten Kosteneinsparung. Vor dem Hintergrund des Global Sourcing gewinnt die Automatisierung zunehmend an Bedeutung, da sowohl unterstützenden Management-Prozesse (automatisierte Qualitätsbewertung von Source Cods als Basis der Dienstleistersteuerung), wie auch Teile der Leistungserbringung automatisiert werden können.

Standardisierung Die intra-kulturelle Kommunikation ist ein oft genannter Grund für das Scheitern von Global Sourcing Projekten. Die Standardisierung der Kommunikation und der Abläufe unterstützt die Vermeidung von Fehlern.

Management Entwicklungsprojekte müssen von dem Kunden kontrolliert und gesteuert werden. In der Praxis findet eine Überprüfung der erbrachten Leistung nur an bestimmten Meilensteinen statt. Der Auftraggeber steht bei strategisch wichtigen Entwicklungen vor dem Problem die Entwicklung nicht zeitnah beeinflussen zu können.

Diesen Herausforderungen kann man durch den Einsatz einer zentralen Entwicklungs- und Projektumgebung begegnen. Eine schematische Darstellung zeigt die Abbildung 1.

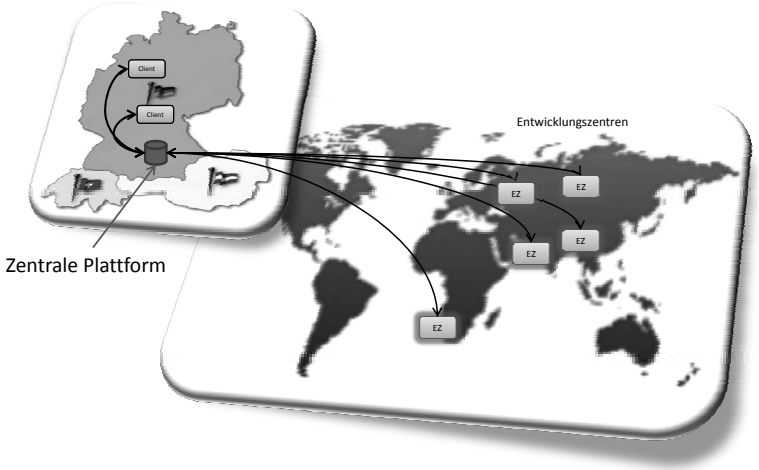


Abbildung 1: Zentrale Plattform als Lösungsansatz

Die zentrale Plattform übernimmt dabei kritische Unterstützungsleistungen für die Software-Entwicklung. Des weiteren werden benötigte Funktionen durch den zentralisierten Ansatz standardisiert (beispielsweise das Anforderungsmanagement oder das Issue Tracking). Die Investition zur Automatisierung von Prozessen sind in diesem Umfeld ökonomisch sinnvoll.

Der Einsatz einer zentralen Entwicklungsplattform zur Industrialisierung der Software-Entwicklung bietet die Möglichkeit Outsourcing Projekte effizient, mit gesicherten Qualitätsstandards und auch schnell umzusetzen.

3 Ergebnisse und nächste Schritte

Das Ziel dieses Vortrages war es die Industrialisierung der Software-Entwicklung, repräsentiert durch die drei Komponenten Automatisierung, Standardisierung, und Steuerung als eine nachhaltige Möglichkeit der Optimierung von Globalen Sourcing Projekten darzustellen. In einem ersten Schritt wurden die zentralen Herausforderungen hinsichtlich der Komponenten identifiziert. Es folgte die Fokussierung auf ausgewählte Entwicklungsprozesse und deren Untersuchung auf die Möglichkeiten der Industrialisierung. Dabei konnten die signifikanten Vorteile aufgezeigt werden. In einem nächsten Schritt werden diese nun qualitativ beschriebenen Vorteile, durch den Einsatz quantitativer Methoden präzisiert.

Stopping (and reversing) the architectural erosion of software systems. An industrial case study

Bernhard Merkle
Research& Development, Software Engineering
SICK AG Waldkirch, Germany
Bernhard.Merkle@sick.de
Bernhard.Merkle@gmail.com

Abstract: During the evolution of a software system it becomes more and more difficult to understand the originally planned software architecture. An architectural erosion happens for various reasons during the development phases.

In this paper we describe an approach to stop and reverse architectural degeneration of software systems. Using this method, the software architecture can even be improved, especially on the long term.

Many projects suffer a loss of the initial software architecture over the project lifetime and end up with a nearly unmanageable code base. Symptoms are e.g. undocumented software architectures, unwanted or unknown dependencies, and an unknown number of changes. In the end such systems often end like the "big ball of mud" and the only solution is to throw them away and start again from scratch. In most cases however one can not afford to lose the investment in the current code base, therefore it is essential to keep an eye on the architecture and monitor it on a regular basis. Recently several tools have become available to accomplish and automate this task and we were able to apply it to a critical project.

SICK AG is the worldwide leading provider of sensor solutions, especially laser scanners. We describe a real case study where we were able to stop and even reverse the architectural erosion.

1 Introduction

This paper describes a case study where we have been able to stop the architectural erosion of a large software system. We first give a short overview in which the terms of the software architecture area are defined and the main problem is outlined, which can manifest itself through different facets. In the following chapter we present a case study: a real software system developed by SICK AG, which suffered from an architectural erosion and was difficult to maintain and further extend.

2 Software Architecture

IEEE 1471-2000 [IEEE 2000] defines software architecture as “The fundamental organization of a system, embodied in its components, their relationship to each other and the environment, and the principles governing its design and evolution”. However two major problems can be found in lots of projects:

- Relationships are insufficiently defined: This means only the “main” relationships are defined in most projects, and they are defined in a very coarse manner, e.g. there is usually no differentiation between required, forbidden and optional relationships.

- Design and Evolution: The initial design is often covered and documented as mentioned above, but most problems arise with the evolution of the software (and hence the architecture).

3 Software Architecture Analysis

Software Architecture Analysis (SAA) can be applied for different purposes: e.g, Checking architectural consistency, Rating, Quantifying, Reviewing an architecture, Discovering an architecture, Measuring (real) facts, Monitoring changes.

3.1 Checking architectural consistency

The general approach is as follows: On the one hand the desired architecture with the requirements is described as the “should”-architecture. On the other hand an “is”-architecture exists which is extracted from the current code base. Via corresponding tool support a difference report can be generated, that shows or enumerates where the “should”-architecture and “is”-architecture differ, e.g. forbidden upward calls in a layered architecture from lower layers to upper layers can be detected.

3.2 Applying SAA on a industrial application: A case study

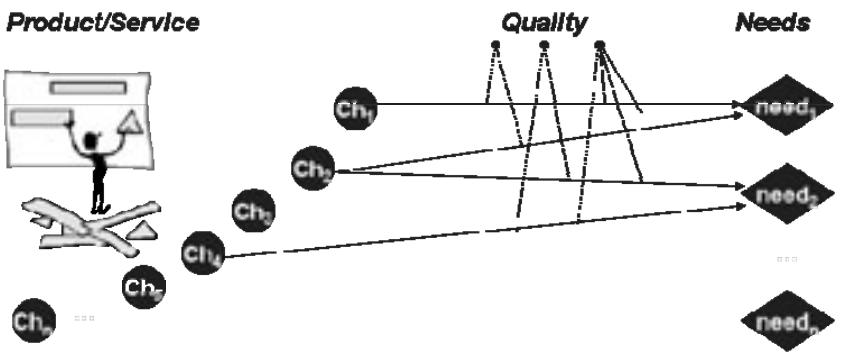
The Java application of our case study consists of about 300 kLOC and grew over about 6 years. Initially the software architecture was well defined, source code and architecture corresponded and the modules were designed in most cases as independent plug-ins. However, meanwhile the architecture degradation proceeded over the years and the modules were now tied together very closely. Most violations were calls from lower layers to upper layers and were able to remove them via moving the relevant classes to higher level packages. The problem with bidirectional dependencies was also resolved. The SAA-Tool displays the number of dependencies for each direction. A typical pattern in our software was a high number of dependencies in one direction and a lower number in the opposite direction.

360° Quality: Functional quality is not enough

Dr. Frank Simon

Software Quality Systems AG
SQS Research & Innovation
Stollwerckstraße 11
51149 Cologne
frank.simon@sqs.de

Most surveys about the success rate of IT-Projects demonstrate very dramatically that there is still missing a silver-bullet for doing large IT-Projects in time, in cost and in quality. Neither new techniques and tools nor new processes are increasing the success-rate. Instead of improving single steps within the Application Lifecycle it should be tried to change the general view on an IT-project. The idea is to have a very holistic view on project success, like already established in the context of quality management since many years. Understanding quality as "the characteristics of a product or service that bear on its ability to satisfy stated or implied needs" quality in general can be modelled like in the following Figure.



Corresponding to this model an IT-project depends on the following two factors: a) Holistic and systematic consideration of all needs and products/services as well as b) continuous transparency about the degree to which characteristics fulfil the needs.

Holistic View

Looking at the reasons for failed IT-projects one interesting fact is that needs/requirements can not be reduced to functional requirements in today's projects. If such a mono-dimensional view on quality is used there exist many high project risks. Typical quality aspects relevant for today's IT-Projects are:

- Maintainability: Most activities around an IT-System are understanding, analysing,

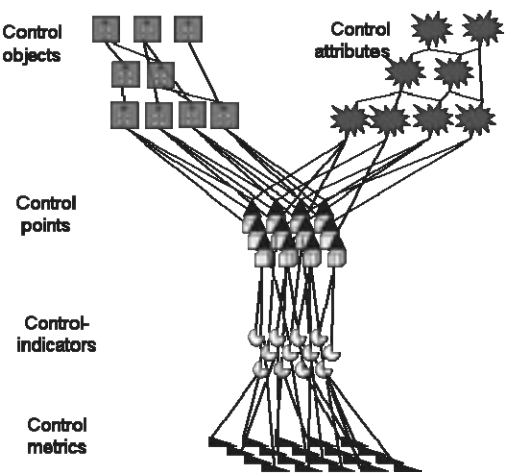
modifying and extending it. To enable these activities it is very important that the system has a corresponding high internal quality. Typical aspects there are to follow programming style-guides, application of well established design patterns as well as the reuse of experienced frameworks with an extensible architecture.

- License Compliance: To reuse existing source-code-snippets, libraries and frameworks is state-of-the-art for today's software development. Unfortunately there exist many licensing aspects that have to be considered when doing so. Ignoring this aspect can immediately stop every project. This is particular relevant when reusing components under the Gnu General Public License (GPL).

Another type of holistic view has to consider, that software can not be reduced to the delivered executable. The overall success of a project also depends on the Quality of the documentation (technical, domain-specific, operational, etc.), the traceability of the requirements, test-infrastructure etc.

Transparency

To have a continuous transparency about the degree to what the holistic requirements are fulfilled it is helpful to see an IT-Project as a typical engineering discipline: The most important characteristics of doing so is to have a systematic approach, to work in a very disciplined way and to concentrate on measurable check points. All of them are fulfilled by the so called Y-Model (see Figure), handling with control objects and control attributes to have a holistic view on the project. Control indicators based on control metrics ensures a high potential for automation and makes sure that the generated view is objective and repeatable.



Software Quality Systems AG

The SQS AG is the world-market-leader in the area of quality management and testing. SQS has about 1.500 employees and about 150 Mio. revenues and knows about all relevant control-points and how to check them continuously as early as possible. For real projects SQS hosts an experienced set of Best-Practices to make IT-Projects successful.

Frank Simon is the head of the SQS-Research & Innovation Group that has the task to maintain and update the set of experienced Best-Practices as well as to work on new services like model-based testing or architecture-verification.

Wirtschaftlich und wirksam: Entwicklungsprozesse auf Basis des Eclipse Process Frameworks

Rainer Singvogel

Cross-Bereich Software-Technologie
msg systems ag
Robert-Bürkle-Straße 1
85737 Ismaning
rainer.singvogel@msg-systems.com

1 Aufgabenstellung

Vorgehensmodelle zur Softwareentwicklung dienen insbesondere in größeren Projekten der Beherrschung der Komplexität. Oft sind diese Entwicklungsprozesse jedoch zu starr, zu allgemein oder zu umfangreich, weshalb sie nicht in dem gewünschten Maß zum Projekterfolg beitragen.

Der Beitrag zeigt, wie die msg systems die eigenen Entwicklungsprozesse gestaltet und dabei die genannten Schwierigkeiten adressiert. Die msg systems ist eines der 10 größten IT-Beratungs- und Systemintegrationsunternehmen in Deutschland und entwickelt sowohl individuelle Anwendungssysteme als auch branchenspezifische Standardsoftware.

2 msg.PROFI – Prozessrahmen für IT-Projekte der msg systems

Das Tätigkeitsspektrum der msg systems erfordert unterschiedliche Vorgehensmodelle, welche in dem **Pro**zessrahmen **f**ür **I**T-Projekte (msg.PROFI) organisiert sind.

Aktuell umfasst msg.PROFI Vorgehensmodelle insbesondere für die Anwendungsentwicklung von Individualsoftware und die Implementierung von SAP-Standardsoftware. Dabei wurden jeweils die wesentlichen Elemente von vorhandenen (Industrie-)Standards (dem Open Unified Process [UP], der Einführungsmethodik „ASAP“ der SAP AG sowie dem Projektmanagementstandard nach GPM [GPM]) übernommen und um msg-spezifische Erfordernisse und Erfahrungen ergänzt.

Als Modellierungswerkzeug wurde das Eclipse Process Framework [EPF] ausgewählt, das die Modellierung von Entwicklungsprozessen auf Basis des Meta-Modells SPEM 2.0 der OMG [OMG] ermöglicht, Repository-basiert arbeitet und die Publikation der Entwicklungsprozesse in Form statischer Webseiten ermöglicht.

Die Gestaltung des Intranet-Auftritts von msg.PROFI berücksichtigt die Ergebnisse einer Zielgruppenanalyse für die msg systems. So gibt es für die verschiedenen Anwendungsfälle zu msg.PROFI spezifische Sichten bzw. Zugänge. Für Nutzer, die die Inhalte von msg.PROFI bereits gut kennen, wurde besonderer Wert darauf gelegt, dass Hilfsmittel (u.a. Templates, Checklisten) auf möglichst kurzem Wege erreichbar sind.

Besonders die Vererbungsmechanismen des EPF und die Anpassbarkeit dieses Werkzeugs wurden ausgenutzt, um Gemeinsamkeiten der verschiedenen Vorgehensmodelle redundanzfrei zu modellieren und die Übersichtlichkeit des Webauftritts zu optimieren. So werden nicht alle Elemente von SPEM 2.0 verwendet. Vor allem auf die Nutzung von Descriptoren, die spezifische Ausprägungen von Aktivitäten, Rollen oder Ergebnissen in unterschiedlichen Projektphasen beinhalten, wurde weitgehend verzichtet. Durch eine msg eigene Erweiterung des EPF stehen die Inhalte von msg.PROFI auch in Form von MS-Word-Dokumenten zur Verfügung.

Die Inhalte des Webauftritts können mit Hilfe eines angebundenen Wikis von allen Mitarbeitern erweitert werden, wodurch eine immer breitere Wissensbasis entsteht. Die Akzeptanz und damit Wirksamkeit der Prozesse wird erhöht durch eine stark ziel- und zweckorientierte Formulierung von Rollen, Aktivitäten und Ergebnistypen. Die Automatisierung des Publikationsprozesses ermöglicht einen effizienten Betrieb von msg.PROFI und fördert eine hohe Aktualität der Inhalte und Hilfsmittel. Ein umfangreiches Schulungsprogramm und entsprechende Prozess-Coaches unterstützen die Wirksamkeit von msg.PROFI.

3 Ausblick

Die Entwicklung und der Einsatz von msg.PROFI in den letzten 6 Jahren zeigen, dass die einfache Nutzbarkeit sowie eine hohe Aktualität der Inhalte wesentliche Erfolgskriterien für Entwicklungsprozesse sind.

Durch neue Projekt-Erfahrungen und Methoden nimmt die Menge an verfügbaren Informationen in der Regel stetig zu. Es ist daher eine dauerhafte Herausforderung, den Projekten einfache und übersichtliche Zugänge zu diesen Informationen zu ermöglichen, was für das bedarfsgerechte, projektspezifische Tailoring eine wichtige Voraussetzung ist. Ansatzpunkte dabei sind, die Inhalte top-down noch stärker zu strukturieren z.B. in Form von Practices, wie sie in [EssUP] vorgeschlagen werden.

Literaturverzeichnis

[UP] Eclipse Foundation, Inc.: Open Unified Process; http://epf.eclipse.org/wikis/openup/

[GPM] Deutsche Gesellschaft für Projektmanagement e.V.: http://www.gpm-ipma.de

[OMG] Object Management Group, Inc.: Software Process Engineering Meta-Model, Version 2.0; http://www.omg.org/technology/documents/formal/spem.htm

[EPF] Eclipse Foundation, Inc.: Eclipse Process Framework; http://www.eclipse.org/epf/

[EssUP] Ivar Jacobson Int.: Essential UP; http://www.ivarjacobson.com/products/essup.cfm

Management großer Projekte – Ein modellbasierter Ansatz

Dehla Sokenou
GEBIT Solutions, Koenigsallee 75b, D-14193 Berlin
dehla.sokenou@gebit.de

1 Motivation

Betrachtet man ein typisches Softwareprojekt, so werden oft Aussagen wie "Die Komponente X ist zu 80% fertig" gefällt. Solche Aussagen haben den großen Nachteil, dass sie keinerlei Rückschlüsse auf den aktuellen Stand des Projekts erlauben, z.B. ist nicht gesagt, wie viel Zeit die fehlenden 20 % noch kosten. Wenn solche Aussagen zu einem Gesamtstatus kumuliert werden, kann dies zu einer deutlichen Überschätzung des Projektfortschritts führen. Verzögerungen, Fehlplanungen und ungenügende Steuerung sind oft die Folge.

2 Lösungsansatz und Vorgehen

Ziel ist die Planbarkeit und Verfolgbarkeit auch großer Projekte, ohne dass sich dadurch der Aufwand für die Projektleitung in einer Weise vergrößert, dass eine sichere Aussage über den aktuellen Projektstatus nicht mehr möglich ist. Um dieses Ziel zu erreichen, schlagen wir ein Verfahren des Projektmanagements vor, das auf wenigen Regeln basiert und einfach während des Projekts gepflegt werden kann. Unser Verfahren beruht auf Anforderungen, die als Use Cases formuliert werden. Neben funktionalen Anforderungen werden auch nichtfunktionale Anforderungen als Use Cases formuliert, um das Verfahren einheitlich auf alle Anforderungen anwenden zu können. Die Regeln sind im Einzelnen:

Regel 1: "*Die Größe eines Use Case muss überschaubar sein.*"
Aus unserer Erfahrung hat sich gezeigt, dass der Umfang eines Use Cases möglichst klein gehalten werden sollte. Typischerweise sollte man Use Cases mit einem Umfang von etwa 5 Personentagen planen. Eine so feingranulare Planung ist meist nicht bei Projektbeginn möglich. Erleichtert wird die Zerlegung durch die Aufteilung des Projekts in Phasen, denen einzelne grobgranulare Use Cases zugeordnet werden. Diese grobgranularen Use Cases werden zu Beginn einer Phase in kleinere Use Cases verfeinert, bis der geforderte Umfang erreicht ist. Feingranulare Use Cases ändern schneller ihren Status und sind deshalb besser planbar. Hat ein Use Case im Umfang von 5 Tagen eine Verzögerung von z.B. 20%, so fällt dies viel früher auf, als bei einem Use Case mit geplanten 100 Tagen. So lässt sich zu jedem Zeitpunkt eine sehr gute Abschätzung des Gesamtprojektstatus erreichen. Massnahmen zur Gegensteuerung können sehr viel schneller ergriffen werden.

Regel 2: "*Es gibt eine begrenzte Menge von Status. Erst wenn ein Use Case einen neuen Status vollständig erreicht hat, wird sein Status geändert.*"
Oft wird der Projektstatus aus Aussagen kummuliert, die eine teilweise Fertigstellung von Use Cases enthalten. Das betrifft meist nicht nur die Implementierung, sondern auch "teilweise spezifiziert" oder "teilweise getestet" kommen vor. Solche Aussage sind jedoch wenig hilfreich, denn ein teilweise implementierter Use Case kann in der Regel noch nicht in das Gesamtsystem integriert oder getestet werden. Statt Teilaussagen zu machen, wird der Status eines Use Cases hier aus einer begrenzten Menge definiert. Für das erfolgreiche Erreichen eines Status müssen genaue Kriterien definiert werden. Farbliche Kodierung der Use Cases je nach Status erlaubt auch bei großen Projekten einen einfachen Überblick, wieviele Use Cases bereits implementiert oder abgenommen sind. Gleichzeitig kann eine genauere Auswertung und somit eine tagesaktuelle Statusdokumentation durch die Generierung entsprechender Dokumente "on-demand" erzeugt werden.

Regel 3: "*Erstellte Use Cases sollten möglichst schnell fachlich abgenommen werden.*"
Diese Regel garantiert eine schnelle Kontrolle der implementierten Anforderungen durch den Kunden und verhindert ein Implementieren an den Kundenwünschen vorbei. Da jeder Use Case eine in sich geschlossene Einheit implementiert, kann diese auch getrennt fachlich getestet und abgenommen werden. Im Gegensatz dazu stellt ein sehr späte Abnahme durch den Kunden ein hohes Projektrisiko dar.

Regel 4: "*Es gibt keinen Weg zurück.*"
Eine Änderung des Status eines Use Case erfolgt immer nur in eine Richtung, d.h. ein einmal erreichter Status kann nicht wieder zurück auf den vorherigen Status gesetzt werden. Werden z.B. nach der Abnahme Änderungswünsche von Kundenseite geäußert, so werden diese als Change Requests aufgenommen. Die Regel garantiert, dass ein einmal erreichter Projektstatus beibehalten wird. Für alle Projektbeteiligten gibt es keine Überraschungen, so ist z.B. klar, wie mit neuen Kundenanforderungen umzugehen ist.

3 Erfahrungen und Ergebnisse

Das beschriebene Verfahren wurde bereits erfolgreich in diversen, auch größeren Softwareprojekten eingesetzt. Auch wenn es zunächst problematisch erscheint, dass ein Projekt eine große Menge feingranularer Use Case definiert, ist unsere Erfahrung, dass diese zu einer deutlich besseren Übersicht und Planbarkeit des Projekts führen. Durch feingranulare Use Cases weicht der kumulierte Projektstatus nur wenig vom tatsächlichen Status ab. Der Projektstatus ist immer eine "Worst Case" Betrachtung, was eine deutlich höhere Projektsicherheit garantiert als eine zu optimistische Schätzung. Natürlich geht dies nicht ohne die entsprechende Werkzeugunterstützung. GEBIT hat dazu eine modellbasierte Werkzeugkette entwickelt, die von den Anforderungen bis zu Test und Projektmanagement eine durchgängige Softwareentwicklung unterstützt. Als Beispiel sei hier das in Zusammenarbeit von GEBIT und DfG für die OBI-Baufachmärkte entwickelte Marktwarenwirtschaftssystem genannt. In diesem durchgängig modellbasierten Projekt wurden innerhalb von 1,5 Jahren über 1.500 Use Cases realisiert. Trotz des großen Projektumfangs erfolgte die Umsetzung in Time & Budget und in hoher Qualität.

Modellgetriebene Entwicklung eingebetteter Systeme mit Eclipse

Axel Terfloth, Lothar Wendehals

itemis AG
Am Brambusch 15-24
44536 Lünen
axel.terfloth@itemis.de, lothar.wendehals@itemis.de

1 Integrierte Werkzeugketten für eingebettete Systeme

Entwicklungsmethoden und -werkzeuge für eingebettete Systeme und Enterprise Applikationen nähern sich zusehends an. Insbesondere die modellgetriebene Entwicklung hilft beim Umgang mit zunehmend komplexeren, eingebetteten Softwaresystemen. Zur Automatisierung der Entwicklung ist die Werkzeugunterstützung entscheidend. Die Open-Source-Entwicklungsplattform Eclipse stellt mit dem Eclipse Modeling Project eine leistungsfähige Infrastruktur zur Entwicklung der Werkzeuge. Außerdem ist Eclipse eine Integrationsplattform für Werkzeugketten. Anhand praktischer Beispiele werden wir Aspekte Eclipse-basierter MDSD-Werkzeugketten für eingebettete Systeme diskutieren.

1.1 Werkzeug- & Modellintegration

Die Werkzeuglandschaft für eingebettete Systeme ist von Speziallösungen bis hin zu universell einsetzbaren Entwicklungsumgebungen weit gefächert. Zur Integration der Modelle verschiedener Werkzeuge gibt es grundsätzlich zwei Methoden: ein gemeinsames Modell oder die Synchronisation der verschiedenen Modelle. Bei einem gemeinsamen Modell wird das spezialisierte Modell eines Werkzeugs durch Modell-zu-Modell-Transformation aus dem gemeinsamen Modell gewonnen. Nach der Bearbeitung werden Änderungen des spezialisierten Modells zurück in das gemeinsame Modell transformiert. Bei der Synchronisation einzelner Modelle werden dagegen Änderungen paarweise von einem in das andere Modell propagiert und eventuelle Transformationen durchgeführt.

1.2 Metamodellierung, DSLs und Editoren

Zur Entwicklung eingebetteter Systeme findet zunehmend die UML mit speziell zugeschnittenen Profilen wie MARTE Anwendung. Daneben sind jedoch auch etablierte Metamodelle wie Blockschaltsysteme, die Architecture Analysis and Design Language (AADL) oder Standards wie AUTOSAR im Automotive-Bereich relevant. Eine weitere Klasse sind die Domain Specific Languages (DSL), die sowohl in grafischer als auch textueller Form vorliegen können. Das Eclipse Modeling Project bietet eine Basis für DSLs. Das darin enthaltene Eclipse Modeling Framework (EMF) hat sich als Standard für Metamodellierung und -modelle etabliert. Neben grafischen Modellierungswerkzeugen werden auch Grammatik-basierte Ansätze zur textuellen Modellierung unterstützt.

1.3 Validierung

Die sehr detaillierten Modelle eingebetteter Systeme erfordern eine frühe, automatisierte Überprüfung der Korrektheit, damit Auswirkungen von Änderungen nachvollziehbar bleiben und unnötige Build/Deploy/Test-Zyklen vermieden werden. Metamodelle können jedoch nur teilweise die syntaktische Korrektheit des Modells sicherstellen. Auch Einschränkungen bei der Freiheit der Modelleditoren sind nur bis zu einem gewissen Grad praktikabel. Einige syntaktische, aber vor allem semantische Vorgaben an das Modell können weder vom Metamodell noch von speziellen Editoren gewährleistet werden. Deshalb ist eine explizite Validierung des Modells durch teils sehr umfangreiche Regelsätze notwendig.

1.4 Simulation und Visualisierung

Die Simulation von Teilsystemen ist wesentlich in der Entwicklung eingebetteter Systeme, da während der Softwareentwicklung das eigentliche Zielsystem für Tests oftmals nicht zur Verfügung steht. Die Simulation ist prinzipiell auf alle Verhaltensmodelle anwendbar. Existieren Simulations-Engines, können diese mit dem Editor gekoppelt werden. Dafür sollte die Notation der Modelle um zusätzliche dynamische Elemente zur Darstellung der Simulation erweitert werden. Bei der Simulation eines Zustandsautomaten können so zur Visualisierung zum Beispiel aktive Zustände oder durchgeführte Transitionen hervorgehoben werden.

1.5 Codegenerierung

Die Codegenerierung für eingebettete Systeme unterscheidet sich bezüglich der zugrunde liegenden Techniken nicht wesentlich von der für Enterprise-Anwendungen. Da eingebettete Systeme oftmals jedoch sicherheitskritisch sind, sind formale Testverfahren, die unter anderem Code-Coverage-Messungen des Generator-Codes enthalten, notwendig. Bei der Produktlinien-Entwicklung, bedingt durch die extrem große Vielfalt der Zielplattformen, ist des Weiteren ein modularer Aufbau der Codegenerierungs-Templates hilfreich. Die für die Zielplattform benötigten Templates können dadurch gezielt zusammengesetzt werden. Durch Aspekt-orientierte Programmierung (AOP) können außerdem je nach Zielplattform zusätzliche Informationen in die Templates eingebracht werden. Xpand aus dem openArchitectureWare-Projekt unterstützt diese Techniken.

2 Das Beispiel

Die angesprochenen Aspekte werden an einer konkreten, Eclipse-basierten Beispiel-Werkzeugkette vorgestellt, die als Open-Source verfügbar ist. Die Werkzeugkette umfasst Editoren zur grafischen Modellierung von Zustandsautomaten sowie zur grafischen und textuellen Modellierung von Blockschaltsystemen. Die Validierung und Simulation der Modelle sowie die Codegenerierung gehören ebenfalls zur Werkzeugkette. Die verschiedenen Modelle werden mit Hilfe von Modelltransformationen in die Werkzeugkette integriert.

Software Architects and Testers – Collaboration Required!

Peter Zimmerer

Corporate Technology Software & Engineering (CT SE 1)
Siemens AG
Otto-Hahn-Ring 6
D-81739 Munich
peter.zimmerer@siemens.com

1 Introduction

Today there is common sense that collaboration is a key success factor in software development. Especially agile methodologies emphasize this fact (see [AM01]).

This article focuses on the collaboration between two important stakeholders: software architects who play a very essential role in the development and maintenance of products and systems and testers.

> *"Architects build software systems and testers test these systems" – so why should an architect need to know anything about testing?*

To answer this question section 2 motivates testing by providing a multidimensional view on what testing really is. Then in section 3 practices in testing regarding software architecture are presented which require active involvement and contributions by software architects. This results not only in better quality but also speeds up development by facilitating change and decreasing maintenance efforts.

2 Testing – why? What is the value of testing?

First of all software architects and testers must have a common understanding what testing is all about and appreciate the value of testing. Unfortunately there are still out-dated definitions used for testing (see [IEEE 610.12-1990] or [IEEE 829-2008]) which just limit testing to "dynamically execute a piece of software to detect bugs" but there is a growing perception (at least in the testing community) on the different dimensions of testing (see [GH88] and [IG07]):

- Demonstrate, check: show it works, gain confidence (confirmatory, constructive)
- Detect, search: find bugs as early as possible (investigative, exploratory, destructive)
- Mitigate and reduce risks: "*no risk, no test*"
- Assess, evaluate, and predict: measure quality attributes (e.g. performance)
- Prevent: Control, influence, and drive quality

These dimensions result in the overall mission and value of testing by *providing information related to quality* as stated by Cem Kaner (see for example [CK06]):

> *Testing is an empirical technical investigation of the product / system / artifact / service under test conducted to provide stakeholders with information about the quality.*

3 Practices in testing regarding software architecture

There are several testing practices which require active involvement by software architects:

- Understand the mission and the value of testing (see 2) and promote it in the development team.
- Risk-based testing strategy – actively participate in a product risk analysis workshop as one important stakeholder, perform architecture testing (see [GO08]), and map architectural risks to appropriate testing levels.
- Test-driven development – do not use it only for unit and acceptance testing but rather use the fundamental idea of preventive testing (see [GH88]).
- Design for testability – must be built into the architecture and design (by architects).
- Integration testing – is *the* important testing level to check the architecture.
- Test architectures, test automation – use architecture excellence to support and improve the test architecture and the test automation system.
- Regression testing – use the knowledge about the architecture and design of the system during change and impact analysis to select required regression test cases effectively based on priority, criticality, risks, and frequency of usage.
- Architectural quality – continuously check the internal software quality of the developed system or product and perform adequate code quality management.

4 Summary

Software architects must cooperate closely with testers to define, motivate, drive, and enforce a comprehensive understanding of and attitude to testing and quality in the whole team. Thereby, they address quality from different viewpoints:

- Software architects: build quality into the system (or rather into the architecture)
- Testers: provide information related to quality (feedback)

Software architects must know and understand their job w.r.t. testing: the different dimensions of testing (see section 2) and their required active involvement in the area of testing (see section 3).

Testers must bring to light what software architects should do w.r.t. testing (see 3). Use these practices to drive the needed collaboration and to educate software architects in better testing which definitely improves effectiveness and efficiency in software testing.

References

[AM01] Agile Manifesto: http://agilemanifesto.org/

[CK06] http://www.kaner.com/pdfs/ETatQAI.pdf

[GH88] D. Gelperin, B. Hetzel, The Growth of Software Testing, Communications of the ACM, Vol. 31, Issue 6, June 1988, pp. 687-695

[GO08] Google search for “architecture testing”, http://www.google.de/search?hl=de&q=architecture+testing&meta=

[IG07] ISTQB Standard Glossary of Terms used in Software Testing V.2.0, Dec, 2nd 2007, http://www.istqb.org/downloads/glossary-current.pdf

Forschungsarbeiten

Modellierung

A Comprehensive Survey of UML Compliance in Current Modelling Tools

Holger Eichelberger, Yilmaz Eldogan, Klaus Schmid

Software Systems Engineering
University of Hildesheim
Marienburger Platz 22
31134 Hildesheim
{eichelberger, eldogan, schmid}@sse.uni-hildesheim.de

Abstract: The Unified Modeling Language (UML) specification is widely adopted in software engineering. When tools do not fully implement the UML specification, the user might be locked-in to a modeling tool, e.g. when exported models are not compatible among tools or tools implement different subsets of the UML. These compatibility problems also have significant impact on the effectiveness of model-driven development approaches. Compliance, as defined by the UML standard, is intended to characterize tools and to highlight such problems. In this paper we describe an approach to asses the UML compliance levels of modeling tools. Using UML definition of compliance, we could only identify 4 out of 68 tools as being acceptable.

1 Introduction

The OMG specifications on the Unified Modeling Language (UML) are widely used in software engineering, e.g. for the design and the documentation of software systems [DP07]. Various tools support these activities by implementing the OMG specifications. In several situations, problems may occur when the tool being used does not sufficiently adopt the specification, e.g. when required model elements are not supported or when incompatible formats prevent migration or further processing. In order to ensure interoperability among tools, the OMG defined a compliance schema consisting of several compliance levels.

Recently, we conducted a detailed and encompassing study [EES08] analyzing the functionality of current UML modeling tools. This study was developed mainly in order to provide detailed information to decision makers in industry and research, when facing the situation of a tool selection. Here, we will use the subset of results of this analysis that directly relates to UML compliance as defined by the UML. Most tool comparisons published so far provide only generic high-level information, e.g. the price of a tool, the supported diagram types or the version of XMI implemented by a tool, thus we decided to focus in this paper on the aspect of compliance as defined in the UML specification.

Therefore, we describe the evaluation approach of our study and discuss the results, i.e. the current status of professional modeling tools in this regard.

In contrast to existing UML tool evaluations, we describe in this paper a feature-based evaluation approach driven by the OMG specifications, and in particular the definition of UML compliance as given by the UML. This detailed evaluation was executed on a sample set of 68 UML tools, covering – in particular – all major professional tools that are widespread in practice. The result of our approach provides detailed tool profiles including the individual UML compliance level per tool. Thus, another important aspect of our approach is the consistent application of an evaluation approach based on a specification to a large set of tools. In fact, the data obtained by the underlying study can be used to support a systematic decision process to guide the selection and adoption of UML modeling tools. As we focus only on UML compliance in this paper, we will not discuss aspects of tool selection and adoption in general here.

The remainder of this paper is structured as follows: In the next section, we review related work on tool comparisons and research approaches for tool evaluations. Then, in Section 3, we describe our evaluation methodology and the evaluation process. In Section 4, we present an overview on the aggregated results of the study. In the final section, we draw some conclusions and discuss future work.

2 Related Work

In this section, we discuss relevant related work. The OMG defined the compliance to the UML specification in [OMG07b, OMG07a]. To our knowledge, no other evaluation directly relates to UML compliance. Even in [BFSC08] the authors describe the generation of compliance test suites for UML taking the meta model and the wellformedness constraints into account but explicitly characterizing the results in terms of the UML compliance definition, e.g. the compliance levels.

We first cover existing tool lists and tool comparisons. Then we review the research approaches on rating, testing and evaluating UML tools from different perspectives. Finally, we discuss work on cross-tool XMI compatibility as an example for work on specific compliance issues.

Several lists enumerating tool and vendor data can be found in the literature and on the internet, e.g. the OMG tool vendor listing [OMG08] or the Wikipedia page on UML tools [www08]. For a decision maker in particular tool comparison matrices like [Jec04, OOS08] summarize for up to 100 different tools high-level features like the supported types of UML diagrams, formats for import and export, target languages for code generation etc. Also, a commercial study covering several aspects on tool support for software development [BFO+05] reviews twelve selected UML tools by comparing the implemented types of UML diagrams, prices and the supported versions of the XMI format. The major drawback of these comparisons and listings is that the provided results are rather generic, i.e. the reader cannot conclude from the presence of a certain UML diagram that required modeling facilities are actually implemented. As a result, an

in-depth discussion of the compliance with the UML specification is also in the case of the commercial study out of scope and left to the reader. Furthermore, there is a significant difference in the quality of the information provided, i.e. whether data is collected from vendor statements or by directly analyzing a tool implementation as done in our study.

Research work in this area focuses often specifically on the evaluation approach itself. In [KF04] an evaluation framework for UML tools including 29 UML criteria, economic issues and usability issues is applied for the evaluation of 7 UML tools. A hierarchical, feature-oriented approach characterizing high-level tool functionality is given in [FDSP05]. In fact, the work in [FDSP05] is most closely related to ours, but the data aggregation is more complex, does not consider unclear feature implementations and the criteria list consists only of high-level tool functionality. Compared with the tool lists discussed above, some of the evaluations in research approaches consider more detailed but also incomplete feature lists [KF04, FDSP05]. Furthermore, all research driven evaluations rely on rather small sample sets. This is not only a drawback of these studies from a practical perspective, but also from a research perspective, as it leads to an insufficient validation of the evaluation frameworks themselves.

While all related work mentioned above is intended to provide an overview on several features of the considered tools, there is also some work on individual aspects, e.g. the ability to exchange models given in XMI format. For example, in [LLPM06] the authors describe a cross-tool compatibility test for Eclipse-based UML modeling tools. The same authors published also further evaluations on that topic for open source or commercial tools. As a result, only few tools are able to pass a backward compatibility test and also few tool combinations are able to share XMI models among each other. The final conclusion of these evaluations was that the user is locked-in to the selected modeling tool. A better model format compliance would also stabilize the technological foundation for automated test-suites e.g. as described in [BFSC08]. A drawback of the work on analyzing XMI compliance of only modeling tools is that further tools being used in practice like model transformation or code generation frameworks providing tool-specific XMI filters are not considered at all. Thus, a detailed analysis of the modeling facilities provides relevant information beyond a pure analysis of XMI compliance.

As discussed in this section, the work published so far has two major shortcomings: Most tool comparisons offer rather generic information, e.g. the provided types of UML diagrams, or rely on a small tool sample set. Regarding the level of detail, we characterize the abilities of individual tools by collecting information on modeling features defined by the UML specification. By aggregating that data, we obtain detailed tool profiles on the high-level facilities, e.g. on the fraction of features implemented for a certain diagram. This leads to a more realistic description of the tools. Regarding the sample set, we initially scheduled 200 UML modeling tools for evaluation instead of selecting an arbitrary subset as done in several studies. Due to the fact that many of these tools are not available anymore, we carried out an exhaustive feature-oriented evaluation on the subset of 68 available tools.

3 Methodology

In this section, we describe the methodology used in our evaluation. The basis of our evaluation is UML compliance as defined by the OMG, in particular the defined compliance levels. In Section 3.1, we introduce the UML compliance schema. In Section 3.2, we discuss the systematic construction of the evaluation criteria based on the UML compliance schema and we outline the evaluation procedure. In Section 3.3, we sketch the overall course of the evaluation.

3.1 Compliance to the UML Specification

The UML specification documents explicitly define the conformance to the specification in the UML Infrastructure [OMG07b], the UML superstructure[OMG07c], the XMI specification [OMG07a] and the diagram interchange specification [OMG06]. Our evaluation relies on UML specification version 2.1.1 which was the published version when preparing the evaluation. Our results are also valid for the current version 2.1.2 of UML, because the minimal changes do not interfere with our work.

The UML defines a two-dimensional compliance schema in the UML infrastructure and the UML superstructure. We outline both dimensions below.

The first dimension introduces the so called **compliance levels** in terms of parts of the UML language. *Level 0* requires the implementation of the simple class-based modelling language as specified in the UML infrastructure. *Level LM* adds profiles and more detailed class modelling to Level 0. *Level 1* extends the class diagrams from Level 0 to meet the UML superstructure specification and furthermore requires use case, sequence, timing, component, composite structure and activity diagrams. For compliance with *Level 2* also deployment diagrams and state machines must be implemented. Finally, *Level 3* represents the entire UML superstructure specification, also including information flows, templates and model management. Except for the metamodel level LM, which is defined in the UML infrastructure, the UML superstructure subsumes the UML infrastructure in this regard.

The second dimension details the **degree of realization**. For *abstract syntax compliance* both, UML metamodel and model persistence according to the XMI format must be implemented. The UML diagramming language as given in the UML superstructure must be realized for *concrete syntax compliance*. Realizing both, the metamodel and the diagramming language leads to *abstract and concrete syntax compliance*. When a tool also provides diagram interchange according to the UML specification, the related level is called *abstract and concrete syntax compliance with the ability of model and diagram persistence.*

In the remainder of this paper we refer to the compliance levels according to the following notation. L2-0, L2-M, L2-1, L2-2 and L2-3 denote the compliance levels of UML 2 as introduced above. To indicate the syntax compliance, we add appropriate abbreviations to the compliance levels, e.g. L2-0A as abstract syntax compliance on Level 0 or L2-1AC for abstract and concrete syntax compliance on Level 1. As some

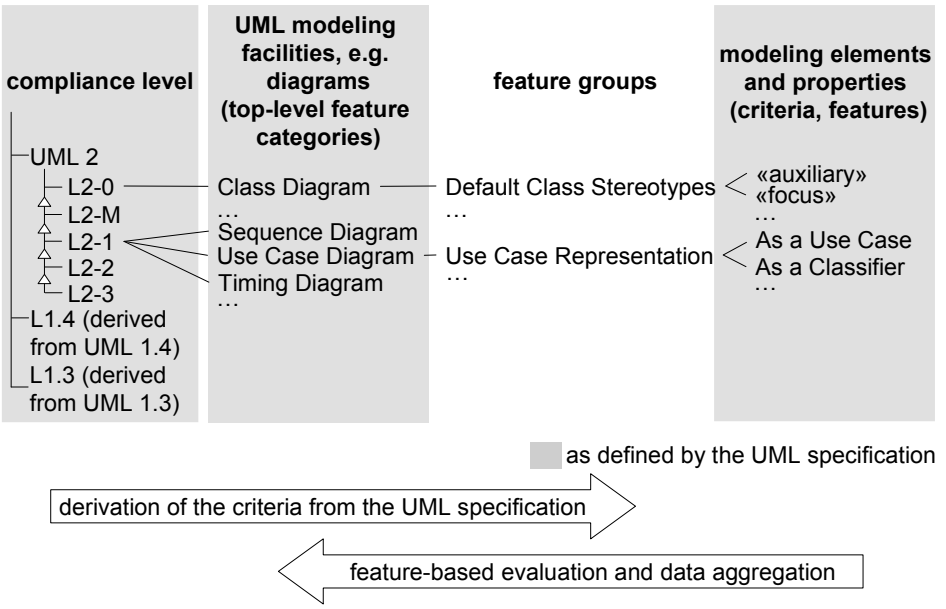


Figure 1: Derivation of features as evaluation criteria and feature-based evaluation.

tools still implement an earlier version of the UML, we use L1.3 and L1.4 to denote the compliance to all elements defined either in UML version 1.3 or 1.4, even though these versions of the UML do not define an explicit compliance mechanism. Thus, L1.3 and L1.4 denote the compliance to the entire specification of the respective version and no additional abbreviations are appended. When deriving the concrete compliance level for a tool, in some cases a tool may fulfil multiple levels simultaneously. In fact, L2-M subsumes L2-0 and L2-3 the levels L2-2, L2-1, L2-0 in the given sequence. For the analysis of our findings, we assign the maximum compliance level to a concrete tool. Therefore, we use the following ordinal scale L2-0, L2-M, L1.3, L1.4, L2-1, L2-2, L2-3 so that the subsumption of the levels as described above is still valid and tools being compliant to earlier versions of UML and only to L2-0 or L2-M are assigned to an appropriate UML 1.x level.

3.2 Evaluation Criteria and Procedure

We used an empirical approach to evaluate the tools and to determine the compliance levels, because for only few tools the implemented meta-model can directly be inspected, e.g. as source code. Figure 1 depicts the derivation of the evaluation criteria from the UML as a top-down process and the evaluation as a bottom-up process. Starting with the UML compliance levels as described in Section 3.1, we systematically dissected the UML specification of the referenced diagram types to obtain the individual modeling elements, relations and their properties. These were mapped to features. We also considered the related standard stereotypes summarized in the annex of the UML superstructure as features. Then we validated the feature list against the example

diagrams and the particular diagram elements summaries given in the UML specification. Afterwards, we reviewed and consolidated the initial feature list.

We assigned 476 features obtained in this step to 131 feature groups to simplify the evaluation and the final data analysis. Thus, for example all features directly related to the use case modeling element were grouped together. These feature groups do not influence the results and are used in our approach only to simplify the evaluation and the presentation of the results. Finally, we summarized the feature groups as a feature hierarchy using the modeling facilities defined by the UML, e.g. the UML diagram types, as top-level categories. Figure 2 depicts a view on the feature hierarchy, in which all top-level categories, the feature groups for use cases and some individual features are shown. In particular the category "Use Cases", the feature groups for use case diagrams and the individual features for the representation of an individual use case are displayed in Figure 2. The category "model persistence" is needed for determining the compliance level and, thus, contains the supported XMI and DI versions as well as the results of the structural validation of files produced by a tool.

The features are aggregated in a bottom-up fashion in order to gain an overview, e.g. on the degree of compliance for the entire UML or for individual diagrams. This aggregation, as depicted in Figure 1, is done according to the following schema: A feature, i.e. a certain functionality realized by a tool, is counted by 1.

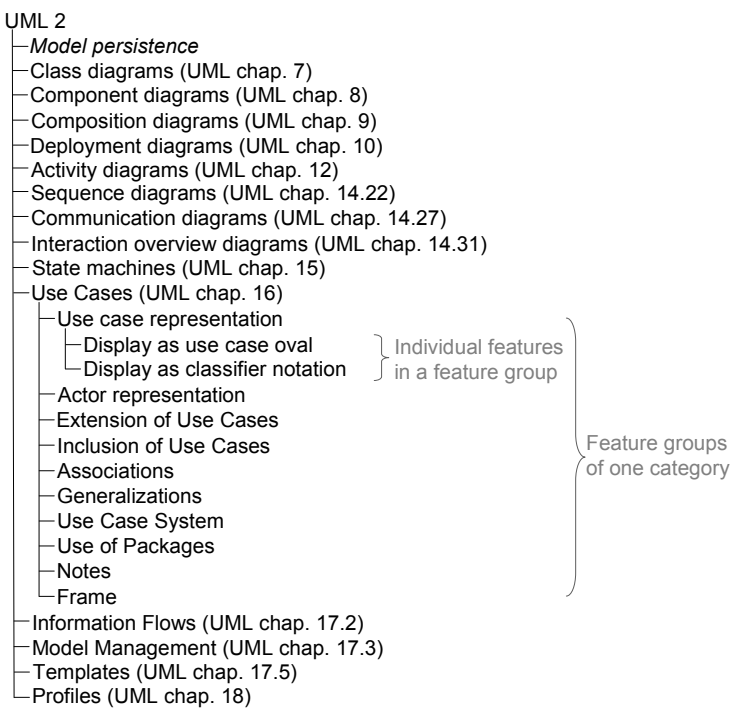


Figure 2: View on the feature hierarchy created while analyzing the UML specification.

After constructing the feature hierarchy, the next action was the actual evaluation.,The following steps were performed to evaluate a tool:

1. **Acquisition:** The evaluator obtained the concrete tool from the vendor e.g. by download. As a part of this activity, an evaluation license was requested if needed. The evaluator recorded also the reason if a tool was not available, e.g. the tool home page does not exist anymore.

2. **Installation:** The evaluator installed the tool into a virtual machine according to the individual installation instructions.

3. **UML evaluation:** The evaluator created an UML model and tested each supported diagram type. In particular, we specified reference examples for class diagrams and use case diagrams to obtain comparable results. The examples aimed at combining most of the features of the diagram in a meaningful manner. For the other diagram types the evaluator just collects whether the features are present and functional by modelling a diagram.

4. **Data collection:** The evaluator collected the data in a spreadsheet prepared according to the feature hierarchy. Thus the evaluator collects data on the realization of the concrete syntax of UML following our evaluation approach as described in Section 3.2. After collecting all requested information, he saved the set of diagrams in XMI or DI format if supported by the tool. In the case that XMI or DI was produced, he validated the resulting file structurally against the OMG specifications and recorded the results using Altova XMLSpy with the appropriate DTD or XSD file provided by the OMG. Furthermore, he produced a screenshot of the class diagram and the use case diagram. Finally, the evaluator stored all collected information in an evaluation repository.

For a given compliance level only a subset of the features in the feature hierarchy is relevant. Thus, we could derive for each compliance level in Section 3.1 a compliance profile, i.e. a projection of the relevant features in the feature hierarchy to automate the compliance level calculation. By using the compliance profiles as a feature selection mechanism, the results of an individual tool could be calculated with respect to the profile. Thus, we could obtain a feature fulfillment degree per profile and per tool. In fact, the UML requires the (complete) realization of certain diagram types per compliance level. From this strict viewpoint no tool in our evaluation would receive any compliance level at all! Thus, in this study we attested for each candidate level with 50%-75% fulfillment a partial compliance, for more than 75% fulfillment an (acceptable) full compliance.

3.3 Planning, Preparing and Conducting the Evaluation

We conducted an exhaustive evaluation including all tools being relevant from the industry as well as the research viewpoint. In order to achieve this, we first collected sales information (name, vendor, URL) of all tools published in 8 different tool listings or comparisons. This list was completed by intensive thorough internet search for UML

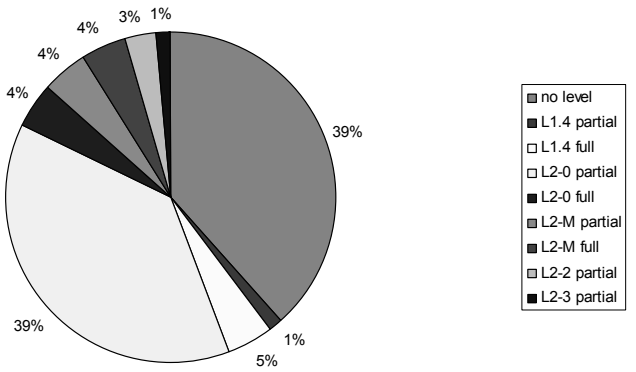


Figure 3: Summary of the maximum compliance levels for all evaluated tools.

tools, so that 200 tools were scheduled for evaluation at the beginning of our work. In parallel, we extracted the feature hierarchy from the UML specification and specified the evaluation procedure as described in Section 3.2.

After the preparation phase, we conducted the evaluation according to the evaluation procedure. Finally, we analyzed the data, e.g. we determined the compliance levels of all evaluated tools upon the collected data and prepared the feature compliance profiles. This provided the basis for the analysis we discuss in the sections below.

4 Results, Findings and Analysis

In this section, we present the results and findings of our evaluation. We aimed at analyzing all commercial and open-source tools being relevant to industry and research. We started with a set of 200 tools, i.e. all tools we could identify. While conducting the evaluation, 66% of the scheduled tools were not available, not maintained after 2003 when UML 2 was introduced, did not provide UML facilities at all or were not available for evaluation. Thus, we evaluated a remaining set of 68 tools, i.e. the currently available tools including all major tools used in industry and research today.

Figure 3 depicts the summary on the UML compliance levels according to the ordering given in Section 3.1. The pie slices in Figure 3 are assigned clockwise starting at the right slice with 39%, i.e. 39% of the tools are assigned to no level, 1% to L1.4 partial, etc. The results are displayed according to partial compliance and acceptable (full) compliance. We attest partial compliance to Level 2.0 for 39% of the tools and do not assign a UML compliance level to 39%. Not all compliance levels are assigned, e.g. we could neither assign full L2-2 nor full L2-3.

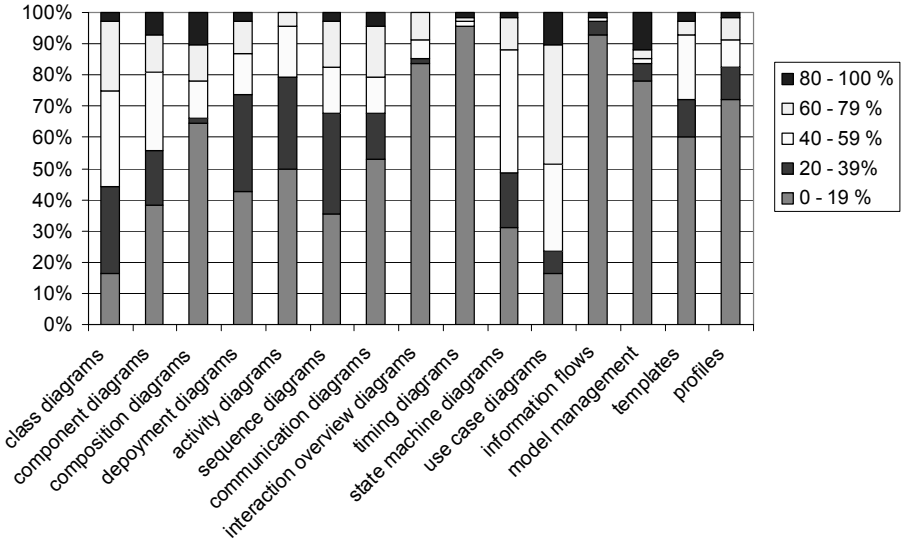


Figure 4: Summary of UML diagram realization measured by feature fulfilment structured according to the categories of the feature hierarchy (excluding model persistence). The legend denotes the degrees of feature realization, the entire chart relates to the evaluated tools.

One can conclude from this summary that currently a lot of tools do not implement all UML diagrams and most tools offer only basic support. Only few tools provide the diagrams introduced with UML 2 as indicated by the fulfilment of L2-2 or L2-3. As a summary of the compliance level rating, 42.6% of the tools are assigned to partial or

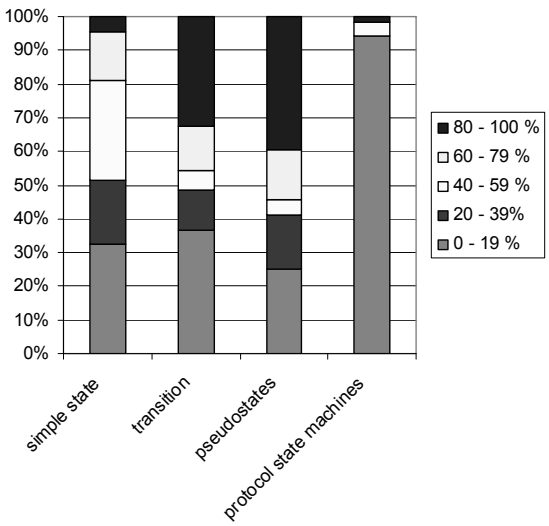


Figure 5: Realization of UML state machine diagrams in terms of feature fulfilment displayed according to selected diagram feature groups. The legend denotes the degrees of feature realization, the entire chart relates to the evaluated tools.

Vendor	Tool name	Version number	Assigned UML Compliance Level	Feature Fulfillment Level
Altova	UModel	Professional 2008	L2.2AC	64.33 %
IDS Scheer	ARIS UML designer	7.0.2.207949	L2.0C	42.78 %
No Magic	Magic Draw	14.0	L1.4 / L2.MAC	77.07 %
Telelogic	Rhapsody	7.11.0	partial L2.0AC	39.92 %
SparxSystems	Enterprise Architect	7.0.817	partial L2.3AC	71.44 %
Visual Paradigm	Visual Paradigm	6.1 sp_1	partial L2.2AC	71.87 %

Table 1: Excerpt from the tool summary table in [EES08].

complete level L2-0, but 82.8% of these tools are considered for level L2-0C, because the tools produced no appropriate XMI. Additional information including detailed characterizations for each tool can be found in [EES08].

Figure 4 displays the realization of features structured by categories, i.e. per UML diagram types and auxiliary UML features. Several of the new UML 2 facilities like composition diagrams, interaction overview diagrams, timing diagrams and information flows are supported only by few tools. We identified also that the ability to work with profiles is not supported by many tools. Despite the fact that a template mechanism was also defined in earlier versions of the UML, templates are also not supported at all by 60% of the tools. The overall support of class diagrams, use case diagrams and state machines appears to be good.

As an example for the data collection on individual diagrams, Figure 5 shows the realization of some selected diagram elements for state machines. Obviously, only few tools consider protocol state machines, but the frequently used elements including the various types of pseudo states are well supported.

Table 1 shows an excerpt from the tool summary table in [EES08]. Only the tools mentioned in this paper are listed in the table. Magic Draw, the tool with the highest feature fulfilment level of 77.07% was assigned to a basic compliance level due to two missing UML 2 diagrams (interaction overview and timing diagrams).

5 Conclusions

UML models are widely used in software engineering. If a UML modelling tool does not properly adopt the specification, this can seriously impede the usefulness of the tool and the user may get locked-in to the tool due to the lack of model exchange capabilities. In particular, this is a drawback for model-driven software engineering approaches. Thus, detailed compliance evaluations can provide an in-depth description of the tools, may

support tool selection and migration and give an overview on the current level of implementation.

In this paper, we presented a feature-based evaluation approach to determine the specification compliance of UML tools. The approach relies on an encompassing feature hierarchy extracted form the UML specification and on compliance profiles for given UML versions or UML compliance levels. We evaluated 68 out of an initial set of 200 tools, covering all major UML tools used in industrial practice today. The remaining 132 tools were not available for evaluation, did not exist anymore, or were not maintained for a longer time, etc.

While four tools reach acceptable compliance levels, more than 39% of all evaluated tools do not implement the UML specification sufficiently. 6% of the tools are classified on UML 1.3 or 1.4 level, 39% on basic UML 2 level. Our results on XMI model exchange fit to related evaluations: 47% of the tools do not pass the structural XMI validity test, 3%, i.e. two tools (ARIS and Rhapsody), pass the XMI validity test while the remaining 50% offer no XMI at all.

There is an obvious discrepancy between the compliance level and the feature fulfilment level in Table 1. Compliance levels focus on larger units of the UML language while the feature fulfilment level can provide a more realistic insight for decision makers. In fact, the data aggregation schema described in this paper is designed for the derivation of compliance as defined in the UML specification. To establish a tailorable decision process considering user or company specific weightings, we will consider alternative feature aggregation schemas in future work, e.g. the logic scoring preference method [FDSP05]. As our focus in this paper was on UML compliance, additional data gathered in our evaluation like code generation was not discussed here.

According to our experience it is a complex task to obtain complete and consistent functional requirements for implementing the UML, because much information is given implicitly, as part of the metamodel specification, in descriptive text, as wellformedness-rules, in example diagrams and in partly inconsistent summaries, e.g. tables. Thus, an improved structure of the UML specification including clear requirements also for the semantics would help implementers validating their individual approaches.

Evaluating complex modeling tools on feature level is an immense effort (at least six person months, fulltime equivalent). As the implementation of most tools is continuously improved, it is impossible to always report on the latest versions of all tools. For some tools in Table 1 the version number indicates that in the meantime newer versions have been released. Thus, we will perform a continuous reevaluation and an analysis of the differences in the future.

Acknowledgements

We thank all the vendors for providing their tools according to their terms of evaluation licenses. For future evaluations we hope that the companies will support us again with download versions and evaluation licenses.

References

[BFO+05] C. Bröcker, F. Ferrandina, B. Oestereich, G. Versteegen, and T. Weilkiens. *iX-Studie "Bessere Software!" Konfigurations- und Änderungsmanagement sowie UML*. Heise Verlag, 2005. (in German).

[BFSC08] P. Bunyakiati, A. Finkelstein, J. Skene, and C. Chapman. Using JULE to generate a compliance test suite for the UML standard. In *ICSE '08: Proceedings of the 30th international conference on Software engineering*, pages 827–830, New York, NY, USA, 2008. ACM.

[DP07] B. Dobing and J. Parsons. How UML is Used. *Open Source Security*, 5(1), February 2007.

[EES08] H. Eichelberger, Y. Eldogan, and K. Schmid. A Comprehensive Survey of UML Tool Capabilites and Compliance. Technical report, Institut für Informatik, University of Hildesheim, 2008. (internal report).

[FDSP05] A. Funes, A. Dasso, C. Salgado, and M. Peralta. UML Tool Evaluation Requirements. In *Proceedings of ASIS 2005*, 2005.

[Jec04] M. Jeckle. UML Tools (CASE & Drawing), 2004. Online available at: http://www.jeckle.de/umltools.htm.

[KF04] L. Kirchner and U. Frank. Evaluierung von UML-Modellierungswerkzeugen. *OBJEKTspektrum*, (1), 2004.

[LLPM06] B. Lundell, B. Lings, A. Persson, and A. Mattsson. UML Model Interchange in Heterogeneous Tool Environments: An Analysis of Adoptions of XMI 2. In O. Nierstrasz, J. Whittle, D. Harel, and G. Reggio, editors, *Model Driven Engineering Languages and Systems, 9th International Conference, MoDELS 2006, Genova, Italy, October 1-6, 2006, Proceedings*, number 4199 in Lecture Notes in Computer Science, pages 619—630. Springer-Verlag Inc., 2006.

[OMG06] OMG. Diagram interchange, v1.0, April 2006. Online available at: http://www.omg.org/docs/formal/06-04-04.pdf.

[OMG07a] OMG. MOF 2.0/XMI Mapping, v2.1.1, December 2007. Online available at: http://www.omg.org/spec/XMI/2.1.1/.

[OMG07b] OMG. Unified Modeling Language: Infrastructure version 2.1.1. Specification v2.11 2007-02-06, Object Management Group, February 2007. Available online at: http://www.omg.org/docs/formal/07-02-06.pdf.

[OMG07c] OMG. Unified Modeling Language: Superstructure version 2.1.1. Specification v2.11 2007-02-05, Object Management Group, February 2007. Available online at: http://www.omg.org/docs/formal/07-02-05.pdf.

[OMG08] OMG. OMG UML Vendor Directory Listing, 2008. Online available at: http://uml-directory.omg.org/vendor/list.htm.

[OOS08] OOSE. UML Tools, 2008. Online available at: http://www.oose.de/umltools.htm.

[www08] Wikipedia, list of UML tools, 2008. Online available at: http://en.wikipedia.org/wiki/List_of_UML_tools.

Embedding Behavioral Models into Object-Oriented Source Code

Moritz Balz, Michael Striewe, Michael Goedicke
Specification of Software Systems
Institute of Computer Science and Business Information Systems
University of Duisburg-Essen, Campus Essen
Essen, Germany
{michael.goedicke, moritz.balz, michael.striewe}@s3.uni-due.de

Abstract: In this paper we address the inevitable problem of aligning a software model with its respective code in such a way that the synchronization between both views of the system can be effectively maintained. In traditional approaches to model-driven software development (MDSD) it is at least necessary to annotate the source code in order to mark its origin for round-trip engineering and to prevent it from being overridden when regenerating code from other models. Instead of just maintaining references to models, we suggest to embed full model semantics into source code. To realize this approach we modify the earlier concept of ViewPoints, so that the necessary configuration of ViewPoints happens inside the source code by using selected constructs of object-oriented programming languages. Our contribution presents the approach to maintain models and source code simultaneously by means of behavioral models.

1 Introduction

Many existing approaches to model-driven software development (MDSD) suffer the problem to relate evolution of source code back to the model [BIJ06]. Several problems prevent automated solutions and thus hinder parallel development of models and source code, especially in large scale industrial software development [HT06, BLW05]. In large applications we can expect different parts of the system to be designed according to different models, such as a process model that works in parallel with a data model. Additionally, parts of source code may be written manually to satisfy the needs of particular frameworks in use or non-functional requirements like performance. To avoid confusion, it is at least necessary to annotate the source code in order to mark its origin for round-trip engineering and to prevent it from being overridden when regenerating code from other models. The result is a huge effort regarding the synchronization. Models have to be kept in sync with each other, each model has to be kept in sync with the code and parts of the source code that are represented by more than one model have to be handled very carefully [VG05].

In cases where modeling technologies are used for system design and not for run time configuration, we want to avoid the overhead to synchronize code and models. Instead of just annotating code with references to models, we suggest to embed full model semantics into source code. If the model is seen rather as an abstract idea than as a visual repre-

sentation, source code structures can be just a different notation for models. The visual representation can be extracted from the code on demand, but is only a partial view on the whole system. This seems to be a promising approach since it is easier to extract a specialized view from the complete system instead of constructing and maintaining the complete system out of specialized views.

To realize this we use the concept of ViewPoints [FKG90, FKN+92] in a slightly modified version, so that the necessary configuration of ViewPoints happens inside the source code by using selected constructs of object-oriented programming languages. Several benefits can be expected from this approach:

First, the existence of an up to date version of the source code is a necessary condition for any useful software system. Once the system is deployed the consistency with any abstract model does not matter until some maintenance or evolution is due. Thus, if the model is maintained in and extracted from the source code for any development step source code and abstract model are in sync automatically. Moreover, different external model representations may be changed concurrently and have to be synchronized manually, but they are automatically in sync when changes are applied to the source code directly.

Second, an application can be aware of the models, since their complete semantic is available at run time via reflection mechanisms. This enables monitoring and model manipulation at run time if desired and appropriate.

Third, our approach integrates smoothly into any phase of the software development process, right after the first lines of source code have been written. In early design phases, single software components can be generated separately from their models. In later phases, these components are integrated manually into a larger system and models can be extracted to validate their respective correct integration. During testing and debugging, models can be extracted to validate whether bug fixes do not affect the intended behaviour.

This contribution is organised as follows: Section 3 introduces the combination of MDSD and ViewPoints at a general level in 3.1 and substantiates the idea for behavioral models in 3.2. Section 4 presents the realization using an actual set of technologies, followed by a case study for state machine models by means of a sample system. This case study is the basis for our assessment in 5. The paper concludes with a discussion of interesting open questions in future research and development in 6.

2 Related Approaches

Various techniques exist to embed semantic information and modeling constraints into object-oriented source code [BHS07]. Examples are the JAVA MODELING LANGUAGE (JML) [LBR99] offering an extensive syntax for specification annotations or the approach to use Smalltalk with its introspection capabilities as a meta language [DG06]. In contrast to such universal approaches we do not aim to present a notation for the specification of all possible system models. Contrary to tools like JAVA PATHFINDER [VHB+03] or abstract universal approaches like Introspective Model-Driven Development [BM06], our approach does not consider a whole application as model, but only designated parts of it.

This limited domain of behavioral models can thus be examined more thoroughly and with respect to a formally well founded background.

The approach to use declarative code structures to maintain relations to models has already been explored, for example in the context of UML models [WS05]. This relies on the existence of external models that have to be reconciled with the source code or vice versa. The same applies to Framework Specific Modeling Languages [AC06]. In contrast, our approach facilitates the principle of having only one representation for model and source code and thus avoids round trip engineering.

Executable models, for example "executable UML" [MB02], automate the transformation of models into executable systems or directly execute model specifications. While this approach does allow a clean and model-centric view of systems, it assumes that entire applications can be expressed as models. Especially in heterogeneous environments this assumption will fail often since different frameworks are in use or a high priority of non-functional requirements, e.g. performance demand the application of certain patterns which causes a deviation from the structure an abstract model would imply. In such cases an automated integration into existing systems is not easy to achieve. In our approach the model structures in the source code can co-exist with non-model parts of an application, thus enabling a seamless integration into larger systems. In addition, our approach executes only compiled byte code and interprets no external representations, which avoids possible performance problems.

3 ViewPoints, MDSD and Embedded Model Semantics

Here we discuss briefly how MDSD and the concept of ViewPoints can be brought together and how this is related to embedded model semantics. At first we discuss this at a general level. Then we substantiate the idea by talking about behavioral models and imperative programs using Java as sample programming language.

3.1 The general Problem

Each step in MDSD is characterized by a certain view on the system under construction or evolution. A computational independent model (CIM) neglects any algorithmic aspect and puts the focus on domain knowledge and business processes, for example. In terms of ViewPoints, this would be part of the view of an business expert. The same can be said for platform independent models (PIM), that may be part of the view of a system architect, and platform specific models (PSM), that may be part of the view of a database expert. While MDSD implies a sequential order of these models, ViewPoints allow to use these models in parallel. Moreover, ViewPoints may be defined in ways that do not clearly correspond to the levels of CIM, PIM or PSM.

The obvious goal of the software development process is to create a system that covers the semantics of all models. In terms of ViewPoints this is realized by finding a consistent

configuration of all views. In terms of MDSD it is realized by incremental annotation and generation of models according to their sequential order, so that the semantics of a CIM is embedded in a PIM and the semantics of a PIM is embedded in a PSM and finally the generated code realizes all the semantics expressed in a PSM.

Comparing both ideas, we can observe two problems. On the one hand, a strong benefit of ViewPoints is the concept of configurations and establishing relationships between the involved ViewPoints. In general, they are easy to manage and to evolve and they ensure that any necessary update is made, but nothing more. If one actor in the development process makes a change in his/her view, the system must be updated by changing the configuration, but never the other views. This is not valid for MDSD. If a CIM changes, all subsequent models have to be updated or regenerated, but if a PSM changes, there is no check whether it still conforms to the original CIM. On the other hand, this order of models and derivation of subsequent models by annotating previous ones is a strong benefit of MDSD. In the concept of ViewPoints, parallel changes in all models are allowed and relationships have to be established manually but systematically. This can lead to ambiguous cases in which more than one relation is possible.

The solution suggested in this paper is to use the source code as the superior and unambiguous representation of semantics. In terms of ViewPoints, source code configures views by identifying and annotating their objects and sharing them between views if necessary. In terms of MDSD, the semantics of each model are directly embedded into source code instead of being deferred to another model. Hence any change in one view or model affects the source code directly and thus updates all other models and views if and only if it is necessary.

The key of ViewPoints to the case proposed here is that establishing explicit relations between the code view and the model(s) view defines the necessary semantic elements in representing development artefacts at different levels of detail and abstraction. Such a consistent multi-view representation scheme enables the application of methods and tools at the corresponding abstraction level and allows the effective translation of related results to the corresponding other levels.

We now discuss how such different levels of abstraction and detail can look for imperative programs.

3.2 Behavioral Models and Programs

In order to embed behavioral models in object-oriented structures we will extract the relevant features from behavioral model specifications that we want to use explicitly. For our purpose we define behavioral models to consist of (at least some of) the following parts:

- States are specific moments in the behavior of a system.
- Activities represent arbitrary functionality being executed by this system. When activities are executed, control flow is handed over to application logic and returned back to the model when the activity has finished.

- Transitions are possible paths between states.
- Guards represent conditions that indicate if a transition can fire.
- Updates specify how activities are expected to change the system state.
- Compositions may encapsulate other elements to simplify the depiction of complex systems and thus build hierarchies of such behavioral models.

This abstraction is applicable to various behavioral modeling techniques. For finite state machines [PGS01] the elements are composed using the following rules: (1) States do not modify a system, but represent a certain condition of that system; (2) guards decide which transition emanating from the current state will fire; (3) activities are executed during transitions; (4) updates describe the changes in the system state after transition activities have been executed. The application of the aforementioned abstraction to process models [Coa95] is slightly different: (1) Activities are executed during states; (2) after execution, guards decide which transition to a next state will fire; (3) updates are usually not part of process models, but may be included in states to validate activity execution by describing the expected changes in system state. Similar rules may be found for e.g. UML activity diagrams [OMG04].

Now we need to define representations in the source code that allow us to specify the model structures in systems at run time. The related programming language constructs must be explicitly defined in the source code and the corresponding information must be accessible at run time, i.e. after compilation, e.g. by means of structural reflection [DM95]. In the context of modern object-oriented languages, for example Java [GJSB05], basic constructs like classes and interfaces are static definitions of application structures and data. The code of actual business logic is contained in methods. Meta data constructs (such as Java Annotations [Sun04]) annotate other constructs with additional information. When model structures are embedded in the source code, reasonable compositions of these constructs must be defined. For this purpose relations between the constructs are used – be they implicit, e.g., that methods are contained in a certain class, or explicit, for example a reference to another class in meta data. The implementation in Java will be explained in section 4 and demonstrated with a larger example. The basic structure is defined by the following rules:

- Each state is represented by a class definition.
- Each state class may contain methods that represent related activities. The actual activity is constituted by the source code in the method body.
- Transitions are defined by meta data referring to the target state and a guard. Depending on the model type, these meta data can be attached to activity methods (e.g. in the case of state machines) or state classes (for process models).
- Guards and updates are implemented as methods without parameters that return a boolean value. The method content consists of logical expressions, either single statements for a certain value or pairs of statements that define value ranges. The

contained code must completely follow these rules so that logical expressions can be extracted for modeling.

- Updates may be referred to from transitions or state classes, depending on if and where a system validation is reasonable after activities.
- To decouple source code elements related to models and arbitrary source code for reasons of clarity, a façade type will be defined that encapsulates the business logic. An instance of this façade, denoted ACTOR in the following, is passed to activity methods by the execution framework.
- The variables defining the considerable system state must be available explicitly to the model, even if they are defined and modified in arbitrary source code. For this purpose another façade type, denoted VARIABLES in the following, will be used to make variable values available to the model execution. This way variables are easy to identify by the method names and thus usable for model extraction.

These simple definitions allow to execute activity models with a simple framework using static reflection: The class structures of an application are examined and state classes are identified. Based on the initial state, possible transition methods are detected and their guards invoked. If a transition can fire, the activity code is executed and connects the state machine to the actual application logic. Afterwards the update method may be called to validate the system state. The next state is then selected from the method meta data.

4 An Instance: Automata and Java Components

The concept sketched above has been used with finite state machines as a specific instance of behavioral models [BSG08]. We will now explain how the concept applies to the selected technologies and describe a non-trivial real-world application (see section 4.2) that has been modeled by means of our approach.

4.1 Concept

The rules for source code structures mentioned above can be applied to Java constructs without change:

- Each state is represented by a class definition implementing a given interface named `IState`, which defines no methods, but allows to type-safely identify these state types. Since Java allows to implement multiple interfaces in one class, a class could be marked to represent a part of more than one model if necessary.
- Each state type contains methods that represent activities. The actual activity is constituted by the source code in the method body. All activity methods must take

no parameters except one instance of a type that encapsulates the business logic of the actual application.

- The activity methods are defined as transitions by meta data (in this case Java annotations) that refers to a target state and a class containing guards and updates.
- Guards and updates are implemented as two methods without parameters that return a boolean value. The method content consists of logical expressions, either single statements for a certain value or pairs of statements that define value ranges. Both methods are executed at run time to decide if a transition can fire and if the activity left the system in an expected state.

4.2 Sample System

To validate that the approach is suitable for the mentioned purposes, we use a load generator component as a sample system. It is part of a larger system for performance testing and is able to generate load according to different measurement modes. The actions of the load generator for each mode are modelled using state machine models. Here, we discuss an exploration mode, in which load is generated by worker threads whose number is adjusted depending on the turnaround time of measurement requests and the acceptable turnaround time range being given by the user.

The state machine to control this application consists of 10 states and 27 transitions. The start state can be left by only one transition to do some preparations and reach a standby state. From here, there is again only one possible transition to perform load generation and reach a state called "AfterMeasurement". Depending on the result of the measurement, the number of load generating workers is adjusted, leading to different states, and another measurement is taken. This process continues until a performance mark is determined and the state machine terminates.

4.3 Code Structures

Some code structures needed in this scenario to represent states, transitions and activities can be seen in figure 1, which is cut out of the source code of the actual system. All model structures are represented by roughly 350 lines of code, being less than 3% of the whole application source code. The class `AfterMeasurementState` shown here implements the interface `IState` and is thus marked to be a state of a state machine. Three transitions are visible in this example, all represented by methods with a `@Transition` annotation. The first parameter of these annotations refers to a class that has implemented the `IState` interface as well. The second parameter refers to classes that implement guards and updates. Activities are simple statements in the method body. In our case, all possible activities are encapsulated by method calls to the `actor` instance.

Figure 2 gives a closer look at the way guards and updates are represented in our frame-

```
public class AfterMeasurementState implements IState {
  @Transition(target = AfterMeasurementState.class,
          contract = RestartContract.class)
  public void restartMeasurement(MeasurementModule actor) {
    actor.increaseNumberOfRestarts();
    actor.doMeasure("Restarted measurement");
  }

  @Transition(target = UpUpState.class,
          contract = BeginUpUpContract.class)
  public void beginUpUp(MeasurementModule actor) {
    actor.resetRestarts();
    actor.beginUpUp();
    actor.doMeasure("Exploration by distance upwards");
  }

  // . . .

  @Transition(target = TerminationState.class,
          contract = AbortContract.class)
  public void abortMeasurement(MeasurementModule actor) {
    actor.terminateMeasurement();
  }
}
```

Figure 1: Class `AfterMeasurementState` with some outgoing transitions

```
public class BeginUpUpContract implements
        IContract<IMeasurementVariables> {
  public boolean checkCondition(IMeasurementVariables vars) {
    return (!vars.getAbort() && !vars.getRestart() &&
            vars.getTooLow());
  }

  public boolean validate(IMeasurementVariables before,
          IMeasurementVariables after) {
    return (after.getNumberOfWorkers() ==
          (before.getNumberOfWorkers() + before.getWorkerDistance()));
  }
}
```

Figure 2: Guards and updates in `BeginUpUpContract`

```
public boolean validate(IMeasurementVariables before,
        IMeasurementVariables after) {
  return (after.getCurrentTurnaroundTime() >= 0 &&
          after.getCurrentTurnaroundTime() <= 10000);
}
```

Figure 3: A validation method that checks for a non-deterministic update.

work. The execution component expects both to be represented in one class that implements an interface named `IContract`. This interface enforces to implement the two methods `checkCondition` for guards and `validate` for updates. Obviously guards can be represented directly by boolean operations on variable values that are retrieved from the VARIABLES façade `vars` via simple method calls. A more interesting case is the representation of updates. Since the actual update is performed by the activities inside the transitions, update definitions from the model can be transformed into validation conditions to monitor the system during run time. Thus, if the model defines that a transition increases the number of load workers by a given delta, the method can check this as shown in the source code example.

Modelling languages for state machines sometimes allow not only updates with fixed values, but also non-deterministic choices from a range of values. This can be expressed in the `validate` method, too, as shown in figure 3. Of course, both in guards and updates the simplest method content would be to return `true`, when a transition is always enabled or a update does not need any validation.

4.4 Generated Models

In our case study, the original implementation was coded manually, but of course the initial code skeletons can be generated from models. Much more important for our approach is the possibility to extract the models from the source code. In our case study, we used graph transformation techniques [Str08] to analyse the syntax and define a mapping to the data format of the UPPAAL system for timed automata [LPY97]. The resulting state machine model is shown in figure 4.

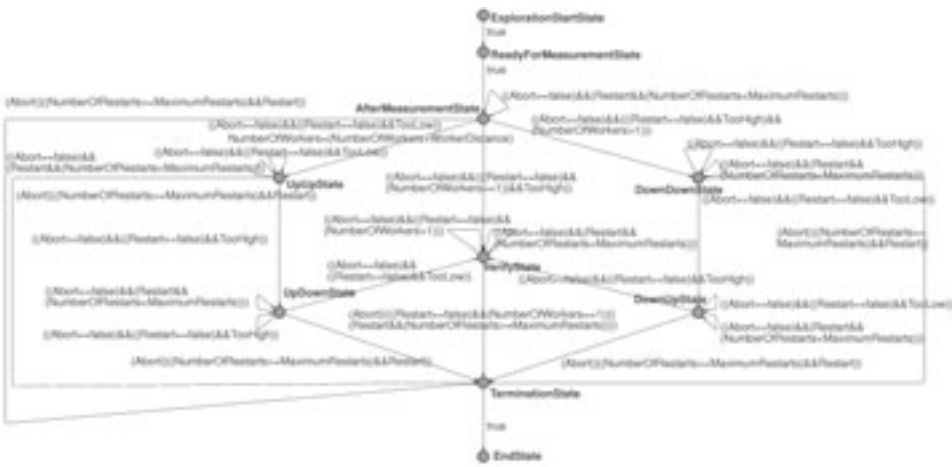


Figure 4: The UPPAAL model

This shows that a model can be easily recovered by a tool from source code which has been

prepared according to the proposed rules. Since the model is independent from the choice of Java, the entire approach is not limited to a single programming language. Moreover, it is thereby possible to relate a PIM directly to source code without touching the PSM. The extracted model could be used to generate state machine structure implementations in other languages as well. The content of these structures, i.e. access to Variables and Actor components, would have to be manually added since they are intentionally considered a black box in our approach.

5 Conclusion

We have seen that object-oriented constructs and basic metaprogramming facilities are sufficient to embed behavioral models into source code and extract all model semantics for development purposes. The real-world application serving as our example shows that a complete state machine representation can be extracted from given JAVA source code and validated in the UPPAAL modeling tool. The way to execute, monitor and debug the model at run time has been outlined. So we have shown that our objective to develop applications in a larger context simultaneously to model specification, validation and simulation for parts of the application is fulfilled. Since no double effort to maintain different representations is required we can state that our approach can effectively be used to avoid maintaining and merging different abstraction layers.

It is important to note that we don't propose our approach as a replacement of existing MDSD technologies for every development task. Systems that need descriptive model specifications being interpreted and potentially modified at run time are clearly not the focus of this code-oriented approach. Instead, integrated systems can be modeled as such and then be used including model semantics. Our approach is applicable in cases when modeling technologies are used for development purposes like system design, validation, documentation and monitoring. Under these circumstances the seamless integration of model semantics into source code structures offers clear advantages over existing approaches.

It also brings the programmer who likes to stick to code to the designer thinking in abstract models together. This is due to the fact that the entire structures – code and models – are kept closely together.

6 Future Work

The approach of embedding models semantics into object-oriented structures being available at development and run time enables a usage of the model information in running systems: We plan to enhance the execution framework so that the system state with variable values, selected transitions and current states can be monitored in real-time. Another similar extension could record all system states and decisions to make events comprehensible ex post. At run time the meta view to the system could be of use to pause and continue applications for debugging or maintenance purposes, as far as the application it-

self utilizes the framework to store its state. Since our approach is not limited to Java as a programming language, we plan to realize the framework in other languages that offer more capabilities for changing code at run time. The framework can then be used as a skeleton for self-adaptive systems that can reflect and change their own model.

At the conceptual level we will explore our approach for applicability with more model types. This concerns behavioral models with different semantics, for example UML interaction diagrams, as well as other model types like data models or component models. We plan not only to validate that our approach supports this models in theory, but also that helpful monitoring tools can exist at run time. For the existing state machine model support we will supplement the creation of hierarchies of embedded models to allow building even larger systems.

Most important to gain acceptance for this approach and to help efficient development processes is the creation of development tools. We plan to implement editors for existing Java development environments that allow graphical creation, manipulation and validation of behavioral models and persist these model structures in the source code constructs mentioned above. This would allow a parallel editing of model semantics and arbitrary source code and not only allow a faster development process, but also a simpler comprehension and validation of existing systems.

References

[AC06] Michal Antkiewicz and Krzysztof Czarnecki. Framework-Specific Modeling Languages with Round-Trip Engineering. In Nierstrasz et al. [NWHR06], pages 692–706.

[BHS07] Bernhard Beckert, Reiner Hähnle, and Peter H. Schmitt. *Verification of Object-Oriented Software. The KeY Approach*. Springer-Verlag New York, Inc., 2007.

[BIJ06] A. W. Brown, S. Iyengar, and S. Johnston. A Rational approach to model-driven development. *IBM Systems Journal*, 45(3):463–480, 2006.

[BLW05] Paul Baker, Shiou Loh, and Frank Weil. Model-Driven Engineering in a Large Industrial Context – Motorola Case Study. In Briand and Williams [BW05], pages 476–491.

[BM06] Thomas Büchner and Florian Matthes. Introspective Model-Driven Development. In *Software Architecture, Third European Workshop, EWSA 2006, Nantes, France, September 4-5, 2006*, volume 4344 of *Lecture Notes in Computer Science*, pages 33–49. Springer, 2006.

[BSG08] Moritz Balz, Michael Striewe, and Michael Goedicke. Embedding State Machine Models in Object-Oriented Source Code. In *Proceedings of the 3rd Workshop on Models@run.time at MODELS 2008*, pages 6–15, 2008.

[BW05] Lionel C. Briand and Clay Williams, editors. *Model Driven Engineering Languages and Systems, 8th International Conference, MoDELS 2005, Montego Bay, Jamaica, October 2-7, 2005, Proceedings*, volume 3713 of *Lecture Notes in Computer Science*. Springer, 2005.

[Coa95] Workflow Management Coalition. The Workflow Reference Model. Technical Report WFMC-TC-1003 Version 1.1, Workflow Management Coalition, 1995.

[DG06] Stéphane Ducasse and Tudor Gîrba. Using Smalltalk as a Reflective Executable Metalanguage. In Nierstrasz et al. [NWHR06], pages 604–618.

[DM95] François-Nicola Demers and Jacques Malenfant. Reflection in logic, functional and object-oriented programming: a short comparative study. In *In IJCAI '95 Workshop on Reflection and Metalevel Architectures and their Applications in AI*, pages 29–38, 1995.

[FKG90] A. Finkelstein, J. Kramer, and M. Goedicke. ViewPoint Oriented Software Development. In *International Workshop on Software Engineering and its Applications*, 1990.

[FKN+92] A. Finkelstein, J. Kramer, B. Nuseibeh, L. Finkelstein, and M. Goedicke. Viewpoints: A Framework for Integrating Multiple Perspectives in System Development. *International Journal of Software Engineering and Knowledge Engineering*, 2, 1992.

[GJSB05] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *Java(TM) Language Specification, The 3rd Edition*. Addison-Wesley Professional, 2005.

[HT06] B. Hailpern and P. Tarr. Model-driven development: The good, the bad, and the ugly. *IBM Systems Journal*, 45(3):451–461, 2006.

[LBR99] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. JML: A Notation for Detailed Design. In Haim Kilov, Bernhard Rumpe, and Ian Simmonds, editors, *Behavioral Specifications of Businesses and Systems*, pages 175–188. Kluwer, 1999.

[LPY97] Kim G. Larsen, Paul Pettersson, and Wang Yi. UPPAAL in a Nutshell. *Int. Journal on Software Tools for Technology Transfer*, 1(1–2):134–152, Oct 1997.

[MB02] Stephan J. Mellor and Marc J. Balcer. *Executable UML*. Addison-Wesley, 2002.

[NWHR06] Oscar Nierstrasz, Jon Whittle, David Harel, and Gianna Reggio, editors. *Model Driven Engineering Languages and Systems, 9th International Conference, MoDELS 2006, Genova, Italy, October 1-6, 2006, Proceedings*, volume 4199 of *Lecture Notes in Computer Science*. Springer, 2006.

[OMG04] OMG. UML 2.0 superstructure specification. Technical report, Object Management Group, 2004.

[PGS01] Doron A. Peled, David Gries, and Fred B. Schneider, editors. *Software reliability methods*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2001.

[Str08] Michael Striewe. Using a Triple Graph Grammar for State Machine Implementations. In Hartmut Ehrig, Reiko Heckel, Grzegorz Rozenberg, and Gabriele Taentzer, editors, *Proceedings of the 4th International Conference on Graph Transformations (ICGT) 2008, Leicester*, volume 5214 of *LNCS*, pages 514–516, 2008.

[Sun04] Sun Microsystems, Inc. JSR 175: A Metadata Facility for the Java™Programming Language, 2004. `http://jcp.org/en/jsr/detail?id=175`.

[VG05] Marek Vokáč and Jens M. Glattetre. Using a Domain-Specific Language and Custom Tools to Model a Multi-tier Service-Oriented Application – Experiences and Challenges. In Briand and Williams [BW05], pages 492–506.

[VHB+03] Willem Visser, Klaus Havelund, Guillaume Brat, SeungJoon Park, and Flavio Lerda. Model Checking Programs. *Automated Software Engineering Journal*, 10(2), 2003.

[WS05] Hiroshi Wada and Junichi Suzuki. Modeling Turnpike Frontend System: A Model-Driven Development Framework Leveraging UML Metamodeling and Attribute-Oriented Programming. In Briand and Williams [BW05], pages 584–600.

Kontextsensitive Dialogmodellierung

Jürgen Rückert, Barbara Paech
Institut für Informatik, Universität Heidelberg
Im Neuenheimer Feld 326
69120 Heidelberg
{rueckert,paech}@informatik.uni-heidelberg.de

Abstract: Die Anpassung von graphischen Benutzungsschnittstellen wird in vielen Anwendungsszenarien benötigt. Anwenderinnen möchten beispielsweise die Oberfläche personalisieren oder diese auf den Kontext des Einsatzes reagieren lassen. Wenn es gelingt die Anpassung zu modellieren, die entweder von Menschen definiert werden kann oder von der Maschine automatisch erlernt werden kann, dann kann das Software Engineering von Benutzungsschnittstellen verkürzt werden. Einerseits entsteht durch die Modellierung Mehraufwand. Andererseits verringert sich der Implementierungsaufwand durch modellgesteuerte, wiederverwertbare Komponenten oder durch Modell-zu-Quellcode-Transformationen.
Der Beitrag fokussiert auf die gleichzeitige Modellierung anpassbarer Strukturen, Eigenschaften und Verhalten von Benutzungsschnittstellen. Der Beitrag stellt ein technologieunabhängiges, kontextsensitives Dialogmodell vor. Das Dialogmodell wurde zur Modellierung eines Web Portals verwendet.

1 Einleitung

Die Anpassung von graphischen Benutzungsschnittstellen (BSS) wird in vielen Anwendungsszenarien benötigt. AnwenderInnen möchten beispielsweise die Oberfläche personalisieren oder diese auf den Kontext des Einsatzes reagieren lassen, wie auf den aktuellen Standort oder das verwendete mobile Gerät.
BSS sind bereits aus Aufgaben- [Pat99] und Domänendatenmodellen [Bal94] mit Hilfe von Modelltransformationen ableitbar. In diesen Transformationen müssen viele Entscheidungen und Annahmen festgelegt werden. Wenn die Entscheidungen und Annahmen nicht ausreichend wartbar und änderbar sind, besteht die Gefahr, dass BSS entstehen, die sehr schwer an die Bedürfnisse von AnwenderInnen anpassbar sind.
Um dies zu vermeiden wurden Modelle vorgeschlagen, die die BSS aus technischer Perspektive modellieren und Zwischenergebnisse der Transformationen darstellen, beispielsweise Präsentations- [LV04, Goo01, OAS04] und Dialogmodelle [Træ03, Bas04, W3Cb, Wei94]. Präsentationsmodelle modellieren den graphischen Teil der BSS. Dialogmodelle modellieren das Verhalten der BSS, welches für AnwenderInnen durch die Reaktion auf Ereignisse erfahrbar wird. Aufgrund seiner Komplexität, wird das Dialogmodell häufig bei der Transformation von Aufgaben- und Domänendatenmodellen zu Quellcode vernachlässigt. Es kann aber einen wichtigen Nutzen bringen – es kann entweder die Anleitung für einen Mechanismus sein, der die BSS zur Laufzeit steuert [HHS05, RP08] oder es kann die Grundlage einer Modell-zu-Quellcode-Transformation bilden [BMP04]. Soll

die BSS kontextabhängig sein, dann muss das Dialogmodell kontextsensitiv werden.
Dieser Beitrag stellt ein kontextsensitives Dialogmodell vor. Abschnitt 2 stellt wichtige Anforderungen an kontextsensitive Dialogmodelle auf. Abschnitt 3 stellt bestehende Ansätze zur Dialog- und Kontextmodellierung vor und beurteilt diese auf Erfüllung der in Abschnitt 2 definierten Anforderungen. Abschnitt 4 stellt kurz den bestehenden Dialogmodellierungsansatz *Guilets* [RP08] vor (4.1), der als Basis für das kontextsensitives Dialogmodell dient, erklärt das zur Modellierung der Anpassung verwendbare kontextsensitive *Guilet*-Dialogmodell (4.3) und veranschaulicht in Abschnitt 4.2 dessen Verwendung anhand des realen Web Portals *Heidelberg Mobil* [HMI08]. Abschnitt 5 fasst den Beitrag zusammen und gibt einen Ausblick auf zukünftige Vorhaben.

2 Anforderungen an kontextsensitive Dialogmodelle

Kontextabhängige BSS berücksichtigen beim Übergang von einem Zustand A in einen Zustand B den Nutzungskontext der AnwenderInnen. Kontextabhängige BSS können in vielen Anwendungsszenarien benötigt werden [CK00]. Der Nutzungskontext besteht aus einem Satz von Kontextdaten, die die konkrete Situation beschreiben, in der die Zustandsänderung der BSS gewünscht wird. Kategorisierungen für diese Kontextdaten wurden vielfach vorgeschlagen (z.B. in [Bol07, Kap03]). Die folgende Auflistung zeigt exemplarisch Kontextdaten und Beispiele.

- **Rechte oder Rolle der AnwenderInnen**. Abhängig von der Rolle sind unterschiedliche Aktionen möglich.
- **Persönliche Daten der AnwenderInnen**. Abhängig von individuellen Einstellungen (Profil) werden unterschiedliche Sichten einer Seite angezeigt. Sprache, Nationalität, Geschlecht bestimmen das Layout und die Datendarstellung. Abhängig vom Geburtsdatum und Altersgruppe werden interessierende Produkte angezeigt.
- **Eingegebene oder ausgewählte Daten**. Abhängig vom Einkaufsbetrag wird eine Kreditprüfung veranlasst.
- **Hardware**. Abhängig von der verwendeten Hardware werden Seiten in mehrere kleinere Seiten zerlegt und die Abfolge der Seiten angepasst.
- **Umgebung**. Abhängig vom Wetter werden die angezeigten Daten sortiert (bei Regen zuerst die Museen, bei Sonne die Schwimmbäder).
- **Zeitpunkt**. Abhängig von der Tageszeit (Tag oder Nacht) wird das Layout (Farben) angepasst.
- **Ort**. Abhängig vom Standort werden interessante Orte in der Nähe angezeigt.

Üblicherweise dienen Kontextregeln dazu die Kontextdaten, die die konkrete Situation beschreiben, in eine Liste von Anpassungen zu überführen. Die Kontextregel besitzt eine Wenn-Seite (left hand side, LHS) mit bool'schen Bedingungen und eine Dann-Seite (right hand side, RHS) mit Anpassungen für die BSS. Es ist wahrscheinlich, dass die Regeln während der Entwicklungszeit der BSS noch nicht alle bekannt sind oder, dass sich diese während des Betriebs der BSS ändern. Die BSS kann sich auf mehrere Arten und Weisen

anpassen. Im Folgenden sind die möglichen Anpassungen dargestellt. Wir nutzen diese Anpassungen zur Definition von Anforderungen an kontextsensitive Dialogmodelle.

- Sichten können aus anderen Sichten zusammengesetzt werden. Beispiel: Die Startseite eines Web Portals zum eLearning ist zusammengesetzt aus Wochenübersicht-, Newsforum- und Dateiablage-Sichten. 1. Diese Kombination kann allerdings von den AnwenderInnen in ihrem Profil geändert werden. 2. Für ein mobiles Gerät darf die Startseite nur aus einer einzigen Sicht bestehen, da das Display des mobilen Geräts nicht ausreichend groß ist.
  Anforderung 1. Das kontextsensitive Dialogmodell soll die Beschreibung der Anpassung der Komposition von Seiten und Sichten erlauben. Ziel dieser Anforderung ist, ein Dialogmodell zu schaffen, welches in der Lage ist deutlich zu machen, bei welchem Ereignis Seiten und Sichten neu zusammengestellt werden.
- Sichten und ihre Bestandteile können ihre Eigenschaften ändern. Beispiel: Neues Layout der Sichten, dies umfasst Farbe und Position.
  Anforderung 2. Das kontextsensitive Dialogmodell soll die Beschreibung der Anpassung von Eigenschaften von Sichten, Seiten und Widgets erlauben. Ziel dieser Anforderung ist, ein Dialogmodell zu schaffen, welches in der Lage ist deutlich zu machen, bei welchem Ereignis die Eigenschaften neu gesetzt werden.
- Sichten können mit anderen Daten aus anderen Anwendungsdiensten versehen werden. Beispiel für die Änderung der Datenquellen: Im Kontext *Freizeit* werden die Veranstaltungen von anderen externen Diensten geladen wie im Kontext *Geschäftsbetrieb*. Beispiel für die Änderung der Datenabfragen: Eine Seite zur Buchung eines Mietwagens wird bei angemeldeten AnwenderInnen mit maßgeschneiderten Auswahlen für Fahrzeugtyp, -farbe und Abholort belegt [LKZ06].
  Anforderung 3. Das kontextsensitive Dialogmodell soll die Beschreibung der Anpassung der Datenweitergabe an Sichten und Widgets erlauben und damit deren kontextabhängige Datenverwendung. Ziel dieser Anforderung ist, ein Dialogmodell zu schaffen, welches in der Lage ist deutlich zu machen, dass die angesprochenen Dienste zur Datenabfrage und deren Ergebnis kontextabhängig variieren.
- Sichten können andere Anwendungsdienste verwenden, um die Daten der Widgets zu verarbeiten. Beispiel: Die Prüfung der Kreditkarte erfolgt bei unterschiedlichen Banken (Web Services) – je nach Land in dem sich die AnwenderInnen befinden.
  Anforderung 4. Das kontextsensitive Dialogmodell soll die Beschreibung der Anpassung der Verwendung von Daten ermöglichen. Die Daten werden verwendet von Widgets. Ziel dieser Anforderung ist, ein Dialogmodell zu schaffen, welches in der Lage ist auszudrücken, dass die angesprochenen semantischen Dienste (zur permanenten Datenänderung) variieren.
- Sichten können unterschiedliche Aktionen für AnwenderInnen anbieten. Beispiel: Menüeinträge können kontextabhängig verfügbar sein.
  Anforderung 5. Das kontextsensitive Dialogmodell soll die Beschreibung der Anpassung der zur Verfügung stehenden AnwenderInnen-Aktionen erlauben. Ziel dieser Anforderung ist, ein Dialogmodell zu schaffen, welches in der Lage ist zu beschreiben, welche Ereignisse abhängig vom Kontext ausgelöst werden können.

3 Verwandte Arbeiten

Man unterscheidet Ansätze zur Modellierung des Kontextes (z.B. UsiXML [LV04]), zur Modellierung der Anpassung der BSS (z.B. Catwalk [LKZ06], UWE [KKK07] und aufgabenbasierte Ansätze wie Contextual CTT [BC04] und [WFTS07]) und zur Modellierung und Auswertung von Kontextregeln (z.B. [Che04]). Letztere sind für uns nicht von Interesse, da die Auswertung von Regeln (Context Reasoning) nicht Bestandteil der Dialogmodellierung ist.
Stellvertretend wird der populäre Ansatz UsiXML und der auf semantischen Technologien basierende Ansatz Catwalk in den Abschnitten 3.1 und 3.2 vorgestellt. Die Ansätze erfüllen die in Abschnitt 2 aufgestellten Anforderungen in unterschiedlich hohem Maße. Dies liegt daran, dass die Ansätze nicht primär die Dialogmodellierung fokussieren. Tabelle 1 zeigt die Erfüllung der Anforderungen. Die Begründung der Beurteilung wird in nachfolgenden Abschnitten 3.1 und 3.2 gegeben.

Anforderung	Kontextabhängige...	UsiXML	Catwalk
1	...Komposition der Präsentation	modellierbar	modellierbar
2	...Eigenschaften der Präsentation	modellierbar	modellierbar
3	...Datenanpassung	-	modellierbar
4	...Datenverwendung	-	-
5	...Aktionen	modellierbar	-

Tabelle 1: Erfüllung der Anforderungen durch bestehende Ansätze für kontextabhängige BSS

3.1 UsiXML

Der zur Entwicklung von modellbasierten BSS populäre Ansatz UsiXML [LV04] bietet neben einem Präsentationsmodell ein Kontextmodell. Ein explizites Dialogmodell ist nicht Bestandteil von UsiXML. Das Kontextmodell von UsiXml modelliert die Umgebungen, AnwenderInnen und Plattformen. Das Plattformmodell basiert auf dem *User Agent Profile* [WAP01], welches eine Verfeinerung des *Composite Capabilities / Preferences Profile* [W3Ca] des W3C darstellt. Entsprechend ist das Plattformmodell sehr technologisch orientiert. Das Nutzermodell besteht aus einer einzigen Klasse *userStereotype* mit fünf Eigenschaften. Das Umgebungsmodell besteht aus einer einzigen Klasse *environment* mit drei Eigenschaften. Zur Modellierung der Anpassung bietet UsiXML keine speziellen Möglichkeiten. Es besteht nur die allgemeine Möglichkeit das Präsentationsmodell durch XML-Transformation unter Verwendung des Kontextmodells in ein neues UsiXML-Präsentationsmodell zu transformieren [Sta05]. Die Entwicklung und Wartung der XML-Transformation ist sehr aufwendig, wenig transparent und erlaubt keine ereignisbasierte Anpassung der BSS und, durch Fehlen des Dialogmodells, keine Modellierung der Verhaltensanpassung. UsiXML erlaubt durch die XML-Transformation unter Verwendung des Werkzeugs *TransformiXML* [Sta05] (unter Verwendung der *Attributed Graph Grammar*) die Anpassung der Komposition (Anforderung 1) und der Eigenschaften (Anforderung 2) der Präsentation. Da Aktionen von AnwenderInnen durch graphische Komponenten ausgelöst werden, kann durch Ein- und Ausblenden die Anforderung 5 rudimentär erfüllt werden.

3.2 Catwalk

Catwalk [LKZ06] ist ein komponentenorientiertes Rahmenwerk für Web-Anwendungen. Die BSS wird durch verschiedene Ontologien beschrieben: eine *domain ontology*, mehrere *context ontologies* und ein *context relations model*, welches die beiden vorherigen Ontologien in Beziehung setzt. Das *navigation*, *view* und *presentation model* enthalten Regeln zur Anpassung des Systems auf Basis der Ontologien. Catwalk unterscheidet die *selection*, *parameterization* und *presentation* der Präsentation, was der Anpassung der Komposition (Anforderung 1), der Eigenschaften (Anforderung 2) und Datenpräsentation (Anforderung 3) entspricht. Catwalk erlaubt die Anpassung der Komposition und der Eigenschaften der Präsentation, sowie der Datenabfragen – berücksichtigt aber nicht die Anpassung der Datenauswertung und der zur Verfügung stehenden Aktionen.

Die derzeitigen Ansätze zur Modellierung von BSS, von denen hier nur eine sehr kleine Anzahl von repräsentativen Vertretern vorgestellt werden konnte (mächtigere Ansätze sind uns nicht bekannt) bieten zwar viele Möglichkeiten zur Beschreibung des Kontexts, zur Modellierung der Präsentation oder bieten BSS-Referenz-Architekturen an, die klar separierte Komponenten (*Context Reasoner*) zur Ermittlung der notwendigen Anpassungen enthalten. Die Beschreibung der Änderung der BSS in einem Dialogmodell wurde bisher allerdings noch nicht vollständig addressiert. Dies ermöglicht die Dialogmodellierung mit *Guilets*, die im nächsten Kapitel erläutert wird.

4 Guilets

4.1 Der Lösungsansatz

Der Lösungsansatz *Guilets* [RP08] stellt zum Erstellen der anwendungsspezifischen *Guilet*-Dialogmodelle in XML während des Entwurfs bereit: **1.** Das *Guilet*-Metamodell in XML Schema. **2.** Den *Guilet*-Editor, als Eclipse-Feature auf Basis des *Eclipse Graphical Modeling Framework* in Java (und auf Basis des *Guilet*-Metamodells).
Der Lösungsansatz *Guilets* stellt (auf Basis des *Guilet*-Metamodells) zur schnellen Entwicklung eines BSS-Prototyps und zur Simulation bereit: **3.** Die *Guilet*-Referenzarchitektur für BSS. **4.** Ein Werkzeug in Java zur Generierung von Quellcode von Verhaltenskomponenten (so genannte Exekutoren, eine Default-Implementierung) und deren Schnittstellen. **5.** Ein Werkzeug in Java zur Generierung von Quellcode der Präsentationen (eine Default-Implementierung) und deren Schnittstellen. **6.** Ein *Guilet*-Dialogkern in Java zur Koordination der BSS auf Basis aller oben genannten Einheiten.
Das Charakteristische am *Guilet*-Dialogmodell ist, und das unterscheidet es von anderen Ansätzen zur Dialogmodellierung [Træ03, Bas04, W3Cb, Wei94], die integrierte Modellierung des hierarchischen Aufbaus von Sichten, die Abstraktion von graphischen Komponenten (*Pisa Interactor Abstraction* [Mar97]), die Modellierung von Sichtenabfolgen (UML Statecharts [Har87]) und die Verwendung von Modellelementen für Anwendungs- und BSS-Dienste innerhalb von Sichten.
Im Rahmen, des im Abschnitt 4.2 vorgestellten Beispiels, wurde ein Dialogmodell erstellt (1. und 2.), aus diesem Modell Default-Verhaltenskomponenten und Default-Präsentationen generiert (4. und 5.), Präsentationen und kontextabhängige Verhaltenskomponen-

Abbildung 1: Ausschnitt des Web Portals *Heidelberg Mobil* (dicke Linien zeigen die Abgrenzung der Sichten)

ten an die konkreten Bedürfnisse angepasst (implementiert) und alle Komponenten durch den Dialogkern zur Laufzeit koordiniert (6.).

4.2 Das Anwendungsbeispiel

Zur Veranschaulichung einer kontextabhängigen, graphischen BSS verwenden wir das Web Portal *Heidelberg Mobil* [HMI08]. Abbildung 1 zeigt die Webseite des Portals vor (links) und nach (rechts) der Anpassung. Die Webseite ist eine Komposition einzelner Sichten, die eindeutig identifizierbar sind (`06`) und einen Namen haben (`Landkarte`). Die mobile Webseite [HMI08] von *Heidelberg Mobil* wird ebenfalls aus Sichten kombiniert. Durch die größenbeschränkten Geräte (PDA, iPhone, Handy) können nicht alle der in Abbildung 1 dargestellten Sichten verwendet werden. Zudem entfallen die Sichten zur Suche und Auswahl der aktuellen Lokation durch die automatische Ermittlung des Standorts, welche bei mobilen Geräten ohne Zutun der AnwenderInnen geschieht. Der in diesem Abschnitt 4 vorgestellte Lösungsansatz kann für beliebige Kontextänderungen angewandt werden (nicht nur für Lokationsänderungen) und ist damit für beliebige mobile und nicht-mobile Web Portale verwendbar. Die Anpassung erfolgt unter Berücksichtigung der Kontextregeln aus Tabelle 2. Die Kontextregeln werten Stamm- und Bewegungsdaten der AnwenderInnen aus. Es wird angenommen, dass die AnwenderInnen dem System bekannt sind (durch Einloggen). Bewegungsdaten sind beispielsweise das von Anwende-

Bedingung (LHS)		Aktion (RHS)		
Ausdruck	**Kategorie**	**ID**	**Ausdruck**	**Anf.**
Nutzer.Ereignis = SucheVeranstaltung	Ereignis	1	Entferne Sicht 5Tops	1
		2	Entferne Sicht 3Tops	1
		3	Zeige Sicht Veranstaltungübersicht	1
		4	Entferne Sicht Interessante Orte	1
		5	Zeige Sicht VeranstaltungenFinden	1
		6	Zeige Sicht VeranstaltungsorteFinden	1
Nutzer.Name = Mayer	Stammdaten	7	Lade die persönlichen Bookmarks in Sicht Bookmarks	3
		8	Setze Hintergrundfarbe weiß im Fenster	2
Nutzer.Typ = Heidelberger	Stammdaten	9	Verschiebe die Sicht Landkarte nach unten	2
		10	Verschiebe die Sicht Bookmarks nach oben	2
		11	Entferne die Sicht BesuchHeidelberg	1
Lokation = Unbestimmt	Lokation	12	Positioniere die Sicht Landkarte auf den Bismarckplatz	3
Uhrzeit > 22:00 und < 23:00	Uhrzeit	13	Zeige in der Sicht Veranstaltungsorte zuerst die Diskotheken an	3
Gerät = Desktop	Plattform	14	Zeige in Sicht Bookmarks die Aktion Mitnehmen an	5

Tabelle 2: Kontextregeln am Beispiel eines kontextabhängigen Web Portals (Anf. referenziert die Anforderungen aus Abschnitt 2)

rInnen ausgelöste Ereignis (Zeile 1), die zur Laufzeit ermittelten Werte Lokation (Zeile 4) und Uhrzeit (Zeile 5) und das Gerät, an dessen Hardware die BSS angepasst werden soll (Zeile 6). Zu den Stammdaten zählen die Personalisierungseinstellungen, die am Namen (im Sinne ID) festgemacht werden (Zeile 2), verschiedene Eigenschaften (ob ortskundig oder nicht, Zeile 3). Die Spalte *Anforderung* zeigt, dass die Sichten des Portals häufig neu komponiert werden (Anforderung 1), dass Eigenschaften von Sichten und Widgets geändert werden (Anforderung 2), dass die Widgets mit kontextabhängigen Daten gefüllt werden (Anforderung 3) und dass AnwenderInnen kontextabhängig Ereignisse auslösen können (Anforderung 5). Das Portal besitzt allerdings keine kontextabhängige Auswahl von Anwendungsdiensten (Anforderung 4). Bei der Dialogmodellierung des Portals gehen wir davon aus, dass die BSS aus Seiten besteht, die in einem Rahmen angezeigt werden, beispielsweise eine Webseite in einem Browser oder eine Seite zum Einloggen in einem neu erscheinenden modalen Fenster. Die Seiten sind aus geschachtelten Sichten aufgebaut. Die Sichten beinhalten elementare, graphische Komponenten (Widgets), beispielsweise Textfelder, Tabellen oder Landkarten. In den Seiten, Sichten und Widgets können AnwenderInnen Ereignisse auslösen. Wie in Abschnitt 2 gezeigt, kann der Kontext die Präsentation und das Verhalten der BSS umfangreich ändern – prinzipiell bei jedem von AnwenderInnen ausgelösten Ereignis. Dialogmodelle beschreiben das Verhalten bei Eintreten eines Ereignisses, wobei üblicherweise das Ereignis einer Seite, Sicht oder Widget zugeordnet wird. Das Dialogmodell referenziert Elemente des Präsentationsmodells. Normalerweise folgt die Reaktion auf Ereignisse einer festgelegten Beschreibung (man denke an Statecharts oder Aktivitätsdiagramme). Die von uns in Abschnitt 2 aufgestellten Anforderungen sollen Dialogmodelle ermöglichen, die in der Lage sind auszudrücken, wo es in dieser festgelegten Beschreibung zu kontextmotivierten Anpassungen kommen kann.

4.3 Das Modell zur kontextsensitiven Dialogmodellierung

Zur Umsetzung der Anforderungen aus Abschnitt 2, wurde das *Guilet*-Dialogmodell aus [RP08] um drei Konzepte erweitert:

- **Konzept 1.** Zwei Untertypen von Exekutoren wurden eingeführt, die die spezielle Semantik *KomponiereSichten* und *SetzeEigenschaften* besitzen.
- **Konzept 2.** Exekutoren, die sich bisher mit Anwendungsdiensten verbunden haben, können nun zusätzlich mit einem *Context Reasoner* verbunden werden, der über die auszulösenden Aktionen entscheidet.
- **Konzept 3.** InEvents, die Ereignissen entsprechen, können nun zusätzlich mit einem *Context Reasoner* verbunden werden, der über deren Auslösungsmöglichkeit entscheidet.

Das *Guilet*-Dialogmodell [RP08] basiert auf der (nicht neuen) Idee, dass atomare Sichten gebildet werden, die danach zu komplexen Sichten zusammengesetzt werden. Neu ist die Art wie das Verhalten der atomaren und der komplexen Sichten modelliert wird. Die in [RP08] vorgestellte Version des *Guilet*-Dialogmodells unterstützt die Berücksichtigung des Kontexts noch nicht, die Lücke soll mit diesem Beitrag geschlossen werden. Im nächsten Absatz wird das *Guilet*-Dialogmodell aus [RP08] vorgestellt, bevor im übernächsten Absatz die kontextsensitiven Erweiterungen genannt werden.
Guilets sind Modelle von graphischen Komponenten wie Fenster, darin enthaltenen Sichten und Widgets. Atomare *Guilets* (**BlackBoxGuilets**) modellieren graphische Komponenten deren Verhalten nicht von Interesse ist. Dies ist der Fall bei Verwendung von graphischen Komponenten aus Bibliotheken, die ein zu kompliziertes Verhalten besitzen, um modelliert zu werden. Komplexe *Guilets* (**WhiteBoxGuilets**) modellieren graphische Komponenten, deren Verhalten von Interesse ist. Komplexe *Guilets* bestehen aus komplexen *Guilets* oder aus atomaren *Guilets*. Durch Auswertung des Kontexts zur Laufzeit können die beinhalteten Sichten variieren (dynamische Komposition). Eine In-Schnittstelle (**InEvent**) wird für jede Nutzeraktion modelliert, die die Sicht anbietet. Über die Schnittstelle können Daten übergeben werden, müssen aber nicht. Eine Out-Schnittstelle (**OutEvent**) wird für jede Nachricht modelliert, die die Sicht senden kann und die andere Sichten empfangen und nutzen können (über das Modellierungselement Transition). **Exekutoren** sind Modelle für Dienst-Komponenten und repräsentieren Anwendungsdienste oder BSS-Dienste. Ein Exekutor bietet eine Schnittstelle, die die Ausführung des Exekutors erlaubt. Während der Ausführung können Exekutoren festgelegte globale Daten lesen und festgelegte globale Daten (dieselben oder andere) schreiben. Ein Exekutor kann mehrere Schnittstellen von Sichten nutzen. Durch Auswertung des Kontexts können die tatsächlich genutzten Schnittstellen variieren bzw. erst zur Laufzeit bekannt werden. Alle Modellierungselemente haben eine ID, einen Namen, eine Beschreibung und können beliebig definierbare Eigenschaften (Schlüssel-Wert-Paare) enthalten. Hiermit können beispielsweise Layout-Vorgaben definiert werden (beispielsweise Position, Anzeige in Fenster).
Wir schlagen die folgenden Erweiterungen des *Guilet*-Dialogmodells vor:

- **Realisierungen von Konzept 1.** Die Anpassung der Komposition (Anforderung 1 aus Abschnitt 2) von Seiten und Sichten wird durch einen speziellen BSS-Dienst

beschrieben, den *ComposeGuiletsExecutor*. Die Anpassung der Eigenschaften (Anf. 2 aus Abschnitt 2) von Seiten, Sichten und Widgets wird durch einen speziellen BSS-Dienst beschrieben, den *SetPropertyExecutor*.

- **Realisierungen von Konzept 2.** Die kontextabhängige Weitergabe von Daten aus Diensten (Exekutoren) an Seiten, Sichten und Widgets (*Guilets*) (Anf. 3 aus Abschnitt 2) wird durch einen *Context Reasoner* entschieden, falls der Exekutor mit Hilfe des *ContextualFlag* markiert wurde. Die kontextabhängige Weitergabe von Daten aus Widgets (*Guilets*) an Dienste (Exekutoren) (Anf. 4 aus Abschnitt 2) wird durch einen *Context Reasoner* entschieden, falls der Exekutor mit Hilfe des *ContextualFlag* markiert wurde.
- **Realisierung von Konzept 3.** Die Verfügbarkeit (Anf. 5 aus Abschnitt 2) von auslösbaren Ereignissen (InEvents) wird durch einen *Context Reasoner* entschieden, falls der InEvent mit Hilfe des *ContextualFlag* markiert wurde.

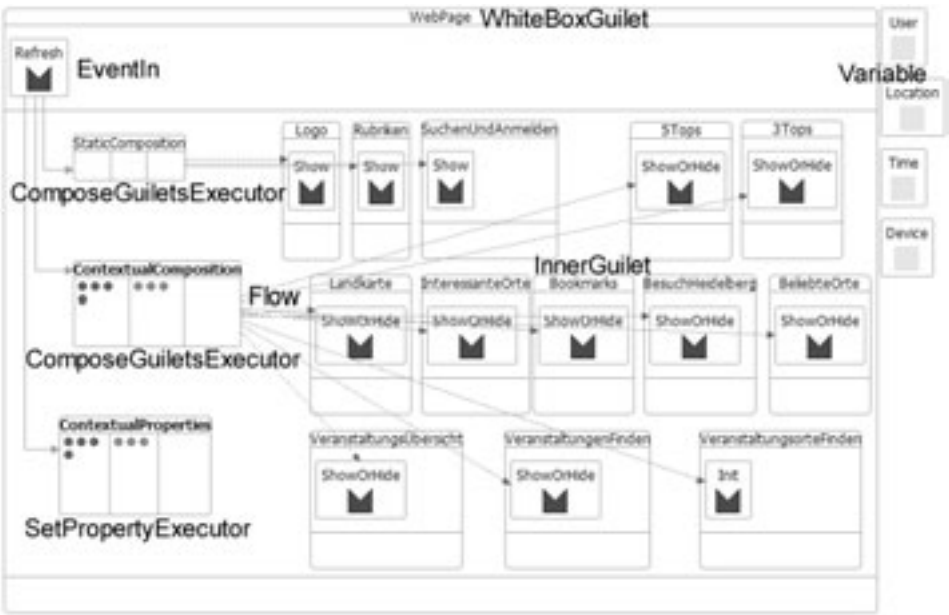


Abbildung 2: Ausschnitt aus dem *Guilet*-Dialogmodell zur Modellierung der Seite (die zugehörigen Typen der Modellelemente sind fettgedruckt)

Abbildung 2 veranschaulicht anhand das *Guilet*-Dialogmodells der Webseite, wie die Anpassung (Tabelle 2) der Sichtenkomposition (1-6, 11) und der Sichteneigenschaften (8-10) modelliert werden. Die Anpassung des Sichtenverhaltens (7, 12-14) erfolgt durch kontextabhängige Verhaltensexekutoren in den einzelnen Sichten und ist hier nicht dargestellt, da im kontextfreien *Guilet*-Dialogmodell nur auf kontextabhängige Exekutoren umgestellt werden muss.
Aus dem Dialogmodell kann folgende Abfolge, zur Durchführung der Anpassung, abge-

lesen werden: 1. Die Webseite empfängt das Ereignis *Refresh*. 2.1 Der Exekutor *StaticComposition* wird aufgerufen, um die 3 immer vorhandenen Sichten anzuzeigen (statische Komposition). 2.2 Der Exekutor *ContextualComposition* (vom neuen, speziellen Typ *ComposeGuiletsExecutor*) wird aufgerufen, um die durch den Kontext bestimmten Sichten zu ermitteln (kontextsensitive Komposition). 2.3 Der Exekutor *ContextualProperties* (vom neuen, speziellen Typ *SetPropertyExecutor*) wird aufgerufen, um die Eigenschaften (Farben und Positionen) der Webseite zu setzen. Bei beiden Exekutoren ist das neue *ContextualFlag* gesetzt, damit diese den *Context Reasoner* ansprechen, der die anzuzeigenden Sichten (IDs) und die Eigenschaften (Schlüssel-Wert-Paare) der Sicht ermittelt. Dem *Context Reasoner* stehen als Datenbasis der Kontextregeln die 4 globalen Variablen *User*, *Location*, *Time* und *Device* zur Verfügung (Punkte der linken Sektion). 3. Die einzelnen Sichten erhalten ein Ereignis, welches nach dem Dialogmodell der einzelnen Sichten abgearbeitet wird (letzteres Modell ist nicht dargestellt).
Bei der modellbasierten Entwicklung des Portals *Heidelberg Mobil* haben wir positive und negative Erfahrungen gesammelt. Dies ist einerseits auf die generelle Verwendung von Modellen zurückzuführen und andererseits auf die spezielle Verwendung eines kontextsensititen Dialogmodells. (+) Die Erweiterung des Dialogmodells auf Kontextsensitivität war einfach durch die bestehende Abstraktion der Dienste (Executors) und Ereignisauslöser (InEvents). (+) Das Dialogmodell dokumentiert diejenigen Sichten, die kontextsensitiv sind, in transparenter Art und Weise. Eine Suche im Quellcode nach den von Anpassung betroffenen Sichten und den zur Anpassung ausgewerteten Daten entfällt. (-+) Einerseits müssen die Kontextregeln des *Context Reasoner* mit dem Dialogmodell synchronisiert werden, weil die Kontextregeln Elemente des Dialogmodells referenzieren. Andererseits ist die Synchronisierung einfach, weil man an zentraler Stelle (im Dialogmodell) nach den Elementen suchen kann. (+) Weder Quellcode noch Dialogmodell mussten zur Realisierung von prototypischen Anpassungsszenarien verändert werden. Die Änderung der Kontextregeln (in einem *Context Reasoner*) war ausreichend. (-) Die erstmalige Erstellung des Modells und die Entwicklung der Transformation von Modell in Quellcode hat Aufwand verursacht. (+) Das Dialogmodell umfasst nun gleichzeitig Sichten der mobilen und der nicht-mobilen Version der Webseiten. Eine Erweiterung um neue Sichten kann automatisch beiden Versionen zu Gute kommen. Ebenso verhält es sich bei dem Umstieg auf ein alternatives Web-Rahmenwerk.

5 Zusammenfassung und Ausblick

In diesem Beitrag haben wir Anforderungen an kontextsensitive Dialogmodelle aufgestellt (Abschnitt 2). Wir haben zwei bestehende Ansätze zur kontextabhängigen Anpassung von BSS vorgestellt (Abschnitt 3) und unsere Lösung zur kontextsensitiven Dialogmodellierung präsentiert (Abschnitte 4.1 und 4.3). Unsere Lösung haben wir anhand des realen Web Portals *Heidelberg Mobil* (Abschnitte 4.2 und 4.3) veranschaulicht.
Das *Guilet*-Dialogmodell ermöglicht die Modellierung der Anpassung der Komposition, der Eigenschaften und des Verhaltens von Sichten, wie sie für kontextabhängige BSS notwendig sind. Zur Anpassung der Komposition von Sichten haben wir einen speziellen BSS-Dienst definiert. Zur Anpassung der Eigenschaften von Seiten, Sichten und graphischen Komponenten haben wir ebenfalls einen speziellen BSS-Dienst definiert. Die

Modellierungselemente für BSS-Dienste und Anwendungsdienste können *kontextsensitiv* sein. Zur Laufzeit verbinden sich die Dienst-Komponenten, die auf Basis dieser kontextsensitiven Modellierungselemente implementiert werden, zu externen *Context Reasoner* Komponenten, die die gewünschten auszulösenden Ereignisse der BSS ermitteln. Das *Guilet*-Metamodell wurde erfolgreich verwendet zur Modellierung des Web Portals *Heidelberg Mobil* im Rahmen eines Reverse Engineering. Wir haben, wie uns die Anwendung auf das Web Portal *Heidelberg Mobil* gezeigt hat, ein attraktives Dialogmodell geschaffen, welches kontextabhängige BSS modellieren kann und die Lücke zwischen abstrakten Modellen (wie Aufgaben- und Domänendatenmodellen) und Quellcode schließen kann, indem es Präsentationsmodelle um Verhaltensbeschreibungen ergänzt und einen Entwurf ermöglicht, der mit der Implementierung synchron ist.
Wir planen die weitere Evaluation des *Guilet*-Metamodells, um die Verwendbarkeit für unterschiedliche Implementierungstechnologien und weitere kontextsensitive Anwendungsszenarien zu überprüfen. Erstens soll unsere Entwurfsentscheidung gefestigt werden: *Es ist für die Wartbarkeit des Dialogmodells vorteilhafter, wenn die Kontextregeln nicht Bestandteil des Dialogmodells sind.* Zweitens soll die Eignung des Dialogmodells für die *kontextabhängige Auswahl von Anwendungsdiensten* (Anforderung 4 aus Abschnitt 2) in einem dienstorientierten Umfeld gezeigt werden.
Wir danken Matthias Jöst von *Heidelberg Mobil International* für die freundliche Zusammenarbeit und Unterstützung der Arbeit.

Literatur

[Bal94] H. Balzert. Das JANUS-System. *Inform., Forsch. Entwickl.*, 9(1):22–35, 1994.

[Bas04] R. Bastide. A model-based approach for real-time embedded multimodal systems in military aircrafts. In *ICMI*, Seiten 243–250, 2004.

[BC04] J. van den Bergh und K. Coninx. Contextual ConcurTaskTrees: Integrating dynamic contexts in task based design. *Pervasive Computing and Communications Workshops, IEEE International Conference on*, 0:13, 2004.

[BMP04] S. Berti, G. Mori und F. Paternò. A transformation-based environment for designing multi-device interactive applications. In *IUI '04: Proceedings of the 9th international conference on Intelligent user interfaces*, Seiten 352–353. ACM, 2004.

[Bol07] C. Bolchini. A data-oriented survey of context models. *SIGMOD*, 36(4):19–26, 2007.

[Che04] H. Chen. SOUPA: Standard Ontology for Ubiquitous and Pervasive Applications. In *Internation Conference on Mobile and Ubiquitous Systems: Networking and Services*, Boston, MA, 2004.

[CK00] G. Chen und D. Kotz. A Survey of Context-Aware Mobile Computing Research. Bericht, 2000.

[Goo01] B. Goodger. XML User Interface Language (XUL) Specification 1.0 and Documentation., 2001. http://www.mozilla.org/projects/xul/.

[Har87] D. Harel. Statecharts: A visual formalism for complex systems. *Sci. Comput. Program.*, 8(3):231–274, 1987.

[HHS05] M. Haft, B. Humm und J. Siedersleben. The Architect's Dilemma - Will Reference Architectures Help?. In *QoSA/SOQUA*, Jgg. 3712 of *LNCS*, Seiten 106–122. Springer, 2005.

[HMI08] Heidelberg Mobil International. Das Web Portal Heidelberg Mobil, Stand vom 25. September 2008. http://www.heidelberg-mobil.de/ http://www.heidelberg-mobil.de/mobile.

[Kap03] G. Kappel. Customisation for ubiquitous web applications a comparison of approaches. *Int. J. Web Eng. Technol.*, 1(1):79–111, 2003.

[KKK07] A. Kraus, A. Knapp und N. Koch. Model-Driven Generation of Web Applications in UWE. In *3rd International Workshop on Model-Driven Web Engineering (MDWE)*, Como, Italy, July 2007.

[LKZ06] S. Lohmann, J. Kaltz und J. Ziegler. Model-Driven Dynamic Generation of Context-Adaptive Web User Interfaces. In *MoDELS Workshops*, Seiten 116–125, 2006.

[LV04] Q. Limbourg und J. Vanderdonckt. USIXML: A User Interface Description Language Supporting Multiple Levels of Independence. In *ICWE Workshops*, Seiten 325–338, 2004.

[Mar97] P. Markopoulos. A compositional model for the formal specifiation of user interface software. PhD thesis at Department of Computer Science, Queen Mary and Westfield College, University of Londo, 1997.

[OAS04] OASIS. User Interface Markup Language Specification (UIML) 3.1 (Working Draft). http://uiml.org/, March 2004.

[Pat99] F. Paternò. *Model-Based Design and Evaluation of Interactive Applications*. Springer, London, 1999.

[RP08] J. Rückert und B. Paech. The Guilet Dialog Model and Dialog Core for Graphical User Interfaces. In P. Forbrig und F. Paternò, Hrsg., *Engineering Interactive Systems 2008*, Jgg. 5247 of *LNCS*, Seiten 197–204. Springer, 2008.

[Sta05] A. Stanciulescu. A transformational approach for multimodal web user interfaces based on UsiXML. In *7th International Conference on Multimodal interfaces (ICMI'05)*, Seiten 259–266. ACM, 2005.

[Træ03] H. Trætteberg. Dialog Modelling with Interactors and UML Statecharts - A Hybrid Approach. In *DSV-IS*, Seiten 346–361, 2003.

[W3Ca] W3C. Composite Capabilities / Preference Profiles: Structure and Vocabularies 2.0. http://www.w3.org/Mobile/CCPP/.

[W3Cb] W3C. State Chart XML (SCXML): State Machine Notation for Control Abstraction. Working Draft, 16 May 2008. http://www.w3.org/TR/scxml/.

[WAP01] WAP Forum. Wireless Application Protocol, User Agent Profile, WAP-248-UAPROF-20011020-a, 2001. http://www.openmobilealliance.org/.

[Wei94] M. Weiser. The world is not a desktop. *Interactions*, 1(1):7–8, 1994.

[WFTS07] M. Wurdel, P. Forbrig, Radhakrishnan T. und D. Sinnig. Patterns for Task- and Dialog-Modeling. In *HCI (1)*, Seiten 1226–1235, 2007.

Modellierung und Qualitätssicherung von UML-Modellen der Geschäftslogik von Informationssystemen

Tobias Brückmann, Volker Gruhn

Professur für Angewandte Telematik / e-Business
Universität Leipzig
Klostergasse 2
04107 Leipzig
brueckmann@ebus.informatik.uni-leipzig.de
gruhn@ebus.informatik.uni-leipzig.de

Abstract: Der zunehmenden Komplexität von Software-Modellen, die als Parameter für Codegenerierung verwendet werden, steht der menschliche Modellierer gegenüber, dessen mögliche Modellierungsfehler sich direkt auf das generierte Systemverhalten auswirken. Der Modellierer muss je nach konkreter Aufgabe die gewünschte Sicht auf das Modell wählen können und darüber hinaus durch automatische Qualitätssicherung während der Modellierung unterstützt werden. Dieser Artikel erläutert wie die Modellierung von UML-Modellen der Geschäftslogik für Informationssysteme im Rahmen der Modellarchitektur AMABULO unterstützt wird. Dabei integriert AMABULO verschiedene Modellierungskonzepte in ein semantisches Metamodell, das durch die Abbildung auf gefärbte Petri-Netze semantisch fundiert wird. Die automatische Übersetzung der logischen Information aus UML-Modellen für Geschäftslogik in gefärbte Petri-Netze bildet die Grundlage für die Funktionen einer automatischen Qualitätssicherung.

1 Einleitung

Seitdem das Paradigma der „Modellbasierten Softwareentwicklung“ mehr und mehr Einzug in die industrielle Softwareentwicklung hält, ändern sich auch die Anforderungen an visuelle Softwaremodelle. Wurden Letztere als Mittel zur zwischenmenschlichen Kommunikation und zur Illustration konzipiert, so werden sie gegenwärtig immer häufiger als Parameter zur automatischen Generierung von Programmcode verwendet. Für diesen Einsatz muss das entsprechende Modell die Anforderungen an ein System vollständig, konsistent und korrekt spezifizieren, wodurch visuelle Modelle komplexer und gleichzeitig Modellierungsfehler kritischer werden: Ein Fehler im Modell resultiert unmittelbar in ein Systemverhalten, das nicht den Anforderungen entspricht. Dem gegenüber steht der manuelle Modellierungsprozess: Modelle werden von Menschen erstellt und verfeinert. Um den Modellierer einen relativ einfachen kognitiven Zugang zu komplexen visuellen Modellen zu ermöglichen, sind verschiedene Sichten auf das Modell in unterschiedlichen Abstraktionsebenen erforderlich. Somit kann der Nutzer je nach konkreter Aufgabe die gewünschte Sicht auf das Modell wählen [WL06].

Um Inkonsistenzen oder mögliche (syntaktische und semantische) Fehler innerhalb eines großen Modells (mit mehreren hundert oder tausend Modellelementen verteilt auf mehrere Sichten) zu erkennen, muss der Modellierungsprozess darüber hinaus durch eine automatische Qualitätssicherung unterstützt werden [FR07]. Eine sinnvolle automatische Qualitätssicherung ist allerdings nur möglich, wenn die Semantik der Modellelemente eindeutig definiert ist.

Unserer Forschungsarbeit zielt auf die semantische Integration derjenigen Modellierungskonzepte, die für die Modellierung von Geschäftslogik relevant sind, sowie auf die automatische Qualitätssicherung des integrierten Modells unter der Voraussetzung einer formal beschriebenen Semantik. Den Fokus legen wir dabei auf Informationssysteme, die bestimmte Geschäfts- oder Verwaltungsprozesse innerhalb einer Organisation oder zwischen mehreren Organisationen unterstützen (e-Government, e-Business). Konkret beschränken wir uns in unseren Betrachtungen auf Modelle und die Modellierung von Geschäftslogik. Den Begriff „Geschäftslogik“ verwenden wir in diesem Zusammenhang um die Teile eines Informationssystems zu bezeichnen, die die eigentliche Anwendungslogik (das Verhalten) implementieren und von Details der Datenstrukturen, der konkreten Persistenzschicht und der konkreten Realisierung der Nutzeroberfläche abstrahieren. Uns interessieren in diesem Kontext hauptsächlich die frühen Phasen eines modellbasierten Entwicklungsprozesses, in denen das (zum Beispiel als Prozessmodell) vorliegende Analysemodell Schritt für Schritt verfeinert und sukzessiv um technische Aspekte erweitert wird, bis der nötige Detailgrad zur Generierung von Teilen des Programmcodes erreicht wurde.

Dieser Artikel gliedert sich wie folgt: Nach der Diskussion verwandter Arbeiten (Abschnitt 2) erläutern wir, welche Modellierungskonzepte wir für die Modellierung von Geschäftslogik unterstützen und welche UML-Diagramme dafür eingesetzt werden können (Abschnitt 3). Anschließend stellen wir die von uns in [BG08] vorgeschlagene Modellarchitektur vor und gehen hierbei insbesondere auf die Abbildung in gefärbte Petri-Netze ein, die die semantische Grundlage des integrierten Modells bildet (Abschnitt 4). Der Beschreibung einer Werkzeugkette zur automatischen Transformation von UML-Modellen in gefärbte Petri-Netze (Abschnitt 5) folgen ein Überblick über konkrete Ansätze zur automatischen Qualitätssicherung (Abschnitt 6) und die abschließende Zusammenfassung (Abschnitt 7).

2 Verwandte Arbeiten

Die mit unserem Ansatz verwandte Literatur umfasst verschiedene Themengebiete: Integration von verschiedenen Sichten (Diagrammen) in ein Modell, Formalisierung von UML-Modellen oder UML-basierten Modellen sowie automatische Qualitätssicherung von visuellen Softwaremodellen. Im Kontext der Modellintegration von Modellen zum Systemverhalten vergleicht und diskutiert Liang in [Li06] 21 verschiedene Ansätze zur Modellsynthese aus Zustandsdiagrammen und Szenarien, wobei die Kernidee aller Ansätze die direkte Transformation aus Zustandsdiagrammen und Verhaltensdiagrammen darstellt. Da alle diskutierten Ansätze für den Bereich der technischen Steuerungssysteme entwickelt wurden, lassen sie sich nur bedingt auf Informationssysteme anwenden, in

denen verschiedene Abläufe komplexer Geschäftsprozesse durch Nutzereingaben gesteuert werden. Küster stellt zur Modellierung von Informationssystemen in [KR07] einen Ansatz zur Integration der Ablaufsicht in die Zustandssicht vor, um die Konsistenz von Prozessmodellen mit einem definierten Objektlebenszyklus nachzuweisen. Dafür entwickelte er Vorschriften, die aus einer Zustandssicht eine Ablaufsicht generieren. Der von Küster verfolgte Ansatz zur Integration von Zustands- und Ablaufsicht ist unserem Vorhaben ähnlich, allerdings generieren wir keine Ablaufsicht aus Zustandsdiagrammen, sondern integrieren beide (unabhängige) Sichten in ein Modell. Ein weiteres relevantes Feld verwandter Arbeiten liegt im Bereich der Formalisierung von UML-Modellen beziehungsweise der Beschreibung einer formalen Semantik für die UML. Börger beschreibt in [BCR00] die Semantik von Aktivitätsdiagrammen mit abstrakten Zustandsautomaten, Dong in [DS03] mit den Pi-Kalkül und Staines diskutiert in [St08] verschiedene Ansätze zur Formalisierung mit (erweiterten) Petri-Netzen. Diese Arbeiten leisten einen wichtigen Beitrag zur Beschreibung einer eindeutigen Semantik der UML, aber sie beschränken sich nur auf eine Sicht und sind nicht auf ein integriertes Modell mehrerer Sichten übertragbar. Einige akademische Forschungsgruppen, wie das UML 2 Semantics Project [BCR06] schlagen zwar eine formale Fundierung der UML vor, allerdings sind diese Arbeiten noch nicht abgeschlossen beziehungsweise nicht Bestandteil der UML Definition.

Die automatische Qualitätssicherung von UML-Diagrammen, insbesondere die automatische Erkennung von Inkonsistenzen in UML-Modellen, ist ebenfalls ein unserer Arbeit verwandtes Forschungsfeld. Exemplarisch dafür stellt Egyed in [Eg06] eine Methode zur Instantvalidierung von UML-Modellen vor und Van Der Straeten in [Str05] ein Framework zur Definition, Erkennung und Behandlung von Inkonsistenzen in UML-Modellen auf Basis von Beschreibungslogiken (description logics, DL). Diese Ansätze adressieren diagrammübergreifende Inkonsistenzen eines UML-Modells auf einer syntaktischen Ebene: Wird beispielsweise in einem Sequenzdiagramm eine Klasse modelliert, die nicht Element eines Klassendiagramms ist, kann diese Inkonsistenz ermittelt werden. Einen Ansatz zur Erkennung von semantischen Inkonsistenzen zwischen Zustandsmodellen und Prozessmodellen präsentiert Küster in [KR07]. Küster untersucht ein unserer Arbeit ähnliches Problem, jedoch transformieren wir nicht Objektlebenszyklen in eine Ablaufsicht, sondern validieren ein semantisches Metamodell, in das alle relevanten Sichten für Modelle der Geschäftslogik integriert sind.

3 UML-Modelle der Geschäftslogik für Informationssysteme

Der Anwendungsbereich, den wir mit unserem Ansatz adressieren, ist die Erstellung und Wartung von Software-Modellen von Informationssystemen, die bestimmte Geschäfts- oder Verwaltungsprozesse innerhalb einer Organisation oder zwischen mehreren Organisationen unterstützen (e-Government, e-Business). Dabei beschränken wir uns auf Verhaltensmodelle, die die eigentliche Anwendungslogik (Geschäftslogik) spezifizieren.

3.1 Beispielmodell

Ein Beispiel für einen Ausschnitt aus einem Modell für Geschäftslogik wird in Abbildung 1 dargestellt. Das Aktivitätsdiagramm auf der linken Seite enthält den „Hauptprozess“ (Teil eines Angebotsprozesses für Versicherungspolicen), das rechte Diagramm eine 4-Augen Entscheidung, die aus dem Hauptprozess aufgerufen werden kann.

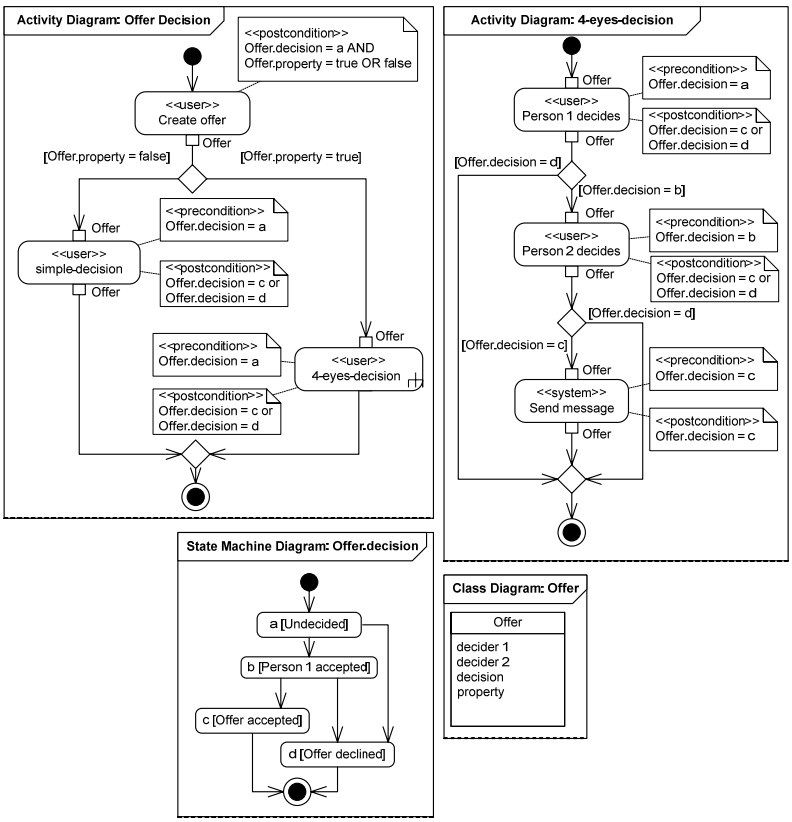


Abbildung 1: Beispielmodell für Geschäftslogik

Das Klassendiagramm spezifiziert das Geschäftsobjekt „Offer“ (Angebot) mit den benötigten Attributen. Das Zustandsdiagramm spezifiziert das Attribut „decision“ des Objekts „Offer“ mit den möglichen Attributwerten und deren Transitionen. Ein (Versicherungs-) Angebot wird von einem Nutzer im System angelegt. Falls nach dem Anlegen eines Angebots (Offer) das Attribut „property“ den Wert „true“ hat, kann es nur nach einer positiven 4-Augen Entscheidung angenommen werden. Das bedeutet, nur wenn beide Entscheider dem Angebot zustimmen, wird es angenommen und eine Nachricht durch

das System versendet. Falls nach dem Anlegen der Wert von „property“ = „false“ ist, kann das Angebot von einer Person bearbeitet werden.

3.2 Ablaufsicht (Aktivitätsdiagramm)

Abläufe lassen sich in Aktivitätsdiagrammen (Activity Diagram, ACD) durch Funktionen (Action) und Funktionsbereiche (Activity) verschachtelt strukturieren. Die Ablaufsteuerung wird dabei durch die Menge der Entscheidungsknoten mit logischen Bedingungen (Guards) beschrieben. Darüber hinaus können zu Actions und Activities sowohl konkrete Vor- und Nachbedingungen (Preconditions, Postconditions) als auch Parameter modelliert werden, wobei zwischen Eingabe- und Ausgabeparametern (Input-, Outputparameter) unterschieden wird. Ein Eingabeparameter bedeutet, dass eine Instanz des jeweiligen Objektes zur Ausführung benötigt wird. Ein Ausgabeparameter bedeutet dagegen, dass die Attribute einer Instanz eines entsprechenden Objektes modifiziert werden.

3.3 Struktursicht (Klassendiagramm)

Wir verwenden im Folgenden den Begriff „Geschäftsobjekt“ (Business Object), um fachliche Klassen zu bezeichnen, deren Existenz bzw. deren Attribute relevant für die Geschäftslogik sind und die in UML Klassendiagrammen modelliert werden. Die Semantik von Geschäftsobjekten entspricht (zumindest aus Sicht der Logik) der von reinen Datenobjekten, die eine Menge von Attributen haben, aber selber keine (wesentliche) Logik implementieren. Objekte werden als Parameter für Elemente des Aktivitätsdiagramms modelliert und Attribute aus diesen Objekten in Guards und Conditions zur Ablaufsteuerung verwendet werden.

3.4 Zustandssicht (Zustandsdiagramm)

Zustandsdiagramme (State Machine Diagram, SMD) verwenden wir zur Beschreibung von fachlichen Zuständen von Objekten, die für die Ablaufsteuerung relevant sind. Dabei setzen wir einen wie folgt beschriebenen Ansatz für die Verwendung der Zustandssicht voraus: Zustände werden als Verfeinerung von Attributen (von Geschäftsobjekten) verwendet, wobei jedes Zustandsdiagramm genau ein Attribut und jeder Zustand einen möglichen Attributwert spezifiziert. Die Transitionen eines SMD entsprechen den möglichen Änderungen von Attributwerten (also den Änderungen von fachlichen Zuständen) während die Anwendung ausgeführt wird. Das bedeutet allerdings nicht, dass sich in einer Aktion nur ein Attribut ändern darf. Im Modell werden zu den fachlich wichtigen Attributen nur weitere Aussagen in Form eines Zustandsdiagramms ergänzt. Zusammenhänge zwischen einzelnen Attributen lassen sich bei Bedarf als Vor- und Nachbedingungen modellieren.

Die Verwendung von Zustandsdiagrammen um konkrete Attributwerte (Zustand = ein konkreter Attributwert eines Attributes eines Objektes) zu spezifizieren und sich dabei auf die für den Kontrollfluss relevanten Zustände zu beschränken, hat im Vergleich zu

Ansätzen, die mit einem Zustandsdiagramm nur den allgemeinen Zustand eines Objektes (Zustand = eine mögliche Kombination von Belegungen aller Attribute eines Objektes) oder zu Ansätzen, die den Zustand eines ganzen Systems beschreiben (Zustand = eine mögliche Kombination von Belegungen aller Attribute jeder möglichen Menge aller Objekte) folgende Vorteile: (1) Zustandsdiagramme werden nicht sehr groß, da sie auf fachliche Zustände beschränkt sind. (2) Nicht jede Änderung eines Attributes in einem Objekt ist „gleich wichtig". So hat die Information, dass eine Versicherungspolice ausgelaufen eine größere fachliche Relevanz als die Information, dass sich die Telefonnummer des Versicherungsnehmers geändert hat. (3) Nicht jede Funktion führt zu einer Änderung des fachlichen Zustands. So kann die Aktion „Telefonnummer ändern" beliebig oft ausgeführt werden, ohne dass sich der fachliche Zustand der Police ändert.

3.5 Semantische Integration verschiedener Sichten

Die semantische Integration verschiedener Sichten auf ein Modell (Zustandssicht, Ablaufsicht, Struktursicht) kann prinzipiell wie in Abbildung 2 dargestellt erfolgen: Entweder durch die Abbildung von Elementen des Metamodells eines Diagrammtyps in Elemente des Metamodells eines anderen Diagrammtyps (links) oder durch die Abbildung von Elementen des Metamodells eines Diagramms auf ein „semantisches" Metamodell, welches allen Sichten zu Grunde liegt (rechts).

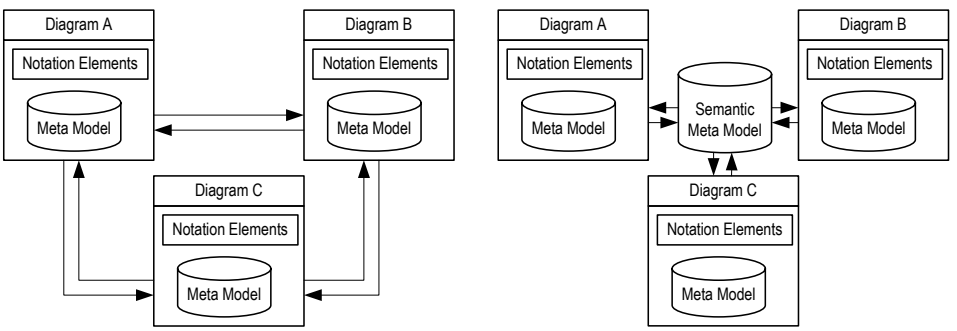


Abbildung 2: Schematische Integration verschiedener Modellierungskonzepte

Dass kein zusätzlicher Aufwand hinsichtlich eines separaten Metamodells besteht, um verschiedene Sichten zu integrieren, ist ein Vorteil der linken Variante gegenüber der Rechten. Ein Nachteil der Variante ohne separates Metamodell ist die schnell zunehmende Anzahl der benötigten Abbildungen zwischen den Modellen bei steigender Anzahl der zu integrierenden Sichten. So müssen für die Integration jeder neuen Sicht *m* genau $2 \cdot (m-1)$ weitere Abbildungen definiert werden.

Ein weiterer Nachteil, insbesondere für einen modellbasierten Entwicklungsprozess besteht darin, dass (unter der Voraussetzung, dass die Verschiedenheit der Sichten nicht nur durch visuelle Transformation beschrieben werden kann, sondern auch durch unterschiedliche logische Information gekennzeichnet ist) die Gesamtheit der im Modell vorhandenen Information nur über alle Sichten „verstreut" vorliegt. Es gibt im Klassen-

diagramm keine Information über Abläufe und im Aktivitätsdiagramm keine Information zu Details der Datenstruktur. Sowohl im Hinblick auf eine automatische Qualitätssicherung als auch auf eine Generierung von Programmcode bietet dagegen ein gemeinsames semantisches Metamodell, das die logische Information aus allen Sichten enthält und dabei von Diagramm-typischen Besonderheiten abstrahiert, eine relativ einfache Schnittstelle. Dass für jede zusätzlich zu integrierende Sicht nur zwei weitere Abbildungen definiert werden müssen, ist ein weiterer Vorteil der Variante mit semantischem Metamodell.

4 AMABULO – Eine Modellarchitektur für Geschäftslogik

Eine Voraussetzung für die automatische Qualitätssicherung von visuellen Modellen der Softwaretechnik ist eine eindeutige Semantik der Modellelemente, die durch die Verwendung von UML allerdings nicht erfüllt ist. Fragen die bei der Verwendung von UML beantwortet werden müssen, sind: Wie können verschiedene Sichten (Ablauf, Struktur, Zustand) in ein Modell semantisch integriert werden? Wie kann man erreichen, dass die Semantik der Notationselemente eindeutig (formal) definiert ist, so dass auf dem Modell „automatisch gerechnet“ werden kann? Eine mögliche Antwort auf diese Fragen gibt die Modellarchitektur für Geschäftslogik *AMABULO* (A Model Architecture for Business Logic).

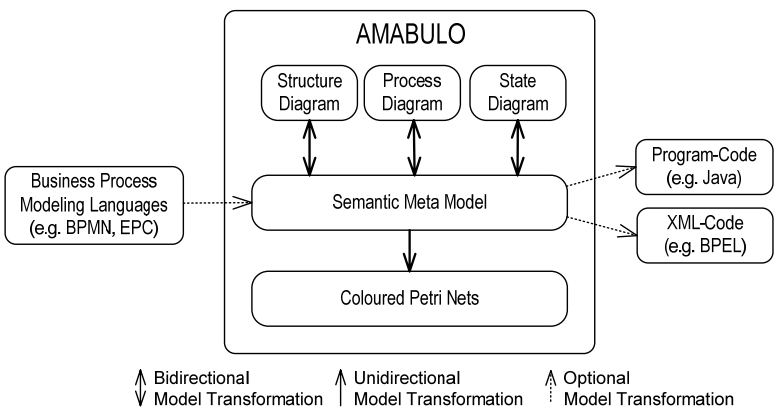


Abbildung 3: AMABULO, Übersicht

Der Begriff „Modellarchitektur“ bezeichnet dabei keine Architektur im Sinne von „Software-Architektur“, sondern eine Menge von Modellen und Modelltransformationen, die in ihrem Zusammenspiel eine Infrastruktur für die Modellierung von Geschäftslogik bilden. AMABULO adressiert insbesondere die frühen Phasen eines modellbasierten Entwicklungsprozesses, in denen das (zum Beispiel als Prozessmodell) vorliegende Analysemodell Schritt für Schritt verfeinert und sukzessiv um technische Aspekte erweitert wird, bis der nötige Detailgrad zur Generierung von Teilen des Programmcodes erreicht wurde. Die von uns in [BG08] vorgeschlagene und in Abbildung 3 dargestellte Modellarchitektur enthält ein zentrales Metamodell für Geschäftslogik. Die semantische

Integration der verschiedenen Sichten erfolgt durch Definition von bidirektionalen Modelltransformationen auf das Metamodell, wobei wir zurzeit die Integration der folgenden UML-Diagramme unterstützen: Zustandsdiagramm, Aktivitätsdiagramm und Klassendiagramm. Dabei beschränken wir uns vorerst auf einfache Elemente, vergleichbar mit denen im Beispielmodell in Abschnitt 3.1, und verzichten auf Elemente die nur der Vereinfachung des Modells dienen („syntaktischer Zucker"). Die detaillierte Liste der unterstützten UML-Elemente, die bei der Modellierung einzuhaltenden Konventionen, sowie eine detaillierte Beschreibung der Modelltransformation und deren Implementierung unter Verwendung der Atlas Transformation Language (ATL) [BVJ08] ist in [No08] beschrieben.

4.1 Semantische Grundlage

Neben der Integration verschiedener Sichten in ein Modell, als Voraussetzung für die Unterstützung des Modellierungsprozesses durch automatische Qualitätssicherung, wird ebenso eine formale Semantik für das Modell benötigt. Dabei ist nicht nur entscheidend, *dass* die Elemente des Metamodells eine eindeutige Semantik tragen, sondern auch *wie* eine eindeutige Semantik beschrieben werden kann, um eine automatische Qualitätssicherung zu ermöglichen. Die Semantik des Metamodells als Abbildung der Elemente und deren Relationen auf gefärbte Petri-Netze (Coloured Petri Nets, CPN) berücksichtigt diesen Aspekt.

Der Einsatz von Petri-Netzen erlaubt die Beschreibung von Systemverhalten in einer definierten Struktur (Stellen und Transitionen, die durch gerichtete Kanten zu einem bipartiten Graphen verbunden werden), deren Elemente durch ein eindeutig definiertes Verhalten charakterisiert sind. Somit kann das Verhalten eines Petri-Netzes (und damit auch das Verhalten des Systems, das durch ein Petri-Netz beschrieben ist) berechnet werden. Darüber hinaus existiert eine Reihe von Programmen, mit denen das Verhalten von Petri-Netzen simuliert werden kann. Nach Jensen in [Je07] sind gefärbte Petri-Netze eine Erweiterung der Petri-Netze um gefärbte Marken (Token). Folglich kann es in einem CPN verschiedene Typen von Token geben, die sich auf der Instanzebene durch einen eigenen inneren Zustand von einander unterscheiden lassen.

Eine detaillierte formale Beschreibung der Abbildung der Elemente des AMABULO Metamodells in Elemente eines gefärbten Petri-Netzes ist auf der Webseite des Projektes (www.amabulo.com) verfügbar und sollen an dieser Stelle nur kurz zusammengefasst werden: Elemente der Zustandssicht und der Geschäftsobjekte werden auf gefärbte Token (Colours) abgebildet. Elemente der Ablaufsicht werden auf Paare von Stellen und Transitionen eines Netzes abgebildet, wobei Vor- und Nachbedingungen als Funktionen von Transitionen umgesetzt werden. Die Ablaufsteuerung wird durch einen bestimmten Kontrollfluss-Token realisiert, wobei modellierte Guards zur Steuerung des Kontrollflusses in Funktionen von Kanten des CPN resultieren.

Falls innerhalb einer Aktion ein Geschäftsobjekt modifiziert wird und mindestens ein Attribut des Geschäftsobjektes durch ein Zustandsdiagramm verfeinert ist, so wird das resultierende CPN um ein Netzteil erweitert. Dieses erweiternde Netzteil prüft dabei den

Zustand aller Attribute vor dem Ausführen der Transition mit dem Zustand der Attribute nach dem Ausführen. Dabei wird sichergestellt, dass nur solche Zustandsübergänge von Actions im ACD erfolgreich ausgeführt werden, die Teil des Zustandsdiagramms sind. Abbildung 4 enthält einen Ausschnitt eines gefärbten Petri-Netzes, was aus dem Beispielmodell in Abschnitt 3.1 unter der Anwendung der Abbildungsvorschriften generiert wurde.

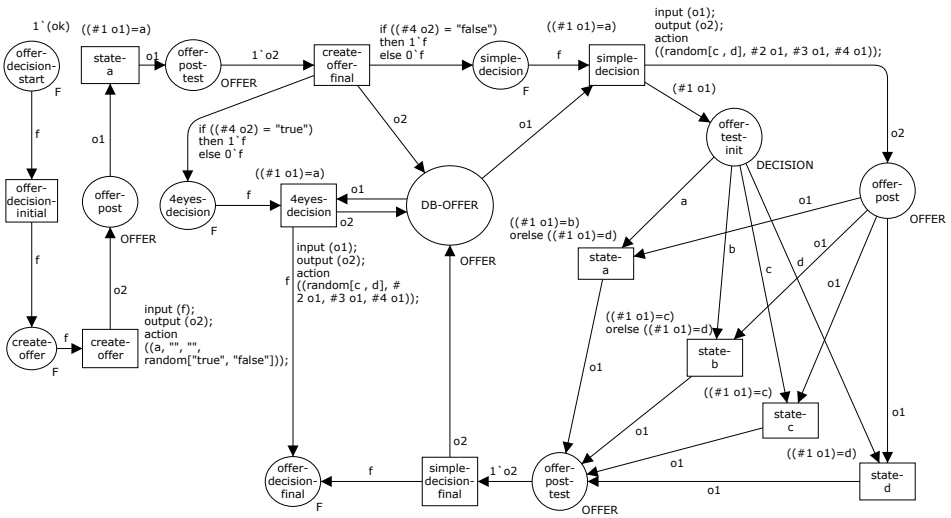


Abbildung 4: Beispielmodell als CPN, Ausschnitt

5 Automatische Generierung von CPN aus UML-Modellen

Gegenwärtig sind wir in der Lage, alle notwendigen Schritte, von der Modellierung in einem Modellierungstool (MagicDraw, EnterpriseArchitect) über die verschiedenen Transformationen bis hin zur Erzeugung von CPNs automatisch auszuführen. Abbildung 5 zeigt schematisch die erforderliche Werkzeugkette und deren Zwischenergebnisse: Nachdem aus einem UML-Werkzeug das Modell in das Modellaustauschformat XMI exportiert wurde, müssen mit einem speziellen Konverter Tool-spezifische Besonderheiten aus dem XMI entfernt und eine XMI-Datei erzeugt werden, die dem XMI-Format des Eclipse-Modeling-Frameworks (EMF) entspricht. Die eigentliche Transformation von UML nach AMABULO wird durch den UML→AMABULO Transformator ausgeführt und resultiert in einer XMI-Datei, die das integrierte Modell enthält. Der AMABULO→CPN Transformator generiert aus diesem Format eine XML-Datei, die dem Austauschformat des Programms CPNTools [Cp08] entspricht. Das auf diese Weise generierte gefärbte Petri-Netz kann dann in das Analyse und Simulationswerkzeug CPNTools geladen und analysiert beziehungsweise simuliert werden.

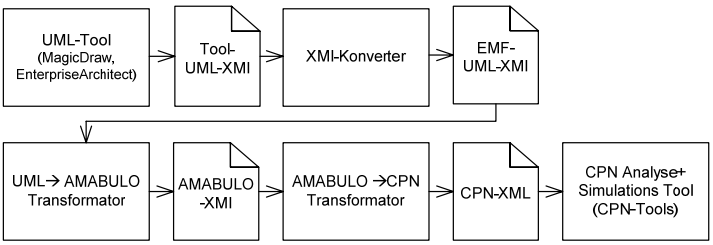


Abbildung 5: Werkzeugkette zur Generierung von CPN

6 Grundlagen für eine automatische Qualitätssicherung

Obwohl die meisten Modellierungswerkzeuge dem Modellierer zunehmend Hilfestellung hinsichtlich der Vermeidung von syntaktischen Fehlern bieten, ist die automatische Erkennung von Modellierungsfehlern immer noch ein aktueller Forschungsgegenstand. Wir fokussieren mit unserem Ansatz semantische Fehler innerhalb eines Modells, die mit Hilfe der statischen und dynamischen Analyse von CPN gefunden werden können. Anhand von drei Beispielfragen wird in diesem Abschnitt kurz erläutert welche Fehlerkategorien wir dabei genauer untersuchen.

(1) *Hängen alle Aktivitätsdiagramme eines Prozesses logisch zusammen?* In großen Modellen mit vielen verschiedenen Diagrammen lässt sich nicht immer überblicken, ob alle Diagramme der Ablaufsicht zusammenhängen. Falls für das generierte CPN mehr als eine Zusammenhangskomponente berechnet wird, ist sicher, dass nicht alle Sichten Teil eines Prozesses sind.

(2) *Können alle modellierten Funktionen und Prozesse ausgeführt werden?* Durch die Verwendung von Constraints und Guards in Aktivitätsdiagrammen sowie durch den Einsatz von Zustandsdiagrammen für Attribute hat der Modellierer die Möglichkeit eine umfangreiche und in ihrem Zusammenspiel sehr komplexe Kombination von Bedingungen zu erstellen. Dies stellt insbesondere in großen Modellen eine häufige Fehlerquelle dar. Falls Teile der modellierten Geschäftslogik aufgrund der Kombination von verschiedenen Bedingungen niemals ausgeführt werden können, führt dies im CPN zu Transitionen, die unter keiner Markierung aktiviert werden können („tote Transitionen“). Die Möglichkeit die Menge der toten Transitionen eines gefärbten Petri-Netzes automatisch zu berechnen (eine entsprechende Funktion bietet CPNTools an), bietet dem Modellierer eine Hilfestellung bei der Suche nach möglichen Modellierungsfehlern.

(3) *Werden alle Funktionen in der richtigen Reihenfolge abgearbeitet?* Während des Modellierens von komplexen Modellen muss der Modellierer prüfen, ob die Funktionen und Prozesse in der richtigen Reihenfolge ausgeführt werden, was wiederum vom aktuellen Zustand der Geschäftsobjekte abhängig ist. Durch schrittweises Simulieren des gefärbten Petri-Netzes kann der Modellierer nachvollziehen, welche Bereiche des Modells gerade aktiv sind. Weiterhin können durch das Einsetzen von Monitor-Stellen im CPN die Abarbeitungsreihenfolge von Transitionen und die Zustände von entsprechen-

den Geschäftsobjekten beobachtet und protokolliert werden, wodurch die automatische Durchführung und Auswertung von mehreren Simulationsläufen hintereinander ermöglicht wird. Die oben beispielhaft aufgeführten Fragen muss der Modellierer während der Modellierung unter dem Aspekt der Qualitätssicherung eines Modells für Geschäftslogik beantworten. Insbesondere bei der Arbeit an großen Modellen ist er dabei auf eine entsprechende Werkzeugunterstützung angewiesen. Eine systematische Klassifizierung weiterer Fragen, die detaillierte Beschreibung der entsprechenden Analyse- und Simulationsverfahren sowie deren Umsetzung sind Gegenstand unserer aktuellen Arbeit.

7 Zusammenfassung

Dieser Artikel erläuterte, welche Modellierungskonzepte für die Modellierung von UML-Modellen der Geschäftslogik für Informationssysteme im Rahmen der von uns vorgestellten Modellarchitektur AMABULO unterstützt werden. Der dabei adressierte Anwendungsbereich war die Erstellung und Wartung von Software-Modellen von Informationssystemen, die bestimmte Geschäfts- oder Verwaltungsprozesse innerhalb einer Organisation oder zwischen mehreren Organisationen unterstützen (e-Government, e-Business). Insbesondere fokussierten wir die frühen Phasen eines modellbasierten Entwicklungsprozesses, in denen das (zum Beispiel als Prozessmodell) vorliegende Analysemodell Schritt für Schritt verfeinert und sukzessiv um technische Aspekte erweitert wird, bis der nötige Detailgrad zur Generierung von Teilen des Programmcodes erreicht ist.

Die Verwendung der entsprechenden UML Diagramme wurde dabei anhand eines Beispielmodells beschrieben. Die Integration der verschiedenen Sichten (Ablauf-, Zustands- und Strukturdiagramm) erfolgte über ein semantisches Metamodell, das von Diagrammtypischen Besonderheiten abstrahiert und nur logische Informationen enthält. Zusammen mit der Beschreibung einer Abbildung auf gefärbte Petri-Netze wurde die Grundlage für eine automatische Qualitätssicherung geliefert. Dadurch konnten wir die logische Information aus mehreren Sichten eines UML-Modells in eine formal definierte Struktur integrieren. Gegenwärtig sind wir in der Lage Modelle, die in einem Modellierungstool (MagicDraw, EnterpriseArchitect) erstellt wurden automatisch in CPN zu überführen. Das so erzeugte gefärbte Petri-Netz kann daraufhin von dem Analyse und Simulationstool CPNTools verarbeitet werden. Weiterhin wurde beschrieben, welche Art von Fehler wir mit Analysemitteln für gefärbte Petri-Netze fokussieren, wobei erste Experimente mit einem Analyse- und Simulationstool für CPNs die prinzipielle Durchführbarkeit einer automatischen Qualitätssicherung bestätigen.

Eine systematische Klassifizierung der zu erkennenden Modellierungsfehler, die detaillierte Beschreibung der entsprechenden Analyse- und Simulationsverfahren sowie deren Umsetzung als auch die Untersuchung von geeigneten Möglichkeiten des Feedbacks an den Modellierer sind Gegenstand unserer zukünftigen Arbeiten.

8 Literaturverzeichnis

[AH02] Aalst, W.v.d./ Hofstede, A.: YAWL: Yet Another Workflow Language. in: QUT Technical report. 2002, Queensland University of Technology.

[BCR00] Börger, E./Cavarra, A./Riccobene, E.: An ASM Semantics for UML Activity Diagrams. In: 8th International Conference on Algebraic Methodology and Software Technology. 2000: Springer.

[BVJ08] Bézivin, J./Valduriez, P./Jouault, F.: ATL (ATLAS Transformation Language). 2006 [cited 13 March 2008]; Available from: http://www.eclipse.org/m2m/atl

[BG08] Brückmann, T./Gruhn, V.: AMABULO - A Model Architecture for Business Logic. in: 15th IEEE International Conference and Workshops of the Engineering of Computer-Based Systems. 2008. Belfast, Northern Ireland: IEEE Computer Society.

[BCR06] Broy, M., et al. 2nd UML 2 Semantics Symposium: Formal Semantics for UML. in: Models 2006 Workshops. 2006: Springer.

[Cp08] CPN Group, U.o.A., Denmark. CPNTOOLS - Computer Tool for Coloured Petri Nets. 2008 [cited 2008 10th January]; Available from: http://wiki.daimi.au.dk/cpntools.

[DS03] Dong, Y./ShenSheng, Z.: Using π - calculus to Formalize UML Activity Diagram. in: 10th IEEE International Conference and Workshop on the Engineering of Computer-Based Systems (ECBS'03). 2003. Huntsville, AL, USA: IEEE Computer Society.

[Eg06] Egyed, A.: Instant Consistency Checking for the UML. in: 20th International Conference on Software Engineering (ICSE). 2006. Shanghai, China: ACM New York, USA.

[FR07] France, R./Rumpe, B.: Model-driven Development of Complex Software: A Research Roadmap. in: Future of Software Engineering. 2007: IEEE Computer Society.

[Je97] Jensen, K.: Coloured Petri Nets: Basic Concepts, Analysis Methods, and Practical Use. 1997: Springer. p. 234.

[KNS92] Keller, G./Nüttgens, M./Scheer, A.-W.: Semantische Prozessmodellierung auf der Grundlage "Ereignisgesteuerter Prozessketten (EPK)". Veröffentlichungen des Institutes für Wirtschaftsinformatik. Universität des Saarlandes. 1992. p. 2-30.

[KR07] Küster, J. M./Ryndina, K./Gall, H.: Generation of Business Process Models for Object Life Cycle Compliance. in: 5th International Conference on Business Process Management (BPM). 2007. Brisbane, Australia: Springer.

[Li06] Liang, H./Dingel, J./Diskin, Z.:. A comparative survey of scenario-based to state-based model synthesis approaches. in 5th Intl.Workshop on Scenarios and State Machines (SCESM). 2006. Shanghai, China: ACM Press.

[No08] Nowak, F.: Semantische Integration verschiedener UML-Diagramme zur Unterstützung der modellzentrierten Software-Entwicklung, Diplomarbeit. 2008. Leipzig: University of Leipzig.

[Om06] OMG. Business Process Modeling Notation 1.0 (BPMN). 2006 [cited 10 October 2007]; Available from: http://www.bpmn.org.

[St08] Staines, T.S. Intuitive Mapping of UML 2 Activity Diagrams into Fundamental Modeling Concept Petri Net Diagrams and Colored Petri Nets. in 15th IEEE International Conference and Workshop on the Engineering of Computer Based Systems (ecbs). 2008. Belfast, Northern Ireland: IEEE Computer Society.

[WL06] Waddington, D./Lardieri, P.: Model-Centric Software Development. IEEE Computer. 2006. 39(2): p. 28-29.

[Str05] Straeten, R.V.D., Inconsistency Management in Model-Driven Engineering, Dissertation. in: Department of Computer Science, System and Software Engineering Lab. 2005. Vrije University Brussel: Brussel. p. 293.

Tracing Quality-Related Design Decisions in a Category-Driven Software Architecture

Stephan Bode, Matthias Riebisch

TU Ilmenau, Faculty of Computer Science and Automation
P.O. Box 100565, 98684 Ilmenau, Germany
{stephan.bode|matthias.riebisch}@tu-ilmenau.de

Abstract: Quality properties, so-called non-functional ones, have a fundamental influence on the development of software systems because they constitute the decisive factors for the design of a system's software architecture. They earn a similar consideration like functional properties. For a high evolvability of the software systems, traceability supports changes by facilitating design decisions, software comprehension and coverage checks. In this paper a method for design traceability is presented, in which links both for functional and quality properties are established in similar ways. A software category based design method is used for a better alignment between requirements and design. As a consequence, the method leads to a reduced number of traceability links. The method has been successfully applied in the development and partial reengineering of an e-commerce system.

1 Introduction

Performance, scalability, flexibility, usability and other so-called non-functional or quality properties are crucial for the success of nearly every software project. They bear even more risk than functional requirements, because they can hardly be implemented after finishing code development. These quality properties represent the determining factors for the software architecture. Therefore, they have a basic influence on the development of software systems. Since they are hard to master and have a high impact on many parts of a software system, several architectural design methods emphasize their analysis [HNS00] and design [Bo00]. However, bridging the gap between requirements analysis and architectural design is still a critical issue, especially since quality requirements change sometimes.

The architectural design methodology Quasar [Si03, Si04] constitutes a successor of advanced design methods. It applies the principle separation of concerns to master the complexity of the design task and to achieve a high flexibility by modularization according to so-called software categories. Moreover, the software categories help to structure the requirements, and thus to bridge the gap between analysis and architectural design [Bo08, We07]. The methodology does not specifically support the implementation of quality features. We extend the methodology in this regard, to achieve both: the high architectural quality in terms of modularity and flexibility, and the satisfaction of the quality requirements balanced with the functional ones.

Traceability is the ability to describe and follow implementation activities. It supports design decisions in many ways. Traceability links facilitate the elaboration and refinement of requirements; they relate design decisions to constraints for rationales' determination, they support change impact analysis and effort estimation. Furthermore, traceability can be utilized for example for the verification of forward engineering activities by checking the completeness of changes, and for tracing the dependencies for program comprehension. The tracing of quality properties however, requires a very high number of links, since typically each quality feature influences a high portion of a system's components. The high number leads to a high human effort for establishing, maintaining, and validating the traceability links. Our design method can contribute to a reduction of the number of links.

In this paper we show a method for the architectural design that explicitly supports quality features. We chose Quasar as the basic method and adapt and extend its architectural design activities. To manage the implementation of the non-functional requirements we have introduced a so-called Goal Solution Scheme as a part of our method. Furthermore, we present a traceability framework customized to it. We show the benefits from the combination of these in terms of a highly detailed description of the design activities, an improved modularization due to the use of software categories, and a reduced number of traceability links. In this way we provide preconditions for an extended tool support in design and traceability management.

This research work has been performed for the quality property usability; however the method can be transformed to other quality properties as well. It has been validated in a reengineering project for an e-commerce system, where it was applied for the design of the upper layers including the user interface. Herpel [He07] performed the reengineering of the business logic layer, while the GUI components were developed by Bode [Bo08], together with an evaluation of the method. We will use a cut-out of the e-commerce trading system as a case study for illustrating our method of quality-oriented design and of establishing traceability links. The system named *Vendorbase* is of a middle size and manages vendors, contracts with them including discounts, controlling, and calculation.

2 State of the Art

2.1 Traceability from Requirements to Design and Between Design Artifacts

Requirements traceability enables to trace back the origin of a requirement from its elicitation and to document every change made to it. Important works in this field are the ones of Gotel and Finkelstein [GF94], Ramesh and Jarke [RJ01], Letelier [Le02] as well as Pinheiro [Pi04]. Other approaches for traceability link establishment consider links between requirements and test cases, for example the scenario-driven approach by Egyed [Eg01] and the approach by Olsson and Grundy [OG02].

For our work, approaches regarding traceability from requirements to design and between design artifacts have to be considered, because our method comes to play after the requirements have been specified. Traceability links for design shall be established and adapted while design activities are performed, for example during the building or

manipulation of models. Therefore, the link establishment steps have to be embedded into the steps of design methods. We have to state that—to our best knowledge—there are no approaches of this kind. For establishing traceability links regarding non-functional requirements and design artifacts there is the probability-based retrieval approach by Clelang-Huang et al. [Cl05] called *Goal-Centric Traceability*. Their user evaluation step to discard incorrectly retrieved links to increase precision is valuable. However, they can only identify links that are incidentally included in the descriptions of artifacts that were elaborated regardless of traceability. Therefore, the completeness and correctness of the links can never be optimal. The incremental approach of *Latent Semantic Indexing* by Jiang et al. [Ji07] aims at the identification of related elements with link recovery. It can be helpful for finding links in existing designs and maintaining their change, but it cannot provide all links. For updating the traceability links after changes, the change propagation approach event-based traceability [CCC03] provides a valuable solution; it will be integrated with our approach.

For the definition of relations between design activities and traceability links, syntactical and semantic definitions of the traceability links are necessary. Unfortunately, the definition of a standard set of traceability link types is still an unresolved issue. Due to different research goals, a high number of traceability link type definitions have been established, for example in [Po96] or [RJ01]. As a step towards simplification and abstraction, we will restrict ourselves on a small set of types later.

Tool support for traceability is an important issue due to the high effort of establishing and maintaining traceability links during the whole software life cycle. There is already support for traceability by requirements management tools, e.g. Requisite Pro. However, their support for linking other artifacts than requirements is limited. Mäder et al. [MGP08] present an approach that tackles the problem of automated traceability for UML-based development. Their *traceMaintainer* is a rule-based prototype tool for traceability link establishment. Nonetheless, there is future work because tool support should not be limited to models specified with the UML. The definition of proper rules for an automated link establishment is an important issue we want to prepare by describing the several development steps of our method in detail.

2.2 Quality-Related Software-Architectural Design

For establishing and maintaining traceability links between design artifacts we are seeking for a design method with (1) a clear definition of relations between design artifacts and model elements and (2) a precisely defined, fine-grained sequence of design activities. Furthermore the ideal method provides (3) a structured way of implementing the non-functional requirements in a non-scattered way. By selection of a method with such a way of implementation we reduce the number of traceability links. We want to extend a design method which is accepted in practice (4) because our work aims at practical application.

For the consideration of non-functional requirements in software architecture development a frequently performed way of work is described by Bosch's QASAR method [Bo00]. As the third of three design phases, appropriate functional solutions are

established for the implementation of as many as possible non-functional requirements. This activity leads to a reduced number of relation between these non-functional requirements and design artifacts and thus, to a reduced number of traceability links. We will integrate this core concept into our approach because it fulfils our criterion (3). Afterwards all remaining non-functional requirements are implemented by changing all affected components, thus in a scattered way – which is leading to a high number of traceability links for these remaining requirements. This would hamper maintainability, therefore, we want to address this is issue by a special consideration of non-functional properties according to Figure 3.

A structured and explicit refinement of non-functional properties provides additional potential for reducing the number of traceability links. Software quality criteria are helpful for aligning design decisions with these requirements. Standards as for example ISO 9126 define a refinement to subcharacteristics for product quality. Quality models and other works that describe subcharacteristics of certain quality goals (e.g. [Ni93], [Sh92]) can be utilized for refining these goals to subgoals. In our method we use them for refining goals in Figure 3.

A suitable approach for the refinement of the property usability is provided by the so-called usability framework of Folmer and Bosch [FB03]. First, usability is decomposed to attributes, e.g. learnability or satisfaction. Then, solutions in form of design principles heuristics and patterns are related to them. This work is a valuable approach, however, it has to be extended to other non-functional properties. Furthermore, a traceability concept is needed for an explicit mapping.

The Non-Functional Requirements (NFR) framework by Chung et al. [Ch00] represents another approach considering with implementation of non-functional properties. The non-functional requirements – so-called softgoals – and their interdependencies are represented by a Softgoal Interdependency Graph. The approach describes the activities to build such a graph. First, the softgoals are established and decomposed to subgoals. Then, sets of architectural solutions are developed as so-called operationalizations for the softgoals. Beyond, the softgoals are refined according to domain characteristics and the developers' expertise. Furthermore, the priority of different goals is evaluated. Finally, a selection of a solution is made, and its contribution to the non-functional requirements is evaluated. In this way the NFR framework helps to build software architectures that explicitly consider non-functional properties. However, Chung et al. concentrate on certain categories of non-functional requirements, e.g. accuracy, security, and performance requirements. Furthermore, they do not consider the application of general design principles for decomposition and refinement. Moreover, the authors do not provide a traceability model.

Many widely accepted (4) design methods e.g. the Unified Process and Fusion fail to provide a seamless sequence of fine-grained activities (2) for architectural design. A much finer granularity in terms of activities and relations (1) is provided by the design method Quasar — an acronym for QUAlity Software Architecture — which extends the methods for component-based systems design. It achieves an improved decomposition of the software into components by distinguishing so-called software categories. By this

way of separation according to the responsibilities and to the knowledge covered, dependencies between components are minimized [Si03, Si04]. Quasar defines a sequence of design steps and uses the categories to structure components according to the requirements. These steps help to bridge the gap between analysis and architectural design in a more precise way than other design methods [Bo08, We07]. Furthermore, the steps enable a more precise definition of rules for the establishment and the maintenance of traceability links (2). Quasar was developed in industrial practice (4) and combines Best Practices and principles for a good component-based software architectural design.

3 Embedding Traceability to the Architectural Design Process

For a discussion of the traceability concept two aspects are investigated in this section. First, the design activities for a software system are explained for the domain functionality of the application kernel. Second, for each activity the corresponding traceability link types have to be defined, because later these types control the tracking and the utilization of the traceability links.

3.1 Traceability Link Types

As introduced by earlier works [MPR07], we distinguish 4 different link types.
- *refine* – for an activity increasing the level of detail, either by decomposition or by specialization.
- *realize* – represents a step towards the solution (e.g. between a usability goal and a design principle)
- *verify* – compares the behavior and the properties of requirements and of the developed solution or its parts (for example between a use case and a test case) and
- *define* – relates the establishment of an identifier and its usage.

The link type *implemented by* corresponds to realize but with an opposite direction. Furthermore, a distinction between implicit and explicit links is necessary. Implicit traceability represents existing associations between elements of the system model using identifiers, for example between an analysis and a design artifact. These traceability links are references, but they are evaluated, if traceability links are tracked during their utilization. Explicit traceability links are established, while a developer performs a software development activity. Additional information can be stored attached to the link, e.g. design decisions. An explicit traceability link consists of:
- a unique identifier for its recognition and to avoid ambiguity,
- a start element as source of the link, including type and context of this element,
- an end element as destination of the link, including type and context,
- the type of the link.

The link can contain additional information:
- a reference to a design rule for this specific activity,
- the decision connected with the development activity, including the goal of the decision, alternatives, rating of the alternatives and the choice,
- the link status concerning the certainty of correctness (e.g. after changes of the connected elements or during reverse engineering activities),

- the creator of the link,
- a temporary priority to control the tracking of the links.

3.2 Design steps for the application kernel

This section shows the several design steps according to our extended Quasar method and explains the establishment of traceability links.

Step 1 Mapping use cases to functional features, with assignment to architectural layers
First, the use case model with a textural description of the functional requirements of the system [Co00] is analyzed for relations to functional features to establish a mapping. In the simplest case one use case is directly connected to one feature. Possibly one use case has to be mapped to more than one functional feature, or more than one use case to one functional feature. The fact, that every use case and every feature has to be assigned at least once, is expressed by a rule [Sch08] and can be checked by a tool. Furthermore, the include and extends relationships between use cases have to be corresponded by feature relations, which are assessable by tools as well. While establishing the function tree, the functional features are structured according to their responsibilities to the intended architectural layers, e.g. of a three-tier architecture. A case study example for a function tree can be seen in the upper part of Figure 1 with the two layers *Domain Functionality* and *Interface Functionality*. In this step traceability links of the type *refine* have to be established, both for each mapping between use cases and functional features and for each assignment to an architectural layer.

Step 2 Grouping the functional features
In the following the functional features are examined for commonalities to build abstractions. For structuring and refining functional features a function tree is introduced as proposed by Herpel [He07] The elaboration of software categories and components from a function tree or a feature model [Ka90] is much easier than directly from the requirements. Hierarchy relations within the tree are traceability links of the type *refine*. This step is an extension to the Quasar methodology for bridging the gap between analysis and architectural design. By structuring the system in a functional way, candidates for later components can be determined and the identification of the categories is supported. The previously existing traceability links have to be maintained according to the changes in the function tree. In our case study the feature *Vendor Management* is introduced for grouping three sub-features, and it is assigned to the *Domain Functionality* layer (see Figure 1).

At this point we give a brief introduction to the QUASAR categories. All components have to be based on the standard categories A, T, 0, and R. For a concrete application design, these categories are therefore refined in a category model. A-components are application specific but independent of technical issues. They contain the application logic and entity classes for the realization of the domain functionality. T-components cover technical knowledge about a system, and they frequently provide an application programming interface (API) for example for database connectivity or for the GUI

elements. They are independent of concrete application functions. Software of the 0-category is neutral concerning the application's functionality and independent of technical aspects and it creates no undesired dependencies. Modules, classes and interfaces with a high degree of reusability belong to 0-software, e.g. class libraries. R-software refers to representation; it establishes a connection between A- and T-components, however, minimizing the dependencies between them. This is achieved by transformation, for example to external data presentation formats like XML. Other ways of directly connecting or even mixing A and T – the so-called AT-software – are prohibited, because they would re-introduce stronger dependencies.

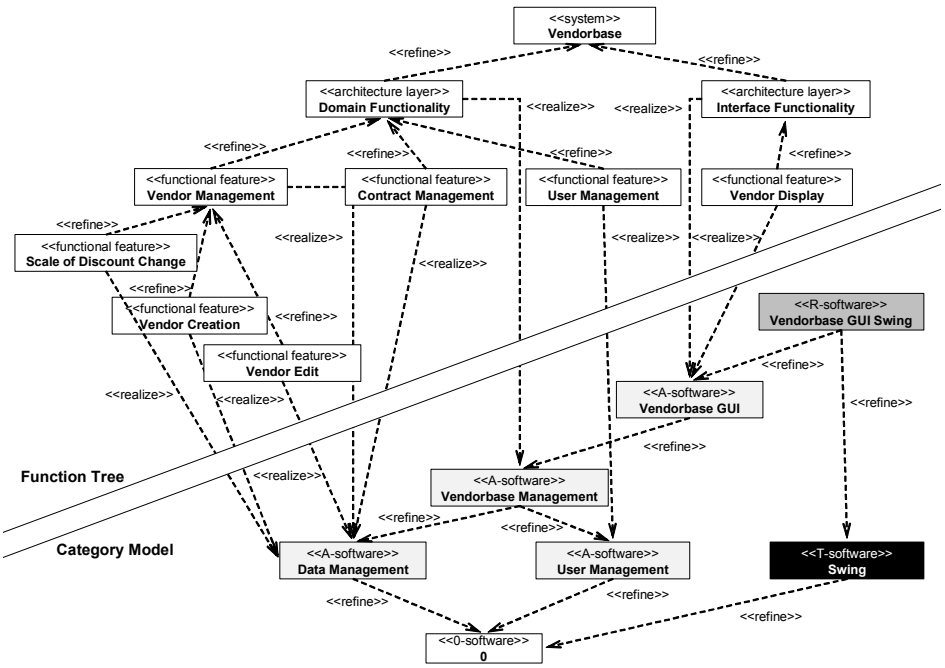


Figure 1. Tracing the identification of the categories between function tree and category model.

Step 3 Identification of categories

The input for the identification of the categories is the function tree refined to architectural layers and functional features. In Figure 1 the two functional features are related to this aspect: the *Vendor Management* and *Vendor Display* are shown with their relation within the function tree (upper part) and to the category model [Si04, Ad07] (lower part). Layers are related via a *realize* link to upper level categories, for example *Vendorbase Management* as a super category for all A parts of the application and *Vendorbase GUI* for all functional features for visualization. Grouped features are candidates for A categories, and they are related via a *realize* traceability link as well. *Vendor Display* represents an example. Functional features become lower level categories only in rare cases.

Generally, the T categories are derived from technical influence factors [HNS00]. In our case, Java Swing is predefined as the GUI framework. Therefore, *Swing* is introduced as a T category. Because a direct coupling of A and T has to be avoided according to Quasar, *Vendorbase GUI Swing* is introduced as an R category accessing the Swing API for visualizing the *Vendorbase GUI* elements. All decisions made, while identifying the categories, are traced with links from the function tree to the category model as shown in Figure 1. Again a tool can support the designer by checking the assignment of features to categories. The refinement relations among features and among categories have be consistent; a tool can determine conflicts or even suggest proper categories for functional features. Furthermore, cyclic dependencies in the category graph can be detected.

Step 4 Identification of application kernel components as domain abstractions
The identification of the application kernel components is guided by the rule that there must be one managing component per each entity class in the data model [Ad07]. Figure 2 shows *VendorManager* for the entity class *Vendor*. Additionally, the functional features from the function tree are assigned to components, so that each functional feature with domain functionality is covered by a component. In the upper part of Figure 2 the functional feature *Contract Management* is assigned to the corresponding component as a responsibility. For these decisions, traceability links of the type *realize* are established for connecting goal and solution. By assessing the established links from a component to categories and features back to the use cases, a tool can be help to analyze the impact of changes of particular requirements for the component.

Step 5 Definition of the interfaces of the application kernel
The provided and required interfaces of the components are specified following the design principle design by contract. According to the architectural style mentioned in section 2, there are two types of provided interfaces:
(1) the so-called programming interfaces according to the functional features as assigned in step 4 (*IVendorManager* for calling the functionality)
(2) interfaces for each entity type to provide getter and setter methods (*IVendor*)

The case study situation is presented in the lower part of Figure 2. *Vendor Manager* and *Contract Manager* both are derived from entity classes; therefore, Quasar demands two provided interfaces for them.

The required interface is determined by two sources:
(1) relations to other application kernel components which result from associations between entity classes in the data model, e.g. the use of the *Vendor Manager* interfaces by the *Contract Manager* which can be traced back to the association between *Contract* and *Vendor* in the data model.
(2) provided interfaces of the underlying framework, e.g. the *Catalog* component.

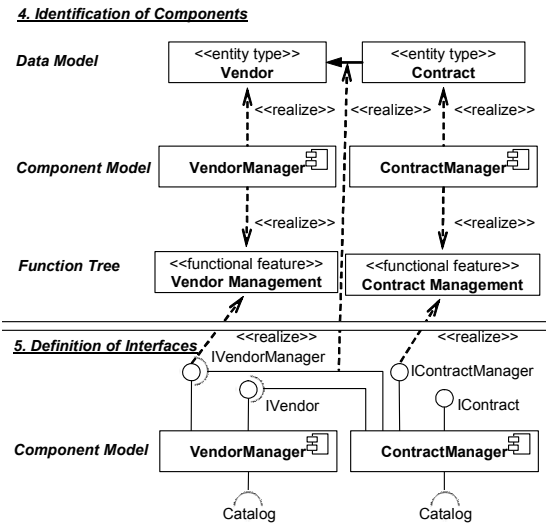


Figure 2. Traceability between the model elements for the steps 4 and 5

Figure 2 also shows the traceability links of the type *realize* from the functional features to the corresponding programming interfaces, which were defined. A rule enables a validation if each functional feature is accessible by at least one interface [Sch08].

Step 6 Definition of the inner structure of the components
In this step sub-components are specified to build components by integrating a bottom-up and a top-down procedure. The composition of components is carried out similarly to the composed categories like *Vendorbase Management* in the category model of Figure 1. Simple components consist only of interface classes, implementing classes and entity classes. Even if later relevant for the package assignment this is not visible in our example.

Step 7 Decision about loose or tight coupling of components
Decoupling is an important issue of the Quasar methodology. For Quasar, the standard way of coupling within the application kernel is a tight coupling. It is usually implemented by programming interfaces using references to entity class interfaces in an object-oriented style. However, for crossing borders between architecture layers or in distributed systems, a loose coupling is appropriate. In terms of Quasar, so-called service oriented interfaces are used wherein transportation classes and data values are minimizing the coupling. Implicit traceability links are established between the corresponding interfaces. A traceability tool can suggest an appropriate link to an interface based on rules and on a comparison of interface identifiers.

3.3 Design Steps for GUI components

Because not covered by Quasar, the authors introduced the design steps for GUI components in an earlier work [Bo08a]. The fundamental sequence of the three steps—

component identification—interface specification—inner structuring—was originally defined by Quasar (steps 4 to 6, see above), and has been adopted for the design of the GUI related components.

For the identification and specification of the GUI components usability as a quality property is an important issue. For an analysis of influencing factors and the development of appropriate solutions a Goal Solution Scheme was developed by the authors, similar to tree diagrams used in the Failure Mode effect analysis. Leading from quality goals through a refinement to influencing factors and further to principles and functional solutions, the scheme supports the transformation of non-functional requirements into functional and technical solutions and components akin to the third phase of QASAR. Figure 3 shows the part of the Goal Solution Scheme corresponding to the case study. Usability is refined to sub-goals representing the six main factors according to [La05] and to the ISO 9241-11 standard. The nodes of the diagram are mapped to architectural elements by design decisions, leading to a lower number of traceability links by reducing the scattering effect.

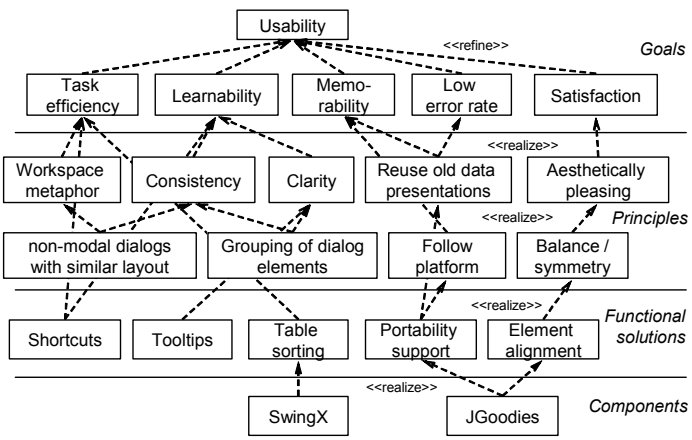


Figure 3. Assignment of principles and solutions to usability goals in a goal solution diagram

In the three design steps, *GUI component identification*, *Definition of the component interfaces*, and *Definition of the GUI components' inner structure*, we realize quality properties especially for usability. Therefore, the Goal Solution Scheme from Figure 3 is applied. To achieve for example the subgoal *satisfaction* of the users, we considered the principles *aesthetically pleasing* using *balanced and symmetric* dialog layouts as important. A technical component helping to accomplish these principles, for example by paying enormous attention on *element alignment* when using Java Swing, is JGoodies, which was utilized in the case study.

This procedure represents a contribution for supporting a systematic implementation of quality features. All corresponding design decisions for the mapping of non-functional requirements to technical solutions are documented using traceability links as shown in Figure 3. Therefore, is becomes possible to check whether and how a particular non-functional requirement is realized by a certain solution. Further, Quasar's guidelines for

component and interface design are traced as presented in the design steps for the application kernel in the preceding subsection.

4 Conclusion and Future Work

In this paper, a method for architectural design concerning quality requirements is presented, which is based on software categories as means of design. A strong alignment between the design for functional and quality requirements (so-called non-functional ones) is established. The advantages of software categories for component identification and decoupling are utilized for the architectural design concerning quality properties. This is achieved by adopting the Quasar procedure. For a detailed, systematic design process and an engineering way of reusing principles and solutions, new intermediary activities and artifacts are introduced, e.g. the arrangement of features to groups and the Goal Solution Scheme. This is done in a way that the overall design effort is reduced by a strict goal-orientation of the design decisions and by the reduced rework.

The improvements regarding evolvability by the introduction of traceability links constitute another important contribution of this work. Traceability links are used because of their great support for software comprehension, decision-making, completeness checks and dependency analyses and other aspects. For minimizing the maintenance effort for the traceability links, the paper provides an assignment of traceability link types to the artifacts and activities of the process. Furthermore, it facilitates the link management by a significant reduction of the number of traceability links by reducing the scattering effect. This reduction is achieved by the improved alignment between quality requirements and the solution elements via features and the Goal Solution Scheme. Moreover, the detailed activities enable a more precise tool support for the link management. The earlier established link types are assigned to the artifact relations to provide the prerequisites for an efficient tracking of the links while evaluating a design.

The work has been performed and evaluated in a reengineering project for a business information system for vendor management. As the next steps, an initial tool support for the architectural design process using traceability shall be provided by generating traceability links in repositories and in the background of design activities. Further steps towards an automated traceability will be performed using this tool support in larger projects, e.g. to establish heuristics for rules and guidelines for tracking the links.

References

[Ad07] Adersberger, J.: Consistency Constraints for the Architectural Development Using the Quasar Methodology (in German: Konsistenzbedingungen bei der Entwicklung einer Softwarearchitektur nach der QUASAR Methode). Erlangen-Nürnberg, Univ., 2007.

[Bo00] Bosch, J.: Design & Use of Software Architectures. Addison Wesley, 2000.

[Bo08] Bode, S.: Traceability Design Decisions for Software Architectures using the Quasar Method (in German: Traceability und Entwurfsentscheidungen für Softwarearchitekturen mit der Quasar-Methode). Diploma Thesis, Ilmenau, Techn. Univ., 2008.

[Bo08a] Bode, S.; Riebisch, M.: Usability-Focused Architectural Design for Graphical User Interface Components. In: Proc. International Conference on Innovation in Software Engineering (ISE'08), 10-12 Dec. 2008, Vienna, Austria (in press).
[CCC03] Cleland-Huang, J.; Chang, C. K.; Christensen, M.: Event-Based Traceability for Managing Evolutionary Change. IEEE Trans. Software Eng. 29(9), 796-810, 2003.
[Ch00] Chung, L. et al.: Non-functional Requirements in Software Engineering. Kluwer, 2000.
[Cl05] Cleland-Huang, J.; Settimi, R.; BenKhadra, O.; Berezhanskaya, E.; Christina, S.: Goal-Centric Traceability for Managing Non-Functional Requirements. ICSE'05, ACM, 2005.
[Co00] Cockburn, A.: Writing Effective Use Cases. Addison Wesley, 2000.
[Eg01] Egyed, A.: A Scenario-Driven Approach to Traceability. In: Proc. 23rd International Conference on Software Engineering ICSE'01, pp. 123-132, IEEE, 2001.
[FB03] Folmer, E.; Bosch, J.: Usability Patterns in Software Architecture. In: Proc. 10th Int. Conf. on Human-Computer Interaction (HCII2003) Volume I pp. 93-97, 2003.
[GF94] Gotel, O. C. Z.; Finkelstein, A. C. W: An Analysis of the Requirements Traceability Problem. Proc. 1st Int. Conf. on Requirements Engineering, IEEE, pp. 94-101, 1994.
[He07] Herpel, K.: Refactoring and Identification of Components (in German: Refactoring und Identifikation von Komponenten). Diploma Thesis, Ilmenau, Techn. Univ., 2007.
[HNS00] Hofmeister, C.; Nord, R.; Soni: Applied Software Architecture. AddisonWesley 2000.
[Ji07] Jiang, Hsin-yi; Nguyen, T. N.; Chang, C. K.; Dong, Fei: Traceability Link Evolution Management with Latent Semantic Indexing. COMPSAC 2007, IEEE, 2007.
[Ka90] K. Kang, S. Cohen, J. Hess, W. Novak, and A. Peterson. Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical Report CMU/SEI-90-TR-021, SEI, Carnegie Mellon University, USA, 1990.
[KKC00] Kazman, R., Klein, M., Clements, P.: ATAM - Method for Architecture Evaluation. (Tech. Rep. CMU/SEI-2000-TR-004). Pittsburgh, USA: CMU, SEI, 2000.
[La05] Lauesen, S.: User interface design: a software engineering perspective. Addison Wesley, 2005.
[Le02] Letelier, P.: A Framework for Requirements Traceability in UML-based Projects. 1st Int. Workshop on Traceability in Emerging Forms of SE (TEFSE'02), pp. 32-41, 2002.
[MGP08] Maeder, P.; Gotel, O.; Philippow, I.: Rule-based maintenance of post-requirements traceability relations. In Proc. 16th Int'l Req. Eng. Conf., 2008.
[MPR07] Maeder, P.; Philippow, I.; Riebisch, M.: Customizing Traceability Links for the Unified Process. In: Software Architectures, Components, and Applications. QoSA 2007 - Revised Selected Papers. Springer LNCS pp. 53-71, 2008.
[Ni93] Nielsen, J.: Usability Engineering. Interactive Technologies. Academic Press, Boston, USA, 1993.
[OG02] Olsson, T.; Grundy, J.: Supporting Traceability and Inconsistency Management between Software Artifacts. In: Proc. Int. Conf. on Software Engineering and Application, 2002.
[Pi04] Pinheiro, F. A. C.: Requirements Traceability. In: Leite, J.; Doorn J. (Eds.) Requirements Traceability in Perspectives on Software Requirements, pp. 91-113, Kluwer, 2004.
[Po96] Pohl, K.: PRO-ART:Enabling Requriements Pre-Traceability. In: Proc. of the 2nd Int. Conference on Requirements Engineering ICRE'96, pp. 76-84, IEEE, 1996.
[RJ01] Ramesh, B.; Jarke, M.: Toward reference models of requirements traceability. IEEE Trans. Software Eng. 27(1), 58–93, 2001.
[Sh92] Shneiderman, B.: Designing the user interface: strategies for effective human-computer interaction. Addison-Wesley, Boston, MA, USA, 2nd edition, 1992.
[Sch08] Schröter, M.: Rules for the Assessment of Traceability Relations and Dependencies for Application Design Using Quasar (in German). Thesis, Ilmenau, Techn. Univ., 2008.
[Si03] Siedersleben, J.: Quasar Standardarchitektur (German). Munich, sd&m research, 2003.
[Si04] Siedersleben, J.: Modern Software Architecture (in German). Heidelberg, dpunkt, 2004.
[We07] Wendler, S.: Design Decisions for Software Architectures (in German) Diploma Thesis, Ilmenau, Techn. Univ., 2007.

Plattform & Produktlinien

Vergleich und Integration von Komposition und Annotation zur Implementierung von Produktlinien*

Sven Apel†, Christian Kästner‡, and Christian Lengauer†

† Fakultät für Informatik und Mathematik, Universität Passau
{apel, lengauer}@uni-passau.de
‡ Fakultät für Informatik, Universität Magdeburg
kaestner@iti.cs.uni-magdeburg.de

Abstract: Es gibt eine Vielzahl sehr unterschiedlicher Techniken, Sprachen und Werkzeuge zur Entwicklung von Softwareproduktlinien. Trotzdem liegen gemeinsame Mechanismen zu Grunde, die eine Klassifikation in Kompositions- und Annotationsansatz erlauben. Während der Kompositionsansatz in der Forschung große Beachtung findet, kommt im industriellen Umfeld hauptsächlich der Annotationsansatz zur Anwendung. Wir analysieren und vergleichen beide Ansätze anhand von drei repräsentativen Vertretern und identifizieren anhand von sechs Kriterien individuelle Stärken und Schwächen. Wir stellen fest, dass die jeweiligen Stärken und Schwächen komplementär sind. Aus diesem Grund schlagen wir die Integration des Kompositions- und Annotationsansatzes vor, um so die Vorteile beider zu vereinen, dem Entwickler eine breiteres Spektrum an Implementierungsmechanismen zu Verfügung zu stellen und die Einführung von Produktlinientechnologie in bestehende Softwareprojekte zu erleichtern.

1 Einleitung

In den letzten Jahren hat die Bedeutung von Produktlinien in der Softwareentwicklung stetig zugenommen [BCK98, PBvdL05]. Als Alternative zur individuellen Neuimplementierung eines jeden Programms, erlauben Softwareproduktlinien die systematische Wiederverwendung von Code innerhalb einer Domäne, d. h. innerhalb einer Familie von Programmen [CE00]. Programme, die zu einer Familie bzw. Softwareproduktlinie gehören und aus einer gemeinsamen Codebasis generiert werden, heißen auch *Varianten*.

Für Produktlinien hat das Konzept des Features besondere Bedeutung. *Features* repräsentieren die Unterschiede und Gemeinsamkeiten von Varianten einer Produktlinie [CE00]. In der Domäne der Datenbanksysteme sind beispielsweise die Transaktionsverwaltung und die Speicherverwaltung Features. Während eine grundlegende Speicherverwaltung in allen Varianten von Datenbanksystemen unabdingbar ist, wird nicht in allen Anwendungsfällen eine Transaktionsverwaltung benötigt [KAB07].

Typischerweise werden Features durch das Hinzufügen neuer und das Erweitern bestehender

*Diese Arbeit wurde durch die Deutsche Forschungsgemeinschaft unterstützt, Projektnummer AP 206/2-1.

Strukturen in der Codebasis einer Produktlinie implementiert, d. h. wir beschränken uns hier auf solche Features, die Inkremente in der Funktionalität eines Softwaresystems darstellen, und betrachten nicht etwa Features wie Ausführungszeit oder Sicherheit.

Unternehmen, die Produktlinientechnologie in ihren Entwicklungsprozess einführen wollen, können aus einer Vielzahl von Methoden, Sprachen und Werkzeugen wählen: von Versionsverwaltungssystemen, über die Verwendung einfacher *#ifdef*s, über Frameworks und Komponenten [BCK98] bis hin zu diversen spezialisierten Programmiersprachen und -werkzeugen [Pre97, BSR04, KHH+01, Gri00]. All diese haben verschiedene Stärken und Schwächen, welche in einschlägigen Foren umfassend diskutiert werden [SvGB05, LHBC05, MO04, KAK08, MP03, ALS08].

In unseren bisherigen Arbeiten haben wir uns mit verschiedenen Techniken zur Entwicklung von Softwareproduktlinien näher beschäftigt. Wir teilen Methoden, Werkzeuge und Sprachen zur Entwicklung von Produktlinien in zwei grundlegende Ansätze ein: dem Kompositionsansatz und dem Annotationsansatz [KAK08]. Wir wählen diese Einteilung, da sie uns erlaubt, eine signifikante Anzahl von Techniken zur Implementierung von Features systematisch zu vergleichen. Techniken, die sich nicht in eine der beiden Kategorien einteilen lassen, wie z. B. prozessorientierte Ansätze, werden nicht weiter betrachtet.

Im *Kompositionsansatz* werden die Produkte einer Produktlinie durch das Zusammensetzen von Codeeinheiten generiert. Wir haben in früheren Arbeiten einige Vertreter dieses Ansatzes, z. B. AHEAD [BSR04] und AspectJ [KHH+01], untersucht [ALS08, ALMK08, AKL08, KAB07, AL08] und sind auf einige prinzipielle Einschränkungen gestoßen [KAB07, KAK08], insbesondere hinsichtlich der Einführung von Produktlinientechnologie in bestehende Softwareprojekte. Aus diesem Grund wendeten wir uns gleichzeitig dem Annotationsansatz zu, dem auch Standardtechniken wie z. B. Präprozessoren und Frameprozessoren angehören. Im *Annotationsansatz* werden Produkte auf Basis von Annotationen einer gemeinsamen Codebasis generiert. Wir haben verschiedene Vertreter analysiert und Erweiterungen und Verbesserungen vorgeschlagenen [KAK08, KA08].

Während wir die spezifischen Probleme beider Ansätze (Komposition und Annotation) untersuchten, z. B. hinsichtlich Granularität, Sprachunabhängigkeit, Ausdruckskraft oder Typsicherheit [KAK08, AL08, ALS08, KAB07, KA08, AKL08], stellten wir nach und nach fest, dass beide komplementär in ihren Stärken sind, d. h. in Punkten, in denen der eine Ansatz Stärken hat, zeigt der andere Schwächen und umgekehrt. Zum Beispiel spielt der Kompositionsansatz seine Stärken bei der Modularisierung von großen, wiederverwendbaren Codeeinheiten aus. Der Annotationsansatz ist vorteilhaft auf kleineren Skalen, z. B. wenn es darum geht, Variabilität im Kontrollfluss von Funktionen auszudrücken.

Dieser Beitrag gibt einen Überblick über Erfahrungen mit beiden Ansätzen anhand von den drei repräsentativen Werkzeugen AHEAD [BSR04], FeatureHouse [AKL09] und CIDE [KAK08], diskutiert Unterschiede und Gemeinsamkeiten, und zeigt auf, wie beide Ansätze gewinnbringend kombiniert werden können. Unser Ziel ist, beide Ansätze so zu kombinieren, dass die Stärken des einen die Schwächen des anderen aufwiegen. Besonderes Augenmerk legen wir auf die Einführung von Produktlinientechnologie in großen Softwareprojekten. Generell ist die Einführung von Produktlinientechnologien schwierig, insbesondere wenn ein bestehendes Softwareprojekt nicht bereits als Produktlinie

entwickelt wurde. Dies bezeichnen wir im Weiteren als *Adoptionsbarriere*. In früheren Analysen wurde beobachtet, dass zwar leichtgewichtige Annotationsansätze wie Frames und Präprozessoren die Adoptionsbarriere verringern, aber auf lange Sicht die Softwarequalität und Architekturstabilität beeinträchtigen können [CK02]. Wir argumentieren, dass gerade die Verbindung von Komposition und Annotation die Adoptionsbarriere verringert und trotzdem Softwarequalität und Architekturstabilität nicht beeinträchtigt.

2 Ansätze zur Implementierung von Produktlinien

Es gibt viele verschiedene Technologien zur Implementierung von Softwareproduktlinien. Die meisten kann man in zwei Ansätze gliedern: den Kompositionsansatz und den Annotationsansatz [KAK08]. Technologien, die nicht in einer der beiden Kategorien fallen, werden nicht weiter betrachtet, z. B. prozessorientierte Ansätze, Versionsverwaltungssysteme oder Generatoren.

Kompositionsansatz. Im Kompositionsansatz werden die Features einer Produktlinie als eigenständige (physisch getrennte) Codeeinheiten[1] implementiert. Zur Generierung einer Variante einer Produktlinie, werden auf Grundlage einer Auswahl von Features die entsprechenden Codeeinheiten ermittelt und zusammengesetzt, üblicherweise zur Übersetzungs- oder Startzeit. Es gibt eine Vielzahl von Technologien zur Implementierung und Komposition von Codeeinheiten in Produktlinien, z. B. Komponenten und Frameworks [BCK98], Featuremodule [Pre97, BSR04, ALS08], Hypermodule [TOHS99], und Aspekte [Gri00]. Abhängig vom jeweiligen Ansatz variieren die verwendeten Kompositionsmechanismen von Plugins über Vererbung bis hin zu komplexen Codetransformationen. Die generelle Idee des Kompositionsansatzes ist in Abbildung 1 am Beispiel des featureorientierten Generators AHEAD [BSR04] illustriert.

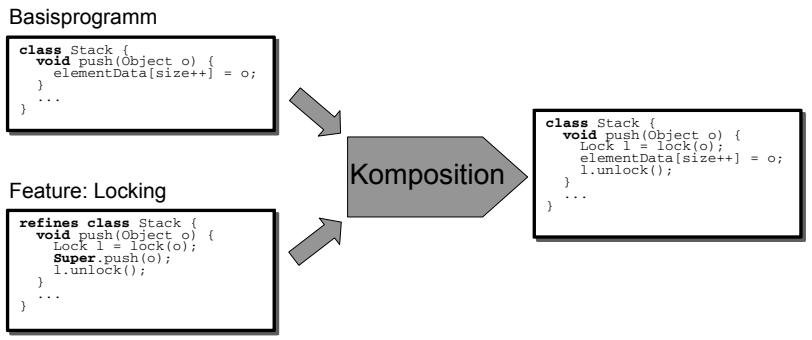


Abbildung 1: Generierung von Varianten mittels Komposition von Codeeinheiten.

[1] Der Kompositionsansatz ist auch auf andere Arten von Dokumenten, z. B. HTML-Dokumentationen oder Übersetzungsskripte, anwendbar.

```
1 class Stack {
2   void push(Object o) { ... }
3   Object pop() { ... }
4 }
```

```
5 refines class Stack {
6   void backup() { ... }
7   void restore() { ... }
8   void push(Object o) { backup(); Super.push(o); }
9 }
```

Abbildung 2: Eine Stackimplementierung in AHEAD (Kompositionsansatz).

Im Weiteren verwenden wir AHEAD sowie das kompatible und sprachunabhängige FeatureHouse [AKL09] als repräsentative Vertreter des Kompositionsansatzes. Wenn die Bewertung für andere Vertreter des Kompositionsansatzes abweicht, diskutieren wir das explizit. In AHEAD und FeatureHouse werden Features als Module implementiert, die, wenn einem Basisprogramm hinzugefügt, diesem neue Pakete und Klassen beigeben und bestehende Pakete, Klassen und Methoden erweitern. Abbildung 2 gibt ein einfaches Codebeispiel. Das Basisprogramm implementiert einen Stack, und ein separates Modul implementiert das Feature *UnDo*. Das Schlüsselwort *refines* zeigt an, dass ein Feature eine bereits vorhandene Klasse verfeinert. Die Verfeinerung in unserem Beispiel fügt zwei neue Methoden ein (*backup* und *restore*) und erweitert die Methode *push* durch Überschreiben. Das Schlüsselwort *Super* verweist auf die verfeinerte Klasse. Das Basisprogramm und das Featuremodul werden mit AHEAD bzw. FeatureHouse komponiert. Dabei werden die gewählten Module überlagert, d. h. im Beispiel entsteht eine zusammengesetzte Klasse *Stack* mit den Methoden *push*, *pop*, *backup* und *restore*, wobei die Methode *push* um den zusätzlichen Aufruf der Methode *backup* erweitert ist.

Abhängig von den im Kompositionsprozess ausgewählten Features, werden unterschiedliche Varianten der Produktlinie generiert. In unserem Beispiel gibt es nur zwei Varianten: einen Stack mit und einen ohne *UnDo*-Feature. Andere Kompositionswerkzeuge funktionieren sehr ähnlich, z. B. AspectJ [KHH+01].

Annotationsansatz. Im Annotationsansatz werden Features nicht als separate Codeeinheiten implementiert, sondern liegen in einer gemeinsamen Codebasis vor. Mittels Annotationen (z. B. textuellen Markierungen) werden die Codefragmente, die zu einem Feature gehören, entsprechend ausgezeichnet.[2] Eine Standardtechnik, die oft in industriellen Softwareproduktlinien Verwendung findet, ist die "Umrahmung" von Code mittels *#ifdef* - und *#endif* -Anweisungen. Alternativ dazu können auch andere Technologien wie Annotationen, Frames [JBZZ03] oder Softwarepläne [CPR07] verwendet werden. Diese Technik ist in kommerziellen Produktlinienwerkzeugen, wie z. B. *pure::variants*[3] oder *Gears*[4], verbreitet.

Wir verwenden unser Werkzeug CIDE als Vertreter des Annotationsansatzes [KAK08]. CIDE ähnelt Präprozessoren in der Hinsicht, dass es ermöglicht Codefragmente individuellen Features zuzuordnen, und dass Codefragmente nicht ausgewählter Features vor der Übersetzung entfernt werden. CIDE eliminiert einige Schwächen von präprozessorbasier-

[2] Auch der Annotationsansatz ist auf andere Arten von Dokumenten, z. B. HTML-Dokumentationen oder Übersetzungsskripte, anwendbar.

[3] http://www.pure-systems.com/

[4] http://www.biglever.com/

```
1 class Stack {
2   void push(Object o) { backup(); ... }
3   Object pop() { ... }
4   void backup() { ... }
5   void restore() { ... }
6 }
```

Abbildung 3: Bedingte Übersetzung mit CIDE (Annotationsansatz).

ten Techniken [KAK08]: (a) Annotationen werden als Hintergrundfarben präsentiert und "verunstalten" damit nicht den Quellcode; (b) Annotationen basieren auf der syntaktischen Struktur des Quellcodes, so dass nur optionale Elemente einer Sprache annotiert werden können (z. B. ist es nicht möglich, nur eine öffnende Klammer oder nur den Namen eine Klasse zu annotieren); (3) Annotationen werden von einer Werkzeuginfrastruktur verwaltet, die eine Navigationsfunktion und unterschiedliche Sichten auf den Quellcode anbietet (z. B. kann aller Code, der zu einem Feature gehört, ein- und ausgeblendet werden).

In Abbildung 3 ist unser Stackbeispiel zu sehen, diesmal mittels Annotationen implementiert. Die Codestücke des Basisprogramms sowie aller Features sind in einer Codebasis verschmolzen (also nicht physisch separiert). Codefragmente, die zum Feature *UnDo* gehören, sind farbig unterlegt und unterstrichen. Zur Generierung einer Variante ohne das *UnDo*-Feature, wird der markierte Code vor der Übersetzung entfernt (bedingte Übersetzung). Erneut sei betont, dass wir uns hier nicht speziell mit CIDE, sondern mit dem Annotationsansatz im Allgemeinen beschäftigen.

3 Vergleich von Kompositionsansatz und Annotationsansatz

Um die individuellen Stärken und Schwächen von Komposition und Annotation gegenüber zu stellen, analysieren wir beide Ansätze anhand von sechs Kriterien. Die Kriterien sind mit dem Ziel gewählt, Gemeinsamkeiten und Unterschiede aufzudecken, und basieren auf Erfahrungen mit Fallstudien zur Produktlinienentwicklung [TBKC07, AKL09, KAB07, KAK08, R+08]. Das Ergebnis unseres Vergleichs ist durch qualitative Bewertung hinsichtlich einzelner Kriterien in Tabelle 1 zusammengefasst. Wir betonen, dass die Bewertungen und Begründungen sich auf unsere Erfahrung stützen und somit keinesfalls als definitiv anzusehen sind. Vielmehr verstehen wir Tabelle 1 und die Ausführungen dieses Abschnitts als Anregung zur Diskussion.

Auffindbarkeit von Featurecode. Features sind zuerst einmal dem Geist der Entwickler und Domänenexperten entsprungene abstrakte Konstrukte. Zur Erleichterung des Verständnisses und der Wartung des Quelltexts sollte die Abbildung von Features auf Implementierungseinheiten möglichst einfach strukturiert sein [CE00], d. h. der Code, der zu einem Feature gehört, sollte leicht in der Codebasis aufzufinden sein. Dies ist besonders wichtig für die Fehlersuche, um vom fehlerhaften Verhalten (Feature) auf den entsprechenden

Code schließen zu können. Im Kompositionsansatz ist es leicht, die Verbindung zwischen Feature und Code herzustellen, da jeweils ein Feature durch eine separate Codeeinheit implementiert wird. Zum Beispiel ist der Code, der das *UnDo*-Feature implementiert, im entsprechenden Featuremodul zu finden (siehe Abbildung 2, Zeilen 5–9). Im Gegensatz dazu kann das Auffinden des zu einem Feature gehörenden Codes im Annotationsansatz kompliziert sein, besonders dann, wenn der Code über weite Teile der Codebasis verstreut ist [SC92] (siehe auch Abbildung 3). Mit speziellen Werkzeugen wie CIDE kann dieses Problem gelindert werden, indem eine Navigationsfunktion und Sichten zur Verfügung gestellt werden. So kann man sich in CIDE z. B. nur den Code des Features *UnDo* anzeigen lassen. Nichtsdestotrotz ist immer ein extra Werkzeug von Nöten.

Modularität. Modularität von Featurecode wird im Kompositionsansatz naturgemäß besser unterstützt als im Annotationsansatz. Beispielsweise erlauben Komponenten [BCK98] oder Hypermodule [TOHS99] die separate Übersetzung von Code, der zu unterschiedlichen Features gehört. Es gibt allerdings auch einige populäre Sprachen und Werkzeuge, die dem Kompositionsansatz folgen, aber nicht einen ähnlich hohen Grad an Modularität ermöglichen. Zum Beispiel können Featuremodule in AHEAD, FeatureHouse oder AspectJ nicht separat übersetzt werden und verletzen das Geheimnisprinzip. Diese Probleme sowie mögliche Lösungen wurden insbesondere im Kontext von aspektorientierter Programmierung diskutiert [GSS+06]. Im Annotationsansatz werden keine Mechanismen zur Modularität von Featurecode verwendet. Werkzeugunterstützung kann über dieses Problem zum Teil hinweghelfen, aber eine separate Übersetzung und die isolierte Analyse von Features werden unmöglich.

Granularität. Die Granularität eines Implementierungsmechanismus ist eng mit seiner Ausdruckskraft verknüpft. Sehr grobgranulare Mechanismen erlauben nur das Hinzufügen und Entfernen von Verzeichnissen und Dateien. Feingranulare Mechanismen ermöglichen das Hinzufügen, Weglassen und Erweitern von Klassen, Methoden, Parametern und einzelnen Anweisungen [KAK08]. Im Annotationsansatz kann jedes Codefragment einem Feature zugeordnet werden. Dieses Prinzip skaliert in Werkzeugen wie CIDE von ganzen Verzeichnissen bis hin zu Parametern und Ausdrücken (siehe Abbildung 3). Im Kompositionsansatz wird ein weniger breites Spektrum abgedeckt. Üblicherweise wird das Hinzufügen von gesamten Verzeichnissen und Dateien unterstützt. Daneben gibt es auch einige Werkzeuge und Sprachen, die das Hinzufügen und Erweitern von Klassen, Methoden, und Instanzvariablen erlauben (siehe Abbildung 2). Die Manipulation von Anweisungen innerhalb einer Methode oder von Parametern und Ausdrücken wird im Kompositionsansatz nicht unterstützt [KAK08].[5] Stattdessen kann der Programmierer nur auf Umwegen zum Ziel gelangen. Zum Beispiel macht eine nur minimale Änderung unseres Stackbeispiels die Implementierung in AHEAD ungleich komplizierter. Steht der Aufruf der Methode *backup* in Abbildung 2 nicht an erster Stelle im Rumpf der Methode *push*, dann muss der Programmierer eine Hookmethode wie in Abbildung 4 verwenden.

[5] Einige aspektorientierte Sprachen unterstützen die Erweiterung von Anweisungen zu einem gewissen Grad (mittels *'call && withincode'*). Dies birgt aber einige schwerwiegende Probleme [KAB07, KAK08].

```
1  class Stack {
2    void push(Object o) {
3      if(o==null) return; hook(); ...
4    }
5    Object pop() { ... }
6    void hook() {}
7  }
```

```
8  refines class Stack {
9    void backup() { ... }
10   void restore() { ... }
11   void hook() { backup(); }
12 }
```

Abbildung 4: Feingranulare Erweiterungen mit AHEAD.

Sicherheit. Für den Kompositionsansatz und den Annotationsansatz gibt es gleichermaßen Bestrebungen, sicherzustellen, dass alle erlaubten Varianten (potentiell mehrere Millionen) einer Produktlinie korrekt sind [TBKC07, AKL08, KA08]. Im Kompositionsansatz wird syntaktische Korrektheit durch die verwendeten Sprachmechanismen garantiert. Syntaktisch inkorrekte Kompositionen werden vom Übersetzer oder vom Kompositionswerkzeug zurückgewiesen. Im Annotationsansatz ist dies problematischer. Werkzeuge die auf Präprozessoren und Frames basieren, können leicht inkorrekte Varianten generieren [KAK08, KA08]. Man braucht nur fälschlicherweise eine Klammer zuviel zu annotieren, und der Fehler wird erst gefunden, wenn die Variante nach der Generierung übersetzt wird. Bei mehreren Millionen Varianten ist dies unter Umständen zu spät, da nicht alle Varianten getestet werden können und erst bei konkreten Kundenanfrage generiert werden. Modernere Werkzeuge wie CIDE, die disziplinierte Annotationen auf Basis der syntaktischen Struktur erzwingen, erreichen ein vergleichbares Maß an Sicherheit wie im Kompositionsansatz [KAK08]. Auch zur Typsicherheit gibt es erste Arbeiten für beide Ansätze, die vom Konzept her sehr ähnlich sind [TBKC07, AKL08, CP06, KA08, KKB08].

Sprachunabhängigkeit. Hinsichtlich Sprachunabhängigkeit liegen beide Ansätze gleich auf. Präprozessoren und Frames sind komplett sprachunabhängig. Selbst disziplinierte Ansätze wie CIDE benutzen die Grammatik einer beliebigen Sprache zum Annotieren von Code [KAK08]. Im Gegensatz dazu sind Sprachen und Werkzeuge, die dem Kompositionsansatz folgen, üblicher Weise auf eine Artefaktsprache festgelegt, z. B. ist AspectJ auf Java als Zielsprache festgelegt. AHEAD und FeatureHouse sind dabei erwähnenswerte Ausnahmen. Besonders in FeatureHouse wird ein sprachunabhängiger Mechanismus zur Komposition verwendet, der leicht um weitere Sprachen erweitert werden kann [AKL09].

Einführung von Produktlinientechnologie. Unternehmen sind sehr vorsichtig bei der Einführung des Kompositionsansatzes zur Produktlinienentwicklung. Ein Grund ist, das bestehender Code sowie etablierte Prozesse zur sehr abgeändert werden müssten. Selbst in einer mittelgroßen akademischen Studie war dies langwierig und fehleranfällig [KAK08]. Derzeit werden allenfalls Komponenten und Frameworks eingeführt [BCK98, PBvdL05]. In Gegensatz dazu setzt sich der Annotationsansatz besser durch. Dies liegt im Wesentlichen an den unkomplizierten Werkzeugen sowie an dem Umstand, dass bestehender Code und etablierte Prozesse zunächst nicht abgewandelt werden müssen [CK02]. Der Annotationsansatz erleichtert dadurch die Einführung von Produktlinientechnologie enorm.

	Komposition	Annotation	Integration	Signifikanz im Entwicklungsproz.
Auffindbarkeit	++	+	+	alle Phasen
Modularität	+	−−	+	Wartung
Granularität	−	++	++	schnelle Adoption, Impl.
Sicherheit	+	+	+	alle Phasen
Sprachunabh.	+	+	+	alle Phasen
Adoption	−	++	++	Projektbeginn

++ sehr gut unterstützt; + gut unterstützt; − schlecht unterstützt; −− gar nicht unterstützt

Tabelle 1: Vergleich von Kompositionsansatz, Annotationsansatz und integriertem Ansatz.

```
13 class Stack {
14   void push(Object o) {
15     if(o==null) return; backup(); ...
16   }
17   Object pop() { ... }
18 }
```

```
19 refines class Stack {
20   void backup() { ... }
21   void restore() { ... }
22 }
```

Abbildung 5: Implementierung eines Stacks mit Featuremodulen und Annotationen.

4 Ein Integrierter Ansatz

Der Vergleich des Kompositions- und Annotationsansatzes zeigt, dass beide hinsichtlich unserer sechs Kriterien unterschiedliche Stärken und Schwächen aufweisen. Oft sind die Stärken und Schwächen komplementär. In anderen Fällen sind vergleichbare Fortschritte für beide Ansätze gemacht worden. Alles in allem scheint eine Kombination sinnvoll. In der aktuellen Forschung findet der Kompositionsansatz viel Beachtung. Der Annotationsansatz spielt eine untergeordnete Rolle. Im industriellen Umfeld setzen Produktlinienwerkzeuge wie Gears und pure::variants dagegen häufig auf Annotationen. Wie schlagen eine Integration zum beiderseitigen Vorteil vor.

Konzeptionell ist eine Integration beider Ansätze einfach. Dem Kompositionsansatz folgend werden grobgranulare Features in separate Codeeinheiten aufgeteilt. Für feingranulare Features, bei denen die Grenzen dieses Ansatzes erreicht werden, greift der Programmierer stattdessen zu Annotationen. Abbildung 5 illustriert an unserem Beispiel, wie beide Ansätze zur Implementierung eines Stacks kombiniert werden können. Die Methoden *backup* und *restore* werden in einem separaten Featuremodul als Erweiterungen des Basisprogramms implementiert. Der Aufruf der Methode *backup* in der Methode *push* wird auf Grund der in Abbildung 4 gezeigten Probleme nicht herausgelöst, sondern entsprechend annotiert. Zur Generierung einer Programmvariante werden die separaten Codeeinheiten komponiert und dann die Annotationen evaluiert, um Code gegebenenfalls zu entfernen.

Technisch ist die Integration ebenfalls nicht kompliziert. In unserem Beispiel müssen die Funktionalitäten von CIDE und AHEAD/FeatureHouse vereinigt werden. Zum Beispiel könnte CIDE so erweitert werden, dass es das Schlüsselwort *refines* versteht und eine Komposition mit bestehenden Kompositionswerkzeugen durchführt, nachdem Annotationen

ausgewertet werden. Darüber hinaus müssen die Funktionalitäten für Navigation und Sichten entsprechend angepasst werden. In diesem Zusammenhang haben wir bereits zwei Transformationswerkzeuge entwickelt, die annotierten Code in physisch separierten Code [KKB07] und umgekehrt überführen [KKB08].

Die Integration von Kompositionsansatz und Annotationsansatz vereint nicht automatisch nur die Stärken beider Ansätze. Zum Beispiel, wenn Code innerhalb von Modulen annotiert wird, ist eine separate Übersetzung ausgeschlossen. Der Vorteil der Integration ist aber, dass Entwickler jederzeit entscheiden können, welches Kriterium Priorität hat. Zusammengefasst ergibt sich für die sechs Kriterien folgendes Bild (siehe Tabelle 1).

Die Abbildung von Features auf Code wird durch Annotationen komplizierter. Ein Feature ist nicht nur durch ein Modul implementiert, sondern möglicherweise durch weitere verstreute, annotierte Codefragmente (siehe Abbildung 5). Wie schon beim Annotationsansatz erwähnt, helfen hier die Navigationsfunktion und Sichten von Werkzeugen wie CIDE. Das heißt, im schlechtesten Fall (es gibt nur ein Modul, dass Code aller Features enthält) ist die Integration vergleichbar mit dem Annotationsansatz und im besten Fall (pro Feature wird genau ein Featuremodul implementiert) mit dem Kompositionsansatz. In der Praxis wird die Lösung irgendwo dazwischen liegen und somit besser sein als der Annotationsansatz allein und trotzdem von dessen Vorteilen auf kleinen Skalen profitieren.

Bei der Integration von Kompositions- und Annotationsansatz kann der Entwickler ein Feature auf zwei Arten implementieren: entweder modular mittels Klassen, Featuremodulen, Aspekten, etc., oder nicht-modular mittels Annotationen. Ähnlich dem Kriterium der Auffindbarkeit, muss der Entwickler eine Entscheidung treffen, welcher Grad an Modularität gewünscht ist.

Der integrierte Ansatz profitiert von der feinen Granularität des Annotationsansatzes. Grobgranulare Features werden immer noch als Module implementiert und für die Implementierung von feingranularen Features kommen dann Annotationen hinzu. Annotationen können natürlich auch für die Implementierung von grobgranularen Feaures verwendet werden, verschlechtern aber die Modularität.

Da sich beide Ansätze hinsichtlich der Sicherheit und Sprachunabhängigkeit sehr ähneln, ändern sich diese Eigenschaften nicht im Gesamtansatz. Bestehende Methoden sind sehr ähnlich und können integriert werden [AKL08, KA08], was im Wesentlichen ein Implementierungsproblem ist.

Der vielleicht interessanteste Vorteil einer Integration liegt im Prozess der Einführung von Produktlinientechnologie in ein Unternehmen. In den frühen Phasen einer Einführung verwenden die Entwickler hauptsächlich Annotationen. Dies ermöglicht die schnelle Definition neuer Varianten, ohne die Codebasis oder den Entwicklungsprozess verändern zu müssen. In späteren Phasen, nachdem die Idee von Produktlinien im Projektmanagement und im Entwicklungsprozess verankert ist, können annotierte Codefragmente *schrittweise* in separate Featuremodule überführt werden. Transformationswerkzeuge helfen diesen Prozess zu automatisieren [KKB07, KKB08]. Die Entwickler können zu jedem Zeitpunkt entscheiden, inwiefern sie weiter Annotationen in Module überführen. Unsere hier vorgestellten Kriterien weisen den Weg und beleuchten die Effekte einer solchen Entscheidung. Eine Evaluierung in der Praxis steht allerdings noch aus.

5 Verwandte Arbeiten

In der Literatur finden sich einige vergleichende Studien, die verschiedene Implementierungsansätze für Produktlinien vergleichen. Lopez-Herrejon et al. [LHBC05] und Mezini und Ostermann [MO04] vergleichen Sprachen, die dem Kompositionsansatz folgen. Beide untersuchen im Wesentlichen Modularität. Der Annotationsansatz wird nicht betrachtet.

Anastasopoulos und Gacek [AG01] und Muthig und Patzke [MP03] vergleichen elf bzw. sechs Implementierungsansätze für Produktlinien. Beide Studien beziehen bedingte Übersetzung und Frames (Annotationsansätze) mit ein. In beiden Fällen stehen die Ausdruckskraft und verschiedene Details der verwendeten Sprachen im Vordergrund. Eine Integration des Kompositionsansatzes und des Annotationsansatzes wird nicht in Betracht gezogen. Des Weiteren vergleichen wir die generellen Ansätze von Komposition und Annotation und nicht spezielle Programmiersprachen und –werkzeuge.

Svahnberg et al. [SvGB05] schlagen eine Taxonomie für Variabilitätsmechanismen vor. Unsere Gliederung in Kompositionsansatz und Annotationsansatz ist orthogonal zu ihrer Klassifizierung.

Die Idee der schrittweisen Einführung von Produktlinientechnologie auf Grund von Komposition und Annotation erwuchs aus einer Diskussion im Dagstuhl-Seminar 08281 ‘Software Engineering for Tailor-made Data Management’. Die Migration von Produktlinientechnologie in ein Unternehmen sollte von ‘in Produktlinien denken’, über die Nutzung von Präprozessoren und Objektorientierung, in Richtung moderner Kompositionsansätze wie Aspektorientierung oder Featureorientierung verlaufen, bis hin zu modellgetriebener Entwicklung. Wir unterstützen diesen Migrationspfad mit unserem Ansatz.

FeatureC++ [ALRS05], eine C++-Erweiterung für featureorientierte und aspektorientierte Programmierung, erlaubt derzeit schon die direkte Kombination von Annotationen in Form von *#ifdef*s und Featuremodulen. Damit kann FeatureC++ als erster Vertreter des hier vorgeschlagenen integrierten Ansatzes verstanden werden. Allerdings fallen *#ifdef*s mit all ihren Problemen hinter moderneren Ansätzen wie CIDE zurück [KAK08].

6 Zusammenfassung

Es gibt eine Vielzahl von Techniken, Sprachen und Werkzeugen zur Entwicklung von Softwareproduktlinien. Wir haben diese in den Kompositionsansatz und den Annotationsansatz gegliedert, und haben beide Ansätze anhand von drei Werkzeugen und sechs Kriterien analysiert und verglichen. Wir sind zu dem Schluss gekommen, dass die Stärken und Schwächen beider komplementär sind, und wir schlagen eine Integration beider Ansätze vor.

Anstatt den Entwickler zu zwingen, sich für einen Ansatz zu entscheiden, erlaubt die hier vorgeschlagene Integration, mit dem Annotationsansatz zu beginnen und schrittweise Teile eines Softwareprojekts mittels des Kompositionsansatzes zu transformieren. Für jedes Feature können Entwickler entscheiden, welcher Ansatz am besten geeigneten ist.

Unsere Kriterien bilden dafür eine Analyse- und Diskussionsgrundlage. Anstatt eines Alles-Oder-Nichts-Ansatzes mit hohem Risiko schlagen wir eine schrittweise Einführung mit abschätzbarem Risiko vor.

Literatur

[AG01] M. Anastasopoules und C. Gacek. Implementing Product Line Variabilities. *SIGSOFT Softw. Eng. Notes*, 26(3):109–117, 2001.

[AKL08] S. Apel, C. Kästner und C. Lengauer. Feature Featherweight Java: A Calculus for Feature-Oriented Programming and Stepwise Refinement. In *Proc. Int'l Conf. Generative Programming and Component Engineering*. ACM Press, 2008.

[AKL09] S. Apel, C. Kästner und C. Lengauer. FeatureHouse: Language-Independent, Automatic Software Composition. In *Proc. Int'l Conf. on Software Engineering*. IEEE CS Press, 2009.

[AL08] S. Apel und C. Lengauer. Superimposition: A Language-Independent Approach to Software Composition. In *Proc. Int'l Symposium on Software Composition*, Jgg. 4954 of *LNCS*, Seiten 20–35. Springer-Verlag, 2008.

[ALMK08] S. Apel, C. Lengauer, B. Möller und C. Kästner. An Algebra for Features and Feature Composition. In *Proc. Int'l. Conf. Algebraic Methodology and Software Technology*, Jgg. 5140 of *LNCS*, Seiten 36–50. Springer-Verlag, 2008.

[ALRS05] S. Apel, T. Leich, M. Rosenmüller und G. Saake. FeatureC++: On the Symbiosis of Feature-Oriented and Aspect-Oriented Programming. In *Proc. Int'l Conf. Generative Programming and Component Engineering*, Jgg. 3676 of *LNCS*, Seiten 125–140. Springer-Verlag, 2005.

[ALS08] S. Apel, T. Leich und G. Saake. Aspectual Feature Modules. *IEEE Trans. Softw. Eng.*, 34(2):162–180, 2008.

[BCK98] L. Bass, P. Clements und R. Kazman. *Software Architecture in Practice*. Addison-Wesley, 1998.

[BSR04] D. Batory, J. Sarvela und A. Rauschmayer. Scaling Step-Wise Refinement. *IEEE Trans. Softw. Eng.*, 30(6):355–371, 2004.

[CE00] K. Czarnecki und U. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, 2000.

[CK02] P. Clements und C. Krueger. Point/Counterpoint: Being Proactive Pays Off/Eliminating the Adoption Barrier. *IEEE Software*, 19(4):28–31, 2002.

[CP06] K. Czarnecki und K. Pietroszek. Verifying Feature-based Model Templates against well-formedness OCL Constraints. In *Proc. Int'l Conf. Generative Programming and Component Engineering*, Seiten 211–220. ACM Press, 2006.

[CPR07] D. Coppit, R. Painter und M. Revelle. Spotlight: A Prototype Tool for Software Plans. In *Proc. Int'l Conf. on Software Engineering*, Seiten 754–757. IEEE CS Press, 2007.

[Gri00] M. Griss. Implementing Product-Line Features by Composing Aspects. In *Proc. Int'l Software Product Line Conference*, Seiten 271–288. Kluwer Academic Publishers, 2000.

[GSS+06] W. Griswold, K. Sullivan, Y. Song, M. Shonle, N. Tewari, Y. Cai und H. Rajan. Modular Software Design with Crosscutting Interfaces. *IEEE Software*, 23(1):51–60, 2006.

[JBZZ03] S. Jarzabek, P. Bassett, H. Zhang und W. Zhang. XVCL: XML-based Variant Configuration Language. In *Proc. Int'l Conf. on Software Engineering*, Seiten 810–811. IEEE CS Press, 2003.

[KA08] C. Kästner und S. Apel. Type-checking Software Product Lines – A Formal Approach. In *Proc. Int'l Conf. Automated Software Engineering*. IEEE CS Press, 2008.

[KAB07] C. Kästner, S. Apel und D. Batory. A Case Study Implementing Features Using AspectJ. In *Proc. Int'l Software Product Line Conference*, Seiten 223–232. IEEE CS Press, 2007.

[KAK08] C. Kästner, S. Apel und M. Kuhlemann. Granularity in Software Product Lines. In *Proc. Int'l Conf. on Software Engineering*, Seiten 311–320. ACM Press, 2008.

[KHH+01] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm und W. Griswold. An Overview of AspectJ. In *Proc. Europ. Conf. Object-Oriented Programming*, Jgg. 2072 of *LNCS*, Seiten 327–353. Springer-Verlag, 2001.

[KKB07] C. Kästner, M. Kuhlemann und D. Batory. Automating Feature-Oriented Refactoring of Legacy Applications. In *Poster at Europ. Conf. Object-Oriented Programming*, 2007.

[KKB08] C. Kim, C. Kästner und D. Batory. On the Modularity of Feature Interactions. In *Proc. Int'l Conf. Generative Programming and Component Engineering*. ACM Press, 2008.

[LHBC05] R. Lopez-Herrejon, D. Batory und W. Cook. Evaluating Support for Features in Advanced Modularization Technologies. In *Proc. Europ. Conf. Object-Oriented Programming*, Jgg. 3586 of *LNCS*, Seiten 169–194. Springer-Verlag, 2005.

[MO04] M. Mezini und K. Ostermann. Variability Management with Feature-Oriented Programming and Aspects. *SIGSOFT Softw. Eng. Notes*, 29(6):127–136, 2004.

[MP03] D. Muthig und T. Patzke. Generic Implementation of Product Line Components. In *Proc. Net.ObjectDays*, Seiten 313–329. Springer-Verlag, 2003.

[PBvdL05] K. Pohl, G. Böckle und F. van der Linden. *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer-Verlag, 2005.

[Pre97] C. Prehofer. Feature-Oriented Programming: A Fresh Look at Objects. In *Proc. Europ. Conf. Object-Oriented Programming*, Jgg. 1241 of *LNCS*, Seiten 419–443. Springer-Verlag, 1997.

[R+08] M. Rosenmüller et al. FAME-DBMS: Talor-made Data Management Solutions for Embedded Systems. In *Proc. EDBT Workshop on Software Engineering for Tailor-made Data Management*, Seiten 1–6. ACM Press, 2008.

[SC92] H. Spencer und G. Collyer. #ifdef Considered Harmful or Portability Experience With C News. In *Proc. USENIX Conf.*, Seiten 185–198, 1992.

[SvGB05] M. Svahnberg, J. van Gurp und J. Bosch. A Taxonomy of Variability Realization Techniques. *Software: Practice and Experience*, 35(8):705–754, 2005.

[TBKC07] S. Thaker, D. Batory, D. Kitchin und W. Cook. Safe Composition of Product Lines. In *Proc. Int'l Conf. Generative Programming and Component Engineering*, Seiten 95–104. ACM Press, 2007.

[TOHS99] P. Tarr, H. Ossher, W. Harrison und S. Sutton, Jr. N Degrees of Separation: Multi-Dimensional Separation of Concerns. In *Proc. Int'l Conf. on Software Engineering*, Seiten 107–119. IEEE CS Press, 1999.

Wiederverwendbare Generatorkomponenten in heterogenen Projektumfeldern

Norbert Klein, Robert Nikonowicz

Capgemini sd&m Research
Capgemini sd&m AG
Mühlheimer Str. 9a
53840 Troisdorf
{norbert.klein, robert.nikonowicz}@capgemini-sdm.com

Abstract: Capgemini sd&m setzt seit über 15 Jahren modellgetriebene Generatoren in seinen Projekten ein. Bisher wurden immer wieder neue Infrastrukturen aufgesetzt, Modellierungssprachen entwickelt und Generatoren gebaut. Jetzt entwickeln wir einen einheitlichen Modellierungs- und Generierungs-Ansatz. Dabei setzen wir auf die MDD-Werkzeuge im Umfeld der Eclipse-Plattform. Wir integrieren Werkzeuge wie Eclipse, openArchitectureWare, Maven2 und Enterprise Architect mit eigenen fertigen Generator-Komponenten (Cartridges) zu einem flexiblen und erweiterbaren MDD-Baukasten: dem Capgemini sd&m MDD-Toolkit. Die im MDD-Toolkit enthaltenen Cartridges können zu individuellen projektspezifischen Generatoren zusammengesetzt werden. Ein Projekt wählt die Cartridges aus, die die gewünschte Ziel-Plattform und Architektur unterstützen. Die Cartridges und der von ihnen generierte Code lassen sich weitgehend frei konfigurieren. In diesem Artikel beschreiben wir die Konzepte durch die wir Cartridges kombinierbar und konfigurierbar machen, wie Cartridges im MDD-Toolkit umgesetzt wurden und welche Erfahrungen erste Projekte mit dem MDD-Toolkit gemacht haben.

1 Einleitung

Model-Driven Development (MDD) ist ein Ansatz, der stetig wachsenden Komplexität heutiger Software-Systeme zu begegnen. MDD rückt Modelle in das Zentrum des Entwicklungsprozesses und hat zum Ziel, große Teile einer Anwendung zu generieren. Der Einsatz von Modellen und Generatoren ist nicht neu und beweist seinen Wert bei der Erstellung von Anwendungen bei Capgemini sd&m seit über 15 Jahren [Den93, Sch93].

Ein einheitlicher Ansatz zum Einsatz von MDD existierte bisher aber nicht. Modellierungssprachen und Generatoren wurden projekt- und kundenspezifisch immer wieder neu erstellt. Projektmitarbeiter benutzten immer gerade die Technologien, in denen sie sich auskannten. Da die Projekte von Capgemini sd&m sich sowohl in Fachlichkeit als auch in Technologie stark unterscheiden, war es nicht möglich, Generatoren oder Teile von ihnen mit vertretbarem Aufwand wiederzuverwenden.

Das Ziel von Capgemini sd&m Research ist es, dies zu ändern. Wir entwickeln eine ein-

heitliche Methode zum Einsatz von MDD, und schaffen einheitliche Werkzeuge zur Umsetzung von MDD. Die bei Capgemini sd&m am häufigsten eingesetzten Modellierungswerkzeuge Enterprise Architect und MID Innovator müssen dazu ebenso nutzbar sein, wie eventuell vom Kunden präferierte weitere Modellierungswerkzeuge.

Eine Sammlung von Bausteinen (Cartridges) soll es ermöglichen, Generatoren wie in einem Baukasten aus fertigen Teilen zusammenzustecken. Projekte sollen neue Cartridges selbst erstellen und sie in das MDD-Toolkit zurückfließen lassen können, damit diese zukünftigen Projekten zur Verfügung stehen. Die Cartridges müssen umfassend konfigurierbar sein, damit sie in der heterogenen Projektwelt von Capgemini sd&m bestehen können. Ein Software-Architekt soll den generierten Code seinen Vorstellungen und den Anforderungen des Projektes entsprechend anpassen können.

Existierende MDD-Werkzeuge versprechen einige der geforderten Punkte abzudecken. Die beiden Open-Source Toolkits openArchitectureWare (oAW siehe [OAW]) und AndroMDA [And] können beispielsweise sowohl mit Enterprise Architect als auch mit Innovator zusammenarbeiten. Im Umfeld der beiden MDD-Projekte existieren auch einige Cartridges für Java-Frameworks, wie Hibernate, Spring oder EJB. Die Sammlung der Cartridges für oAW ist aber eher spärlich und diese sind nicht aufeinander abgestimmt und nur schwer zu kombinieren. AndroMDA bietet eine umfangreiche Sammlung von Cartridges, die aber auch nicht immer einfach kombinierbar sind. Die Konfiguration der Cartridges und die Anpassung des generierten Codes ist bei beiden Toolkits nur sehr eingeschränkt möglich. Auch bei AndroMDA scheint man erkannt zu haben, dass Wiederverwendung und Anpassung von Cartridges ein wichtiges Thema sind und will dies in Version 4.0 mit einer grundlegend neuen Architektur verbessern [Boh06].

Die Lösung von Capgemini sd&m ist das MDD-Toolkit. Es bietet Methoden, Werkzeuge und Bausteine zum Einsatz von MDD.

Als gemeinsame Basis haben wir dazu die MDD-Werkzeuge der Eclipse-Plattform gewählt, da Eclipse die primäre Entwicklungsumgebung bei Capgemini sd&m ist. Wir stellen eine MDD-Eclipse-Distribution mit openArchitectureWare und allen benötigten Plugins zur Verfügung. Adapter zur Nutzung von Enterprise Architect und MID Innovator sind ebenfalls in der Distribution enthalten.

Einen Mehrwert bietet das MDD-Toolkit vor allem durch die enthaltenen Cartridges. Es bietet eine Auswahl an Cartridges, die die zur Generierung am besten geeigneten technischen Domänen [HSS05] abdecken.

Die Cartridges basieren auf einem Komponenten-Modell eines modularen Generators, das feste Schnittstellen für Cartridges definiert und dadurch eine Komposition der Cartridges ermöglicht.

Zur umfangreichen Konfiguration der Cartridges greifen wir neben Property-Dateien auf den Ansatz von [VG07] zurück, Generatoren über Aspektorientierung (AO) anzupassen. Die Kombination von Property-Dateien und AO, erfüllt alle Anforderungen an die Anpassungsfähigkeit einer Cartridge.

Neben den einheitlichen Schnittstellen vereinfacht auch die Nutzung eines gemeinsamen Architektur-Metamodells das Zusammenspiel von Cartridges. Basierend auf Quasar, der

Quality Software Architecture von Capgemini sd&m (vgl. [Sie04, Sie03]) haben wir das Metamodell einer Anwendung als Grundlage für alle Cartridges erstellt.

Wir beschreiben hier, wie wir diese Grundgedanken im Capgemini sd&m MDD-Toolkit umsetzen. Abschnitt 2 zeigt die Architektur des MDD-Toolkit und seiner Bausteine. In Abschnitt 3 stellen wir die konkrete Umsetzung der Konzepte vor.

2 Die Architektur des MDD-Toolkit

Das MDD-Toolkit bietet alle Komponenten einer MDD-Umgebung als fertige Bausteine an. Abbildung 1 zeigt die Referenz-Architektur einer solchen MDD-Umgebung.

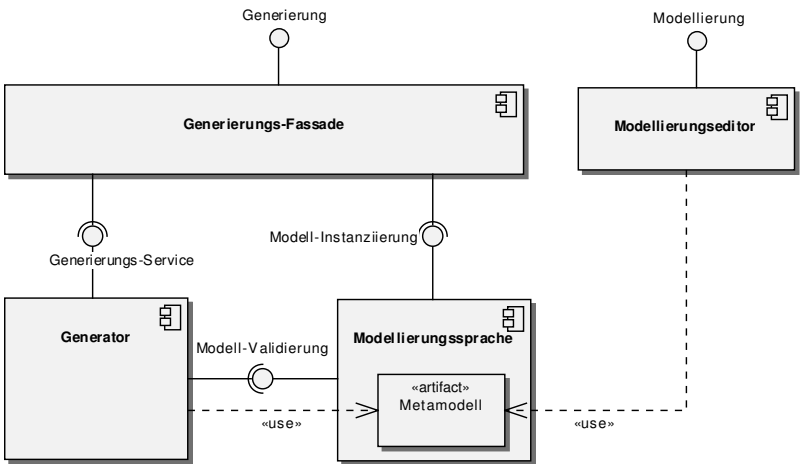


Abbildung 1: MDD-Referenzarchitektur

2.1 Generierungs-Fassade

Die Generierungs-Fassade ist der Berührungspunkt des Anwenders zur MDD-Umgebung. Sie abstrahiert Details von Modellierungssprache und Generator. Der Benutzer hat durch sie eine einfache Schnittstelle, über die er die Generierung anstoßen kann.

2.2 Modellierungssprache und Modellierungseditor

Eine Modellierungssprache definiert im MDD-Toolkit das Metamodell für den Generator und die Modellierungseditoren. Sie bietet auch Schnittstellen zur Instanziierung von auf dem Metamodell basierenden Modellen und zu deren Validierung. Das Metamodell formalisiert die modellierbaren Aspekte einer Anwendung. Das MDD-Toolkit enthält ein allen Cartridges gemeinsames in EMF-ecore realisiertes Metamodell der Quasar-Referenz-Architektur. Es stellt die zentralen Konzepte der Quasar-Referenz-Architektur wie Komponenten, Schnittstellen, Entitäten, Datentypen und Services als Modellierungs-Elemente bereit.

Der Modellierungseditor dient der Erstellung und Pflege der Modelle. Das MDD-Toolkit unterstützt zunächst die Modellierungswerkzeuge Enterprise Architect und MID Innovator. Die Modellierungseditoren benötigen eine externe Notation des Metamodells als Definition der konkreten Modell-Syntax. Dies geschieht im Fall von Enterprise Architect und Innovator durch UML-Profile. MDD-Toolkit liefert UML-Profile für die Bereiche Daten, Komponenten, Schnittstellen, Dienste und Remoting, sowie Modell-zu-Modell-Transformationen um Modelle aus dem Format des Modellierungswerkzeugs in Quasar-Metamodell-konforme Modelle zu überführen. Projekte können auch eigene UML-Profile definieren, die dann über selbstgeschriebene Modell-zu-Modell-Transformationen überführt werden müssen.

2.3 Generator

Der Generator erzeugt aus dem Modell die Ziel-Artefakte. Mit Hilfe des MDD-Toolkit werden Generatoren aus fertigen Cartridges zusammengebaut. Diese Cartridges enthalten jeweils für Teilaspekte der Anwendung die eigentliche Generierungsfunktionalität und werden vom Generator gesteuert.

Um solche Cartridges als Mittel zur Modularisierung und Wiederverwendung zu erstellen, werden sie im MDD-Toolkit als Komponenten mit festen Schnittstellen definiert (siehe Abbildung 2).

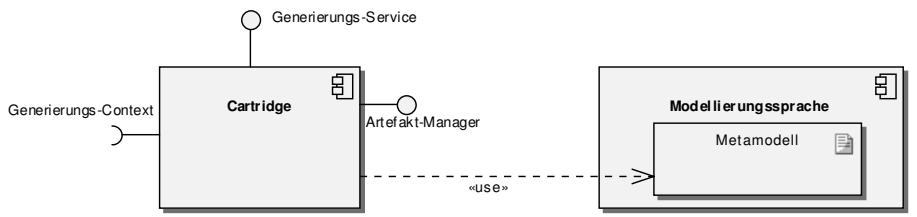


Abbildung 2: Cartridge als Komponente

Die Generierungs-Funktionalität wird über die Schnittstelle Generierungs-Service angeboten. Der Schnittstellen-Operation wird dabei das Modell übergeben. Die Schnittstelle Artefakt-Manager liefert Informationen über die generierten Artefakte, beispielsweise Referenzen auf Java-Klassen. Die Operationen der Generierungs-Context Schnittstelle definieren die Anforderungen der Cartridge an ihre Umgebung. Die Cartridge ist erst betriebsbereit, nachdem diese Schnittstelle versorgt ist. Es ist die Aufgabe des Cartridge-Nutzers, eine geeignete Implementierung bereitzustellen.

Die Schnittstellen Artefakt-Manager und Generierungs-Context stellen das wesentliche Mittel zur Komposition von Cartridges dar. Über Artefakt-Manager liefert eine Cartridge die Informationen, die eine andere Cartridge über die Schnittstelle Generierungs-Context anfordert.

Durch diese Architektur wird die Modularisierung und Wiederverwendung von Generatoren und Cartridges erreicht. Cartridges sind voneinander unabhängig und in unterschiedlichen Generatoren einsetzbar. Der Generator übernimmt die Kopplung der Cartridges (siehe Abbildung 3). Da er selbst als Cartridge gebaut wird, kann auch er in Kompositionen eingesetzt werden.

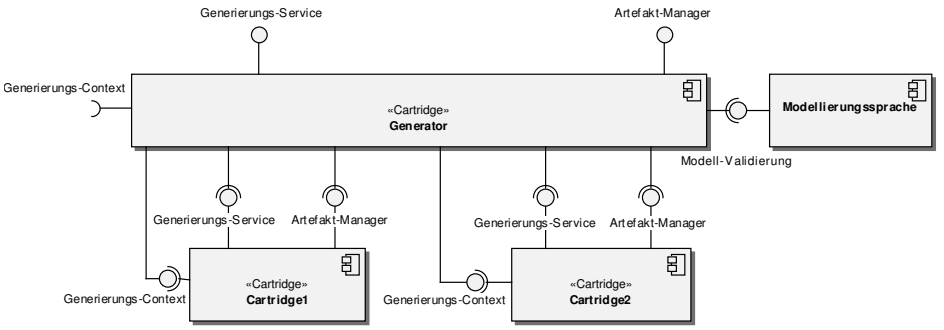


Abbildung 3: Der Generator als Komposition von Cartridges

3 Umsetzung von Cartridges mit openArchitectureWare

3.1 Komposition über Schnittstellen

Eine Cartridge muss die drei Schnittstellen Generierungs-Service, Artefakt-Manager und Generierungs-Context definieren. Leider bietet openArchitectureWare kein Sprachmittel zur Definition von Schnittstellen. Im MDD-Toolkit greifen wir daher auf die oAW-Mittel Workflow und Xtend zurück, um Schnittstellen zu realisieren. Durch ihre Package-Zugehörigkeit werden Workflows und Xtend-Funktionen als Teil der Schnittstelle gekennzeichnet. Abbildung 4 zeigt den Aufbau einer Cartridge mit oAW-Mitteln.

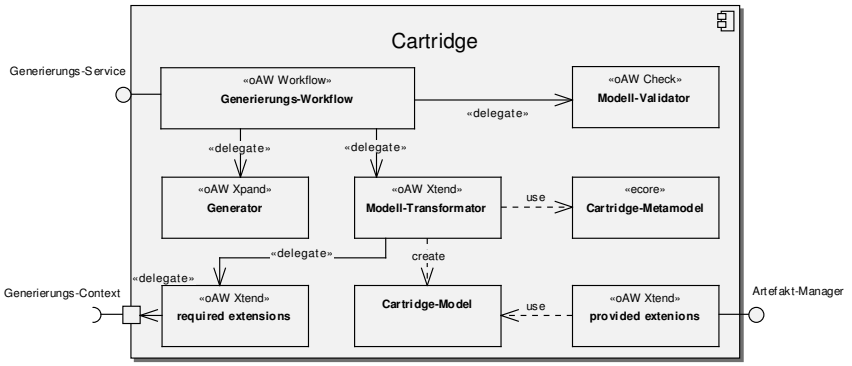


Abbildung 4: Cartridge - Innensicht

Die Schnittstelle Generierungs-Service wird als oAW-Workflow umgesetzt. Der Workflow validiert das übergebene Modell und stößt dann eine Transformation des Modells in ein cartridge-spezifisches Modell an. Dieses Modell basiert auf einem internen Metamodell der Cartridge, das stark an die generierten Artefakte angelehnt ist. Die Abbildung des internen Modells auf den generierten Code durch den Generator ist dadurch nahezu eine 1:1 Transformation. Die Xpand-Templates, die in oAW zur eigentlichen Generierung eingesetzt werden, können daher sehr einfach gehalten werden.

Zur Transformation dient in oAW die Sprache Xtend. Xtend ist eine funktionale Sprache mit einer an OCL angelehnten Syntax. Funktionalität, die die Transformation vom Kontext der Cartridge benötigt, wird als Xtend-Funktion ohne echte Implementierung definiert. Die so definierten Funktionen bilden die Generierungs-Context-Schnittstelle der Cartridge. So lange diese Funktionen nicht vom Cartridge-Nutzer mit einer echten Implementierung versorgt werden, bricht die Transformation in der Cartridge mit einer Fehlermeldung ab.

Die Schnittstelle Artefakt-Manager wird ebenfalls über Xtend-Funktionen realisiert und bietet Zugriff auf das interne Modell der Cartridge.

Die konkrete Umsetzung dieser Schnittstellen zeigen wir am Beispiel der Komposition von Data- und Hibernate-Cartridge im AWK-Generator. Der AWK-Generator ist ein fertiger Generator für die Capgemini sd&m Entwicklungsplattform [LNH07]. Er dient als Proof of Concept für die vorgestellten Konzepte und zeigt beispielhaft die Komposition von Cartridges. Abbildung 5 zeigt den Generierungsablauf des AWK-Generators.

Die Data-Cartridge generiert Entitäten, Datentypen und DataAcessObjects technologieneutral als reine Java-Klassen. Die Hibernate-Cartridge generiert anschließend die notwendigen Mapping-Dateien für Hibernate. Durch diese Trennung könnte die Hibernate-Cartridge in einem konkreten Projekt etwa durch eine Toplink-Cartridge ersetzt werden. Die von der Hibernate-Cartridge des MDD-Toolkits benötigten Informationen über die in der Data-Cartridge generierten Entitäten werden in ihrer Generierungs-Context-Schnittstelle folgendermaßen definiert:

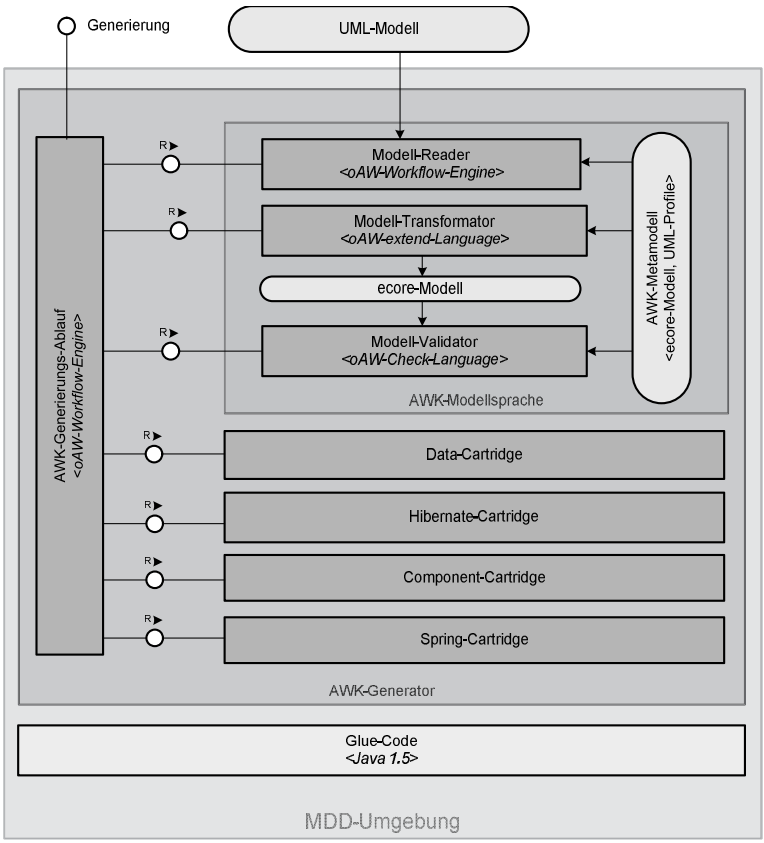


Abbildung 5: AWK-Generator Innensicht

```
/* resolve entity references */
cached String pltRef(Entity this):
                        unsupportedExtension("pltRef");
```

Die Standardimplementierung mittels `unsupportedExtension()` bricht die Transformation mit einer Fehlermeldung ab.

Um die Cartridge zu verwenden muss, der AWK-Generator eine andere Implementierung der Funktion einbinden. Dies geschieht über Aspect-Oriented Programming (vgl. [EFH+08] Kapitel 5.5.10 und 5.6.9). Im Generator wird für die Funktion `pltRef()` ein `around`-Advice angelegt. In diesem wird die Funktione durch den Aufruf der Xtend-Funktion `entityRef()` aus der Artefakt-Manager-Schnittstelle der Data-Cartridge implementiert. Diese liefert die notwendige Information zurück, so dass der Aufruf von `pltRef()` im Transformationsschritt der Hibernate-Cartridge die Referenz auf die gerade behandelte Entität liefert.

```
extension hibernate::gencontext;
extension data::artefactmanager;

/* resolve entity references */
around hibernate::gencontext::pltRef(Entity this):
                        entityRef(this.type);
```

Der AWK-Generator verknüpft also die Schnittstelle Artefakt-Manager der Data-Cartridge mit der Schnittstelle Generierungs-Context der Hibernate-Cartridge und komponiert diese zu einem lauffähigen Ganzen.

3.2 Anpassung des generierten Codes

Neben der Komposition der Cartridges ist der Generator auch dafür zuständig, den von den Cartridges generierten Code an projektspezifische Anforderungen anzupassen.

Zur einfachen Anpassung von Namen hat sich in den Projekten die Definition über Properties bewährt. Jede Cartridge definiert einen festen Satz von Properties mit Standardwerten. Diese können vom Generator überschrieben werden indem eine Datei mit Namen `global.properties` für die globalen Properties und eine cartridge-spezifische Datei (etwa `hibernate.properties`) im Klassenpfad abgelegt werden. Wird in der Datei `global.properties` die Property `pkg.base=com.capgemini-sdm` definiert, so beginnen die Packagenamen aller zu generierenden Artefakte mit dem Präfix `com.capgemini-sdm`.

Um Umfangreiche Änderungen am generierten Code durchzuführen, bietet oAW die Möglichkeit auch in ein Xpand-Generierungs-Template mittels AOP einzugreifen. Dies hat sich jedoch in der Beta-Phase des MDD-Toolkit nicht bewährt. Ein Eingriff mittels AOP benötigt detaillierte Kenntnis der Templates und damit der Interna einer Cartridge. Zusätzlich müssen die Templates für jeden angedachten Eingriff mittels AOP vorbereitet sein, wodurch die Templates unnötig aufgebläht werden.

Als wesentlich flexibler und sauberer herausgestellt hat sich die Möglichkeit, eine Schnittstelle zum internen Modell einer Cartridge zu bieten. Repräsentiert das interne Modell den generierten Code sehr nah, so sind die Templates nur noch eine 1:1 Übersetzung des Modells in den Code. Änderungen am Code können jetzt über Zugriff auf das interne Modell geschehen. Dazu wird als letzer Schritt der M2M-Transformation in einer Cartridge immer die Xtend-Funktion `postProcess()` mit dem internen Modell als Argument aufgerufen. Diese Funktion wird als weiterer Teil der Generierungs-Context-Schnittstelle definiert. Sie liefert in ihrer Standardimplementierung das Modell ohne Veränderung zurück. Um dieses Verhalten zu ändern, muss der Generator wieder AOP einsetzen und dadurch seine eigene Transformation in das Cartridge injizieren.

Diese Möglichkeit wird von Projekten ausgiebig genutzt, um Code zu verändern. Beispielsweise generiert das Data-Cartridge pro modellierter Entität genau ein DataAcessObject (DAO). In einem konkreten Projekt sollte aber pro Package nur ein DAO existieren,

das die Operationen für alle Entitäten des Packages enthält. Dies konnte durch eine zusätzliche Transformation des Cartridge-Modells verändert werden, die DAOs eines Packages wurden vor der eigentlichen Generierung zu einem DAO zusammengezogen.

Einige Transformationen konnten als typische Pattern für einzelne Cartridges erkannt werden. Sie werden im nächsten Release als Pattern-Transformationen in die Cartridges aufgenommen und in der Schnittstelle Artefakt-Manager als Xtend-Funktionen zur Verfügung gestellt.

4 Fazit

Capgemini sd&m Research ist es gelungen mit dem MDD-Toolkit einen Baukasten für die Nutzung modellgetriebener Entwicklung zu erstellen.

Wir stellen Modellierungssprachen, Modellierungseditoren, eine integrierte Generierungsumgebung, sowie Cartridges zum Bau von Generatoren zur Verfügung.

In der Version 1.1 vom Dezember 2008 enthält das Toolkit folgende Bausteine:

MDD-Eclipse-Distribution Eine Eclipse 3.3 Distribution mit allen benötigten MDD-Plugins, insbesondere oAW 4.3, EMF 2.3, M2Eclipse.

Maven-Repository Ein Maven-Repository mit allen benötigten Bibliotheken. Auch die im MDD-Toolkit enthaltenen Cartridges werden als Bibliotheken im Maven Repository zur Verfügung gestellt.

Integration Modellierungseditoren Adapter für die Modellierungswerkzeuge Enterprise Architect und MID Innovator.

Quasar-Metamodell Das Metamodell einer Anwendung basierend auf der Referenz-Architektur Quasar.

UML-Profile UML-Profile, die die Konzepte des Metamodells für die Modellierung in Enterprise Architect und Innovator bereitstellen.

Data-Cartridge Generiert Java-Klassen für fachliche Entitäten und Datentypen, sowie DAO-Klassen mit CRUD-Operationen.

Component-Cartridge Generiert die fachlichen Schnittstellen und den Code-Rahmen für die fachlichen Komponenten.

Hibernate-Cartridge Generiert das OR-Mapping für Hibernate-Persistence.

Spring-Cartridge Generiert Spring-Konfigurationen für die Komposition der fachlichen Komponenten zu einem Anwendungskern. Die Integration der fachlichen Komponenten mit den technischen Komponenten der Ziel-Plattform erfolgt ebenfalls über die generierte Spring-Konfiguration.

EJB2.1-Cartridge Generiert alle Java-Artefakte von Session-Beans, die Deployment-Deskriptoren und einen generischen Service-Locator zum Aufruf der Beans.

EJB3.0-Cartridge Generiert Java-Artefakte für Session-Beans und Entitäten mit Annotationen.

AWK-Generator Generator als Komposition mitgelieferter Cartridges, um alle architektonischen Aspekte eines Anwendungskerns zu generieren.

Durch die Komponentenarchitektur mit festen Schnittstellen können Cartridges beliebig kombiniert und wiederverwendet werden.

Der Chef-Architekt eines Projektes kann durch die weitgehenden Konfigurationsmöglichkeiten der Cartridges die generierten Artefakte anpassen, um an Kundenwünsche und Besonderheiten des Projektes angepasste Ziel-Plattformen und Architekturen zu unterstützen.

Das MDD-Toolkit wurde Ende Juni 2008 veröffentlicht und wird zur Zeit in 2 Großprojekten pilotiert. Ein weiteres Projekt setzt eine Vorabversion des AWK-Generator seit Ende 2007 ein.

Die Projekte waren in der Lage vorhandenen Cartridges, wie das Data-Cartridge und das EJB 2.1 Cartridge zu nutzen und für Ihre Zwecke zu konfigurieren. Eines der Großprojekte hatte die Designphase bereits abgeschlossen und besaß ein umfangreiches Designmodell in Enterprise Architekt (EA) mit eigenen UML-Profilen. Für dieses Projekt konnte die Modell-zu-Modell-Transformation im EA-Adapter innerhalb eines Tages so angepasst werden, dass das Modell zur Generierung nicht verändert werden musste.

Eines der Projekte benötigte für die Zielplattform EJB 3.0 eine neue Cartridge. Diese wurde mit 2 Bearbeiterwochen Aufwand erstellt und ist seit Release 1.1 fester Bestandteil des MDD-Toolkit.

In zukünftigen Versionen des MDD-Toolkits sollen weitere Cartridges ausgeliefert werden. Insbesondere werden zur Zeit für den Client-Anteil einer Anwendung Modellierungssprachen und Cartridges entwickelt, die im nächsten Release Mitte 2009 ausgeliefert werden.

Literatur

[And] AndroMDA. http://www.andromda.org.

[Boh06] Matthias Bohlen. The AndroMDA 4.0 architecture. http://galaxy.andromda.org/index.php?option=com_content&task=blogsection&id=10&Itemid=78, 2006.

[Den93] Ernst Denert. Dokumentenorientierte Software-Entwicklung. In *Informatik-Spektrum 16*, Seiten 159–164, 1993.

[EFH+08] Sven Efftinge, Peter Friese, Arno Haase, Clemens Kadura, Bernd Kolb, Dieter Moroff, Karsten Thoms und Markus Völter. *openArchitectureWare User Guide, Version 4.3*, 2004-2008.

[EFK08] Sven Efftinge, Peter Friese und Jan Köhnlein. Best Practices für modellgetriebene Softwareentwicklung. *Java Magazin*, 5 2008.

[HO07] Martin Haft und Bernd Olleck. Komponentenbasierte Clientarchitektur. In *Informatik-Spektrum, Band 30, Heft 3*. Gesellschaft für Informatik, Juni 2007.

[HSS05] Bernhard Humm, Ulf Schreier und Johannes Siedersleben. Model-driven development - hot spots in business information systems. In A. Hartmann und D. Kreische, Hrsg., *Proceedings of the European Conference on Model Driven Architecture (ECMDE-FA 2005), Lecture Notes in Computer Science 3748*, Seiten 103–114, Berlin Heidelberg, 2005. Springer-Verlag.

[Hum04] Bernhard Humm. Technische Open Source Komponenten implementieren die Referenzarchitektur Quasar. In Olaf Zukunft Helmut Eirund, Heinrich Jasper, Hrsg., *ISOS 2004 - Informationsysteme mit Open Source, Proceedings GI-Workshop*, Seiten 77–87, Berlin, 2004. Gesellschaft für Informatik.

[LNH07] Martin Lehmann, Robert Nikonowicz und Martin Haft. Eine Entwicklungsplattform für die architekturzentrierte Erstellung betrieblicher Anwendungssysteme auf Basis von Open-Source. In *GI Jahrestagung 2007, Workshop ISOS*, 2007.

[OAW] openArchitectureWare. The leading platform for professional model-driven software development. http://www.openarchitectureware.org.

[Sch93] Gero Scholz. Maßgeschneiderte Software-Generatoren. In *Proceedings ONLINE 1993 Congress VI - C636*, 1993.

[Sie03] Johannes Siedersleben. Quasar: Die sd&m Standardarchitektur. Teile 1 und 2. Bericht, sd&m Research, 2003.

[Sie04] Johannes Siedersleben. *Moderne Softwarearchitektur*. dpunkt Verlag, Bonn, 2004.

[SVE07] Thomas Stahl, Markus Völter und Sven Efftinge. *Modellgetriebene Softwareentwicklung. Techniken, Engineering, Management*. d.punkt Verlag, 2. Auflage, Mai 2007.

[VG07] Markus Völter und Iris Groher. Product Line Implementation using Aspect-Oriented and Model-Driven Software Development. In *11th International Software Product Line Conference*, Seiten 233–242. IEEE Computer Society, 2007.

Qualitätssicherung

Using Template Analysis as Background Reading Technique for Requirements Elicitation

Sebastian Stein, Yves Lauer, Marwane El Kharbili
IDS Scheer AG
Altenkesseler Str. 17
66115 Saarbrücken
Germany
sebastian.stein, yves.lauer, marwane.elkharbili@ids-scheer.com

Abstract: Requirements cannot be collected, but must be elicited from people's tacit and systems' embedded knowledge. It is a proven approach to use existing systems and manuals as a source. Literature suggests using background reading to elicit requirements from domain descriptions and manuals. Besides content analysis, no concrete technique describing how to actually conduct background reading is available in literature. In this paper, we evaluate the usage of template analysis as a technique for background reading. We applied template analysis in a project to extract requirements from 35 success stories about process performance management. We found template analysis to be very useful not just for eliciting requirements and creating a shared understanding of the studied domain, but also for helping new employees to get familiar with the domain. We also formulated competence questions to document and communicate requirements, but this did not prove helpful and hence we would not recommend it.

1 Introduction

Today, it is a well-known fact that carefully defining and managing requirements is a fundamental base for successful software engineering projects [DC06]. In the past decades, requirements engineering has evolved from being a sub-discipline of software engineering into a well established research discipline. Standard requirements engineering literature like [KS04, Dav93] shows that requirements engineering comprises many different aspects such as the requirements engineering process, elicitation and analysis, validation, change management, viewpoint analysis, formal specification, etc. Not surprisingly, research contributions usually focus on a specific aspect of requirements engineering. This paper focuses on requirements elicitation.

Requirements elicitation is an early activity in the requirements engineering process aiming at acquiring and understanding requirements. Usually, this is not a task of just collecting clearly articulated requirements, but instead intensive work is needed to transform embedded knowledge into explicit one. There are many possible knowledge sources like human beings (domain experts, current users, stakeholders, etc.) but also artefacts like manuals, domain descriptions, laws and regulations, legacy system, and competing soft-

ware products [KS04, Dav93]. Depending on the knowledge source, different requirements elicitation techniques are applicable.

Requirements elicitation consists of two phases [Zha07]: problem analysis and product specification. In the problem analysis phase, the requirements engineer creates an understanding of the problem domain and the context, whereas in the product specification phase concrete requirements for the product to be developed are defined. Existing requirement elicitation techniques can be grouped in four categories [Zha07]: conversational methods (e. g. interview), observational methods (e. g. ethnography), analytic methods (e. g. background reading), and synthetic methods (e. g. prototyping).

Byrd et al. [BCZ92] provide a similar categorisation, but they do not just cover requirements elicitation techniques but also include knowledge acquisition techniques used in knowledge engineering. They show that a great similarity of techniques exist in both domains. Zhang [Zha07] extends this work by creating a selection matrix supporting requirements engineers in selecting an appropriate requirements elicitation technique. For example, if no access is given to domain experts, conversational methods cannot be applied. However, the selection matrix by Zhang is not based on empirical findings.

In this paper we present a technique to acquire requirements from existing texts. We use the technique template analysis to analyse texts and we document the acquired requirements by formulating competence questions. We show that template analysis is a promising background reading technique, but that the procedure of formulating competence questions was not very helpful. We applied template analysis in a public funded research project in order to document current requirements of users and customers in process performance management projects.

This paper is structured as follows: In the following section we discuss other background reading techniques and we show why we selected template analysis in our case. Sect. 3 describes the research material used and the research context where we conducted the study. In Sect. 4 we first give a brief overview of template analysis, show how we have done it, how we analysed the results, and how we used competence questions to document the requirements extracted. We provide a comprehensive discussion in Sect. 5 about the lessons learnt. The discussion will be useful for other requirements engineers to judge if template analysis is applicable in their specific context. The paper is shortly summarised at the end.

2 Background and Related Work

In a recent research effort, we wanted to document typical requirements requested in performance management projects. We had no direct access to domain experts, but we had a corpus of 35 success stories from past projects. Using the selection matrix [Zha07], we decided to use an analytical method, namely background reading. Using existing documents and manuals as a source for requirements elicitation is an accepted approach discussed or at least listed by many authors [KS04, Zha07, BCZ92]. Different terms are used for such techniques like background reading, document analysis or documentation studies. Back-

ground reading is not just used to acquire requirements, but also to understand the overall culture in an organisation or project [Zha07].

Even though many authors point to the value of background reading and document analysis, Rayson et al. [RGS99] show that there are neither specific techniques nor instructions documented or even evaluated in literature. Therefore, they evaluate the usage of natural language processing (NLP) techniques. In NLP, texts are analysed by linguistic algorithms to extract common themes and topics. Ideally, no human interaction is needed. Using NLP techniques to extract knowledge from existing documents is a common approach in other disciplines as well. For example, Aussenac-Gilles et al. [AGBS00] demonstrate the usage in the ontological engineering research domain. Here, it is important not to confuse this with approaches like Fliedl et al. [FKM+07], who are using NLP techniques to analyse requirements documents in contrast to gathering requirements. NLP is a good approach if lengthy texts are available. This was not the case, because we only had 35 success stories. Therefore, we could not apply NLP techniques.

Besides NLP based techniques, we also found publications using content analysis [Kri03] as a reading technique. For example, Shea and Exton [OE04] use content analysis to extract requirements for a software visualisation tool from a set of texts like bug reports. They use predefined categories. They analyse the results through a quantitative analysis measuring how often a certain category is used. Content analysis is a good requirements elicitation technique if a set of categories can be defined prior analysing the actual texts. This requires a certain degree of understanding of the domain to be studied, but it is not applicable in case the texts should be used to get an understanding of the domain and to elicit requirements. We had no prior understanding of the domain covered by the success stories and we could not create a categorisation. Therefore, we could not apply content analysis. We looked for another technique, which is similar to content analysis, but does not require a fixed categorisation. We found template analysis [Kin98], which we are evaluating in this paper.

Our research partners asked us to document the acquired requirements as competence questions. Competence questions [MI96] are either literal questions like "How often was process XYZ executed?" or formal ones like formulating WSML queries. The questions are used during ontology development. The created ontology is checked whether it can be used to answer the competence questions defined before. It was the task of our research partners to create an ontology based on the requirements we gathered from the success stories. Therefore, we decided to try documenting the requirements as competence questions. During this paper, we also reflect on this usage of competence questions for requirements documentation.

3 Use-case and Research Material

The work presented here is part of the research project SUPER[1]. The SUPER research project investigates how semantic technologies like ontologies and reasoners can be used

[1] http://www.ip-super.org/

in business process management (BPM) [SS08]. An important aspect of BPM is monitoring and analysing existing IT systems and executed processes. This is known as process performance management. The SUPER research project investigates how semantic technologies can support process performance management [CAdMZ+07, AdMPvdA+07]. However, just focusing on what might be possible in process performance management using semantics does not automatically guarantee the gain of useful research insights. Therefore, it is important to take into account current user requirements and what is already possible with today's technologies, so that the state of the art can be extended and not just reinvented using another technology.

We, as a vendor of BPM software were asked to provide an overview of today's problems tackled within process performance management projects. Our company is doing such projects since several years all over the world. Fortunately, some of the past projects are documented as marketing oriented success stories. Each success story consists of few presentation slides describing the initial situation, the customer, the customer problem, the applied solution, and the benefits gained. The success stories are created by project leaders or other project members, both having extensive domain knowledge. At the time this research was conducted, we had access to 35 success stories from different industrial domains like chemical/pharma, finance, automotive, logistics, insurance, capital goods, telecommunications, and utilities. The success stories are written by different individuals and as they are meant to be marketing tools, they are written bluntly. Therefore, the material is too subjective and weak to be used in any quantitative study. Also, there is too little text to apply NLP. Still, the success stories contain valuable domain knowledge, which we want to make explicit. The success stories are not available to the public. The following section details how we applied template analysis to extract requirements from the success stories and how we used competence questions to document the requirements.

4 Applying Template Analysis

4.1 Overview Template Analysis

This section provides a short introduction to template analysis and demonstrates how we applied template analysis to acquire requirements from the success stories. Template analysis[2] [Kin98] is a qualitative research method for analysing research material given as texts (i. e. the corpus). Codes are terms representing possible themes in the corpus. A code template consists of a set of codes. The person performing template analysis is called coder.

Template analysis is an iterative process. The first step in template analysis is creating the corpus. Afterwards, the coder has to get familiar with the text by reading it. Next, the code template is created by the coder. The first version of the code template should be based on the coder's understanding or perception of the studied domain. Usually, the coder uses background knowledge of theories he expects to be present in the corpus. Another possible

[2] http://www.hud.ac.uk/hhs/research/template_analysis/index.htm

source is existing classifications or taxonomies. A code template can be a plain list of terms or a hierarchy of terms. Afterwards, the corpus is analysed by one or more coders who read the text and assign codes from the code template to segments of the corpus. While coding the corpus, the code template is reworked to better reflect the content. Creating the code template and coding the corpus is an intertwined iterative process. The code template is reworked several times, new codes get introduced, and other codes are removed or split into more detailed codes. This is a major difference compared to content analysis [Kri03], because in content analysis the categories (code template) are fix.

After some time, usually after coding one third of the corpus, the code template stabilises. After having a stable code template, the actual coding starts. Each coder marks segments of the corpus and assigns codes from the code template to the marked segments. It is possible to assign multiple codes to one segment. Segments can be freely selected, so one word can belong to multiple segments. Coding can be done manually using for example a printout of the corpus and marking segments with different colours, where each colour represents a code. There are also different software tools available for this task. Such tools are usually called qualitative data analysis (QDA) tools.

After coding the whole corpus, coders have a better understanding of the content. This understanding can be further extended by sharing the coding results, for example in a group discussion. It is also common to reflect on the coding process itself and how the own perception of the domain changed while coding. In summary, template analysis is a tool for sense-making. It is not meant to cluster the text according to a predefined set of criteria. Instead, template analysis is a tool to work with the text to understand it and gain insights. It is not the aim of analysing the coding results to judge if coding was done correctly. For example, if a coder applied a code to completely different segments than all other coders, it does not mean he did the coding wrong. Instead, it should be discussed why the coder used the code in a different way. Such discussions usually challenge the current individual perception and help to better understand the corpus.

4.2 Application of Template Analysis

We started doing template analysis with two people. We both have profound knowledge in BPM, but only initial insights into process performance management. We have not done any process performance management projects nor used the specific software provided by our company. First, we copied the content of the different success stories into one ASCII text file removing sections like customer descriptions. We created an initial version of the code template using general textbooks about service level agreements (SLA) and key performance indicators (KPI). We expected those topics to be main themes in process performance management projects. We structured the code template around the core terms *costs*, *time*, and *quality*. Beneath those terms, we formulated concrete codes like *cost reduction*, *defect reduction* or *quality improvement*. Both of us created the code template individually. Afterwards, we merged the two code templates after discussing and clarifying the scope of each code. We learnt that we both had different ideas and expectations about process performance management projects.

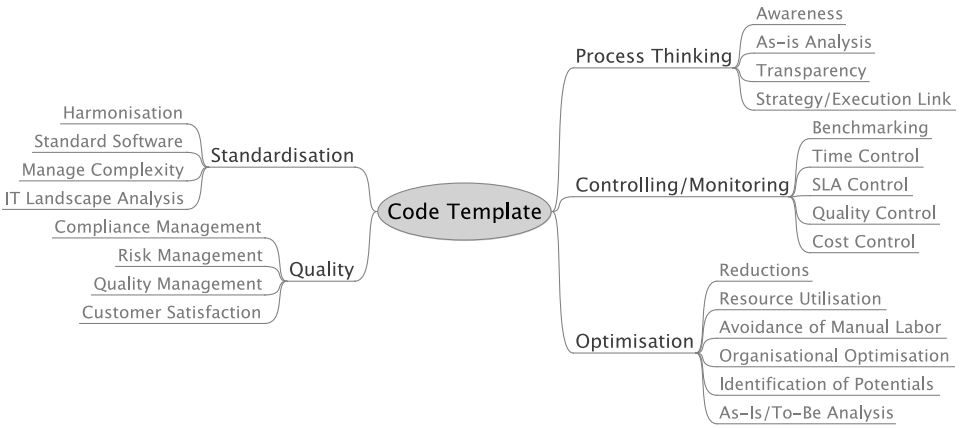


Figure 1: Final code template

We did a test coding with the initial version of the code template. Testing the code template revealed that the initial code template was not reflecting very well the content of the corpus. For example, many customers wanted to stimulate process awareness among their employees by mining and documenting existing processes. Therefore, we completely restructured the code template. The final version of the code template is shown in Fig. 1. We only used leafs for coding. Before we started coding, we created a textual description for each category and each code. We discussed this description to ensure that we have a similar understanding of the codes' meanings. We can confirm that the code template is not just defined at the beginning but revised several times as the coder gets a better understanding of the studied texts.

After having a stable code template, both of us coded the whole text. Coding the whole text took one day for each of us. We used Weft QDA[3] for coding, because it is freely available. Unfortunately, Weft QDA does not support multiple coders. Therefore, each coder had to copy the database containing the text and code template. Having several databases at the end makes comparing the results much harder, because inter-coder analysis is not supported by the tool but instead must be done manually outside of Weft QDA. However, this is not an inherent problem of template analysis, but just related to the tool we used.

A third researcher joined after both of us had finished coding. We wanted to test what happens if a coder has to use an existing code template. First, the new coder read the code template with the corresponding textual descriptions. Afterwards, we discussed with the coder the meaning of the different codes and how he understood the code template. This already revealed that he had thought of additional things, which were not present in the code template. Afterwards, he performed the coding using the unchanged code template. We did not change the code template, so that the two of us, who already finished coding, do not had to do the coding again. We included the results of the new coder in the overall analysis process. The following subsection describes how we analysed the coding results.

[3] http://www.pressure.to/qda/

4.3 Results of Template Analysis

After all coders had coded the whole text, we had three Weft QDA databases containing the results. Weft QDA has some basic analysis and reporting functionality, but which was not sufficient in our case. Therefore, we did the actual analysis outside of Weft QDA.

We created a list for each code containing the text segments marked with this code by each coder. We used those lists to discuss the outcome. For example, the discussion revealed that we had a different interpretation of some of the codes. This was not avoidable, even though we had one common textual description for each code. We were able to learn about why a certain coder applied a code. Having coded the text differently was therefore a good way to initiate a discussion between the participants and to strengthen the understanding. We also learnt that some codes were not applied very often, because the software provided by our company only supports those use-cases partially and therefore such use-cases are not within the focus of customers buying our software and services. It is important to understand that template analysis is not finished when the coding is complete. Instead, discussing the results between the different coders is an important part of template analysis. Template analysis is done to work with the text, but not to manually cluster the text according to the code template. This also means that if a certain code is only used a few times, this code can still represent a major aspect.

4.4 Documenting Requirements as Competence Questions

As discussed before in Sect. 2, formulating competence questions is a common approach in ontology engineering. To support our research partners and to communicate the insights we gained by analysing the success stories, we formulated a set of competence questions. For each code we looked at the text segments marked with the code and transformed the text segments into questions. For example, we created the following five competence questions for the code *as-is analysis*:

- How is process XYZ looking like?
- Can we extract the executed processes from our running IT systems?
- Where do we have weak spots and bottlenecks?
- Which processes are used/executed very often?
- What is our organisational structure?

We formulated the questions from the viewpoint of the users based on the corpus. We did not create one question for each marked text segment, but tried to summarise several coded text segments in one question. For example, the first question above is based on the following three original text segments marked with the code *as-is analysis*:

- “Process analysis of as is HR-processes”

- "visualize the as-is-processes in different regions"
- "Documentation of lived SAP-SD processes"

The example competence questions given above show that it is impossible to simply answer them using some kind of software. It was an important outcome to demonstrate to our research partners that many questions raised in process performance management projects require intensive consulting services and not just software. However, the initially formulated competence questions were too abstract to be useful for our research partners, because they it was their task to create a formal description of the domain by means of an ontology. The questions provoked a discussion among the partners about the nature and scope of process performance management projects, but they could not be used to judge how semantic technologies can be applied in this domain. Therefore, we refined the questions further to provide a more detailed view by interpreting the statements given in the success stories. The limitations of this refinement are discussed in Sect. 5. The following list shows some of them for the code *as-is analysis*:

- What data do we use in our business processes?
- What processes use what data?
- What is the configuration of my business process execution system?
- Can we have a global schematic map of the organisations/roles involved in processes/activities?
- Can we view a dashboard of the number of instances of each process over a period of time?
- Can we view a dashboard of the frequency of execution of each process over a period of time?

5 Discussion

5.1 Template Analysis as Requirements Elicitation Technique

In the past sections, we presented how we used template analysis as a background reading technique for requirements elicitation. In this section, we reflect on this usage of template analysis. This discussion also includes the usage of competence questions to define and communicate the requirements extracted.

We used template analysis as a tool to analyse an existing comprehensive set of texts. The analysis was done in a small team of three people. We noticed that applying template analysis requires a certain degree of open-minded thinking and a high level of collaborative disposition. It is not the aim of template analysis to prove or disprove a certain

expectation, but instead to develop the perception of the participants. Reworking and discussing the code template is an important element of template analysis enabling creative thinking. For example, in our case study a third coder joined at a later point. Adding his view to the analysis showed us some shortcomings in our code template. The resulting discussions helped all participants to better understand the corpus. We also used basic statistical analysis to visualise different understandings and perceptions.

Using a tool like Weft QDA for coding the text allows having a direct link between the codes and the coded segments. This ensures that one can easily look up to which text segments a code was applied to. Even though this allows an easy navigation between source and result, it is unclear how to integrate such a qualitative data analysis tool with existing requirements management tools.

If we compare our usage of template analysis to previous usages by other researchers, we can see that there are different implementations of the approach. For example, Carey et al. [CMO96] also had several coders, but they were trying to do template analysis in a much stricter way. They quantitatively tested the code template and also made a detailed statistical assessment about the application of the code template in the coding process. We did not have such a formal approach to template analysis. Instead, we tried to use it more as a creative thinking technique to provoke discussions and disagreements between the participants.

5.2 Template Analysis, NLP and Content Analysis

Template analysis and NLP are very different approaches to do requirements elicitation on existing texts. NLP is meant to be an automated approach creating a knowledge base, which can be used by a requirements engineer. In contrast, template analysis is a manual approach, where requirements engineers create a knowledge base according to their own needs. As a result, requirements engineers gain a better understanding of the domain. Template analysis enables collaboration between several people. We expect template analysis to be easier to use, because no complicated installation and setup of software is required. We also expect that it is easier to understand the results and how they were created, because explaining and understanding NLP is not an easy task. On the other hand, template analysis does not scale very well with the corpus length and number of coders involved. Each coder has to be familiar with the whole text. In case of very long texts, this can be almost impossible and it may take too much time and resources. As both techniques are very different, we do not expect a competition between both. Instead, both techniques might be combined. For example, NLP can be used to create an initial version of the code template by analysing the text for common terms. Such a generated code template might be more accurate in reflecting the content of the text and hence provide a better starting point for the coders continuing with template analysis.

Content analysis is more similar to template analysis than NLP. Content analysis is usually used to analyse communication like interview transcripts, but it can also be applied to other texts. In contrast to template analysis, content analysis uses a fixed set of categories.

Content analysis has a very strong focus on providing quantifiable results through statistical analysis. We expect content analysis to be a better background reading technique than template analysis if a) the requirements engineers are already familiar with the domain and b) quantifiable results are needed. Both conditions were not given in our case.

5.3 Additional Advantages and Disadvantages of Template Analysis

A big advantage of using template analysis is that it is easy to learn and teach. After a third coder joined, we gave him a short introduction how to do template analysis and provided the available literature. This small training was enough that he could directly start coding the corpus and discuss with us the results.

Creating and changing the code template is easy for someone able to do requirements analysis. A high acquaintance with conceptual and abstract thinking is required. On the other hand, no special knowledge like computer science or qualitative research background is needed. This ensures that very different people can do template analysis. Having people with different know-how will even strengthen the result, because more perceptions of the domain are included and mixed. All of us had no specific background in the domain of process performance projects. Still, we were able to perform the analysis and we gained a valuable overview. All of us confirmed that template analysis and especially the coding step is an exhaustive task. One has to concentrate very much to use the codes always in the same way and not to change the meaning of codes while coding. This may happen, because while coding the own view on the domain changes. Here, we found a glossary to look up the meaning of each code to be very useful.

Maybe one of the biggest disadvantages is the time needed to carefully do template analysis. One has to read the corpus several times to develop the code template. Also, coding a text takes time and splitting this time over several days is not easy. During coding it is important to have a quiet surrounding. The best results are achieved if several people participate. However, this can be a problem, since people tend to be busy and thus booking meetings for group discussions can be a tricky task. If such constraints exist, template analysis might not be the technique to choose.

5.4 Competence Questions as Requirements Documentation

Besides using template analysis, we also evaluated the idea of formulating competence questions to document and communicate the results to our research partners. Formulating the competence questions was challenging for us. We felt it was an artificial approach. We often just turned a customer statement like “I want to extract my running processes!” around into questions like “Can I extract my running processes?” It is unclear to us if we really gained anything by doing that. Transforming statements found in the corpus into questions involves a reinterpretation of the person doing the transformation. We are not convinced that this helps in clearly defining requirements or if it just adds another level of

indirection.

In the end, the competence questions provided to the other research partners were not detailed enough to be useful for them. We had to refine the questions by providing more detailed ones. This step is hard to justify, because such detailed requirements are not supported anymore by the content of the success stories. Therefore, we would not suggest using competence questions to document the understanding gained by template analysis, unless the competence questions are defined by domain experts. If we take the requirement abstraction model [GW06] into account, we can say that template analysis is a valuable technique to deliver requirements on the product or feature level, but not to provide requirements needed on the function or component level. However, this might also depend on the source material like how detailed the textual descriptions are. A final judgement is not possible at this point and further evaluations are needed.

6 Summary

In this paper, we evaluated template analysis as a background reading technique for requirements elicitation. We also formulated competence questions to communicate the requirements found. Template analysis proved to be successful in our case study, but formulating competence questions was not helpful. To better understand the conditions under which template analysis can be used as a background reading technique, further case studies and experiments are needed. It is also important to compare template analysis to other requirements elicitation techniques. If possible, we will use template analysis again and evaluate it. However, we also encourage other practitioners and researchers to use template analysis in future requirements elicitation efforts and to report on their experiences.

References

[AdMPvdA+07] A. K. Alves de Medeiros, C. Pedrinaci, W. M. P. van der Aalst, J. Domingue, M. Song, A. Rozinat, B. Norton, and L. Cabral. An outlook on semantic business process mining and monitoring. In *On the Move to Meaningful Internet Systems 2007: OTM 2007 Workshops*, volume 4806 of *LNCS*, pages 1244–1255, Vilamoura, Portugal, November 2007.

[AGBS00] N. Aussenac-Gilles, B. Biébow, and S. Szulman. Revisiting Ontology Design: A Method Based on Corpus Analysis. In *Knowledge Engineering and Knowledge Management. Methods, Models, and Tools: 12th International Conference (EKAW)*, volume 1937 of *LNCS*, pages 27–66, Juan-les-Pins, France, October 2000.

[BCZ92] T. A. Byrd, K. L. Cossick, and R. W. Zmud. A Synthesis of Research on Requirements Analysis and Knowledge Acquisition Techniques. *MIS Quarterly*, 16(1):117–139, 1992.

[CAdMZ+07] I. Celino, A. K. Alves de Medeiros, G. Zeissler, M. Oppitz, F. Facca, and S. Zöller. Semantic Business Process Analysis. In *Workshop on Semantic Business Process*

and Product Lifecycle Management (SBPM), volume 251 of *CEUR Workshop Proceedings*, pages 44–47, Innsbruck, Austria, June 2007.

[CMO96] J. W. Carey, M. Morgan, and M. J. Oxtoby. Intercoder agreement in analysis of responses to open-ended interview questions: Examples from tuberculosis research. *Cultural Anthropology Methods Journal*, 8(3):1–5, 1996.

[Dav93] A. M. Davis. *Software Requirements: objects, functions and states*. Prentice Hall, New Jersey, USA, 1993.

[DC06] D. Damian and J. Chisan. An Empirical Study of the Complex Relationships between Requirements Engineering Processes and Other Processes that Lead to Payoffs in Productivity, Quality, and Risk Management. *IEEE Transactions on Software Engineering*, 32(7):433–453, 2006.

[FKM+07] G. Fliedl, C. Kop, H. C. Mayr, A. Salbrechter, J. Vöhringer, G. Weber, and C. Winkler. Deriving static and dynamic concepts from software requirements using sophisticated taggingstar, open. *Data & Knowledge Engineering*, 61(3):433–448, 2007.

[GW06] T. Gorschek and C. Wohlin. Requirements Abstraction Model. *Requirements Engineering*, 11(1):79–101, 2006.

[Kin98] N. King. Template Analysis. In Gillian Symon and Catherine Cassel, editors, *Qualitative Methods and Analysis in Organizational Research: A Practical Guide*, pages 118–134. Sage Publications, London, UK, 1998.

[Kri03] K. Krippendorff. *Content Analysis: An Introduction to Its Methodology*. Sage Publications, London, UK, 2nd edition, 2003.

[KS04] G. Kotonya and I. Sommerville. *Requirements Engineering: Processes and Techniques*. Wiley, Chichester, UK, 2004.

[MI96] R. Mizoguchi and M. Ikeda. Towards Ontology Engineering. Technical Report AI-TR-96-1, I.S.I.R., Osaka University, 1996.

[OE04] P. O'Shea and C. Exton. The Application of Content Analysis to Programmer Mailing Lists as a Requirements Method for a Software Visualisation Tool. In *12th International Workshop on Software Technology Practice (STEP)*, pages 30–39, Los Alamitos, CA, USA, 2004. IEEE CS.

[RGS99] P. Rayson, R. Garside, and P. Sawyer. Language engineering for the recovery of requirements from legacy documents. Technical report, REVERVE project report, Lancaster University, May 1999.

[SS08] H. J. Schmelzer and W. Sesselmann. *Geschäftsprozessmanagement in der Praxis*. Carl Hanser Verlag, München, Germany, 6th revised edition, 2008.

[Zha07] Z. Zhang. Effective Requirements Development – A Comparison of Requirements Elicitation techniques. In *INSPIRE2007*, Tampere, Finland, 2007.

Testfokusauswahl im Integrationstestprozess

Lars Borner, Barbara Paech

Lehrstuhl für Software Systeme
Universität Heidelberg, Institut für Informatik
Im Neuenheimer Feld 326
69120 Heidelberg
borner@informatik.uni-heidelberg.de
paech@informatik.uni-heidelberg.de

Abstract: Im Integrationstestprozess werden Bausteine eines Softwaresystems schrittweise zusammengesetzt und deren Abhängigkeiten untereinander getestet. Die große Anzahl Abhängigkeiten heutiger Systeme macht es unmöglich, alle Abhängigkeiten zu testen. Tester müssen sich daher auf die fehlerhaften Abhängigkeiten fokussieren. In diesem Beitrag stellen wir einen Ansatz vor, der Informationen über Abhängigkeitseigenschaften nutzt, um den Testfokus für den Integrationstest festzulegen. Der Ansatz umfasst eine Liste von Eigenschaften, um Abhängigkeiten zu klassifizieren. Darüber hinaus gibt er eine Methode vor, um mit Hilfe von Fehlerdaten früherer Versionen eines Softwaresystems sowie deren Eigenschaften fehlerhafte Abhängigkeiten zu identifizieren. Evaluiert wurde der Ansatz mit Hilfe der Entwicklungsumgebung Eclipse.

1 Einleitung

Softwaresysteme bestehen heutzutage aus einer großen Anzahl von (Software-) Bausteinen[1], die unzählige Anforderungen realisieren. Um diese Anforderungen zu erfüllen, müssen die realisierenden Bausteine miteinander interagieren oder sind voneinander abhängig. Das Überprüfen der Abhängigkeiten zwischen Bausteinen erfolgt innerhalb des Integrationstestprozesses. Ziel ist es, „… Fehler in Schnittstellen und im Zusammenspiel zwischen integrierten Komponenten[2] zu finden“ ([SL06], Seite 202). Nachdem die einzelnen Bausteine im Komponententest getestet wurden, werden sie im Integrationstest schrittweise zu einem Gesamtsystem zusammengesetzt und die Abhängigkeit zwischen ihnen getestet. Der Fokus liegt dabei auf typischen Fehlern im Zusammenspiel der Bausteine. Anschließend kann der Systemtestprozess durchgeführt werden, um die korrekte Umsetzung der Anforderungen zu überprüfen.

Innerhalb des Integrationstestprozesses werden verschiedene Entscheidungen getroffen (vgl. [BIP07]). Eine der wichtigsten Entscheidungen ist die Auswahl des Testfokus, d.h. die Auswahl der zu testenden Abhängigkeiten. Die Festlegung des Testfokus ist

[1] Ein Baustein repräsentiert einen Teil eines Softwaresystems (z.B. eine Klasse, eine Modul, eine Komponente, ein Teilsystem, eine Datei, …), der von anderen Teilen abhängig ist bzw. von dem andere Teile abhängig sind.
[2] In diesem Fall synonym mit dem Begriff Baustein

notwendig, da vollständiges Testen eines Softwaresystems nicht möglich ist (vgl. [Me79]). Diese Entscheidung schränkt die Anzahl der zu testenden Abhängigkeiten auf die vermutlich fehlerhaften ein. In unserer Arbeit wird eine Abhängigkeit als *fehlerhaft bezeichnet*, wenn mindestens einer der beteiligten Bausteine eine hohe Fehleranzahl aufweist, da das Fehlschlagen einer Abhängigkeit immer durch Fehler in den beteiligten Bausteinen verursacht wird. In der Literatur über den Integrationstest sind nur wenige Informationen über die Testfokusauswahl für den Integrationstest zu finden. Sneed und Winter verweisen in [SW02] auf eine Strategie zur Ermittlung der Integrationsreihenfolge auf Basis der Kritikalität der Bausteine. Ein Vorgehen, wie diese Kritikalität ermittelt werden kann, wird jedoch nicht beschrieben. Ansätze zur Fehlervorhersage können diese Lücke schließen. Die Ansätze verwenden statistische Verfahren, um Zusammenhänge zwischen der Fehleranzahl von Bausteinen und ihren Bausteineigenschaften (z.B. in [BBM96], [OW07]) in früheren Versionen eines Softwaresystems aufzudecken. Gefunden Zusammenhänge werden in der aktuellen Version ausgenutzt, um vorherzusagen, welche Bausteine fehleranfällig sein werden (z.B. [BBM96]) oder um die voraussichtliche Fehleranzahl pro Baustein zur ermitteln (z.B. [OW07]). Alle Ansätze konzentrieren sich nur auf einzelne Bausteine und nicht auf die Abhängigkeiten und sind für den Integrationstest nur bedingt einsetzbar. Unser Ansatz legt den Fokus auf die Abhängigkeiten und ihre Eigenschaften und sagt fehleranfällige Abhängigkeiten voraus, die als Testfokus für den Integrationstest ausgewählt werden. Wir stellen eine Liste von möglichen Abhängigkeitseigenschaften zur Verfügung, um die Abhängigkeiten eines Softwaresystems zu klassifizieren. Wir verwenden ebenfalls statistische Verfahren, um Zusammenhänge zwischen der Fehleranzahl in den Bausteinen und den Abhängigkeitseigenschaften in früheren Versionen eines Softwaresystems aufzudecken. Diese nutzen wir aus, um den Testfokus für die aktuelle Version festzulegen. Der Ansatz wurde für die Entwicklungsumgebung Eclipse in zwei verschiedenen Versionen angewendet. In diesen Versionen deckten wir Zusammenhänge auf und legten basierend darauf den Testfokus für eine dritte Version fest. Nachfolgend werden die identifizierten Abhängigkeitseigenschaften vorgestellt (Kapitel 2). Im Kapitel 3 beschreiben wir den Ansatz zur Testfokusauswahl. Die Ergebnisse der Anwendung des Ansatzes für Eclipse stellen wir in Kapitel 4 vor. Abschließend gehen wir auf zukünftige Arbeiten ein und geben eine kurze Zusammenfassung.

2 Abhängigkeitseigenschaften

Eine Abhängigkeit ist eine unidirektionale Beziehung zwischen zwei Bausteinen. Ein Baustein übernimmt die Rolle des abhängigen und einer des unabhängigen Bausteins (vgl. Abbildung 1a). Es kann zwischen zwei Bausteinen maximal zwei Abhängigkeiten geben. In Abbildung 1b) existieren die Abhängigkeiten „ab“ und „ba“. In „ab“ ist der Baustein A der abhängige und Baustein B der unabhängige Baustein; in „ba“ entsprechend umgekehrt. Eine Abhängigkeit wird durch Abhängigkeitseigenschaften näher definiert (siehe unten), wobei eine Abhängigkeit mehrere Eigenschaften in sich vereinen kann. Baustein A kann beispielsweise verschiedene Dienste an Baustein B aufrufen und auf mehrere Attribute bei B zugreifen.

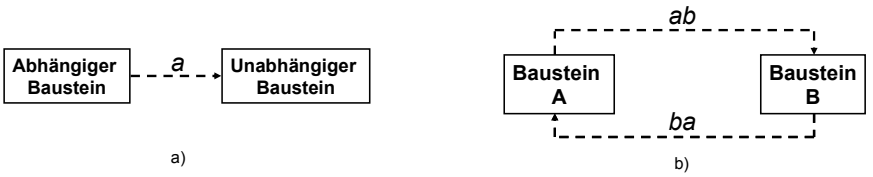


Abbildung 1: Abhängigkeit zwischen Bausteinen

Das Wissen über Abhängigkeitseigenschaften stellt einen Schlüsselfaktor für die erfolgreiche Durchführung eines Integrationstests dar. Eine detaillierte Beschreibung einer Abhängigkeit und ihrer Eigenschaften ermöglicht zum einen, die Abhängigkeit in eine fehlerhafte oder weniger fehlerhafte Abhängigkeit zu klassifizieren. Zum anderen geben die Eigenschaften Auskunft über mögliche Fehler, die innerhalb der Abhängigkeit auftreten können (z.B. wenn bei einem Dienstaufruf zwei Parameter gleichen Typs übergeben werden, kann das zu dem Fehler führen, dass die Parameterwerte versehentlich vertauscht werden, ohne dass es eine Fehlermeldung vom Compiler gibt [SPB+06]). Die Liste der Abhängigkeitseigenschaften wurde mit Hilfe einer Literaturrecherche und Gesprächen mit Entwicklern und Testern erarbeitet. Die identifizierten Eigenschaften lassen sich in drei Kategorien unterteilen: **Laufzeiteigenschaften**, **Entwicklungseigenschaften** und **Deploymenteigenschaften**. Nachfolgend werden die Eigenschaften dieser drei Kategorien kurz beschrieben.

2.1 Laufzeiteigenschaften

Auf oberster Ebene können die Laufzeiteigenschaften in *Client/Server-*, *Vererbungs-* und *indirekte Eigenschaften* eingeteilt werden. Bei einer Client/Server-Eigenschaft ruft ein Baustein bei einem zweiten Dienste auf und/oder greift auf dessen Attribute zu. Die Eigenschaft des *Dienstaufrufs* kann durch weitere Eigenschaften näher spezifiziert werden: übergebene oder zurück gegebene *Parameter*, einzuhaltende *Kommunikationsverträge,* die *Kommunikationsart* (asynchron, synchron) oder „*übergreifende" Zustandsänderungen*. Dienstaufrufe mit Eingabeparameter können eine mögliche Fehlerquelle sein. Die Parameter können von den beteiligten Bausteinen falsch interpretiert und verarbeitet werden [SPB+06] (z.B. ist ein Zahlenwert als Euro übergeben worden, wird aber als Dollar interpretiert). Die Komplexität der Parameter (Eingabe oder Ausgabe) kann ebenfalls ein Indiz auf die Fehleranfälligkeit sein, da sie komplexe Zustände in sich bergen können. Ein Kommunikationsvertrag (vgl. [VR02]) schränkt die Interaktion zwischen zwei Bausteinen ein. Eine vordefinierte Reihenfolge muss eingehalten werden, um einen bestimmten Dienst zu erbringen. Zeitliche Beschränkungen liegen vor, wenn der Server seine Antwort in einer bestimmten Zeit liefern muss. Darüber hinaus können weitere nichtfunktionale Anforderungen (NFA) an die Interaktion gestellt werden, z.B. die Genauigkeit der Ergebnisse. Für den Integrationstester ist es von Interesse, ob die Dienstaufrufe synchron oder asynchron verlaufen und die beteiligten Bausteine in einem oder in zwei parallelen Kontrollprozessen ausgeführt werden. Beispielsweise kann im Falle eines Kontrollprozesses eine wechselseitige Kommunikation zu einem Deadlock führen, wenn die Kommunikation synchron verläuft. Die übergreifende Zustandsänderung beschreibt die Folgen einer Kommunikation. Diese Folgen können das Erzeugen von Instanzen,

Modifikation an existierenden Instanzen und Bausteinen sowie das Löschen von Instanzen sein. Zur Laufzeit können die Änderungen an falschen Instanzen oder an falschen Bausteintypen durchgeführt werden. Attributzugriffe beschreiben das Lesen und/oder Modifizieren von Attributen (oder Variablen) am Server. Der Client hat dabei Zugriff auf Attribute des Servers und kann sie beliebig modifizieren. Ein Zugriff auf ein nicht initialisiertes Attribut beispielsweise (vgl. [Bi96]) kann eine mögliche Fehlerursache sein.

Unter einer Vererbungseigenschaft *„versteht man die Möglichkeit, ein neues Objekt von einem vorhandenen Objekt abzuleiten, wobei das neue Objekt alle Merkmale und Fähigkeiten des alten besitzt.*“ (in [MSH03], Seite 62). Der neue Baustein (Unterklasse) erbt alle Eigenschaften, d.h. alle Zustände, Attribute, Dienste und Abhängigkeiten des vorhandenen Bausteins (Oberklasse). Diese Art der Eigenschaft ermöglicht es, eine Instanz einer Oberklasse durch eine Instanz der Unterklasse zu ersetzen. Wichtige Eigenschaften in einer Vererbung sind die *Modifikation* und die *Vererbungstiefe*. Die Modifikation beschreibt das Ausmaß der Veränderung der Eigenschaften der Oberklasse durch die Unterklasse. Die Modifikation kann durch das Hinzufügen von neuen Diensten und Attributen sowie durch das Überschreiben von existierenden Diensten geschehen. Beim Überschreiben eines Dienstes implementiert die Unterklasse das Verhalten des Dienstes der Oberklasse neu und ändert damit das erwartete Verhalten. Dies bedeutet, dass das Verhalten der Unterklasse nicht zwangsläufig konsistent zum Verhalten der Oberklasse ist. Da jedoch jede Instanz einer Oberklasse durch eine Instanz einer Unterklasseklasse substituiert [Or98] werden kann, wird das erwartete Verhalten (das der Oberklasse) nicht garantiert. Die Vererbungstiefe beschreibt den „Abstand“ zwischen der betrachteten Unterklasse und der Oberklasse. Je größer der Abstand ist, desto mehr Klassen sind zwischen den beiden Klassen der Vererbung zu finden. Alle Klassen in der Klassenhierarchie müssen als potentielle Modifikatoren[3] angesehen werden. Je größer die Vererbungstiefe ist, desto stärker wurde die Oberklasse modifiziert.

Indirekte Abhängigkeiten entstehen, wenn zwei Bausteine voneinander abhängig sind, obwohl sie nicht direkt miteinander kommunizieren oder Vererbungseigenschaften besitzen. Die Abhängigkeit entsteht durch die gemeinsame Verwendung interner und externer Ressourcen, wodurch sich die Bausteine gegenseitig beeinflussen. Die betrachteten Ressourcen sind in dieser Abhängigkeit in keiner der beiden Bausteine zu finden. Verschiedene Arten von Ressourcen sind denkbar: *globale Variablen*, *externe Datenquellen*, gemeinsam genutzte *Hardware* oder *externe Dienste*.

2.2 Entwicklungs- & Deploymenteigenschaften

Die Klasse der Entwicklungs- und Deploymenteigenschaften ist im Vergleich zu den Laufzeiteigenschaften sehr klein. Aber sie liefern ergänzende Informationen über eine bereits vorliegende Abhängigkeit mit Laufzeiteigenschaften. Entwicklungseigenschaften

[3] Ein Modifikator ist eine Klasse, die die Eigenschaften der Oberklasse modifiziert (durch neue Attribute, Dienste oder das Überschreiben)

beschreiben näher, unter welchen Umständen die Abhängigkeit und die daran beteiligten Bausteine entstanden sind. Durch Offshoring [CR05] oder durch große Softwareentwicklungsprojekte entstehen *inhomogene Entwicklungsteams*, die *geographisch verteilt* sind und unter Umständen auch mit *unterschiedlichen Technologien* (z.B. Programmiersprachen) arbeiten. Die Realisierung von abhängigen Bausteinen an unterschiedlichen Standorten kann zu Missverständnissen und somit zu Fehlern führen. Beispielsweise kann die Verwendung von unterschiedlichen Maßzahlen, insofern sie nicht eindeutig spezifiziert wurden, zu unerwartetem Verhalten in der Kommunikation führen. Die Abhängigkeiten zwischen Bausteinen, die in unterschiedlichen Programmiersprachen realisiert werden, können aufgrund unterschiedlich realisiertet Programmiersprachenkonzepte fehlschlagen. Die Deploymenteigenschaften beziehen die Hardwareumgebung mit ein, in der die betrachteten Bausteine installiert sind und ausgeführt werden. Hierbei wird unterschieden, ob die Bausteine *verteilt installiert* sind und ausgeführt werden oder sich auf dem gleichen Hardwareknoten befinden. „*Two components that unconsciously interact via the infrastructure are potential source of failures*“ ([Ma03] Seite 7).

3 Testfokusauswahl

„There is no universal metric or prediction model hat applies to all projects“ (vgl. [ZN08] Seite 1). Daher stützt sich unser Ansatz zur Testfokusauswahl auf Erkenntnisse, die aus früheren Versionen eines Softwaresystems gewonnen werden. Wir empfehlen, die Daten, die über frühere Versionen bereits vorliegen, zu nutzen, um Erkenntnisse für die aktuelle Version des Softwaresystems zu gewinnen. Existierende Ansätze legen den Schwerpunkt auf die Identifikation von fehlerhaften Bausteinen. Unser Ansatz identifiziert fehleranfällige Abhängigkeiten. Hierfür setzen wir für frühere Versionen Abhängigkeitseigenschaften mit der Fehleranfälligkeit der beteiligten Bausteine in Beziehung. Die Fehleranfälligkeit definiert sich als Anzahl der Fehler innerhalb eines Bausteins für eine bestimmte Version. Mit Hilfe statistischer Verfahren decken wir Zusammenhänge zwischen der Fehleranzahl in den Bausteinen und den Abhängigkeitseigenschaften auf. Die statistischen Untersuchungen werden für mehrere frühere Versionen durchgeführt, um versionsübergreifende Korrelationen aufzudecken. Nur Korrelationen, die über mehrere Versionen Bestand haben, sollten zur Auswahl des Testfokus verwendet werden.

Die Testfokusauswahl mit Hilfe unserer Methode besteht aus drei Hauptschritten: *Abhängigkeitseigenschaften festlegen* (1), *frühere Versionen analysieren* (2) und *Testfokusauswahl für die aktuelle Version* (3) (vgl. Abbildung 2). Die Tatsache, dass Informationen über Abhängigkeitseigenschaften in den Artefakten verschiedener Entwicklungsphasen eines Projektes verstreut sind, erschwert die Identifikation der Abhängigkeiten und ihrer Eigenschaften. Aus diesem Grund müssen im ersten Schritt unseres Ansatzes die Abhängigkeitseigenschaften identifiziert werden, die für statistische Untersuchungen zur Verfügung stehen und zu denen schneller Zugang möglich ist. Der Quelltext der Bausteine liefert viele Informationen über die Laufzeiteigenschaften (z.B. Dienstaufrufe, Parameter, Vererbung und Modifikation). Der Projektplan informiert über Entwicklungseigenschaften. Architektur- und

Entwurfsmodelle geben einen Überblick über die Verteilung, die Art der Kommunikation (synchron, asynchron), Kommunikationsverträge usw. In unseren Untersuchungen haben wir uns nur auf den Quelltext als Informationsquelle konzentriert. Wir analysierten Abhängigkeiten (und ihre Eigenschaften) zwischen Quelltextdateien. (siehe Kapitel 4). Nachdem festgelegt wurde, welche Eigenschaften erhoben werden, müssen die Wertebereiche der Eigenschaften definiert werden. In den meisten Fällen handelt es sich bei dem Wertebereich einer Eigenschaft um eine bestimmte Anzahl (z.B. Anzahl aufgerufener Dienste vom abhängigen zum unabhängigen Baustein).

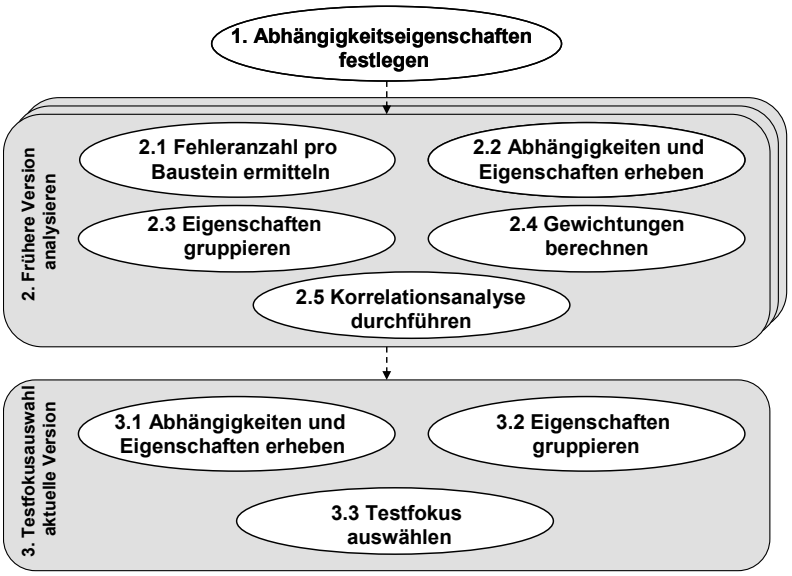


Abbildung 2: Einzelschritte der Testfokusauswahl

Im zweiten Schritt müssen frühere Versionen eines Softwaresystems analysiert werden, um Zusammenhänge zwischen Abhängigkeitseigenschaften und Fehleranzahl der beteiligten Bausteine aufzudecken (2). Für möglichst viele frühere Versionen müssen folgende Teilschritten ausgeführt werden (vgl. Abbildung 2): *Fehleranzahl pro Baustein ermitteln (2.1)*, *Abhängigkeiten und Eigenschaften erheben (2.2)*, *Eigenschaften gruppieren (2.3)*, *Gewichtungen berechnen (2.4)*, *Korrelationsanalyse durchführen (2.5)*. In unserer Methode verwenden wir das Verfahren aus [ZPZ07] um die Fehleranzahl pro Baustein zu berechnen (2.1). Es werden Informationen aus Bugtrackingsystemen, die zur Verwaltung von Fehlermeldungen dienen und Informationen aus Versionsveraltungssystemen (z.B. CVS, SVN) verwendet, um die Fehleranzahl pro Baustein und Version zu ermitteln. Parallel dazu können die Abhängigkeiten und ihre Eigenschaften erhoben werden (2.2). Hierbei werden nur die Eigenschaften betrachtet, die im ersten Schritt (*Abhängigkeitseigenschaften festlegen*) ausgewählt worden sind. Eine übersichtliche Darstellung in Tabellenform als Ergebnis bietet sich hierbei an (eine Tabelle pro Version), wobei jede Zeile eine Abhängigkeit und die Spalten die Eigenschaften darstellen. Diese Tabelle wird um die erhobene

Fehleranzahl der beteiligten Bausteine ergänzt. Die Fehleranzahl des abhängigen und des unabhängigen Bausteins wird für jede Abhängigkeit in die letzte Spalte der Tabelle eingetragen. Abbildung 3 stellt ein Beispiel für eine Abhängigkeitstabelle (Ausschnitt) dar. Die ersten zwei Spalten enthalten die Bezeichnung des abhängigen und unabhängigen Bausteins. Spalten zwei bis fünf listen die Ausprägungen der Eigenschaften für die einzelnen Abhängigkeiten auf. Es ist in Zeile zwei zu erkennen, dass in der Abhängigkeit zwischen dem Baustein `GC.java` und `OS.java` der abhängige Baustein 75 unterschiedliche Dienste am unabhängigen Baustein aufruft, auf 37 unterschiedliche Attribute zugreift und es zwischen beiden Bausteinen keine Vererbung gibt. Die letzten zwei Spalten geben die Fehleranzahl der beteiligten Bausteine wieder (2 für `GC.java` und 8 für `OS.java`). Die entstandene Tabelle kann anschließend von einem Statistik-Werkzeug (z.B. SPSS [SP08]) eingelesen und weiter verarbeitet werden. Bevor die statistischen Analysen durchgeführt werden können, müssen Eigenschaften jeweils gruppiert werden (2.3), da wir als statistisches Verfahren den Mittelwert verwenden. In unseren Untersuchungen hat sich gezeigt, dass eine 10er Einteilung gute Ergebnisse liefert. Dabei wird jeder Wert einer Eigenschaft in eine von 10 Gruppen umkodiert, wobei darauf geachtet wird, dass die entstehenden Gruppen annähernd gleich groß sind. Dezile bieten sich hierbei an. Bevor die statistischen Auswertungen durchgeführt werden können, müssen die einzelnen Abhängigkeiten gewichtet werden (2.4). Dies ist notwendig, da ein Baustein an mehreren Abhängigkeiten als abhängiger und/oder unabhängiger Baustein beteiligt sein kann. Die Gewichtung ergibt sich aus dem Kehrwert der Häufigkeit, mit der ein Baustein als abhängiger bzw. unabhängiger Baustein an einer Abhängigkeit beteiligt ist. Abschließend können wir die Korrelationsanalyse durchführen (2.5). Da an einer Abhängigkeit immer genau zwei Bausteine beteiligt sind, können die Korrelationsanalysen mit der Fehleranzahl des abhängigen, des unabhängigen oder der Summe der Fehleranzahl beider Bausteine durchgeführt werden. In unserem Ansatz verwenden wir die Fehleranzahl beider einzelnen Bausteine, d.h. wir prüfen, ob es einen Zusammenhang zwischen der Eigenschaft und der Fehleranzahl im abhängigen Baustein und zusätzlich ob es einen Zusammenhang zwischen Eigenschaft und Fehleranzahl im unabhängigen Baustein gibt.

Abhängige Datei	Unabhängige Datei	# Attributzugriffe	# Serviceaufrufe	Vererbung	...	# Fehler im abhängiger Baustein	# Fehler im unabhängiger Baustein
CodeAttribute.java	IBytecodeVisitor.java	0	205	0	...	12	2
GC.java	OS.java	37	75	0	...	2	8
Decorations.java	OS.java	79	72	0	...	5	8
Control.java	OS.java	164	71	0	...	8	8
BinaryExpression.java	CodeStream.java	1	64	0	...	12	2
ASTConverter.java	AST.java	0	63	0	...	3	24
AstMatchingNodeFinder.java	ASTVisitor.java	0	62	1	...	5	1
Display.java	OS.java	80	52	0	...	45	8
CodeStream.java	ConstantPool.java	4	51	0	...	2	0
Shell.java	OS.java	49	47	0	...	3	8
⋮	⋮	⋮	⋮	⋮	...	⋮	⋮

Abbildung 3: Ausschnitt einer Abhängigkeitstabelle eines Softwaresystems

Für das Aufdecken von Zusammenhängen zwischen Fehleranzahl und Eigenschaften verwenden wir den Mittelwertvergleich [JL02]. Aufgrund der großen Streuung in den Ergebnissen wurden Zusammenhänge, die durch den Mittelwertvergleich aufgedeckt

wurden, nicht von anderen Verfahren (z.B. Rangkorrelation nach Bravais-Pearson [JL02] oder Korrelationskoeffizient nach Spearman [JL02]) aufgedeckt. Beim Mittelwertvergleich wird die durchschnittliche Fehleranzahl in den gruppierten Eigenschaften (vgl. Schritt 2.3) miteinander verglichen. Gesucht wird nach positiven Zusammenhängen zwischen der durchschnittlichen Fehleranzahl und der untersuchten Eigenschaft, d.h. je größer der Wert (z.B. Anzahl der aufgerufenen Dienste), einer Eigenschaft ist, desto größer ist die durchschnittliche Fehleranzahl. Diese Korrelationsanalyse wird für alle Eigenschaften, die in Schritt 1 festgelegt wurden, durchgeführt. Nachdem der zweite Schritt (und alle seine Teilschritte 2.1-2.5) für mehrere frühere Versionen durchgeführt wurden, können Korrelationen ausgewählt werden, die über mehrere Versionen Bestand haben.

Im dritten Schritt wird die Testfokusauswahl für die aktuelle Version durchgeführt. Hierzu werden die Abhängigkeiten und Eigenschaften für die aktuelle Version identifiziert (3.1) und nach dem gleichen Schema wie in Schritt 2.3 gruppiert (3.2). Anschließend wird der Testfokus ausgewählt. (3.3) Hierzu wird jede Abhängigkeit markiert, die mindestens eine Eigenschaft besitzt, die 1.) eine Korrelation mit der Fehleranfälligkeit aufweist und 2.) deren Wert in den letzten (höchsten) Gruppen eingeordnet werden kann. Hierbei kann unterschieden werden, ob die Eigenschaft eine Korrelation mit der Fehleranzahl des abhängigen Bausteins, mit der Fehleranzahl des unabhängigen Bausteins oder mit der Fehleranzahl beider Bausteine aufweist. Anschließend werden Markierungen verwendet, um den Testfokus auszuwählen. Hat eine Abhängigkeit nur Eigenschaften, die keine Korrelation zur Fehleranzahl aufweisen oder deren Werte der Eigenschaften nicht in den höchsten Gruppen zu finden sind, so wird sie nicht zum Testfokus gezählt. Gibt es mindestens eine Eigenschaft in der Abhängigkeit, die sich in den höchsten Gruppen befindet und eine Korrelation zur Fehleranzahl mit genau einem der beteiligten Bausteine aufzeigt, so wird der Testfokus dieser Abhängigkeit mit einer „1“ kodiert. Bei einer Abhängigkeit, die mindestens einen Eigenschaftswert aufweist, der in den höchsten Gruppen ist, und diese Eigenschaft eine Korrelation mit der Fehleranzahl beider Bausteine aufweist, wird der Testfokus mit „2“ kodiert. Die Abhängigkeiten mit der Kodierung „2“ bekommen die höchste Testpriorität. Abhängigkeiten mit der Kodierung 1 erhalten eine geringere Testpriorität. Entsprechend der vergebenen Testprioritäten werden die zu testenden Abhängigkeiten ausgewählt.

4 Fallstudie Eclipse

Im Rahmen einer Fallstudie haben wir unsere Methode angewendet. Wir wollten zeigen, dass mit Hilfe der Methode, die Abhängigkeiten mit Eigenschaften, die auf fehlerhafte Bausteine hinweisen, ausgewählt werden. Verwendet wurde die Entwicklungsumgebung Eclipse [Ec08] in den Versionen 2.0, 2.1 und 3.0. Es standen die Quelltexte und die Fehleranzahl pro Quelltextdatei zur Verfügung [ZPZ07]. In einem ersten Schritt wurden die Abhängigkeiten und ihre Eigenschaften identifiziert (einfache statistische Daten über die drei Versionen sind in Abbildung 4 zusammengefasst).

	Eclipse 2.0	Eclipse 2.1	Eclipse 3.0
Anzahl Quelltextdateien	6747	7908	10631
Anzahl Abhängigkeiten	56765	71182	96652
Durchschnittliche Fehleranzahl pro Datei	1,38	0,78	0,95

Abbildung 4: Statistische Daten der drei Eclipse-Versionen

12 Abhängigkeitseigenschaften wurden mit Hilfe eines Werkzeugs automatisch aus dem Quelltext extrahiert, um die Abhängigkeiten näher zu charakterisieren. Die untersuchten Eigenschaften waren:

- Vererbungseigenschaften: *Anzahl überschriebener Dienste, Anzahl neuer Dienste, Anzahl neuer Attribute*
- Client/Server-Eigenschaften: *Anzahl Attributzugriffe, Anzahl aufgerufener Dienste, Anzahl Eingabeparameter, Anzahl komplexer Eingabeparameter, Anzahl Dienste mit komplexen Ausgabeparametern, Anzahl Dienste mit mindestens zwei Eingabeparametern gleichen Typs, durchschnittliche Anzahl Eingabeparameter pro Dienstaufruf, durchschnittliche Anzahl komplexer Eingabeparameter pro Dienstaufruf, relative Häufigkeit von Dienstaufrufen mit komplexen Ausgabeparametern*

Im nächsten Schritt wurden für die Versionen 2.0 und 2.1 Mittelwertvergleiche durchgeführt. Dabei konnte nur für vier Eigenschaften (*Anzahl überschriebener Dienste* vgl. Abbildung 5, *Anzahl neuer Dienste*, *Anzahl neuer Attribute* und *Anzahl Dienstaufrufe mit komplexem Ausgabeparameter*) ein signifikanter[4] Zusammenhang hinsichtlich der Fehleranzahl im **abhängigen Baustein** aufgedeckt werden, der in beiden Versionen Bestand hatte.

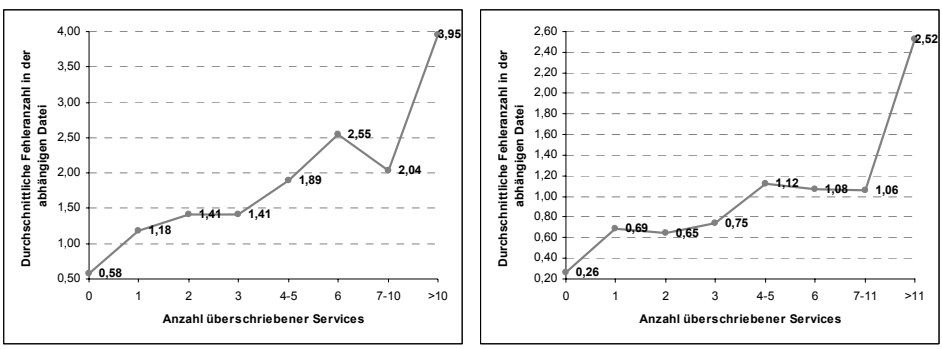


Abbildung 5: Zusammenhang zwischen Anzahl überschriebener Dienste und Fehleranzahl des abhängigen Bausteins (links Version 2.0, rechts: Version 2.1)

Für die **unabhängigen Bausteine** wurden ebenfalls vier Eigenschaften identifiziert, die auf eine erhöhte Fehleranzahl hindeuten (*Anzahl Dienstaufrufe*, *Anzahl Dienstaufrufe mit komplexem Ausgabeparameter*, *Anzahl Eingabeparameter*, *Anzahl komplexer Eingabeparameter*). Die gefundenen Zusammenhänge wurden abschließend verwendet,

[4] Im Rahmen der statistischen Untersuchungen wurde ein Signifikanzniveau von 0,01 verwendet.

um die zu testenden Abhängigkeiten für die Eclipse-Version 3.0 festzulegen. Nach der oben vorgestellten Methode wurden die Abhängigkeiten kodiert (0 = *nicht testen*, 1 = *testen mit geringer Priorität*, 2 = *testen mit hoher Priorität*). Aus den 96652 Abhängigkeiten der Version 3.0 wurden 5318 Abhängigkeiten mit hoher Testpriorität, 12980 Abhängigkeiten mit mittlerer Testpriorität und die verbleibenden 78354 Abhängigkeiten als "nicht testen" eingestuft. Abschließend wurde ein erneuter Mittelwertvergleich für die Version 3.0 durchgeführt, um die Unterschiede der Fehleranzahl in den drei Gruppen (0, 1, 2) miteinander zu vergleichen. Das Ergebnis ist in Abbildung 6 zu sehen. Es zeigt sich, dass die Bausteine, die an Abhängigkeiten beteiligt sind, die nicht als Testfokus ausgewählt wurden, die geringste durchschnittliche Fehleranzahl aufweisen. Für die zweite Gruppe ist die durchschnittliche Fehleranzahl höher als für die erste, aber kleiner als die dritte Gruppe. Die Gruppe mit der höchsten Priorität weist auch die höchste durchschnittliche Fehlerrate auf, sowohl für den abhängigen als auch für den unabhängigen Baustein.

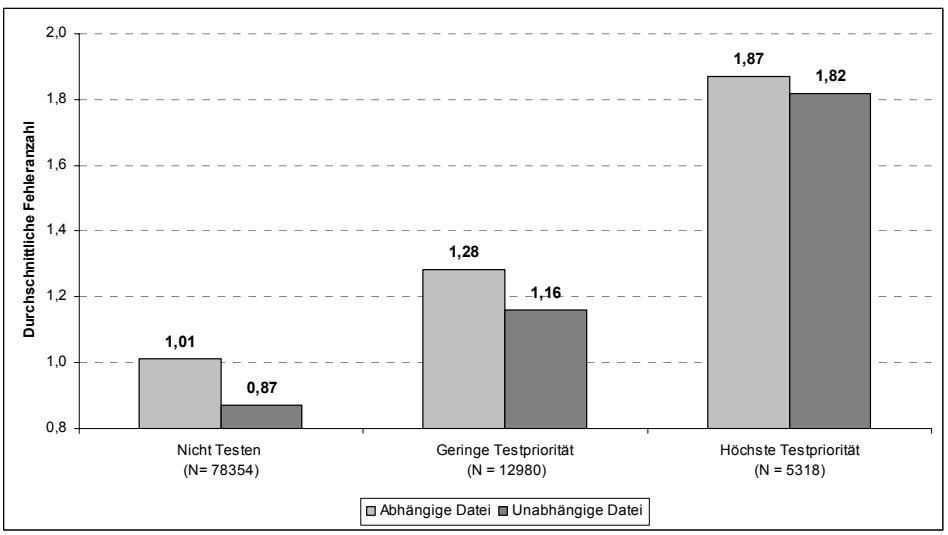


Abbildung 6: Durchschnittliche Fehleranzahl für Dateien nach Zuordnung des Testfokus

Die Untersuchung am Beispielsystem Eclipse hat die Anwendbarkeit unseres Ansatzes an einem großem System gezeigt. Es war möglich, mit Hilfe statischer Analysewerkzeuge die große Anzahl von Abhängigkeiten und ihrer Eigenschaften aufzudecken. Es zeigt sich auch, dass die Eigenschaften, die für Eclipse 2.0 und 2.1 auf fehlerhafte Bausteine hinweisen, auch in Eclipse 3.0 verwendet werden können, um den Testfokus sinnvoll festzulegen. Abbildung 6 zeigt, dass die Abhängigkeiten so ausgewählt worden sind, dass die beteiligten Bausteine eine höhere Fehleranzahl aufweisen. Zusätzlich ermöglicht die Einteilung in „nicht testen", „Testen mit geringer Priorität" und „Testen mit hoher Priorität" eine bessere Planung des Testprozesses. Abhängigkeiten mit hoher Priorität müssen auf jeden Fall getestet werden und wenn anschließend noch Ressourcen zur Verfügung stehen, kann die zweite Gruppe getestet werden.

5 Zusammenfassung

Das vorgestellte Verfahren zur Testfokusauswahl für den Integrationstest bietet eine Möglichkeit, aus den Erfahrungen früherer Versionen eines Softwaresystems zu lernen und dieses Wissen in der aktuellen Version des Systems einzusetzen. Inwieweit sich das gewonnen Wissen auf andere Softwaresysteme übertragen lässt, muss in nachfolgenden Studien untersucht werden.

In zukünftigen Arbeiten wollen wir das Verfahren für die Testfokusauswahl verfeinern, um eine feingranularere Einteilung der Abhängigkeiten nach Testnotwendigkeit durchzuführen. Hierzu werden nicht nur die gefundenen Korrelationen verwendet, sondern auch untersucht, inwieweit der Fehlerdurchschnitt in den einzelnen Gruppen einer Eigenschaft vom Fehlerdurchschnitt aller Dateien abweicht. Je größer die Abweichung ist, desto größer ist der Einfluss der Eigenschaft auf die Fehleranfälligkeit der beteiligten Bausteine. In Abbildung 5 beispielsweise ist zu erkennen, dass für die Version 2.0 die durchschnittliche Fehleranzahl der abhängigen Datei in der letzten Gruppe 3,95 beträgt. Die durchschnittliche Fehleranzahl aller Dateien dieser Version beträgt jedoch „nur“ 1,38 Fehler pro Datei. Somit liegt die Fehleranzahl der abhängigen Dateien in dieser Gruppe ca. 2,8mal höher. Diese Faktoren wollen wir uns zu Nutze machen, um die einzelnen Eigenschaften zu gewichten. Darüber hinaus wird derzeit erarbeitet, inwieweit nicht nur fehlerhafte Bausteine sondern direkt fehlerhafte Abhängigkeiten identifiziert werden können.

In weiteren Arbeiten werden die Ergebnisse der Testfokusauswahl dazu verwendet, geeignete Testentwurfstechniken auszuwählen, um die ausgewählten Abhängigkeiten gezielt testen zu können. Die Eigenschaften geben Auskunft darüber, welche Fehler in dieser Abhängigkeit auftreten können. So können diese mit systematischen Verfahren aufgedeckt werden. Darüber hinaus wird ein Verfahren zur Ermittlung einer optimalen Integrationsreihenfolge erarbeitet, welches den ausgewählten Testfokus berücksichtigt. Es ist das Ziel, Abhängigkeiten, die auf fehlerhafte Bausteine hinweisen, möglichst früh zu integrieren, um viele Fehler früh im Integrationstestprozess aufzudecken. Des Weiteren hat eine frühe Integration von fehlerhaften Bausteinen und Abhängigkeiten den Vorteil, dass sie im Rahmen weiterer Integrationstests indirekt mit getestet werden (vgl. [Bi00]).

Während der Anwendung des Ansatzes im Rahmen der Fallstudie hat sich gezeigt, dass sich der Zeitaufwand in Rahmen hält. Den größten Teil nahm dabei die Quelltextanalyse ein, um die Abhängigkeiten und ihre Eigenschaften zu extrahieren. Das von uns eingesetzte Werkzeug benötigte dafür acht bis zehn Stunden pro Version. Die Korrelationsanalyse wurde mit Hilfe des Werkzeugs SPSS durchgeführt. Darin wurden alle Analysen per Hand angestoßen. Allerdings war eine Version in ca. 120 Minuten analysiert und mögliche Korrelationen aufgedeckt. Da für frühere Versionen die Analyse jeweils nur einmal durchgeführt werden muss, kann gesagt werden, dass mit Hilfe der derzeitigen Werkzeugunterstützung ca. 10 - 12 Stunden für die Analyse einer Version benötigt werden. In naher Zukunft soll das gesamte Verfahren stärker automatisiert werden. Eine einheitliche Werkzeugumgebung, die die statischen Analysen des Quelltextes, das Identifizieren der Abhängigkeiten und ihrer Eigenschaften sowie die

statistischen Tests durchführt, wird derzeit erarbeitet. Das große Ziel ist es, den Integrationstestprozess in Projekten durch eine geeignete Unterstützung besser zu etablieren.

Literaturverzeichnis

[BBM96] Basili, V.R., Briand, L.C., Melo, W.L.: A Validation of Object-Oriented Design Metrics as Quality Indicators, IEEE Transaction on Software Engineering, Vol 22, No. 10, Oktober 1996

[Bi96] Binder, R.: "Testing Object-Oriented Software: A Survey" Journal of Software Testing, Verification and Reliability, Volume 6, Seiten 125-252, 1996.

[Bi00] Binder, R.: "Testing Object-Oriented Systems". Addison-Wesley, 2000.

[BIP07] Borner, L., Illes-Seifert, T., Paech, B.: "The Testing Process - A Decision Based Approach", International Conference on Software Engineering Advances (ICSEA) 25-31 Aug. 2007.

[CR05] Chakraborty, K., Remington, W.: "'Offshoring' of IT services: the impact on the US economy". J. Comput. Small Coll. 20, 4 (Apr. 2005), 112-125.

[JL02] Janssen, J., Laatz, W.: „Statistische Datenanalyse mit SPSS für Windows", Springer Verlag, 2002.

[Ma03] Mariani, L.: "A Fault Taxonomy for Component-based Software", Proceedings of the International Workshop on Test and Analysis of Component-Based Systems; TACoS 2003, September 2003.

[Me79] Meyers, G.J.: "The Art of Software Testing", John Wiley & Sons, New York, 1979.

[MSH03] Middendorf, S., Singer, R., Heid, J.: „JAVA Programmierhandbuch und Referenz", dpunkt.verlag, 2003.

[Or98] Orso, A.: "Integration Testing of Object-Oriented Software", Doktorarbeit, 1998.

[OW07] Ostrand, T.J., Weyuker, E.J.: How to Measure Success of Fault Prediction Models in Proceddings of the Fourth international workshop on Software quality assurance SOQUA 2007, September 2007

[SL06] Spillner, A., Linz, T.: "Basiswissen Softwaretest - Aus- und Weiterbildung zum Certified Tester", dpunkt.verlag, 2006.

[SPB+06] Suliman, D., Paech, B., Borner, L., Atkinson, C., Brenner, D., Merdes, M., Malaka, R.: "The MORABIT Approach to Runtime Component Testing", Proceedings of the 30th Annual International Computer Software and Applications Conference, 2006.

[SW02] Sneed, H., Winter, M.: "Testen objektorientierter Software – Das Praxishandbuch für den Test objektorientierter Client/Server - Systeme", Hanser Verlag, 2002.

[VR02] Vieira, M., Richardson, V.: "The Role of Dependencies in Component-based Systems Evolution" Proceedings of the International Workshop on Principles of Software Evolution 2002.

[ZPZ07] Zimmermann, T., Premraj, R., Zeller, A.: "Predicting Defects for Eclipse", Proceedings of PROMISE 2007.

[ZN08] Zimmermann, T., Nagappan, N.: „Predicting Defects using Nework Analysis on Dependency Graphs", Proceedings of International Conference on Software Engineering 2008.

[Ec08] Eclipse, www.eclipse.org, 2008

[SP08] SPSS, http://www.spss.com/, 2008

The vital few and trivial many: An empirical analysis of the Pareto Distribution of defects

Timea Illes-Seifert, Barbara Paech

Lehrstuhl für Software Systeme, Institut für Informatik
Im Neuenheimer Feld 326
69120 Heidelberg
illes-seifert@informatik.uni-heidelberg.de
paech@informatik.uni-heidelberg.de

Abstract: The Pareto Principle is a universal principle of the "vital few and trivial many". According to this principle, the 80/20 rule has been formulated with the following meaning: For many phenomena, 80% of the consequences originate from 20% of the causes. In this paper, we applied the Pareto Principle to software testing and analysed 9 open source projects (OSPs) across several releases. The results show that a small number of files account for the majority of defects, even across several releases. In contrast, there is no evidence that this small part of files containing most of the defects also makes up only a small part of the system's code size. While this is not the first study about the Pareto Principle, it adds to the body of empirical body of knowledge wrt. software defects.

1 Introduction

The Pareto Principle, also known as the 80-20 rule has been originally analysed by Vilfredo Pareto [Re05] who observed that 80% of property in Italy was owned by 20% of the Italian population. Juran [JG88] generalized this principle he called the "vital few and trivial many", stating that most of the results in any context are raised by a small number of causes. This principle is often being applied in several contexts, e.g. in sales, stating that 20% of the customers are responsible for 80% of the sales volume. One of the first studies that translated this principle to the software engineering area is reported in [En79]. The author analyses the distribution of defects in an operating system developed at IBM laboratories. The distribution of about 430 defects over about 500 modules has been analysed and confirms the Pareto Principle, i.e. approximately 80% of the defects were contained in 20% of the modules.

Two main hypotheses related to the Pareto Principle form the basis of this study. First, we want to analyse whether a small part of files accounts for the majority of defects. Second, if this is the case, we want to determine whether this small part of files also constitutes a small part of the system's code size. Knowing the validity of the Pareto Principle in the testing context is very valuable for testers, because they can focus their testing activities on the "vital few" files accounting for most of the defects. In addition, the Pareto Principle can form the basis for defect prediction algorithms, as presented in [KZ+08] which predicts the subset of the most fault-prone files. From the research

perspective, this study increases the empirical body of knowledge in the area of defect distribution. First, by replicating empirical studies conducted before, more general conclusions can be derived. In any discipline, progress involves the building of a solid empirical body of knowledge [BL99]. Second, to our knowledge, this is one of few studies that focus on the analysis of the Pareto Principle in detail, including data from 9 large OSPs. The availability of OSPs enables to conduct such a comprehensive study.

The reminder of this paper is organized as follows: The design of our study is described in Section 2, including hypotheses, characteristics of the analysed projects, as well as data collection procedures. In Section 3, the results of our empirical study are presented. In Section 4, we discuss the threats to validity and in Section 5 an overview of related work is given. Finally, Section 6 concludes the paper.

2 Study Design

In this paper, we use the definition of defects and failures provided in [IS07]: A **defect or fault** is "a flaw in a component or system that can cause the component or system to fail to perform its required function. A defect, if encountered during execution, may cause a failure of the component or system". Thus, a **failure** is the observable "deviation of the component or system from its expected delivery, service or result". **Defect count** is the number of defects identified in a software entity. In this paper, we count the number of defects of a file. The file *a* is more **fault-prone** than the file *b* if the defect count of the file *a* is higher than the defect count of the file *b*. In the subsequent Sections details on the experiment are described.

Hypotheses. In this paper, the following hypotheses related to the Pareto Principle will be analysed:

- **Hypothesis 1, Pareto distribution of defects in files:** A small number of files accounts for the majority of the defects.
- **Hypothesis 2, Pareto distribution of defects in files across releases:** If the Pareto Principle applies to one release, then it applies to all releases of a software project.
- **Hypothesis 3, Pareto distribution of defects in code:** A small part of the system's code size accounts for the majority of the defects.
- **Hypothesis 4, Pareto distribution of defects in code across releases:** If the Pareto Principle applies to one release, then it applies to all releases of a software project.

Subject Projects. In this study, we analysed 9 OSPs. As required in [Ha08], we applied the following criteria to select the OSPs: (1) The project is of a large size in order to permit significant results. Thus, the size of the selected projects ranges from about 70.000 LOC to about 240.000 LOC. (2) A well documented defect history is available. We searched for projects for which a bug tracking system is available. (3) The project is mature so that effects will have appeared if present. According to this criterion, we selected projects with a number of check-ins (we call them history touches - HT) in a

versioning control system (VCS) greater than 50.000. (4) The source code is available for at least one release. We included one project, OSCache that does not fulfil the criteria defined above, in order to compare the results obtained for all other projects with a smaller but mature[1] project. The URL of all projects are given in the footnote[2].

OS-Project	Project since	# Defects	# HTs	LOC	# Files
1. Ant (*1.7.0*)	2000	4804	62763	234253	1550
2. FOP (*0.94*)	2002*	1478	30772	192792	1020
3. CDK (*1.0.1*)	2001*	602	55757	227037	1038
4. Freenet (*0.7*)	1999*	1598	53887	68238	464
5. Jetspeed2 (*2.1.2*)	2005	630	36235	236254	1410
6. Jmol (*11.2*)	2001*	421	39981	117732	332
7. Oscache (*2.4.1*)	2000	2365	1433	19702	113
8. Pentaho (*1.6.0*)	2005*	856	58673	209540	570
9. TV-Browser (*2.6*)	2003	190	38431	170981	1868

Table 1: Subject Programs

A * behind the data in the column "Project since" denotes the date of the registration of the project in SourceForge[3]. For the rest, the year of the first commit in the versioning system is indicated. The column "OS-Project" contains the name of the project followed by the project's latest version for which the metrics "LOC" (Lines of Code) and the number of files have been computed. The 3rd and the 4th columns contain the number of defects registered in the defect database and the number of HTs extracted from the VCS.

Data collection. Similarly to the approach used in [FPG02], [CM03], [SZZ05], we combine the information contained in VCSs with information contained in defect tracking systems in order to compute the defect count per file. For this purpose, we extract the information of the VSC into a history table of a data base. Additionally, we extract the defects of the corresponding project into a defect table of the same data base. Then, we use a 3-level algorithm. **Direct search:** First, we search for messages in the history table containing the defect-IDs from the defect table. Messages containing the defect-ID and a text pattern, e.g. "fixed" or "removed", are indicators for defects that have been removed. **Keyword search:** In the second step, we search for keywords, e.g. "defect fixed", "problem fixed", within the messages which have not been investigated in the step before. **Multi-defects keyword search:** In the last step, we search for keywords which give some hints that more than one defect has been removed (e.g. „two defects fixed").We used SPSS[4], version 11.5, for all statistical analyses.

[1]The project exists since 2000.
[2]1. http://ant.apache.org/, 2. http://xmlgraphics.apache.org/fop/index.html, 3. http://sourceforge.net/projects/cdk/, 4. http://freenetproject.org/whatis.html, 5. http://portals.apache.org/jetspeed-2/, 6. http://jmol.sourceforge.net/, 7. http://www.opensymphony.com/oscache/, 8. http://sourceforge.net/projects/pentaho/, 9. http://www.tvbrowser.org/
[3]http://sourceforge.net/
[4] SPSS, http://www.spss.com/

3 Results

3.1 Hypothesis 1: Pareto distribution of defects in files

The first hypothesis related to the 80/20 rule concerns the distribution of defects in files. All OSPs presented in Section 2.2 have been analysed graphically in order to verify this hypothesis. Figure 1 shows the Alberg Diagram suggested by Fenton and Ohlsson [FO00] for the graphical analysis of the Pareto Principle. Thus, files are ordered in decreasing order with respect to the number of defects. Then the cumulated number of defects is plotted on the y-Axis of the Albert diagram relative to the percentage of files (plotted on the x-Axis). For example, in the case of the Jetspeed2 project, 80% of the defects are contained in 27% of the files. Figure 1 shows the distribution of defects of all analysed OSPs within one Alberg diagram.

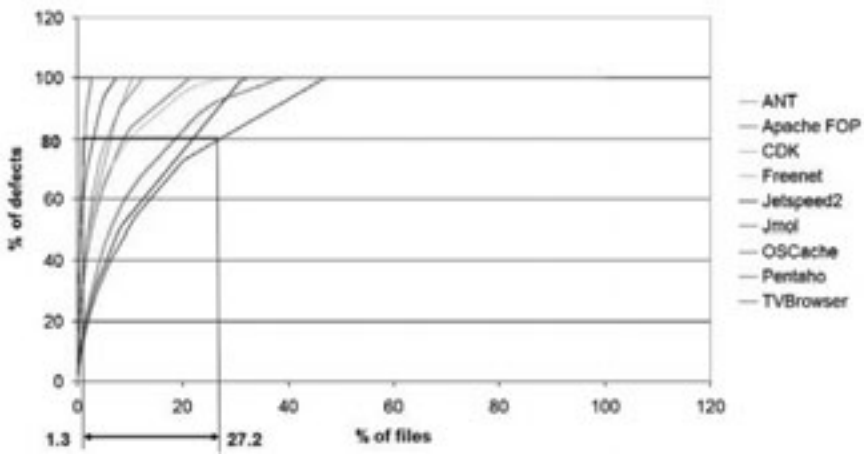


Figure 1: Pareto distribution of defects for each OSP in an Alberg diagram

Approximately 80% of the defects are concentrated in a range of 1.3% (in the case of the TVBrowser project) to 27.2% (in the case of the Jetspeed2 project) of files. Thus, the TVBrowser project shows the strongest focus of defects on a very small part of the files. Only one project shows a concentration of 80% of defects in clearly more than 20% of the files. This is the case of the Jetspeed2 project, with 27.2%. In case of the Pentaho project, 21.9% of the files contain 80% of the defects that can be considered much closer to the 20% formulated in the hypothesis.

Based on this analysis, Hypothesis 1 can be largely confirmed for OSPs*: A small number of files account for the majority of the defects in OSPs.* 7 OSPs show an even stronger focus of the majority of defects on a small part of files than required by the 80/20-rule. This is the case of TVBrowser, Jmol, OSCache, CDK, Ant and Freenet. Two other projects are close to the 80/20 rule.

3.2 Hypothesis 2: Pareto distribution of defects in files across releases

In order to analyse this hypothesis, the percentage of the most fault-prone files containing 80% of the defects has been computed for several releases of the OSPs. Table 2 shows the results. The first column contains the name of the OSP, followed by the number of the analysed releases. The next two columns indicate the absolute and, respectively, the relative number of releases for which about 80% of the defects are concentrated in a small percentage (below 25%) of files. The column "Range" indicates the range for the concentration of defects. For example, the concentration of defects in the ANT project ranges from 8.23% to 24.97% of the files depending on the analysed release.

OSP	Number of analysed releases	Pareto distribution holds for …		Range	100% of defects contained in less than 25% of the files. This holds for …	
		Absolute # of analysed releases	Percentage of the analysed releases		Absolute # of analysed releases	Percentage of the analysed releases
1. ANT	5	5	100%	8.23% - 24.97%	4	80%
2. ApacheFOP	4	4	100%	12.02% - 24.79%	1	25%
3. CDK	7	7	100%	3.14% - 20.56%	7	100%
4. Freenet	6	6	100%	1.95% - 17.29%	2	17%
5. Jetspeed2	3	2	66.67%	19.03% - 67.73%	0	0
6. Jmol	9	9	100%	1.52% - 17.42%	8	89%
7. OSCache	4	4	100%	3.95% - 13.98%	4	100%
8. Pentaho	3	2	66.67%	9.12% - 35.12%	1	33%
9. TVBrowser	4	4	100%	2.69% - 29.32%	3	75%

Table 2: Pareto Distribution of defects in files across releases

For 7 of the 9 analyses OSPs, 80% of the defects are contained in less than 20% of the most fault-prone files. The concentration of the defects ranges from 1.52% in the case of the Jmol project to 35.12%, in case of one analysed release of the Pentaho project. One exception is the Pentaho project. In one of the analysed releases, the defects are not concentrated on a few files but rather distributed among 67.73% of the files. In many releases of the analysed OSPs, we observed a high concentration of defects on a very small number of files. Thus, we additionally determined the percentage of files that account for 100%, i.e. for **all** defects in a system. The last two columns in Table 2 show the absolute and relative number of releases for which 100% of defects are contained in about 20 % of the files. Again, we used 25% of files as a threshold. In two thirds of the analysed releases of the OSPs (30 of 45), 100% of the defects are concentrated in less than 25% of the files. In 27 of 45 analysed releases, 100% of the defects are even contained in less than 20% of the files.

Based on the results of these analyses, it can be concluded that the Pareto Principle largely persists across several releases of a software project. The concentration intensity can vary slightly from release to release.

3.3 Hypothesis 3: Pareto distribution of defects in code

In order to analyse the Pareto hypothesis for code, the percentage of code that accounts for 80% of the defects contained in the most fault-prone files has been computed. Consequently, this analysis determines if the small part of the files responsible for most of the defects also represent a small part of the code. The results of this analysis are shown in Figure 2. On the X-axis, the analysed releases of the OSPs are indicated[5]. The line chart and the bar chart indicate for each release the percentage of files and the corresponding percentage of code that account for approximately 80% of the defects. For example, in the case of the Jmol 9 release, **10.18%** of the files that account for 80% of the defects make up **16.93%** of the system's code. Similarly, in the case of the Ant 1.6.1 release, **24.97%** of the files that account for 80% of the defects make up **89.42%** of the system's code.

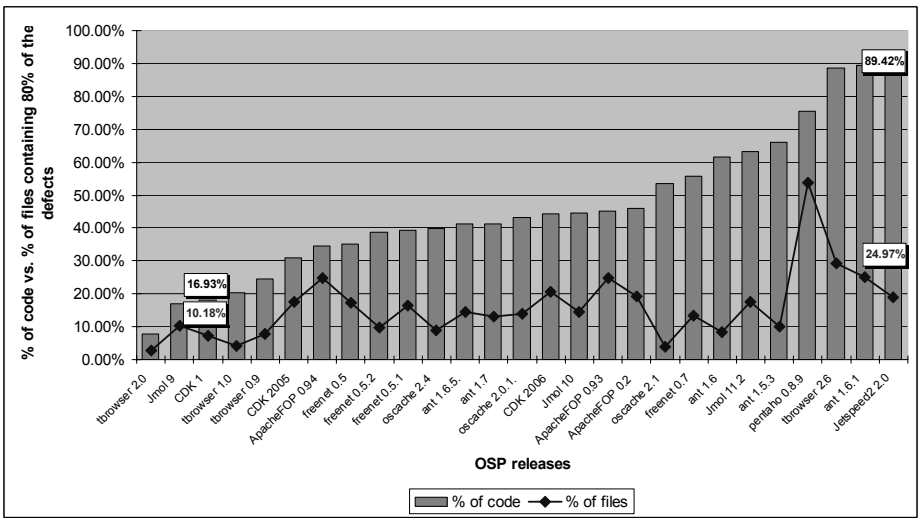


Figure 2: Pareto distribution of defects in code

The concentration of the majority of the defects on a small part of the system's code is true only for a small part of the OSP releases. About 5 releases show a concentration of defects on less than about 25% of the code (TVBrowser 2.0, Jmol 9, CDK 1, TVBrowser 1.0, TVBrowser 0.9). Most of the analysed releases show a distribution of the defects on about 30% to about 60% of the code. For a small part of the releases, the defects are distributed on almost the whole system.

Based on this analysis, the hypothesis has to be rejected. A small part of the code accounts for the majority of the defects only in a few of the analysed cases. In addition, there is no evidence for the contrary hypothesis: If a small number of files account for

[5] This analysis comprises all releases of the OSPs for which the source code is available.

the majority of the defects, it is because these files contain the most code. This statement is true for a small part of the analysed OSPs only.

3.4 Hypothesis 4: Pareto distribution of defects in code across releases

Since the Pareto hypothesis on the distribution of defects in code has been rejected, the hypothesis 4 has to be adjusted. For all cases, in which the Pareto hypotheses could be confirmed: Does the Pareto distribution of defects in code hold for all or at least for the most releases of an OSP? Despite the fact that Hypothesis 3 has been rejected, this research question is important to be analysed. If this is the case, it means that for a small part of OSPs the Pareto Principle is valid and it is worthwhile to perform further analyses in order to determine characteristics of such programs and to find out factors that favour such a distribution.

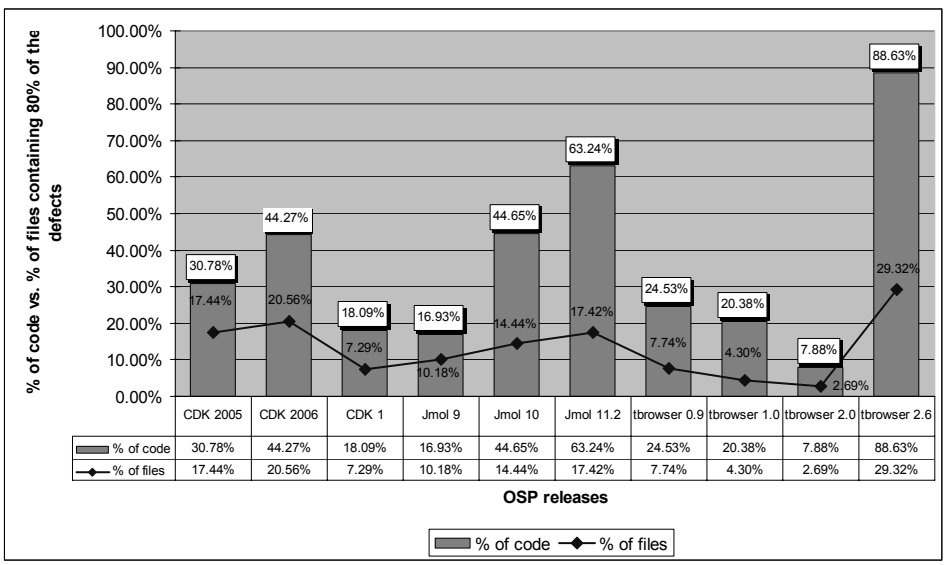


Figure 3: Pareto distribution of defects in code across releases

Figure 3 shows the distribution of 80% of the defects in code across releases for all OSPs for which at least one release shows a concentration of most of the defects on less than 20% of the code. The bar chart shows the percentage of code that contains 80% of the defects, and the line chart shows the percentage of files accounting for 80% of the defects. In the case of the CDK and the Jmol project, only a single release shows a concentration of most of the defects on a small part of the code. For the other analysed releases, the defects are distributed on about 30% to 63% of the code. In the case of the TVBrowser project, three of four analysed releases show a concentration of most of the defects on a small part of the code. The last analysed release, however, shows a high distribution of the defects on about 88% of the code.

Based on the results of this analysis, the hypothesis can not be confirmed. A concentration of most of the defects on a small part of the code in one release does not mean that this concentration will persist in consequent releases.

4 Threats to validity

Internal validity is concerned with the degree to which conclusions about the causal effect of the independent variables on the dependent variable can be drawn [WR+00]. One threat to validity is that not all developers deliver meaningful messages when they check-in files. Developers, for example, can also check in files without specifying any reason, even though they had corrected a defect. Thus, the defect count of a file can be higher than the defect count computed by our algorithm. This concern is alleviated by the size of the analysed OSPs. **External validity** is concerned with the degree to which results can be generalized [WR+00]. This issue is alleviated by the number and diversity of the analysed OSPs. The more OSP programs show the same characteristics, the higher the probability that other OSP programs would also show these characteristics. Additionally, we choose programs from different application domains in order to increase the representativeness of the study results. Furthermore, analyses of additional programs that are intended in our future work would increase the external validity.

5 Related Work

There are several studies that analyse the **Pareto distribution of defects in files**. In contrast to our study, most of the studies concentrate on analysing few or one releases of a system or do not detail the results per release. Exceptions are the studies reported in [OW02] and in [KZ+08]. The Pareto distribution of defects in files is largely confirmed by most of the authors. The extent of the concentration varies from program to program. This is similar to the results obtained by studying OSPs in this paper and in [KZ+08]. In addition, the type of analysed defects differs from study to study. Roughly, the defect types can be categorized into pre-release[6] and. post-release[7] defects. The results of the studies in literature are summarised in Table 3.

In [OW02] and in [KZ+08], the Pareto distribution across several consecutive releases is reported. In [OW02], the authors observe that the concentration of defects on a small part of files becomes stronger when the system matures. In [KZ+08], the authors report that for all analysed OSPs, 10% of the files account for 73%-95% of the defects. The results of the latter study are similar to the results reported in this study: The concentration remains low across nearly all releases of the analysed OSPs, but the extent to which defects are concentrated on a part of the files varies from release to release.

[6]Pre-release defects are defects reported before release, usually by developers and testers.
[7]Post-release defects are defects reported after release, usually reported by developers, testers and in some case also by customers.

Ref.	Characteristics of the analysed projects	Relationship	Kind of defects analysed
[En79]	One release of the operating system DOS/VS	21 – 78	pre-release defects: defects found during system testing
[AR07]	Three projects from a large company in the telecommunications domain.	20 – 87 (P1), 20 – 87 (P2) 20 – 80 (P3)	post-release defects[8]
		20 – 63 (P1), 20 – 70 (P2), 20 – 70 (P3)	pre-release defects
[OA96]	Two consecutive releases of a telecommunication switching system	20 - 60	pre-release and post-release defects altogether: defects reported during function, system and site tests, as well as during the first moths in operation[21]
[KK96]	5 consecutive releases of a commercial telecommunications system	38 – 80	pre-release and post-release defects altogether: defects reported as "Failure Reports" reported from validation teams and from customers
[FO00]	Two releases of a major commercial system developed at Ericsson Telecom AB	20-60	pre-release and post-release defects altogether: defects reported during function test and system test by testers
		10 – 100 1st release 10 – 80 2nd release	post-release defects: **d**efects reported during operation[21]
[MK92]	Two distinct data sets: command and control communication system, medical imaging system	20-65	pre-release and post-release defects altogether: defects recorded during system integration and test phases and for the first year of program deployment
[OW02]	Thirteen releases of a large industrial inventory tracking system	10 - 68 10 -100 (for the last four releases)[9].	pre-release and post-release defects altogether: all kinds of defects recorded in one of these phases: development, unit testing, integration testing, system testing, beta release, controlled release, and general release.
		pre-release, 36 – 80, 3 – 80 (in later releases), post-release 1/2 - 80	separate analysis for pre-release and post-release defects.
[KZ+08]	Seven open source programs	10 – 73%-95%	pre-release and post-release defects altogether

Table 3: Pareto analyses in literature

Similarly to the results of this paper, there is little evidence for the **Pareto distribution of defects in code** reported in literature. The strongest concentration of defects on a small part of a system's code size is reported in [OW02]. 10% of the files that account for a range of 68% - 100% of defects (depending on the analysed release) contain about 35% of the system's code. But the percentage of the code contained in the most fault-prone files always exceeded the percentage of the files that contained the defects. The results reported in [FO00], [AR07], and [KK96] do not provide evidence for the Pareto distribution of defects in code as well. This is the case for both pre-release and post-release defects as reported in [AR07]. The only study analysing the Pareto distribution of defects in code across several releases is reported in [OW02]. In contrast to a decreasing concentration of defects on a small part of files from release to release, the corresponding percentage of code (contained in those fault-prone files) does not show such a trend. 10% of the most fault-prone files that account for the most of the system's

[8]It is not clear, whether post-release defects include the defects reported by the test team only or by the customers, too.

[9]Concentration of defects on a small number of files gets increases as system matures.

defects make up about 35% of the code mass. This result is similar to the results obtained in this paper.

The authors in [NZ+07] aim at predicting vulnerable components. In their case study, they report on the distribution of vulnerabilities across the Mozilla code base with the result that 4% of the components are involved in security fixes. There are several other studies that focus on predicting the defect count of a software entity using several indicators. Amongst others, in [GK+00], [AB06], [KA+98], [OWB05], [BOW06], [SC06], history characteristics (e.g. a file's age or previous defects) and product characteristics (e.g. a file's code complexity) are combined in order to predict the defect count. In [ZNZ08], indicators like complexity, problem domain, evolution and the development process are evaluated with respect to their ability to predict defects. A huge amount of research papers analyse the influence of other metrics of a software entity and its defect count, amongst others in [DP02], [NBZ06], [ZPZ07] and [GFS05]. In contrast to our study, the aim of these studies is defect prediction. Our main goal is to analyse the Pareto Principle on a large data set in order to increase the empirical body of knowledge in this area. If confirmed, the Pareto Principle can be used as basis for defect prediction as presented in [KZ+08].

6 Summary and Conclusions

In this paper, we presented the results of an empirical study on the distribution of defects in software.In contrast to most of the studies considering a small number of commercial systems, we analysed the fault distribution in a wide range of open source projects across several releases. From the research point of view, this study increases the empirical body of knowledge. Replication of studies is advocated in order to gain confidence in the results instead of relying on single studies with specific context [Pf05], [BL99].

Two of our initial hypotheses could be confirmed: A small number of files accounts for the majority of the defects (*Hypothesis 1*). This is true even across several releases of software (*Hypothesis 2*). The results widely correspond to the findings reported in literature.

Similarly to the results reported in literature, we did not find evidence for our initial hypotheses concerning the distribution of defects in code (*Hypothesis 3)*. Defects concentrate on a small part of the files but they **do not** concentrate on a small part of the code. One reason for this could be that a considerable part of an application's logic is concentrated on few files that are error-prone and not well understood. These files are candidates for refactoring and should be considered by maintainers. In addition, for fault-prone parts of the code, the unit test coverage criteria should be intensified. These files should also be higher prioritized during regression testing.

One of the goals of the study has been to analyse the Pareto Distribution of defects. If confirmed, we expected to be able to give advices to testers on which parts of the software under test to concentrate their limited resources. Despite of the confirmation of the Pareto Distribution for files, defects are not concentrated on a small part of code.

Consequently, detecting which 20% of the files account for most of the defects is useful for testers, but not enough to prioritize testing activities because these 20% of the files possibly account for a high part of the code and hence of an application's logic. For this purpose, additional indicators, e.g. a file's age or its complexity should be used in combination with a Pareto analysis in order to give reliable advices to testers on which parts of the software testing activities should be focused. Algorithms like those presented in [KZ+08] that determine the most fault-prone files are also only useful when considering the amount of code covered by these files. The Pareto analysis is useful for maintainers too. The parts of code responsible for a high amount of software's defects are candidates for refactoring.

Acknowledgements

We thank the anonymous referees for their valuable comments.

References

[AB06] Arisholm, E.; Briand, L. C.: Predicting fault-prone components in a java legacy system. In Proceedings of the 2006 ACM/IEEE international Symposium on Empirical Software Engineering, ISESE '06. ACM, New York, NY, 2006, 8-17.

[AR07] Andersson, C.; Runeson, P.. A Replicated Quantitative Analysis of Fault Distributions in Complex Software Systems. IEEE Trans. Softw. Eng. 33, 5, May. 2007.

[BL99] Basili, V. R.; Lanubile, F.: Building Knowledge through Families of Experiments. IEEE Transactions of Software Engineering, Vol 25, Issue 4, 1999, 456-473.

[BOW06] Bell, R. M.; Ostrand, T. J.; Weyuker, E. J.: Looking for bugs in all the right places. In Proceedings of the 2006 international Symposium on Software Testing and Analysis (ISSTA '06). ACM, New York, 2006, 61-72.

[CM03] Cubranic, D.; Murphy, G.C.: Hipikat: Recommending pertinent software development artefacts. 25th International Conference on Software Engineering (ICSE 2003), Portland, Oregon, 2003, 408-418.

[DP02] Denaro, G. and Pezzè, M.: An empirical evaluation of fault-proneness models. In Proceedings of the International Conference on Software Engineering (ICSE), Orlando, Florida, USA, 2002, 241-251.

[En79] Endres, A.: An analysis of errors and their causes in system programs. SIGPLAN Not. 10, 6 (Jun. 1975), 1975, 327-336.

[FO00] Fenton, N. E.; Ohlsson, N.: Quantitative Analysis of Faults and Failures in a Complex Software System. IEEE Trans. Softw. Eng. 26, 8 (Aug. 2000), 2000, 797-814.

[FPG02] Fischer, M.; Pinzger, M., Gall, H.: Populating a release history database from version control and bug tracking systems. Proceedings of the International Conference on Software Engineering (ICSE 2002), Orlando, Florida, USA, 2002, 241-251.

[GFS05] Gyimothy, T.; Ferenc, R.; Siket, I.: Empirical Validation of Object-Oriented Metrics on Open Source Software for Fault Prediction. IEEE Trans. Softw. Eng. 31, 10, 2005, 897-910.

[GK+00] Graves, T. L.; Karr, A. F.; Marron, J. S.; Siy, H.: Predicting fault incidence using software change history. IEEE Transactions on Software Engineering, vol. 26, 2000.

[Ha08] Hatton, L.: The role of empiricism in improving the reliability of future software, Keynote Talk at TAIC PART 2008, http://www.leshatton.org/Documents/TAIC2008-29-08-2008.pdf, last visited October 2008.

[IS07] Standard glossary of terms used in Software Testing Version 2.0 (December, 2nd 2007), Produced by the 'Glossary Working Party' International Software Testing Qualifications Board, http://www.istqb.org/downloads/glossary-current.pdf

[JG88] Juran, J.M.; Gryna, Jr.F.M.: Quality Control Handbook (4th edition), McGraw Hill, 1988.

[KA+98] Khoshgoftaar, T. M.; Allen, E. B.; Halstead, R.; Trio, G. P.; Flass, R. M.: Using Process History to Predict Software Quality. Computer 31, 4, 1998, 66-72.

[KK96] Kaaniche, M.; Kanoun, K.: Reliability of a commercial telecommunications system. In Proceedings of the the Seventh international Symposium on Software Reliability Engineering (ISSRE '96), IEEE Computer Society, Washington, DC, 207, 1996.

[KZ+08] Kim, S.; Zimmermann, T.; Whitehead, E. J.; Zeller, A.: Predicting faults from cached history. Proceedings of the 1st Conference on India Software Engineering Conference ISEC, Hyderabad, India, 15-16, ACM, New York, 2008.

[MK92] Munson, J. C.; Khoshgoftaar, T. M.: The Detection of Fault-Prone Programs. *IEEE Trans. Softw. Eng.* 18, 5 (May. 1992), 1992, 423-433.

[NBZ06] Nagappan, N.; Ball, T.; Zeller, A.: Mining metrics to predict component failures. In Proceedings of the 28th international Conference on Software Engineering ICSE '06. ACM, New York, NY, 2006, 452-461.

[NZ+07] Neuhaus, S.; Zimmermann, T.; Holler, C.; Zeller, A.: Predicting vulnerable software components. In Proceedings of the 14th ACM Conference on Computer and Communications Security, Alexandria, Virginia, USA, ACM, New York, NY, 2007, 529-540.

[OA96] Ohlsson, N.; Alberg, H.: Predicting Fault-Prone Software Modules in Telephone Switches. IEEE Trans. Softw. Eng. 22, 12 (Dec. 1996), 886-894, 1996.

[OW02] Ostrand, T. J.; Weyuker, E. J.: The distribution of faults in a large industrial software system. SIGSOFT Softw. Eng. Notes 27, 4 (Jul. 2002), 2002, 55-64.

[OWB05] Ostrand, T. J.; Weyuker, E. J.; Bell, R. M.: Predicting the location and number of faults in large software systems. IEEE Trans. Software Eng., vol. 31, 2005, 340-355.

[Pf05] Pfleeger, S. L.: Soup or Art? The Role of Evidential Force in Empirical Software Engineering. IEEE Softw. 22, 1 (Jan. 2005), 2005, 66-73.

[Re05] Reh, J.F.: Pareto's Principle - The 80-20 Rule, How the 80/20 rule can help you be more effective, about.com Management, http://management.about.com/cs/generalmanagement/a/Pareto081202_2.htm, last visited, December 2008.

[Sc06] Schröter, A.; Zimmermann,T.; Premraj, R.; Zeller, A.: If Your Bug Database Could Talk. Proceedings of the 5th International Symposium on Empirical Software Engineering, Volume II: Short Papers and Posters, 2006, 18-20.

[SZZ05] Sliwersky, J.; Zimmermann, T.; Zeller; A.: When do changes induce fixes? On Fridays. Proceedings of the International Workshop on Mining Software Repositories (MSR 2005), St. Louis, Missouri, USA, 2005.

[WR+00] Wohlin, C.; Runeson, P.; Höst, M.; Ohlsson, M.C.; Regnell, B.; Wesslén, A.: Experimentation in Software Engineering: An Introduction. Kluwer Academic Publishers, 2000.

[ZNZ08] Zimmermann, T.; Nagappan, N.; Zeller, A.: Predicting Bugs from History. In Software Evolution, 2008, 69-88.

[ZPZ07] Zimmermann, T.; Premraj, R.; Zeller, A.: Predicting Defects for Eclipse. Proceedings of the Third International Workshop on Predictor Models in Software Engineering (PROMISE), Minneapolis, USA, 2007.

Kurzbeiträge

Transaktionsorientiertes Mischen von Modellen

Maik Schmidt, Sven Wenzel, Udo Kelter
Praktische Informatik, Universität Siegen
{mschmidt,wenzel,kelter}@informatik.uni-siegen.de

Abstract: Modellbasierte Softwareentwicklung wird zunehmend gängige Praxis. Da immer umfangreichere Modelle arbeitsteilig in Teams entwickelt werden, sollte das parallele Bearbeiten von Modellen durch ähnliche Mischfunktionen unterstützt werden, wie man es für textuelle Dokumente gewohnt ist. Wir analysieren zunächst die technischen Randbedingungen bei der modellbasierten Softwareentwicklung; demzufolge müssen Mischprozesse bei Modellen anders gestaltet werden als bei Texten. Auf dieser Basis stellt das Papier ein Design für Mischwerkzeuge vor, das einen hohen Grad an Konsistenz der Mischergebnisse erzielt und gleichzeitig Verluste an Arbeit infolge von Konfliktauflösungen minimiert.

1 Motivation

Modellbasierte Softwareentwicklung wird zunehmend gängige Praxis, sowohl unter Nutzung allgemeiner Modellierungssprachen wie der UML oder von Modelltypen wie Matlab/Simulink oder ASCET, die für bestimmte Domänen spezialisiert sind. Zudem werden inzwischen größere Modelle arbeitsteilig entwickelt, was üblicherweise durch Repositories unterstützt wird [EKSW07]. Ein klassisches Problem besteht in der gleichzeitigen Bearbeitung desselben Dokumentes durch verschiedene Teammitglieder, da die parallelen Änderungen wieder zusammengeführt werden müssen. Für die modellbasierte Softwareentwicklung braucht man daher Mischwerkzeuge, die analoge Leistungen wie Mischwerkzeuge für textuelle Dokumente anbieten. Das Mischen von Modellen unterscheidet sich in einigen Randbedingungen erheblich vom Mischen textueller Dokumente. In Abschnitt 4 stellen wir ein Mischverfahren für Modelle vor, das einen hohen Konsistenzgrad der Mischergebnisse gewährleistet und zugleich die Vollständigkeit der Mischung maximiert.

2 Begriffsdefinitionen

3-Wege-Mischen. Beim 3-Wege-Mischen sind zwei Dokumente D1 und D2 gegeben, die beide Revisionen einer Basisversion D0 sein müssen. Wir unterstellen immer, dass alle Dokumente den gleichen Typ haben. Die Mischung ist ein neues Dokument, das möglichst alle Änderungen in D1 bzw. D2 gegenüber D0, wozu auch Löschungen zählen, enthalten soll. Die Änderungen in D1 bzw. D2 gegenüber D0 können im Prinzip auf beliebige Weise bestimmt werden, z.B. durch Protokollierung in Editoren [SZN04]; üblicher ist die

Berechnung einer asymmetrischen Differenz [KSW08] von D0 nach D1 bzw. D2 oder einer speziellen 3-Wege-Differenz.

Editierdatentyp. Wir unterstellen, dass für jeden Modelltyp elementare Editieroperationen definiert sind, u.a. Operationen zum Einfügen, Löschen und Ändern von Modellelementen. Konzeptuell liegen diese Operationen den Kommandos in Editoren zugrunde. Wir unterstellen ferner, dass die Änderungen in D1 bzw. D2 gegenüber D0 mittels dieser Operationen ausgedrückt werden und dass diese Operationen durch einen abstrakten Datentyp definiert sind. Wir bezeichnen diesen Datentyp als den Editierdatentyp. Editierdatentypen unterstellen immer eine bestimmte Repräsentation der Modelle, i.d.R. als abstrakter Syntaxgraph mit mehr oder minder umfangreichen kontextsensitiven Konsistenzbedingungen.

Modelleditoren können nur solche Modelle darstellen und verarbeiten, die gewisse Konsistenzbedingungen einhalten. Mischverfahren arbeiten dagegen oft auf einer "internen" Repräsentation der Modelle, die diese Konsistenzbedingungen nicht einhält. Die Mischergebnisse sind dann nicht konsistent genug für graphische Editoren und müssen z.B. als XML-Datei mit entsprechenden Editoren korrigiert werden.

Konflikte und Mischentscheidungen. Ein Paar von je einer Änderung aus D1 bzw. D2 gegenüber D0 kann unverträglich insofern sein, dass beide Änderungen zusammen technisch nicht möglich sind[1] oder einen Modellzustand erzeugen, der bestimmte, vom Modelltyp abhängende Konsistenzbedingungen verletzt. Ein solches Paar von Änderungen bildet einen **Konflikt**. Änderungen, die nicht in Konflikt stehen, können gemeinsam in das Mischergebnis übernommen werden. Bei einem Konflikt muss hingegen eine der beiden Änderungen weggelassen werden. Dies bezeichnen wir als eine **Mischentscheidung**. Hierfür sind zwei Vorgehensweisen in der Praxis bedeutend:

1. **nichtinteraktives Mischen:** Es wird zunächst ein Mischergebnis erzeugt, in dem *beide Änderungen* dargestellt werden. Hierzu müssen ggf. die veränderten Stellen dupliziert oder zusätzliche *Hilfsdaten* angelegt werden, die weder zu D1 noch D2 gehören[2] Verschiebungen und andere komplexe Operationen können so i.d.R. nicht dargestellt werden. Das Mischergebnis ist durch die Hilfsdaten oder die unverträglichen Änderungen inkonsistent und muss manuell nachbearbeitet werden. Erst hierbei wird die eigentliche Mischentscheidung getroffen. Ein Vorteil dieser Vorgehensweise ist, dass eine beliebige weitergehende Editierung zur Konfliktauflösung möglich ist. Die Variante, die Opfer der Konfliktauflösung ist, kann nach Modifikationen trotzdem in das Mischergebnis übernommen werden und muss nicht komplett neu eingegeben werden.
2. **interaktives Mischen:** Ein Entwickler wird während des Mischens um die Mischentscheidung gebeten, der Mischvorgang wird also interaktiv. Denkbare Entscheidungen sind: Übernahme der Änderung gemäss D1 oder D2, Übernahme keiner Änderung, Übernahme von modifizierten Änderungen. Bei diesem Ansatz wird garantiert, dass das Mischergebnis wieder ein konsistentes Modell ist. Hauptnachteil dieses Ansatzes ist, dass für jeden Modelltyp ein eigenes interaktives Mischwerkzeug realisiert werden muss. 3-Wege-Mischwerkzeuge sind deutlich komplexer als reine Differenzanzeigewerkzeuge [KSW08] bzw. davon relativ leicht ableitbare 2-Wege-Mischwerkzeuge.

[1] Technisch gesehen würde wenigstens eine Ausführungsreihenfolge einen Fehler erzeugen und scheitern.
[2] Die automatische Mischfunktion in RCS (bzw. CVS oder SVN) arbeitet nach diesem Prinzip.

3 Vorhandene Lösungsansätze

Das Mischen von Modellen ähnelt dem Mischen von Programmen, wenn man diese als abstrakte Syntaxgraphen betrachtet. Die Algorithmen hierfür [Mens02] orientieren sich allerdings bzgl. der Charakteristika der Graphen und beim Konfliktbegriff stark an den Strukturen in Quellprogrammen und sind auf fast alle Modelltypen nicht übertragbar. Das "Mischen" von Modellen wird weiterhin adressiert durch einige Papiere, die Repository-Systeme für Modelle vorstellen, z.B. [MCGW08]. Tatsächlich sind die dort als "Mischen" bezeichneten Funktionen eher patch-Funktionen.

Nur sehr wenige praktisch nutzbare Werkzeuge zum Mischen von Modellen sind verfügbar. In der Praxis meidet man daher notgedrungen Situationen, in denen Mischen notwendig wird [BE08], d.h. verschiedene Entwickler dürfen nur strikt nacheinander oder an disjunkt partitionierten Modellen arbeiten.

Das Graphical Merge Tool von Matlab/Simulink[3] ist ein Differenzanzeigewerkzeug für Matlab/Simulink-Modelle, in das rudimentäre Mischfunktionen integriert sind. Es zeigt zwei zu vergleichende Modelle in ihrer grafischen Repräsentation und als Objektbaum an. Im Objektbaum sind die lokalen Differenzen markiert. Man kann für einzelne lokale Differenzen die Zustände vom einen Modell in das andere übertragen. Es handelt sich um ein reines 2-Wege-Mischen, für jede einzelne lokale Differenz ist daher eine manuelle Mischentscheidung nötig; dies ist ein langwieriger und fehlerträchtiger Prozess.

Die Vergleichskomponente des Rational Software Architect[4] unterstützt neben der reinen Differenzdarstellung sowohl das 2-Wege- als auch das 3-Wege-Mischen. In der grafischen Darstellung werden das Basisdokument, die zu mischenden Varianten sowie das vorläufige Mischergebnis in der gewohnten grafischen Notation angezeigt. Zudem wird eine Liste der Mischkonflikte angezeigt. Zusammenhängende Änderungen, die sich z.B. auf das gleiche Modellelement beziehen, werden gruppiert. Bei jedem Konflikt ist eine der möglichen Änderungen interaktiv zu wählen. Für Gruppen von Konflikten können alle Änderungen des einen oder des anderen Dokuments ausgewählt werden, wodurch die Zahl der Einzelentscheidungen reduziert wird. Zwei in Konflikt stehende Änderungen können mit dem Rational Software Architect nicht in das Mischergebnis übernommen werden.

4 Transaktionsorientiertes Mischen von Modellen

In diesem Abschnitt stellen wir ein Verfahren zum 3-Wege-Mischen von Modellen vor, das sich weniger an den aktuellen Zuständen der zu mischenden Modelle orientiert, sondern vielmehr an den elementaren Editierungen, die zu diesen Zuständen geführt haben.

[3] http://www.mathworks.de/products/simulink, 2007
[4] http://www.ibm.com/software/awdtools/architect/swarchitect, 2008

4.1 Transaktionen

Editiervorgänge innerhalb eines Versionszweigs lassen sich in Gruppen einteilen, die jeweils einer bestimmten Arbeitsaufgabe entsprechen, z.B. dem Beheben eines Fehlers oder dem Einfügen einer neuen Funktion, und die das Modell von einem konsistenten Zustand in einen anderen überführen. Eine solche Gruppe von Editieroperationen bezeichnen wir als *Transaktion*. Eine Grundidee des transaktionsorientierten Mischens ist, die Transaktionen, also Sequenzen von Änderungen, als "Bürger erster Ordnung" aufzufassen, die als Datenobjekte editiert und ggf. sogar dauerhaft gespeichert[5] werden.

Werden Modelle mit einem Versionsmanagementsystem verwaltet, lassen sich die Änderungen zwischen zwei Revisionen in erster Näherung als eine Transaktion ansehen. Bei diszipliniertem Gebrauch dokumentiert die *commit message* sogar den Zweck der Transaktion. Das Mischwerkzeug sollte Funktionen anbieten, mit denen man solche "Roh-Transaktionen" zusammenlegen oder aufteilen kann, wenn z.B. mehrere fachliche Änderungen innerhalb einer Revision umgesetzt wurden. Die Editieroperationen in einer Transaktion können im Prinzip durch Vergleich der beiden Revisionen (vgl. [KWN05]) bestimmt werden. Um allerdings Modellelemente über die ganze Versionshistorie hinweg korrekt zu identifizieren, muss die komplette Versionshistorie bis zum gemeinsamen Basisdokument analysiert werden. Eine Technik hierzu ist in [WHK07] beschrieben. Alternativ kann man Transaktionen auch durch Protokollierung in Editoren und anschließende Bereinigung der Protokolle gewinnen.

4.2 Abhängigkeiten und Konflikte

Editieroperationen werden immer auf einem Kontext, der aus referierten Modellelementen besteht, ausgeführt und erzeugen, verändern oder entfernen Modellelemente. Zwischen zwei *Editieroperationen im gleichen Zweig* liegt eine **Abhängigkeit** vor, wenn die spätere Operation auf Ergebnissen der früheren Operation aufbaut. Zwischen zwei *Editieroperationen in verschiedenen Zweigen* liegt ein **Konflikt** vor, wenn sie im einfachsten Fall dasselbe Element verändern oder den benötigten Kontext der anderen Operation verändern. Allgemeiner liegt ein Konflikt vor, wenn beim gegebenen Editierdatentyp die beiden Operationen bei Hintereinanderausführung andere Ergebnisse erzeugen, wenn man die Ausführungsreihenfolge vertauscht. Je nach Editierdatentyp sind "nichtlokale" Konflikte möglich, z.B. durch Kardinalitäten im Metamodell: zwei Editieroperationen, die jeweils ein Element anlegen, können zusammen eine Kardinalitätsgrenze überschreiten, d.h. die zweite Operation würde bei Hintereinanderausführung scheitern.

Zwei *Transaktionen* sind abhängig voneinander bzw. stehen in Konflikt, wenn sie wenigstens ein Paar von Editieroperationen enthalten, die abhängig voneinander sind bzw. in Konflikt stehen.

[5] Bei scharfen Produkthaftungsbedingungen, die z.B. für eingebettete Software in Fahrzeugen typisch sind, müssen ohnehin alle Entwicklungsschritte genauestens dokumentiert werden.

4.3 Transaktionsorientiertes Mischen

Die Gruppierung einzelner Editieroperationen zu Transaktionen kann den Mischprozess in mehrfacher Hinsicht vereinfachen und verbessern: Die Transaktionen gruppieren die aus Anwendersicht i.d.R. zu kleinteiligen Einzeloperationen und bilden in sich geschlossene Änderungen, die ggf. einzelnen Änderungsanforderungen zugeordnet werden können. Mischentscheidungen werden so verständlicher und ggf. ohne Mehraufwand dokumentiert. Die Verwendung von Transaktionen als Granularität der Mischentscheidungen verringert die Anzahl der manuell zu entscheidenden Konflikte und stellt zudem sicher, dass Mischergebnisse im Sinne der intendierten Änderungen semantisch vollständig sind. Der Mischvorgang auf Basis von Transaktionen beinhaltet folgende Hauptschritte:

1. Automatische Analyse der Historien einzelner Modellelemente gemäß [WHK07].
2. Definition der Transaktionen: in einfachsten Fall anhand von check-ins in einem Repository, ggf. manuelle Nachbearbeitung (s.o.)
3. Abhängigkeiten und Konflikten zwischen Operationen und Transaktionen bestimmen.
4. Berechnung des "Transaktionswertes" und anderer Hilfsdaten; der **Wert einer Transaktion** ist typischerweise definiert als die Zahl der enthaltenen Änderungen, die je nach involvierten Elementtypen gewichtet werden. Allgemeiner können Transaktionen mittels Differenzmetriken [Wen08] bewertet werden. Die Bewertung unterstützt später die Auswahl der in das Mischergebnis übernommenen Änderungen.
5. Berechnung eines initialen Vorschlags für Mischentscheidungen auf Transaktionsebene: Oft ist es sinnvoll, einem der beiden Zweige höhere Priorität zuzuordnen und bei allen Konflikten zunächst zugunsten dieses Zweiges zu entscheiden. Alternativ kann es sinnvoll sein, den Gesamtwert der gewählten Transaktionen anhand einfacher Heuristiken zu optimieren, z.B. die jeweils wertvolleren Transaktionen (unter Berücksichtigung von abhängigen Transaktionen) zu wählen. Weitergehende Optimierungen können darin bestehen, z.B. die Zahl bestimmter Modellelemente im Ergebnis zu maximieren.
6. Iterative manuelle Korrektur der Mischentscheidungen auf Transaktionsebene, jeweils Neuberechnung des Gesamtwerts der gewählten Transaktionen
7. Manuelle Behandlung der "geopferten" Transaktionen: i.w. wird hier die Operationsliste einer solchen Transaktion editiert und die modifizierte Transaktion am Ende freigegeben. Hierzu sollten für eine betroffene Transaktion die darin enthaltenen Editieroperationen angezeigt werden. Ein Entwickler mit entsprechendem Domänenwissen kann dann die Operationsliste verändern, indem einzelne Operationen hinzugewählt oder deselektiert werden oder indem Parameter modifiziert werden. Während die vorigen Schritte nur komplette Transaktionen in das Mischergebnis übernehmen und damit annahmegemäß automatisch die Konsistenz der Modelle erhalten, liegt die Verantwortung für die Konsistenzerhaltung hier beim Entwickler.
8. Generierung eines Mischergebnisses auf Basis der Mischentscheidungen

Die ausgewählten Transaktionen bzw. Einzeloperationen, die in das Mischergebnis aufgenommen werden sollen, lassen sich einfach speichern. Dies bietet zwei Vorteile: erstens kann der Mischvorgang jederzeit unterbrochen und später fortgesetzt werden. Zweitens

kann ein Test des Mischergebnisses zeigen, dass bei Editieren der geopferten Transaktionen Fehler gemacht wurden; diese Fehler können behoben werden und das Mischergebnis kann leicht neu generiert werden.

5 Fazit

Das Mischen von Modellen wird bislang nur unzureichend von Werkzeugen unterstützt. In diesem Papier haben wir die wesentlichen Probleme beim Mischen von Modellen analysiert; viele Prinzipien und Details von Mischvorgängen, die sich bei textuellen Dokumenten bewährt haben, können nicht für Modelle übernommen werden. Ferner haben wir ein Mischverfahren vorgestellt, das die Änderungshistorie der Modelle ausnutzt und eine Mischung auf Basis von Transaktionen erlaubt. Hierdurch wird die Zahl der Mischentscheidungen reduziert. Während konventionelle Mischverfahren sich vor allem an den Zuständen der zu mischenden Dokumente orientieren, ist dieses Verfahren änderungs- und historienorientiert und bezieht sich explizit auf einen konsistenzerhaltenden Editierdatentyp für den jeweiligen Modelltyp. Hierdurch und weil Transaktionen konsistente Zustände erhalten, wird die Überwachung der syntaktischen und semantischen Korrektheit der Mischung unterstützt. Das Verfahren erlaubt Mischenentscheidungen sowohl auf der Ebene von der Transaktionen als auch auf der Ebene einzelner Editieroperationen, und reduziert hierdurch den Verlust an geleisteter Entwicklungsarbeit.

Literatur

[BE08] Bendix, L.; Emanuelsson, P.: Diff And Merge Support For Feature Oriented Development; In: Proc. ICSE Workshop CVSM, 2008; S.31-34

[EKSW07] Ebert, J.; Kelter, U.; Schürr, A.; Westfechtel, B.: Workshopbericht Vergleich und Versionierung von UML-Modellen, Hamburg; Softwaretechnik-Trends 27:2; 2007

[KSW08] Kelter, U.; Schmidt, M.; Wenzel, S.: Architekturen von Differenzwerkzeugen für Modelle; S.155-168 in: Software Engineering 2008, München; 2008

[KWN05] Kelter, U.; Wehren, J.; Niere, J.: A Generic Difference Algorithm for UML Models; In: Proc. GI-Fachtagung Software Engineering, Essen, 2005

[Mens02] Mens, T.: A State-of-the-Art Survey on Software Merging; In: IEEE Transactions on Software Engineering 28:5; 2002

[MCGW08] Murta, L. et al: Towards Odyssey-VCS 2: Improvements Over A UML-based Version Control System; In: Proc. ICSE Workshop CVSM, 2008; S.25-30

[SZN04] Schneider, C.; Zündorf, A. ; Niere, J.: CoObRA - a small step for development tools to collaborative environments; In: Proc. WoDiSEE, 2004

[Wen08] Wenzel, S.: Scalable Visualization of Model Differences In: Proc. ICSE Workshop on Comparison and Versioning of Software Models, Leipzig, 2008; S.41-46

[WHK07] Wenzel, S.; Hutter, H.; Kelter, U.: Tracing model elements; In: Proc. of the 23rd International Conference on Software Maintenance, Paris, 2007

Eine Taxonomie für Aufgabenmodelle

Gerrit Meixner, Daniel Görlich

Zentrum für Mensch-Maschine-Interaktion (ZMMI)
Deutsches Forschungszentrum für Künstliche Intelligenz (DFKI)
Trippstadter Str. 122, 67663 Kaiserslautern
{Gerrit.Meixner, Daniel.Goerlich}@dfki.de

Abstract: Dieser Beitrag beschreibt eine Taxonomie für Aufgabenmodelle, die ermöglicht, Aufgabenmodelle auf ihre Verwendbarkeit im Rahmen eines modellbasierten Entwicklungsprozesses hin zu analysieren und zu evaluieren. Dadurch können vorhandene Aufgabenmodelle geprüft und verbessert werden. Zudem erhalten Entwickler durch diese Taxonomie ein Hilfsmittel zur Auswahl eines passenden Aufgabenmodells für ihre eigenen Entwicklungsprozesse. Exemplarisch wird die Taxonomie auf die Useware Markup Language (useML) angewandt, wodurch Verbesserungspotenziale von useML aufgezeigt werden.

1 Einleitung

Schon seit langem wird versucht, die Interaktion zwischen Mensch und Maschine zu vereinfachen und zu verbessern. Heute erfolgt sie zumeist über grafische Benutzungsschnittstellen, doch Szenarien wie die Interaktion von Menschen mit Geräten mittels multimodaler Schnittstellen (z.B. visuell, haptisch, akustisch) werden in Zukunft eine bedeutendere Rolle spielen [Zü04]. Auch die stetig steigende Anzahl heterogener Plattformen (PC, Smartphone, PDA, etc.) führt dazu, dass Benutzungsschnittstellen auf einer Vielzahl von Zielplattformen konsistent gehalten werden müssen, damit ihre intuitive Handhabung und somit die Zufriedenheit der Nutzer gewährleistet wird [Lu04]. Aspekte wie Wiederverwendbarkeit, Flexibilität und Plattformunabhängigkeit gewinnen daher bei der Entwicklung von Benutzungsschnittstellen zunehmend an Bedeutung. Um den wiederkehrenden Entwicklungsaufwand für Einzellösungen jeweils für spezifische Plattformen, Modalitäten oder Nutzungskontexte zu mindern, bietet sich ein modellbasierter Ansatz an, der die Bedürfnisse und Anforderungen der Nutzer in den Fokus rückt. Im Zentrum eines solchen modellbasierten, nutzerzentrierten Entwicklungsprozesses muss ein Aufgabenmodell zur Repräsentation von Nutzeraufgaben stehen [PTV03]. Die Vielzahl verfügbarer Modellierungssprachen für Aufgabenmodelle erschwert allerdings die Auswahl derjenigen Sprache, die am besten zum jeweiligen Projekt passt. Im Rahmen dieses Beitrages wird daher eine Taxonomie eingeführt, die ermöglicht, Aufgabenmodellierungssprachen auf ihre Einsetzbarkeit in ganzheitlichen, modellbasierten Entwicklungsprozessen für Benutzungsschnittstellen hin zu evaluieren. Die Taxonomie wird zunächst allgemein vorgestellt (Kapitel 2) und anschließend exemplarisch auf die Useware Markup Language (useML) angewandt (Kapitel 3). Kapitel 4 gibt nach der Zusammenfassung einen Ausblick auf weiterführende Arbeiten.

2 Die Taxonomie und ihre Bewertungskriterien

Die Taxonomie setzt ihren Bewertungsfokus auf die Integration eines Aufgabenmodells in eine Architektur zur modellbasierten Entwicklung von Benutzungsschnittstellen, die möglichst auch konsistent mit verschiedenen Modalitäten und auf verschiedenen Plattformen bzw. Endgeräten einheitlich zu bedienen ist. Zur Evaluierung von Aufgabenmodellen werden Kriterien definiert, die auf den Arbeiten von [WVE98] und [BOP03] basieren, diese aber um zusätzliche Kriterien ergänzen. Sie alle werden nun näher beschrieben, wobei auch Korrelationen zwischen mehreren Kriterien aufgezeigt werden.

Kriterium 1: Modellierungsmächtigkeit. Ein Aufgabenmodell muss dem Entwickler die Konzentration auf Aufgaben, Aktivitäten und Handlungen erlauben [Pa99]. Dabei stellt die **Granularität der Aufgabenbeschreibung** einen vorrangigen Aspekt dar. Der Einsatz des Aufgabenmodells in einem modellbasierten Entwicklungsprozess erfordert verschiedene Abstraktionsstufen [LPV01]: Mit abstrakten Aufgaben wird die Modellierung übergeordneter Aufgaben beschrieben, während konkrete Aufgaben elementare bzw. atomare Handlungen sind. Allzu feingranulare, konkrete Handlungen wie z.B. Tastendrücke in GOMS [CMN83] sollten aber nicht modelliert werden [Pa03], weil dadurch bereits eine bestimmte Eingabemodalität (hier: eine Tastatur) vorausgesetzt würde.

Nach allgemeiner Auffassung besitzen Menschen mental zumeist individuelle, hierarchische Abbildungen von auszuführenden Aufgaben [Se88]. Die **hierarchische Struktur** entspricht somit der intuitiven Vorgehensweise von Menschen zur Problemlösung und Aufgabenbewältigung. Konsequent werden dabei komplexe Aufgaben in einfacher lösbare Unteraufgaben aufgeteilt [Di00]. Allerdings können Aufgaben aus unterschiedlichen Perspektiven modelliert werden [PTV03] – speziell sollten Aufgabenmodelle zwischen **interaktiven Benutzer-** und **reinen Systemaufgaben** unterscheiden [BS98].

Ein weiterer Aspekt der Modellierungsmächtigkeit betrifft den **Grad der Formalisierung**. Vielfach werden informelle Beschreibungen (Use Cases, Instructional Text) als Ausgangsbasis für die Modellierung genutzt, jedoch mangelt es diesen Beschreibungen an einer formalen Basis, was ihre nahtlose Integration in einen modellbasierten Entwicklungsprozess erschwert. Zur automatischen Prüfung von Aufgabenmodellen auf Korrektheit müssen eine ausdruckfähige Semantik und eine klare Syntax bereitgestellt werden. Als optimal gilt ein **semi-formaler Formalisierungsgrad** [OPB98].

Temporaloperatoren erlauben, zeitliche Abhängigkeiten und Ordnungen semantisch ausdrucksstark zu modellieren [Di04] [Pa99] [Di00]. Für die Generierung von Dialogmodellen aus Aufgabenmodellen werden Transformatoren benötigt, die zunächst Temporaloperationen des Aufgaben- auf Transitionen des Dialogmodells abbilden. Die zeitliche Abfolge einzelner Aufgaben ist ein wesentlicher Aspekt der Entwicklung von Benutzungsschnittstellen [Pa03] und öffnet erst den Weg zu vollständig modellbasierter Entwicklung [LPV01]. Auch die Kennzeichnung der **Optionalität** von Aufgaben ist essentiell [BOP03]. Mit ihr eng verbunden erlaubt die Angabe der **Mehrfachausführung** (Iterationen, Kardinalität), Ausführungshäufigkeiten für Einzelaufgaben zu formalisieren. Zuletzt können **Konditionen** festlegen, unter welchen logischen [Re03] oder zeitlichen Bedingungen Aufgaben überhaupt ausgeführt werden. Letztere werden, im

Gegensatz zu Temporaloperatoren, als quantitative zeitliche Aspekte der Aufgabenmodellierung bezeichnet. Kaum ein Aufgabenmodell ist jedoch in der Lage, quantitative und qualitative zeitliche Aspekte zugleich explizit zu modellieren.

Kriterium 2: Integrationsfähigkeit. Nur wenn ein Aufgabenmodell in eine modellbasierte Architektur zur Generierung von Benutzungsschnittstellen integrierbar ist, können nutzerzentrierte Benutzungsschnittstellen automatisch erzeugt werden [LPV01]. Daher sollten Aufgabenmodelle nicht isoliert, sondern ganzheitlich im Zusammenspiel bspw. mit Dialog- und Präsentationsmodellen betrachtet werden. Dabei ist speziell die eindeutige Identifikation aller modellierten Aufgaben zu beachten, um ihre konsistente Zuordnung zu Interaktionsobjekten der Benutzungsschnittstelle zu gewährleisten.

Kriterium 3: Kommunizierbarkeit. Obwohl nicht explizit für die Kommunikation zwischen Menschen gedacht, eignen sich Aufgabenmodelle jedoch zur Verbesserung der Kommunikation in Teams und zum Endnutzer [PTV03]. Aufgabenmodelle können u.a. zur Darstellung von Nutzeranforderungen [BOP03], zur Evaluation [Re03], zur Simulation und zur interaktiven Validierung genutzt werden. Der Fokus dieses Kriteriums liegt daher auf der Les- und Erlernbarkeit (Verständlichkeit) des Aufgabenmodells. Eine grafische, semi-formale Notation erweist sich dabei als überaus nützlich [OPB98].

Kriterium 4: Manipulationsfähigkeit. Dieses Kriterium beschreibt, wie einfach das Aufgabenmodell durch die Entwickler erstellt und manipuliert werden kann [BB95]. Generell unterscheidet man zwischen rein textuellen Notationen wie GOMS [CMN83] und grafischen Notationen wie CTT [Pa99] und GTA [WVE98]. Bei der Erstellung von Aufgabenmodellen sind grafische Notationen deutlich gebrauchstauglicher als textuelle [Di04], da sie u.a. die hierarchische Aufgabenstruktur deutlicher visualisieren können [Pa03]. Zur Manipulation bzw. Bearbeitung von Aufgabenmodellen sind dementsprechend grafische Notationen und Editoren notwendig [PTV03].

Kriterium 5: Anpassungsfähigkeit. Dieses Kriterium subsumiert die Möglichkeiten der flexiblen Anpassung eines Aufgabenmodells an neue Situationen und Domänen. Es korreliert mit den Kriterien 1 und 6. Umfangreiche Anpassungen können etwa bei der Entwicklung von Benutzungsschnittstellen für verschiedene Modalitäten und Plattformen notwendig werden. Des Weiteren sollte ein Aufgabenmodell Informationen bspw. über Nutzergruppen oder Bedienorte einbinden können.

Kriterium 6: Erweiterbarkeit. Dieses Kriterium, korreliert mit den Kriterien 1 und 5, bewertet, wie einfach eine Notation semantisch erweitert werden kann, da bisher kein Aufgabenmodell allgemeingültig für jeden Anwendungsfall nutzbar ist [BB95]. Generell sind semi-formale Notationen einfacher erweiterbar als formale [BOP03], die auf fundierten mathematischen Theorien basieren und einfache Erweiterungen kaum zulassen.

Kriterium 7: Verarbeitungsfähigkeit. Für den Grad der (automatisierten) Verarbeitung eines Aufgabenmodells ist u.a. relevant, in welcher Form es auf einem Datenträger gespeichert werden kann. Dabei ist insbesondere auf die Verwendung offener Standards wie XML zu achten, da proprietäre Formate Austausch und Verarbeitung von Aufgabenmodellen deutlich erschweren.

3 Anwendung der Taxonomie auf die Useware Markup Language

Zur nutzer- und aufgabenorientierten Bediensystementwicklung wird am Zentrum für Mensch-Maschine-Interaktion der Useware-Entwicklungsprozess verfolgt, der Nutzeraufgaben als zentrale Elemente in so genannten Benutzungsmodellen strukturiert. Ihre Struktur und Semantik sind in der Useware Markup Language (useML 1.0 nach [Re03]) verankert, deren XML-Schema jedoch mittels beliebig definierbarer Attribute (Nutzergruppen, Bedienorte, etc.) an spezielle, projektspezifische Gegebenheiten angepasst werden kann. Die in Analysen ermittelten Aufgabenstrukturen und mentalen Modelle der Nutzer werden dabei in einer hard- und softwareunabhängigen Struktur integriert, die komplexe Aufgaben in Tätigkeiten, Handlungen und zuletzt elementare Tätigkeiten bzw. Operationen unterteilt. Jegliche komplexe Aufgaben aller Nutzergruppen an allen Geräten werden in einer Hierarchie von Benutzungsobjekten (BO's) hinterlegt, die letztlich in elementare Benutzungsobjekte (eBO's) zergliedert werden. useML sieht fünf eBO-Typen vor: ändern, auslösen, auswählen, eingeben sowie informieren [Re03] [Zü04].

Die Aufteilung des Benutzungsmodells in BO's und fünf eBO-Typen wird hier als ausreichend feingranular interpretiert. Anhand der eBO's kann direkt auf korrespondierende, abstrakte Interaktionsobjekte geschlossen werden [Re03], was die gröbere Aufgabentypisierung des de-facto-Standards CTT nicht zulässt [Lu04]. useML erfüllt zudem das Hierarchiekriterium dieser Taxonomie und kann als semi-formal angesehen werden: Zwar basiert useML nicht auf formalen mathematischen Grundsätzen wie etwa die Petri-Netze, jedoch wird der Aufbau der Sprache durch ihr XML-Schema klar definiert, was Syntax- und Konsistenzprüfungen erlaubt.

useML 1.0 rückt die Benutzer in den Vordergrund und kennt daher keine reinen Systemaufgaben. Für eine spätere Anbindung der Applikationslogik an die Benutzungsschnittstelle ist dieser Aufgabentyp aber ebenfalls nötig, da Systemaufgaben durchaus Bestandteil größerer, interaktiver Bedienhandlung sein können. Weiterhin sind in useML 1.0 keine Temporaloperatoren vorgesehen, Mehrfachausführungen von Aufgaben nicht modellierbar und BO's und eBO's können nicht als optional oder erforderlich markiert werden (Das Attribut „gewichtung", das projektspezifische, relative Werte etwa von 1 bis 10 annehmen kann, genügt nicht zur formalen Repräsentation der Optionalität). Zuletzt können zwar logische Vor- und Nachbedingungen spezifiziert werden, nicht jedoch zeitliche Bedingungen und Invarianten. Die Abwägung all jener Unterkriterien lässt daher nur zu, die **Modellierungsmächtigkeit** von useML als **niedrig** zu bewerten.

Die Integrationsfähigkeit des Benutzungsmodells in eine vollständige, modellbasierte Architektur für die Entwicklung von Benutzungsschnittstellen ist schwierig zu beurteilen, da für den Useware-Entwicklungsprozess bislang keine weiteren Modelle definiert wurden. [Lu04] bemängelt insbesondere das Fehlen eines Dialog- und Präsentationsmodells. Für die Integration in eine Gesamtarchitektur müssen das Benutzungsmodell semantisch erweitert und weitere Modelle angebunden werden. Auch eindeutige Kennungen (ID's) von Aufgaben sieht useML nicht vor: BO's und eBO's werden aktuell nur über deren Namen identifiziert, also bloße Zeichenketten, die nicht zur eindeutigen Zuordnung ausreichen. Insgesamt muss die **Integrationsfähigkeit** des Benutzungsmodells in useML daher bisher als **niedrig** eingestuft werden.

Da die Useware-Entwicklung ein interdisziplinäres Vorgehen verlangt, sollten Benutzungsmodelle einfach lesbar und useML leicht erlernbar sein. Für die Betrachtung von useML als XML-Dialekt wird prinzipiell nur ein Text- oder XML-Editor benötigt, was jedoch XML-Kenntnisse erfordert. Besser verständlich ist die webbrowser-ähnlichen Darstellung des useML-Viewers [Re03], der aber weder zur Bearbeitung von Benutzungsmodellen noch für interaktive Simulationen zur Validierung oder Evaluation entwickelt wurde. Insgesamt wird die **Kommunizierbarkeit** von useML lediglich als **mittelmäßig** bewertet. Ein vollwertiger grafischer useML-Editor zur Bearbeitung von Benutzungsmodellen steht aber kurz vor der Fertigstellung [MGS08]. Bis dahin muss useML zwar eine **niedrige Manipulationsfähigkeit** bescheinigt werden; der useML-Editor wird zukünftig aber eine **hohe Manipulationsfähigkeit** gewährleisten und die Effizienz der Erstellung von Benutzungsmodellen mit useML deutlich erhöhen.

useML wurde entwickelt, um die systematische Entwicklung von Bedienschnittstellen zu unterstützen [Re03]. Die Modellbildung ist bereits möglich, bevor Wissen über die Zielplattform(en) vorhanden ist. Zudem kennt useML explizit aus dem Hauptschema ausgelagerte Attribute (Benutzergruppen, Gerätefunktionsmodelle, etc.), die problemlos projektspezifisch angepasst werden können, ohne das Hauptschema anzutasten, wodurch useML für eine Vielzahl von Modalitäten und Plattformen anpassbar ist. So können für zahlreiche Anwendungsdomänen geeignete Schemaattribute vordefiniert werden, weshalb **Anpassungsfähigkeit** und **Erweiterbarkeit** von useML als **hoch** bewertet werden. Zuletzt ist durch die Umsetzung des Benutzungsmodells in useML als XML-Dialekt seine automatische Weiterverarbeitung möglich. Mittels spezieller Transformatoren können Prototypen direkt aus useML erzeugt werden [Re03]. Die automatische **Verarbeitungsfähigkeit** (Transformation) von useML ist somit ebenfalls **hoch** zu bewerten.

Abbildung 1: Visualisierung der Bewertung des Benutzungsmodells

Die hier angewandte Taxonomie auf useML wurde zum Zwecke der Selbstevaluierung entwickelt und um nachzuprüfen, ob useML den praktischen Anforderungen noch genügt. Zusammenfassend visualisiert Abbildung 1 das Ergebnis der Anwendung als Netzdiagramm und zeigt deutlich die aktuellen Defizite, welche wiederum Ansatzpunkte für anstehende und zukünftige Verbesserungen der Useware Markup Language darstellen.

4 Zusammenfassung und Ausblick

Die hier vorgestellte Taxonomie dient der Bewertung und Evaluation von Aufgabenmodellen, die in modellbasierten Entwicklungsprozessen für Benutzungsschnittstellen genutzt werden. Demnächst soll diese Taxonomie auf weitere Aufgabenmodelle angewandt und ein direkter Vergleich zwischen etablierten Methoden zur Aufgabenmodellierung geschaffen werden. In useML identifizierte Defizite wurden zum Teil bereits in der semantisch erweiterten Version 2.0 und im grafischen useML-Editor integriert [MGS08].

Literaturverzeichnis

[BB95] Brun, P.; Beaudouin-Lafon, M.: A taxonomy and evaluation of formalism for the specification of interactive systems. In: Proc. of the HCI'95 Conference on People and Computers X, 1995.

[BOP03] Balbo, S.; Ozkan, N.; Paris, C.: Choosing the right task modelling notation: A Taxonomy. In: Diaper, D.; Stanton, N. (Hrsg.): The Handbook of Task Analysis for Human-Computer Interaction. Lawrence Erlbaum Associates, 2003; S. 445-466.

[BS98] Bomsdorf, B.; Szwillus, G.: From task to dialogue: Task based user interface design. In: SIGCHI Bulletin, Band 30, Heft 4, S. 40-42.

[CMN83] Card, S. K.; Moran, T. P.; Newell, A.: The psychology of human-computer interaction. Lawrence Erlbaum Associates, 1983.

[Di00] Dittmar, A.: More precise descriptions of temporal relations within task models. In: Proc. of the 7th International Workshop on Interactive Systems: Design, Specification and Verification, 2000; S. 151-168.

[Di04] Dix, A. et al.: Human-Computer Interaction. Prentice Hall, 2004.

[LPV01] Limbourg, Q.; Pribeanu, C.; Vanderdonckt, J.: Towards Uniformed Task Models in a Model-Based Approach. In: Proc. of the 8th International Workshop on Interactive Systems: Design, Specification and Verification, 2001; S. 164-182.

[Lu04] Luyten, K.: Dynamic User Interface Generation for Mobile and Embedded Systems with Model-Based User Interface Development. Transnationale Universiteit Limburg, 2004.

[MGS08] Meixner, G; Görlich, D.; Schäfer, R.: Unterstützung des Useware-Engineering Prozesses durch den Einsatz einer modellbasierten Werkzeugkette. USEWARE 2008, Baden-Baden, VDI-Bericht 2041, VDI/VDE-GMA, Düsseldorf, VDI-Verlag, 2008, S. 219-232.

[OPB98] Ozkan, N.; Paris, C.; Balbo, S.: Understanding a Task Model: An Experiment. In: Proc. of HCI on People and Computers, 1998; S. 123-137.

[Pa99] Paternò, F.: Model-based design and evaluation of interactive applications, Springer, 1999.

[Pa03] Paternò, F.: ConcurTaskTrees: An Engineered Notation for Task Models. In: Diaper, D.; Stanton, N. (Hrsg.): The Handbook of Task Analysis for Human-Computer Interaction, Lawrence Erlbaum Associates, 2003; S. 483-501.

[PTV03] Paris, C.; Lu, S.; Vander Linden, K.: Environments for the Construction and Use of Task Models. In: Diaper, D.; Stanton, N. (Hrsg.): The Handbook of Task Analysis for Human-Computer Interaction, Lawrence Erlbaum Associates, 2003; S. 467-482.

[Re03] Reuther, A.: useML – systematische Entwicklung von Maschinenbediensystemen mit XML. Fortschritt-Berichte pak, Band 8. Technische Universität Kaiserslautern, 2003.

[Se88] Sebillotte, S.: Hierarchical planning as a method for task analysis: The example of office task analysis. In: Behavior and Information Technology, Band 7, Heft 3, S. 275-293.

[WVE98] van Welie, M.; van der Veer, G.; Eliens, A.: An ontology for task world models. In: Proc. of the 5th International Workshop on Interactive Systems: Design, Specification and Verification, 1998, S. 57-70.

[Zü04] Zühlke, D.: Useware-Engineering für technische Systeme, Springer, 2004.

Automating Product Derivation in Software Product Line Engineering

Goetz Botterweck
Lero, Univ. of Limerick
Limerick, Ireland
goetz.botterweck@lero.ie

Kwanwoo Lee
Hansung University
Seoul, South Korea
kwlee@hansung.ac.kr

Steffen Thiel
Lero, Univ. of Limerick
Limerick, Ireland
steffen.thiel@lero.ie

Abstract: This paper deals with deriving software products from a software product line (SPL) in an efficient and automated way. We present an approach that (1) represents the SPL with a set of integrated models, (2) specifies variability and configuration options for possible product variants and (3) automatically derives executable products with model transformations and aspect-oriented techniques. The approach is discussed with a sample SPL of scientific calculators.

1 Introduction

A software product line (SPL) is a set of software-intensive systems sharing a common, managed set of features that are developed from a common set of core assets in a prescribed way [CN02]. SPL engineering has rapidly emerged as a viable and important software development activity during the last few years. It allows companies such as Philips, Bosch, and Nokia to build functionally related products with a minimum of technical diversity and thus to achieve significant improvements in time-to-market, cost, productivity, and quality (e.g., [CN02, HFT04]). SPL engineering differs from developing single products in that variability is an inherent part of the development. Products are built by resolving this variability in order to implement customer-specific functionality. This is usually done in a dedicated product derivation process during application engineering.

Many companies attempt to achieve economies of scale in SPL engineering while keeping the number of product variants in their product lines high. Those large-scale product lines easily comprise hundreds of products and incorporate thousands of variation points and configuration parameters. This is especially the case in the mobile devices and automotive domain (e.g., [MH03, STB+04]). However, recent studies show that much of the work on product derivation is carried out manually (e.g., [DSB05]). This makes systematic product derivation extremely difficult, error-prone, and time consuming and compromises the benefits of product line adoption significantly.

In this paper we present an approach and research tool that support automated product derivation in software product lines. The approach integrates feature modelling and aspect-oriented programming (AOP) and is based on model-driven techniques. It provides the expressive means to describe a product line from two perspectives: (i) the variability and configuration options and (ii) implementation strategies for those options.

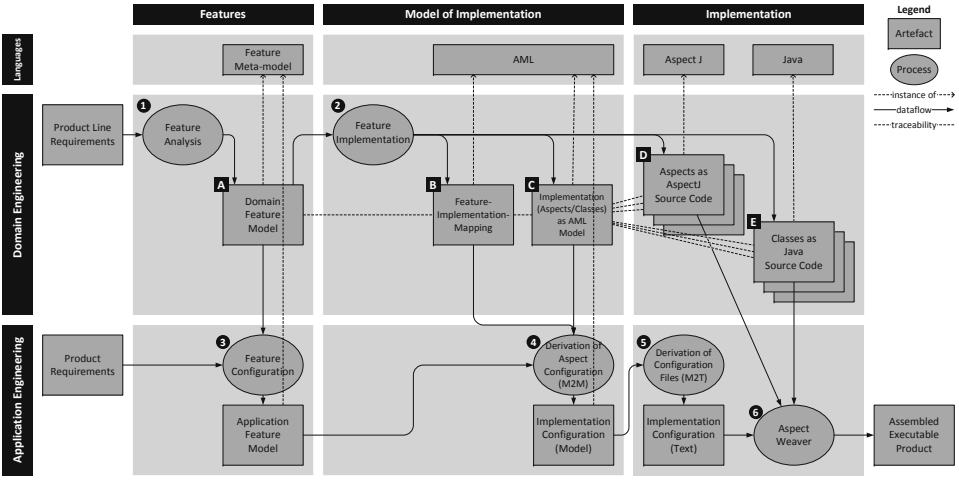


Abbildung 1: Overview of the presented approach.

The presented approach differs from existing research as it provides the explicit modelling of dependencies among features and the intentional separation of dependency implementations from feature implementations. This allows a direct mapping of feature dependencies to the corresponding dependency implementations which results in reduced coupling among feature implementation components, less side-effects, and, consequently, a more effective product derivation.

To facilitate the approach we have developed an interactive and visual derivation tool which supports the automated derivation of executable products based on a feature configuration. The remainder of the paper is structured as follows: Section 2 describes how we model a SPL, Section 3 explains how these models are used in our automated product derivation approach. In Section 4 our approach is compared with other related work. Section 5 concludes the paper.

2 Creating and Modelling a Software Product Line

In our approach the product line is described in terms of three models and two types of source code artefacts (see the markers **A** to **E** in Figure 1). These models and artefacts are created in a two-step process of Feature Analysis and Feature Implementation. We illustrate our approach based on a sample product line of calculator applications, which we created by refactoring the open source Java application *Java Scientific Calculator* [jsc08].

During *Feature Analysis* ❶ a *Domain Feature Model* **A** is created, which describes the capabilities of the SPL and potential functionality of products from a stakeholder's point of view. In addition, this model captures various dependencies among features: *Design-time*

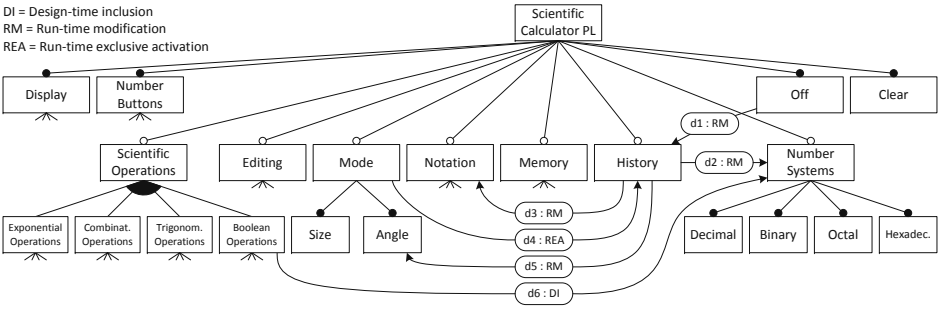


Abbildung 2: Feature Model (excerpt).

dependencies limit the available choices and guide the engineer during feature configuration ❸. *Run-time dependencies* describe the interaction among features in the executed product. Figure 2 shows the domain feature model of our Calculator SPL with mandatory features, e.g., `Display`, and optional features, e.g., `Editing`. It also shows some dependencies, some of which are design-time dependencies (d6=`BooleanOperations` requires `NumberSystems`) and some run-time dependencies (d5=`History` modifies the behaviour of `Angle`).

During *Feature Implementation* ❷ the capabilities of the software product line are implemented with *AspectJ* D and *Java* E. The *Implementation Model* C, described in our AML (Aspect-oriented implementation Modelling Language), provides an abstract view on this implementation. Just like the corresponding textual code, the AML model contains concepts like *Aspect* or *Class* but describes them as model elements. This allows to describe mappings B between features and the corresponding implementation components, which is necessary to select implementation units for a feature configuration.

Figure 3 shows some of these mappings for our Calculator example. For instance, the feature `History` is mapped to the package `history` and the dependency `d5` is mapped to the aspectual component `AngleHistoryDep.aj`.

3 Product Derivation

In the preceding section we discussed how the software product line is created and how the elicited knowledge is captured. This section describes how this knowledge is exploited during product derivation to create the assembled executable products.

Given the product-specific requirements, we start creating the product by first performing interactive *Feature Configuration* ❸. Since this activity involves the interpretation of requirements, it cannot be automated. However, it can be supported by interactive tools which provide visual guidance [BNP+07], e.g., by providing feedback based on constraints which have been captured in the Domain Feature Model A.

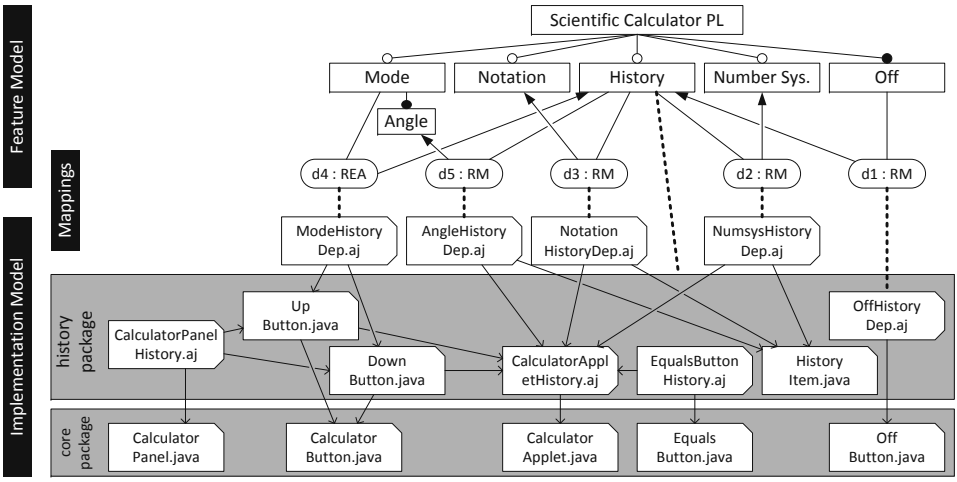


Abbildung 3: Feature-Implementation Mappings.

The feature configuration is used in *Automated Product Derivation* processes (❹ to ❻), to derive assembled executable products. The process starts with a model-to-model transformation ❹, written in ATL (Atlas Transformation Language), that creates an *Implementation Configuration* model. This model describes all components that have to be included to implement this particular feature configuration. The model is then processed by a model-to-text transformation ❺ which generates an equivalent textual configuration. Finally, AspectJ mechanisms are used to perform the assembly of the particular product ❻.

Our goal is to automate the product derivation as far as possible. Within that tool chain, only the feature configuration ❸ has to be performed *interactively*. The remaining processes (❹ to ❻) can be executed mechanically. To fully automate the approach we use Ant scripts, which orchestrate the overall process and custom Ant tasks, which integrate the various tools into this tool chain. In summary, this turns a product-specific feature configuration into an *Assembled Executable Product*.

4 Related Work

AOP techniques were originally developed to modularise crosscutting concerns. Recently, they have been used for modularising feature implementation. Godil et al. [GJ05] and Liu et al. [LBL06] applied AOP to the feature-oriented refactoring of Prevayler, an open source Java application.

Feature-oriented programming (FOP) [Bat04] is another option to implement features. FOP takes features as first-class design and implementation entities, i.e., features are designed and implemented as program refinements and composed to form a complete system.

Other techniques such as Caesar [MO04] or Framed Aspect [LR04], can be used for feature implementation. Such approaches typically support only simple feature-implementation mappings, which can lead to problems when features depend on each other. Recently, Lee et al. [LKK06] identified problems with simple mappings between features and aspects and proposed guidelines on how feature dependency information can be used for implementing features using AOP. The approach presented here differs from [LKK06] in that a systematic product derivation process is introduced.

Our approach deals with product derivation based on a feature configuration. The work in [CA05] uses templates for mapping features to different kinds of models and a model-to-model transformation to instantiate these models. However, in [CA05] the rules for filtering are described as OCL constraints, whereas we describe them as a model transformation which can partly be derived from the underlying meta-model.

The work in [VG07] is similar to our approach in that domain artifacts are expressed using models, which are then transformed. AOP is used to implement crosscutting features on code level. However, the links between features and aspects are not made explicit.

5 Conclusions

Our primary goal is to make product derivation more efficient. To this end, we (1) integrate feature modelling and AOP to structure the SPL implementation to facilitate product derivation and (2) define a model-driven product derivation process, which transforms a feature configuration into an executable product. This process is implemented as a research prototype, which automatically performs the necessary steps.

Although AspectJ has been used to demonstrate the applicability of the proposed method, more advanced techniques can be applied, e.g., Aspect-oriented architecture modelling [KTG+06]. To support software engineers in the handling of *large* SPLs we are currently extending our graphical modelling tools with improved support for the feature-implementation mappings. In addition to this we are working on techniques for reverse engineering, e.g., by extracting an AML implementation model from existing Java/AspectJ code bases. These techniques can then be used for automated analyses, e.g., to check an implementation for consistency with a specified design [JB08].

6 Acknowledgments

This work was supported, in part, by Science Foundation Ireland grant 03/CE2/I303_1 to Lero – the Irish Software Engineering Research Centre (www.lero.ie).

Literatur

[Bat04] Don Batory. Feature-Oriented Programming and the AHEAD Tool Suite. In *ICSE '04*, Seiten 702–703, Washington, DC, USA, 2004. IEEE Computer Society.

[BNP+07] Goetz Botterweck, Daren Nestor, André Preussner, Ciarán Cawley und Steffen Thiel. Towards Supporting Feature Configuration by Interactive Visualisation. In *ViSPLE 2007, collocated with SPLC 2007*, Kyoto, Japan, September 10-14, 2007 2007.

[CA05] Krzysztof Czarnecki und Michal Antkiewicz. Mapping Features to Models: A Template Approach Based on Superimposed Variants. In *GPCE'05*, 2005.

[CN02] Paul Clements und Linda M. Northrop. *Software Product Lines: Practices and Patterns*. The SEI series in software engineering. Addison-Wesley, Boston, 2002.

[DSB05] Sybren Deelstra, Marco Sinnema und Jan Bosch. Product derivation in software product families: a case study. *The Journal of Systems and Software*, 74:173–194, 2005.

[GJ05] Irum Godil und Hans-Arno Jacobsen. Horizontal Decomposition of Prevayler. In *CASCON 2005*, Seiten 83–100, 2005.

[HFT04] A. Hein, T. Fischer und S. Thiel. Produktlinienentwicklung für Fahrerassistenzsysteme. In G. Böckle, P. Knauber, K. Pohl und K. Schmid, Hrsg., *Software-Produktlinien: Methoden, Einführung und Praxis*, Seiten 193–205. dpunkt-Verlag, 2004.

[JB08] Mikolas Janota und Goetz Botterweck. Formal Approach to Integrating Feature and Architecture Models. In *FASE 2008*, Budapest, Hungary, 29 March - 6 April 2008.

[jsc08] Java Scientific Calculator. *http://jscicalc.sourceforge.net*, May 2008.

[KTG+06] Ivan Krechetov, Bedir Tekinerdogan, Alessandro Garcia, Christina Chavez und Uira Kulesza. Towards an Integrated Aspect-Oriented Modeling Approach for Software Architecture Design. In *AOM 06, collocated with AOSD 2006*, 2006.

[LBL06] Jia Liu, Don Batory und Christian Lengauer. Feature Oriented Refactoring of Legacy Applications. In *ICSE 2006*, Seiten 112–121, 2006.

[LKK06] Kwanwoo Lee, Kyo C. Kang und Minseong Kim. Combining Feature-Oriented Analysis and Aspect-Oriented Programming for Product Line Asset Development. In *SPLC 2006*, 2006.

[LR04] Neil Loughran und Awais Rashid. Framed Aspects: Supporting Variability and Configurability for AOP. In *ICSR 2004*, Seiten 127–140, Madrid, Spain, July 2004. Springer.

[MH03] Allessandro Maccari und Anders Heie. Managing Infinite Variability. In *Software Variability Management Workshop*, Seiten 28–34, February 2003.

[MO04] Mira Mezini und Klaus Ostermann. Variability Management with Feature-Oriented Programming and Aspects. In *SIGSOFT 2004*, Seiten 127–136, Newport Beach, CA, 2004.

[STB+04] Mirjam Steger, Christian Tischer, Birgit Boss, Andreas Müller, Oliver Pertler, Wolfgang Stolz und Stefan Ferber. Introducing PLA at Bosch Gasoline Systems: Experiences and Practices. In *SPLC 2004*, Seiten 34–50, Boston, MA, USA, 2004.

[VG07] Markus Völter und Iris Groher. Product Line Implementation using Aspect-Oriented and Model-Driven Software Development. In *SPLC 2007*, Kyoto, Japan, 2007.

Die Entwicklung einer gemeinsamen IT-Plattform im Bereich Logistik

Thomas Hering, André Ludwig, Bogdan Franczyk

Institut für Wirtschaftsinformatik,
Wirtschaftswissenschaftliche Fakultät, Universität Leipzig
Marschnerstraße 31
04109 Leipzig
{hering, ludwig, franczyk}@wifa.uni-leipzig.de

Zusammenfassung: Gegenstand des vorliegenden Beitrags ist ein Phasenmodell, das der Entwicklung einer branchenspezifischen Shared-Services-Plattform dient. Das Phasenmodell wurde im Rahmen eines vom Bundesministerium für Bildung und Forschung finanzierten Projektes entwickelt und wird anhand der Erstellung einer Logistik-Service-Bus-Plattform praktisch eingesetzt. Der Beitrag umreißt das gesamte Vorgehen und fokussiert aufgrund ihrer herausragenden Bedeutung die beiden ersten Phasen.

1 Einleitung[1]

Flexibilitäts-/Kosten- und Zeitdruck zwingt Unternehmen, sich auf ihre Kernkompetenzen zu konzentrieren und Prozesse, die nicht zu den primären Wertschöpfungsaktivitäten zählen an externe Dienstleister auszulagern. Unter anderem betrifft diese Auslagerungsstrategie die Informationssysteme von Unternehmen und deren Bereitstellung durch externe IT-Dienstanbieter. Eine zentrale Grundvoraussetzung an die Architektur solcher ausgelagerter Informationssysteme ist die Bereitstellung klar definierter Funktionalität, die auf modulare Weise verfügbar gemacht und entsprechend vielfältig eingesetzt werden kann. Auf dieser Basis können unterschiedliche Unternehmen die gleichen Informationssysteme nutzen, die Komplexität des einzubindenden Systems verringert sich und mit ihr der Integrationsaufwand.

[1] Das InnoProfile-Projekt Logistik-Service-Bus ist Teil von UnternehmenRegion, der Innovationsinitiative des Bundesministerium für Bildung und Forschung (BMBF). Informationen zum Projekt befinden sich auf http://www.lsb-plattform.de.

Dienstbasierte Architekturen bilden einen vielversprechenden Ansatz diese Voraussetzung zu erfüllen und extern bereitgestellte Informationssysteme über abstrakte Dienste nahtlos in Geschäftsprozesse einzubinden (Shared Services). Bisher wurde eine Aufteilung von Informationssystemen in Komponenten vor allem nach technischen Gesichtspunkten durchgeführt. Für eine Auslagerung von Informationssystemen sollte eine Aufteilung der Geschäftsfunktionalität und Kapselung in Diensten nach fachlichen Gesichtspunkten erfolgen, um eine Wiederverwendung über unterschiedliche Nutzer zu garantieren.

Im vorliegenden Beitrag wird ein Phasenmodell zur Identifizierung und Ableitung von Shared Services unter Berücksichtigung einer Vielzahl fachlicher Aspekte vorgestellt. Die dabei abgeleiteten Services sollen über eine zentrale Infrastruktur (Logistik-Service-Bus-Plattform) unterschiedlichen Logistik-Dienstleistungsunternehmen bereitgestellt werden. Die Logistik-Service-Bus-Plattform ist eine IT-Plattform, die IT-Dienste zur gemeinschaftlichen Nutzung bereitstellt. Die Dienste, die eine solche Plattform erbringt, müssen sich nahtlos in das Geschäft der Anwender integrieren. Dies ist nur möglich, wenn man das Leistungsspektrum dieser Unternehmen kennt, ihre Organisation und Prozesse beurteilen kann. Zahlreiche Themengebiete, etwa das der Unternehmensarchitekturen, des Geschäftsprozessmanagements oder des Business Engineerings können hier Beiträge leisten. Bestimmte Prozesse bzw. Teilprozesse werden nicht mehr von jedem Unternehmen selbständig, sondern durch eine dritte Partei (dem Plattform-Betreiber) erbracht (Geschäftsprozessoutsourcing). Es ist offensichtlich, dass die Entwicklung einer solchen Plattform nur dann möglich und sinnvoll ist, wenn die Funktion („Was“) und die Funktionsweise („Wie“) der einzelnen Prozesse der individuellen Unternehmen verstanden wurden und dieses Wissen auf geeignete Art und Weise zusammengeführt, aggregiert wird.

2 Phasenmodell zur Entwicklung einer Shared-Services-Plattform

Die einschlägige Fachliteratur ist hinsichtlich der Frage, welche Prozessmodelle zur Entwicklung von branchenspezifischen IT-Plattformen verwendbar sind, unergiebig. Im weiteren Verlauf des Gliederungspunktes werden ein Phasenmodell sowie die Ziele und Aktivitäten der Phasen zur Entwicklung einer Shared-Services-Plattform umrissen. Um den Umfang der Darstellung zu begrenzen, wird von der Beschreibung von Rollen, Prinzipien, Werkzeugen etc. abgesehen. Der Fokus der Darstellung liegt dabei auf den beiden ersten Phasen.

2.1 Die Phase der Marktdefinition

Grundlegende Voraussetzung ist die Existenz eines Service-Marktes und von potentiellen Service-Konsumenten. Hinsichtlich der Markdefinition sind insbesondere die folgenden drei Punkte zu beachten.

Branchenstruktur. Je breiter die Branche gewählt wird, desto geringer sind Gemeinsamkeiten zwischen den in der Branche enthaltenen Unternehmen. Breite bezieht sich dabei auf die Verschiedenartigkeit der von den branchenzugehörigen Unternehmen hergestellten Produkte bzw. angebotenen Dienstleistungen, der eingesetzten Verfahren, verwendeten Prozesse und Software. Je enger die Branche gewählt wird, desto homogener ist sie und desto ähnlicher sind deren Informationssystem-Anforderungen.

Branchengröße. Die Entwicklung und der Betrieb einer Service-Plattform basiert auf dem Umstand, dass die gemeinsame Nutzung von Services den Nutzern Kostenvorteile gegenüber der individuellen Selbsterstellung verschafft. Aufgrund der positiven Korrelation zwischen Nutzeranzahl und Wiederverwendungsgrad werden die Entwicklung und der Betrieb einer entsprechenden Plattform für einen Entwickler/Betreiber umso rentabler, je höher die Anzahl der Nutzer ist.

Branchenreife. Neben Branchenstruktur und -größe spielen Aspekte wie Branchenreife und -stabilität im Sinne stabiler, sich selten ändernder Prozesse, Beziehungen zwischen Marktteilnehmern, politischer Rahmenbedingungen eine wichtige Rolle. Dies führt zur Stabilität von Anforderungen.

Zur Definition von Branchen bestehen prinzipiell folgende Möglichkeiten:

(1) Intuition. Die einfachste Möglichkeit zur Branchendefinition besteht darin, intuitiv, d.h. basierend auf allgemeinen Beobachtungen, eine Festlegung zu treffen. Diese Option zeichnet sich durch ihre Einfachheit und Schnelligkeit aus, führt aber zu unsicheren Ergebnissen, die auf Erfahrung und subjektiver Umweltwahrnehmung beruhen.

(2) Imitation. Eine weitere einfache und schnelle Möglichkeit zur Branchendefinition besteht in der Imitation einer anderen Plattform (z.B. in einer anderen Region) oder einer bestimmten Software-Lösung (z.B. zielgruppen-/branchenspezifisches ERP-System). Ebenso wie bei einem intuitiven Ansatz sind bei dieser Variante Qualität, Passgenauigkeit sowie die Tragfähigkeit der Marktdefinition unsicher.

(3) Bestehende Kundenbeziehungen. Branchendefinition über bestehende Kundenbeziehungen ist vor allem dann eine sinnvolle Alternative, wenn der Plattform-Entwickler/-Betreiber bereits über einschlägige Produkte verfügt, eine gewisse Markt- und Branchenkenntnis besitzt und mit der zu entwickelnden Branchen-Plattform die Ergänzung bzw. Ablösung von Teilen oder des gesamten Produktportfolios intendiert.

(4) Branchenanalyse. Durch die Erhebung/Verdichtung/Auswertung von Unternehmensinformationen zur Zielbranche wird eine solide Grundlage für fachliche Geschäftsanalyse geschaffen. Die Abschätzung des Marktpotenzials der zukünftigen Plattform erhält somit eine systematische und vor allem objektive Basis (Informationsquellen z.B. Branchenverbände, Statistisches Bundesamt, Industrie- und Handelskammern). Kennzeichnend für diese Option ist das qualitativ hochwertige Ergebnis, das jedoch mit einem relativ hohen Aufwand für Informationsbeschaffung, Datensichtung und Auswertung bezahlt wird.

2.2 Die Phase der Geschäftsanalyse

In der Phase der Geschäftsanalyse werden die geschäftlichen Anforderungen des Zielmarktes untersucht. Sinnvollerweise setzt eine Geschäftsanalyse auf der Ebene der Geschäftsarchitektur (vgl. [VB06]) an, da diese Ebene ein stabiles Bild über den Aufbau des Unternehmens hinsichtlich Leistungen, Produkte, Kompetenzen etc. liefert. Anspruchsvoll ist eine solche Geschäftsanalyse deshalb, weil es nicht nur um das Verständnis *eines* sondern *mehrerer* Unternehmen geht. Vor diesem Hintergrund schlagen die Autoren vor, als Modellierungskonzept für Geschäftsarchitekturen das Konzept der organisationalen funktionalen Fähigkeiten (vgl. [Co94], [Ho06]) zu verwenden. Zwar kann auf diese Weise keine gesamte Geschäftsarchitektur modelliert werden, sondern nur Kompetenzen und funktionale Aspekte; jedoch ist die überragende Bedeutung der Funktionssicht für Geschäftsarchitekturen unumstritten (vgl. [Mc99], [GM05], [Sc02]).

Die Geschäftsanalyse mittels organisationaler Fähigkeiten umfasst folgende Schritte:

Erfassung der Unternehmensfähigkeiten. Fähigkeiten lassen sich als funktionale Module betrachten, die sich zerlegen und zusammensetzen lassen. Das Ziel der Erfassung der Unternehmensfähigkeiten besteht darin, eine hierarchische Fähigkeitenstruktur zu erfassen. Versteht man grundlegende Fähigkeiten als die Fähigkeiten einer Ebene 1, lassen sich tiefere Ebenen erfassen, indem untersucht wird, welche „Teil"-Fähigkeiten zur Erbringung der Fähigkeiten der Ebene 1 notwendig sind. Das Ergebnis eines solchen Vorgehens ist eine hierarchische Struktur funktional orientierter Fähigkeiten (Fähigkeitenkarte). Dabei kann zwischen Fähigkeiten, die das Unternehmen bereits besitzt und solchen, die es gerne besitzen würde, differenziert werden.

Bewertung der Unternehmensfähigkeiten. Im nächsten Schritt wird eine Prioritätenliste erstellt, die Auskunft darüber gibt, für welche Fähigkeiten IT-Unterstützung sinnvoll bzw. weniger sinnvoll ist. Damit lässt sich ein Ranking aufbauen, das die Fähigkeiten nach der Sinnhaftigkeit von IT-Unterstützung ordnet. Dazu werden die Fähigkeiten hinsichtlich bestimmter Merkmale bewertet (z.B. derzeitige Leistungsfähigkeit, Bedeutung für das Unternehmen, derzeitige IT-Unterstützung, Auslagerungsfähigkeit). Basierend auf diesen Bewertungen wird im nächsten Schritt eine Prioritätenliste über ein Index-Verfahren ermittelt. Zum Zweck der Bewertung ist es sinnvoll, eine Reihe von Kategorien und Eigenschaften zu definieren, die mittels konkreter Fragen von den Unternehmen, z. B. auf dem Wege einer Befragung, beurteilt werden.

Aggregation der Fähigkeitenbewertungen. Als letzte Aktivität dieser Phase sind die Bewertungen der einzelnen Fähigkeiten unternehmens- sowie branchenintern zusammenzufassen. In einem ersten Schritt wird ein unternehmensspezifischer Indexwert für jede Fähigkeit ermittelt (vgl. Abbildung 1 rechts). In einem zweiten Schritt werden diese unternehmensspezifischen Indexwerte zu einem Branchen-Index zusammengefasst.

Das Ergebnis der Phase besteht in einem Ranking von Fähigkeiten, wobei die Positionen der einzelnen Fähigkeiten durch den Branchen-Indexwert bestimmt sind. Hohe Indexwerte sind dabei dahingehend zu interpretieren, dass IT-Unterstützung mittels Services einer branchenspezifischen Services-Plattform sinnvoll ist; für Fähigkeiten mit niedrigen Indexwerten ist das Gegenteil der Fall.

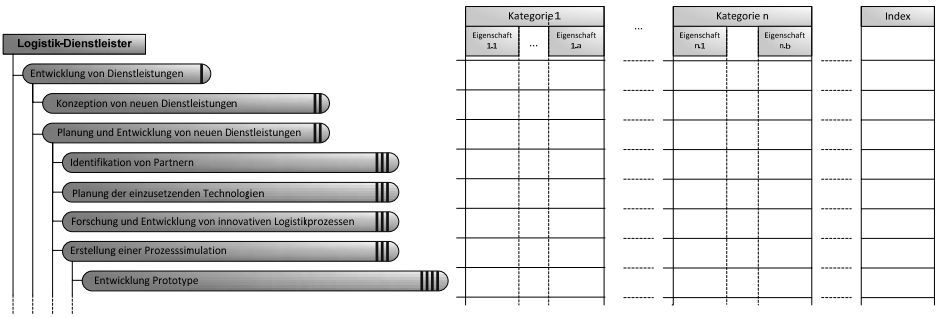


Abbildung 1: Unternehmensspezifische Bewertung aller Fähigkeiten und Aggregation der Bewertungen zu einem unternehmenssepzifischen Indexwert

2.3 Die Phase der Prozess-Analyse und -Aggregation

Die Geschäftsanalyse legt die funktionalen Bereiche offen, die für eine Shared-Services-Plattform relevant sind, gibt jedoch noch keine Auskunft darüber, *wie* die entsprechenden Funktionen erbracht werden (Prozessgestaltung, IT-Unterstützung). Der Gegenstand der Prozess-Analyse besteht in der Untersuchung der Funktionsweise der den Fähigkeiten zugeordneten Prozesse. Das prinzipielle Vorgehen in dieser Phase entspricht dem der Geschäftsanalyse: Analyseergebnisse in einzelnen Unternehmen werden in ein Gesamtbild aggregiert, auf dessen Basis im nächsten Schritt Services erstellt werden können. Da nicht in jedem Fall davon auszugehen ist, dass Prozesse zu identischen Fähigkeiten gleichfalls identisch sind, müssen Unterschiede in Prozessen sowohl erfasst als auch in angebotenen Services realisiert werden. Ein geeignetes Instrument liefert hierzu das Prozess-Familien-Engineering (vgl. z.B. [Ba05]).

2.4 Die Phase der Service-Identifikation und des Designs

Inhalt dieser Phase ist die Identifikation bzw. Abgrenzung der Funktionalität, die als Service bereitgestellt werden soll. In der Literatur werden dazu verschiedene Ansätze diskutiert, die sich häufig einem Top-Down- oder Bottom-Up-Vorgehen zuordnen lassen. Bei einem Top-Down-Ansatz wird von geschäftlichen Belangen, Funktionen, Prozessen einer Geschäftsarchitektur ausgegangen und diese schrittweise in IT-Konstrukte übertragen und in Services implementiert. Bei Bottom-Up-Ansätzen wird Funktionalität als Service bereitgestellt und Services werden zu komplexeren und geschäftsnäheren Services komponiert. Nach Ansicht der Autoren es sinnvoll, beide Wege zu kombinieren. Die ersten beiden Phasen setzen einen Top-Down-Ansatz um. Im Zuge des Service-Designs ist zu prüfen, inwieweit Service-Bedarfe durch vorhandene Services abgedeckt werden. Fragen der Service-Granularität sind zu thematisieren, da zwischen Granularität, Komplexität und Wiederverwendbarkeit enge Wechselwirkungen bestehen.

2.5 Die Phase der Service-Realisierung

Die letzte Phase besteht in der Überführung des Service-Designs in lauffähige Services. Die konkreten Aktivitäten können sowohl darin bestehen, Services komplett neu zu implementieren, aber auch darin, Komponenten bereitzustellen, die von einzelnen Services genutzt werden (Service Kompositionen). Eine weitere Aufgabe besteht in der Bereitstellung einer Service-orientierten Infrastruktur, z.B. Enterprise Service Bus. Auch wenn Service-orientierte Architekturen prinzipiell Technologie-agnostisch sind, ist es sinnvoll, sie auf der Basis von Web-Services aufzubauen, da sie über eine Reihe von wünschenswerten Eigenschaften verfügen (vgl. [Er07]).

3 Zusammenfassung und Ausblick

In dem vorliegenden Beitrag wurde ein Phasenmodell umrissen, das der Entwicklung einer Service-orientieren, Branchen-spezifischen IT-Plattform dient. Der gesamte Entwicklungsprozess wurde in fünf Phasen zerlegt. Die bisherigen Arbeiten der Autoren beschränkten sich auf konzeptionelle Überlegungen für den gesamten Entwicklungsprozess sowie eine Detaillierung und Validierung der ersten zwei Phasen. Zahlreiche technische Fragen und Herausforderungen wurden bisher nur am Rande gestreift. Fragen der Datenintegration, Interoperabilität etc. wurden in diesem Beitrag nicht thematisiert. Gleiches gilt für das Management von *Business Services*, der Abrechnung und Preisbildung und sowie für ein mögliches Betreibermodell der Shared-Services-Plattform.

Literaturverzeichnis

[Ba05] Bayer, J. et al.: Process Family Engineering. Modeling variant-rich processes, Kaiserslautern, 2005.

[Co94] Collis, D.J.: Research Note: How Valuable Are Organizational Capabilities?, Strategic Management Journal, 15, 1994, S. 143-152.

[Er07] Erl, T.: SOA: Principles of Service Design, Upper Saddle River et al., Prentice Hall, 2007.

[GM05] Glushko, R.J.; McGrath, T.: Document engineering: analyzing and designing documents for business informatics and Web services, Cambridge, Massachusetts, The MIT Press, 2005.

[Go06] Goeken, M.: Entwicklung von Data-Warehouse-Systemen: Anforderungsmanagement, Modellierung, Implementierung, Deutscher Universitätsverlag, 2006.

[Ho06] Homann, U.: Building Distributed Applications. A Business-Oriented Foundation for Service Orientation. 2006.

[Mc99] McDavid, D.W.: A standard for business architecture description, IBM Systems Journal, 1999. 38(1).

[MZ07] Maglio, P.; Zysman, J.:Toward a Science of Service Systems, in Sofcon 2007.

[Sc02] Scheer, A.-W.: ARIS - Vom Geschäftsprozess zum Anwendungssystem, 4. Aufl., Berlin, Heidelberg, New York, Springer, 2002

[VB06] Versteeg, G.; Bouwman, H.: Business architecture: A new paradigm to relate business strategy to ICT, Information Systems Frontiers, 8(2), Wiesbaden, 2006, S. 91-102.

Ein Stichprobenverfahren zur Abschätzung der Fehlerzahl

Karl-Heinz Wichert, Melanie Späth

Capgemini sd&m AG
Carl-Wery-Str.42
81739 München
karl-heinz.wichert@capgemini-sdm.com
melanie.spaeth@capgemini-sdm.com

Abstract: Eine häufige Ursache für ineffiziente Tests und mangelnde Testendequalität ist die ungenügende Eingangsqualität der Testobjekte. Es fehlen praktisch anwendbare Verfahren, mit denen die Qualität der Testobjekte frühzeitig im Test bewertet werden kann. Dieses Dokument beschreibt ein Stichprobenverfahren, mit dem die Gesamt-Fehlerzahl eines Softwaresystems nach Durchführung weniger Testfälle abgeschätzt werden kann. Das Verfahren kombiniert ein einfaches mathematisches Modell für Fehlerabdeckung mit der Methodik des Pairwise-Testens. Mit dieser Information kann frühzeitig entschieden werden, ob ein weiterer Test sinnvoll ist oder konstruktive Maßnahmen zu diesem Zeitpunkt besser geeignet sind. Dadurch kann ineffizientes Testen vermieden, der Testaufwand genauer abgeschätzt und die Qualitätsverantwortung weg vom Testen und hin zum Entwicklungsprozess delegiert werden.

1 Einleitung

Testverfahren haben meistens das Ziel, möglichst viele Fehler eines Softwaresystems zu finden. Häufig erfolgt der Test in mehreren Stufen: Zunächst testet jeder Entwickler seinen Code, üblicherweise mit Whitebox-Techniken. Dann beginnen weitere Testschritte, die von Testteams mittels Blackbox-Verfahren durchgeführt werden (z.B. der Systemtest). Sie sollen die verbleibenden Fehler finden. Dabei treten oft folgende Probleme auf:

- **Die Qualitätsverantwortung wird dem Systemtest untergeschoben.**
  Entwickler verlassen sich darauf, dass der Systemtest alle verbleibenden Fehler findet. Damit sinkt die Motivation, bereits bei der Programmierung hochwertige Qualität zu produzieren und durch den Entwicklertest abzusichern.

- **Blackbox-Test ist ineffizient bei hoher Fehlerdichte.**
  Enthält die Software noch viele Fehler, sind Whitebox-Tests, Code-Reviews etc. erwiesenermaßen effizienter, um die Fehler zu beheben [Jo97].

- **Der Testaufwand ist nicht abschätzbar.**
  Beschreibung, Kategorisierung, Nachstellung und Verwaltung der Fehler sind wesentliche Aufwandstreiber beim Systemtest. Eine vernünftige Planung für den Systemtest ist somit nur möglich, wenn die Fehlerzahl ungefähr bekannt ist.

Dies sind alltägliche Erfahrungen in vielen Projekten. Häufig wird jedoch, statt die Ursache zu ergründen, entweder Testen an sich für ineffizient gehalten oder das Testvorgehen im konkreten Projekt angezweifelt. Die Probleme können adressiert werden, wenn möglichst früh und mit wenig Aufwand die Fehlerzahl abgeschätzt und auf dieser Basis entschieden wird, ob die Software bereits die nötige Reife für den Systemtest besitzt.

Ein hierfür geeignetes Verfahren sollte folgende Anforderungen erfüllen:

A1. Es liefert eine Abschätzung der Fehlerzahl, die beim Test auftreten wird. Eine hohe Genauigkeit ist wünschenswert, aber nicht notwendig.

A2. Der Aufwand zur Abschätzung der Fehlerzahl ist sehr viel kleiner als der Aufwand für den gesamten Systemtest.

A3. Das Verfahren ist anwendbar ohne Kenntnis über die interne Struktur und Vorgeschichte der Software. Nur fachliches Wissen wird vorausgesetzt.

A4. Das Ergebnis ist nachvollziehbar.

Unser Vorschlag für ein solches Verfahren besteht aus zwei Schritten: Zunächst wird ermittelt, wie viele Fehler die Software ungefähr enthält. Dann wird entschieden, ob diese Fehlerzahl für den Eingang in den Systemtest akzeptabel ist. Da der zweite Schritt stark abhängig ist von den Qualitätszielen des jeweiligen Projektes, werden wir diesen nur am Rande betrachten.

2 Related Work

Es gibt bereits eine Reihe von Ansätzen zur Fehlerprognose, wobei das Ziel meistens in einer Vorhersage der Zuverlässigkeit der Software für die Produktion ist [FN99]. Solche Prognosen sind schwierig, weil im Test gefundene Fehler sich nur mit erheblicher Unsicherheit auf die Zuverlässigkeit im Betrieb abbilden lassen. Unser Ziel ist bescheidener; wir wollen lediglich die Reife einer Software für den Test nachweisen.

Eine in der Industrie gängige Eingangsprüfung für Systemtests sind sog. Smoke-Tests [SL05], [Bi99]. Diese prüfen, ob das System grundsätzlich lauffähig und testbar ist. Wir halten dieses Kriterium für notwendig, aber nicht hinreichend. Wir wollen darüber hinaus auch prüfen, ob die Software effizient getestet werden kann.

Häufig untersucht wurden z.B. sog. Reliability Growth Models [Ly95], [FP98]. Diese prognostizieren die Zuverlässigkeit eines Systems aus den Ausfalldaten operationeller Tests. Für unsere Zwecke sind die Modelle nicht geeignet, weil wir die Fehlerabschätzung zu einem frühen Zeitpunkt im Entwicklungszyklus benötigen.

Zudem gibt es Veröffentlichungen zur Aufwandschätzung für Tests an Hand gemittelter Erfahrungswerte von Fehlerdichten [KV03]. Erfahrungsgemäß schwanken Fehlerdichten beträchtlich [FO00]. Unser Verfahren soll feststellen, ob die Fehlerdichte in der untersuchten Software im üblichen Bereich liegt oder stark davon abweicht.

Viele Arbeiten, darunter [Bi99], [Wo95], haben sich mit dem Zusammenhang von Codeabdeckung mit Fehlerabdeckung beschäftigt. Durch Messung der Codeabdeckung beim Ausführen von Testfällen lässt sich damit die Gesamt-Fehlerzahl extrapolieren. Diese Methoden betrachten wir als hilfreich zur Überprüfung der Qualität von Entwicklertests. Für unsere Zwecke sind sie nicht anwendbar, da es sich um Whitebox-Verfahren handelt, für deren Durchführung Wissen über die Codestruktur notwendig ist.

3 Eigener Ansatz

Um eine Aussage über die ungefähre Anzahl der Fehler in einem Software-System zu erhalten, verwenden wir ein Stichprobenverfahren. Die Stichprobe ziehen wir aus der Menge aller theoretisch möglichen Testfälle. Die Fehlerzahl, die bei Ausführung aller Testfälle gefunden werden würde, ist genau die unbekannte Fehlerzahl, die wir ermitteln wollen. Klären müssen wir, welchen Anteil A_F der gesamten Fehler wir finden, wenn eine Stichprobe T der gesamten Testfallmenge gezogen wird. Dies hängt von der Wahl der Testfälle ab. Dass die Fehler zudem nicht über die Module hinweg gleichverteilt sind, führt bei Verwendung eines Stichprobenverfahrens naturgemäß zu einer statistischen Ungenauigkeit. Diese nehmen wir in Kauf; sie begrenzt die Prognosegenauigkeit.

Wir diskutieren nun ein einfaches Modell für die Abhängigkeit des Fehleranteils von der Anzahl durchgeführter Testfälle, $A_F(T)$. Die grundlegende Annahme dabei ist, dass die Wahrscheinlichkeit, mit einem Testfall einen neuen Fehler zu finden, linear abnimmt mit der Anzahl der bereits durchgeführten Testfälle. Es gilt also

$$\Delta A_F \sim (1- A_F)\, \Delta T,$$

wobei T: Anzahl bereits durchgeführter Testfälle; A_F: Fehlerabdeckung = Anteil der bereits entdeckten Fehler an allen Fehlern; ΔA_F: Änderung der Fehlerabdeckung; ΔT: Anzahl neu durchgeführter Testfälle.

Bei einer hohen Anzahl durchgeführter Testfälle kann Δ als klein betrachtet werden. Aus obiger Gleichung ergibt sich eine einfache Differentialgleichung mit folgender Lösung:

$$A_F(T) = 1-e^{-\lambda * T}$$

Die Funktion ist eine aus vielen natürlichen Prozessen bekannte Sättigungskurve mit nur einem freien, zunächst unbekannten, Parameter, λ. Bei geschickter Wahl von Testfällen steigt die Fehlerabdeckung schneller an als bei ungeschickter Wahl. Diese Wahl repräsentiert λ. Abbildung 1 zeigt fiktive Kurven für verschiedene Werte von λ.

Für die Fehlerprognose müssen wir das λ finden, das für ein gegebenes System und eine gegebene Abfolge von Testfällen die reale Kurve näherungsweise am besten beschreibt.

Im Folgenden zeigen wir, wie dieser Parameter bestimmt werden kann für Testobjekte, für die sich Pairwise-Testen [Ma85], [Wi00] anwenden lässt. Pairwise-Testen ist eine Blackbox-Testmethode (Anforderung A3), die Testfälle durch gezielte Kombination von Parametern bildet. Zur Erzeugung von Pairwise-Testfällen gibt es verschiedene Algorithmen und Werkzeuge, darunter [Co97], [HR04].

Für den Test mit einem vollständigen Pairwise-Testfallsatz geben [WK01], [KR02] und [KWG04] Fehlerabdeckungen zwischen 50% und 100% an. Dies legen wir zugrunde für die Annahme, dass durch die Pairwise-Methode eine optimale Fehlerabdeckung erreicht wird und diese bei 85% ± 15% liegt. Damit bestimmen wir für ein System bei bekannter Anzahl Pairwise-Testfälle das λ für die Durchführung einer optimalen Testfallmenge:

$$\lambda = -\ln(1- A_p)/ T_p ,$$

wobei T_p: Anzahl aller Pairwise-Testfälle; A_p: Fehlerabdeckung bei deren Ausführung.

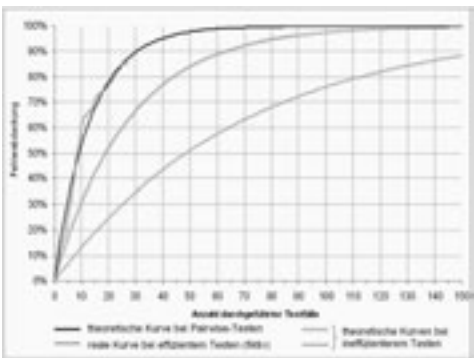


Abbildung 1 - Fehlerabdeckung abhängig von der Effizienz des Testvorgehens

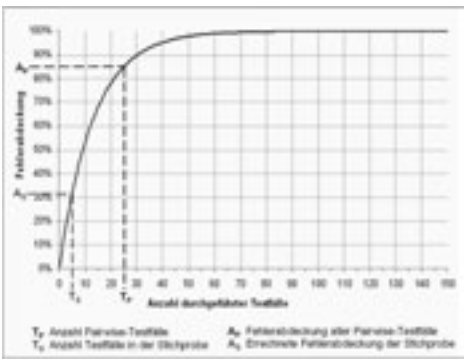


Abbildung 2 - Verlauf der Fehlerabdeckung bei optimaler Testfallwahl

Durch die Kenntnis der Kurve können wir nun für A_p=0,85 bei Wahl einer Testfall-Teilmenge T den zugehörigen Fehlerabdeckungsgrad A_F bestimmen als

$$A_F (T) = 1- e^{-0,15(T/ Tp)}.$$

Die geschätzte Gesamtfehlerzahl im System ist dann:

$$F_{Gesamt} = F_{Test}/ A_T = F_{Test} / (1 - (1 - A_p)^{(T_{Test} / T_p)}),$$

wobei F_{Test}: gefundene Fehleranzahl; T_{Test}: durchgeführte Testfälle; T_p: Anzahl aller Pairwise-Testfälle; A_p: Fehlerabdeckung; A_T: erreichte Fehlerabdeckung.

3.1 Anwendung des Verfahrens

Basierend auf dem mathematischen Modell werden folgende Schritte durchgeführt:

1. Ermittle für das zu prüfende System die Eingabe- und sonstigen Parameter und deren mögliche Werte durch Äquivalenzklassen- und Grenzwertanalyse.

2. Berechne mit Hilfe von Werkzeugen die Anzahl aller Pairwise-Testfälle.

3. Führe eine Teilmenge der Pairwise-Testfälle durch. Faustregel hierbei: Wähle die Testfallanzahl so, dass mindestens 20% Fehlerabdeckung erreicht werden.

4. Zähle die Anzahl der Fehler, die bei der Durchführung auftreten.

5. Wende das entwickelte mathematische Modell an. Die Gesamtfehlerzahl kann mit einem einfachen Tabellenkalkulationsprogramm berechnet werden.

Das Ergebnis liefert die geschätzte Gesamtzahl der Fehler in dem getesteten System (Anforderungen A1, A4). Diese wird gegen das Abbruchkriterium geprüft. Die maximal zu erwartende Anzahl von Fehlern wird in der Testplanung festgelegt: Deren Protokollierung und resultierende Retests sind wesentliche Größen der Testaufwandsschätzung.

Da die Schritte 1 bis 3 ohnehin im Systemtest ausgeführt werden, ist der Zusatzaufwand durch unser Verfahrens gering (Anforderung A2). Das Verfahren anzuwenden ist dann sinnvoll, wenn der Aufwand für die noch verbleibende Testdurchführung deutlich höher ist als die Hochrechnung der Fehlerzahl. Dies ist in der Praxis nur dann nicht der Fall, wenn die Durchführung aller verbleibenden Testfälle vollständig automatisiert wurde.

3.2 Genauigkeit des Verfahrens

Bezüglich der Fehlerabdeckung bei Ausführung aller Pairwise-Testfälle nehmen wir an, dass sie bei 85 ± 15% liegt. Die Anwendung bekannter Regeln zur Fehlerfortpflanzung ergibt eine Genauigkeit der Hochrechnung von ± 50%.

Bei der Hochrechnung der gefundenen Fehler zur Ermittlung der Gesamt-Fehlerzahl tritt zusätzlich ein statistischer Fehler auf. Die Ungenauigkeit ist stark abhängig von der Zahl gefundener Fehler (Gesetz der großen Zahlen). Dies führt dazu, dass Software mit wenigen Fehlern eher zu gut bewertet wird. Bei einer nicht zu kleinen Stichprobe ist der Fehler fast immer geringer als der oben ermittelte Fehler durch die Ungenauigkeit von T_p. Deshalb kann eine Genauigkeit des Verfahrens von ± 50% angenommen werden.

4 Zusammenfassung und Ausblick

Mit unserem Verfahren können wir nach Ausführen weniger Testfälle auf die Gesamtfehlerzahl schließen. Das Modell ist anwendbar auf Systeme, die mittels Pairwise-Testen testbar sind. Die Genauigkeit liegt hierbei bei ± 50%. Dies ist für viele Zwecke ausreichend. Höhere Genauigkeit kann durch Ausführen von mehr Testfällen erreicht werden.

Das Verfahren kann angewendet werden, um die Qualität eines Systems zu prüfen und abhängig von dieser weitere Maßnahmen zu ergreifen.

Insbesondere verhindert es, dass Software mit zu schlechter Qualität in den Systemtest gelangt und damit zu hohen Testaufwänden führt. Bei unkritischen Systemen und gemessener guter Qualität kann der Test unter Umständen schnell beendet werden.

Wir erproben das Verfahren derzeit in konkreten Projekten bei Capgemini sd&m.

Interessant wäre zudem, den Anwendungsbereich auf Systeme zu erweitern, für die Pairwise-Testen keine geeignete Testfallauswahl bietet. Dies ist beispielsweise der Fall für Systeme mit verschiedenen Prozessdurchläufen mit jeweils unterschiedlichen Parametern. Das Prinzip unseres Verfahrens bleibt auch für diese Art von Systemen anwendbar. Nur die Justierung der Fehlerkurve muss dann auf andere Weise erfolgen.

Literaturverzeichnis

[Bi99] Binder, R.: Testing Object-Oriented Systems. Addison-Wesley, 1999.

[Co97] Cohen, D. M. et. al.: The AETG System: An Approach to Testing Based on Combinatorial Design. In (IEEE Computer Society Hrsg.): IEEE Transactions on Software Engineering (TSE), 23(7), 1997; S. 437-444.

[HR04] Hartman, A.; Raskin, L.: Problems and Algorithms for Covering Arrays. In: Discrete Mathematics, 284(1-3), 2004; S. 149-156.

[FN99] Fenton, N.E.; Neil, M.: A Critique of Software Defect Prediction Models. In (IEEE Computer Society Hrsg.): IEEE TSE, 25(5), 1999; S. 675-689.

[FO00] Fenton, N.E.; Ohlsson, N.: Quantitative Analysis of Faults and Failures in a Complex Software System. In (IEEE Computer Society Hrsg.): IEEE TSE, 26(8), 2000; S. 797-814.

[FP98] Fenton, N.E.; Pfleeger, S.L.: Software Metrics: A Rigorous and Practical Approach (2nd ed.). PWS Publishing Co., 1998.

[Jo97] Jones, C.: Software Quality - Analysis and Guidelines for Success. Int. Thomson Computer Press, 1997.

[KR02] Kuhn, D.R.; Reilly, M.J.: An Investigation of the Applicability of Design of Experiments to Software Testing. In (IEEE Computer Society Hrsg.): Proc. 27th Annual NASA Goddard Software Engineering Workshop, Washington, 2002; S. 91-95.

[KV03] Kropfitsch, D.; Vigenschow, U.: Das Fehlermodell: Aufwandsschätzung und Planung von Tests. Objekt-Spektrum, (6), 2003; S. 30-35.

[KWG04] Kuhn, D.R.; Wallace, D.R.; Gallo, A.M.: Software Fault Interactions and Implications for Software Testing. In (IEEE Computer Society Hrsg.): IEEE TSE, 30(6), 2004; S. 418-421.

[Ly95] Lyu, M.R.: Handbook of Software Reliability Engineering. McGraw-Hill, 1995.

[Ma85] Mandl, R.: Orthogonal Latin Squares: An Application of Experiment Design to Compiler Testing. In: Communications of the ACM, 28(10), 1985; S. 1054-1058.

[SL05] Spillner, A., Linz, T.: Basiswissen Softwaretest. dpunkt.verlag, Heidelberg, 2005.

[Wi00] Williams, A.W.: Determination of Test Configurations for Pair-Wise Interaction Coverage. In (Kluwer, B.V. Hrsg.): Proc. 13th Int. Conf. on Testing Communicating Systems, 2000; S. 59-74.

[WK01] Wallace, D.R.; Kuhn, D.R.: Failure Modes in Medical Device Software: An Analysis of 15 Years of Recall Data, In (World Scientific Publishing Company Hrsg.): International Journal of Reliability, Quality and Safety Engineering, 8(4), 2001; S. 351-371.

[Wo95] Wong, W.E. et. al.: Effect of Test Set Minimization on Fault Detection Effectiveness. In (ACM Hrsg.): Proc. 17th Int. Conf. on Software Engineering, Seattle, 1995; S. 41-50.

Workshops

Human-Computer Interaction und Visualisierung (HCIV 2009)

Achim Ebert[1], Peter Dannenmann[2]

[1]Technische Universität Kaiserslautern
ebert@informatik.uni-kl.de
[2]DFKI GmbH Kaiserslautern
peter.dannenmann@dfki.de

Wichtige Ziele in den Forschungsgebieten Human-Computer Interaction (HCI) und Visualisierung sind die Erforschung, Entwicklung und Evaluierung von Werkzeugen und Strategien, um die Möglichkeiten der Interaktion mit großen Displays besser verstehen und unterstützen zu können. Bislang existieren aber nur wenige Anwendungen, die die neuen Voraussetzungen großer Displays gewinnbringend und effektiv einsetzen können. Eine dieser Anwendungen könnte die Software Visualisierung darstellen, welche auf der äußerst erfolgreichen Software Engineering-Thematik „Software Measurement/Software Analysis aufbaut. Nach Price et. al. lässt sich das Gebiet der Software Visualisierung in „Algorithm Animation“ und „Program Visualization“ klassifizieren. Der erste Begriff bezieht sich dabei auf die dynamische Darstellung von High-Level-Beschreibungen von Software (Algorithmen). Der zweite Begriff – im Zusammenhang mit dem Workshop der wichtigere – umschreibt die Verwendung dynamischer Visualisierungstechniken zur Steigerung des Verständnisses der Umsetzungen von Programmen oder Datenstrukturen. Program Visualization-Ansätze versuchen, die Features der Software zu extrahieren und damit diese besser greifbar zu machen. Sie liefern damit einen wichtigen Zugang zum abstrakten Produkt „Software“. Im Rahmen des Workshops werden Vorträge aus den Bereichen HCI und (Software) Visualisierung aktuelle Entwicklungen und den State-of-the-Art dieser Gebiete in Bezug auf Software Engineering näher beleuchten. In gemeinsamen Diskussionen mit den eigentlichen Nutzern der Konzepte, den Experten aus dem Software Engineering, werden diese Inhalte anschließend diskutiert und Visionen für zukünftige Forschungen entwickelt. Der Workshop wird unterstützt vom Internationalen Graduiertenkolleg „Visualisierung großer und unstrukturierter Datenmengen“ der DFG.

Modellgetriebene Softwarearchitektur - Evolution, Integration und Migration (MSEIM 2009)

Michael Goedicke[1], Maritta Heisel[1], Sascha Hunold[2],
Stefan Kühne[3], Matthias Riebisch[4], Niels Streekmann[5]

[1]Universität Duisburg-Essen; michael.goedicke@s3.uni-due.de,
maritta.heisel@uni-duisburg-essen.de
[2]Universität Bayreuth; hunold@uni-bayreuth.de
[3]Universität Leipzig; kuehne@informatik.uni-leipzig.de
[4]TU Ilmenau; matthias.riebisch@tu-ilmenau.de
[5]OFFIS; niels.streekmann@offis.de

Softwarearchitekturen sind wichtige Artefakte zur Unterstützung der Software-Entwicklung und Software-Evolution. Sie dienen der grundlegenden Beschreibung von Softwaresystemen und dienen damit auch als Ausgangspunkt zur Weiterentwicklung bestehender Systeme. Diese Weiterentwicklung umfasst neben der Evolution bestehender Softwaresysteme auch die Integration mit anderen Systemen und die Migration in neue Systemumgebungen. Die modellgetriebene Softwareentwicklung, die zunehmende Verbreitung sowohl in der Wissenschaft als auch im Einsatz in der Praxis findet, stellt mit den verfolgten Zielstellungen Automatisierung, Komplexitätsreduzierung sowie Qualitätssteigerung einen viel versprechenden Ansatz zur Unterstützung der dafür notwendigen Aufgaben dar. Der Einsatz modellgetriebener Entwicklungsansätze im Kontext bestehender Legacy-Systeme wirft jedoch auch zahlreiche Fragen hinsichtlich Vorgehen, Methoden, Werkzeuge und Strategien auf.
Ziel des Workshops ist der Austausch von Erfahrungen und Forschungsergebnissen zum Thema modellgetriebene Softwareentwicklung. Ein besonderer Fokus liegt dabei auf der Softwarearchitektur im Zusammenhang mit Evolution, Integration und Migration bestehender Systeme.

Patterns in Enterprise Architecture Management (PEAM 2009)

Florian Matthes, Alexander Ernst

Technische Universität München
{matthes,ernst}@in.tum.de

There is a growing interest in academia and industry to identify, collect, document and exchange best practices in the management of very large software application landscapes in a structured manner. This workshop addresses European researchers and practitioners with experience in enterprise architecture management topics. The goal of the workshop is to improve this knowledge exchange by using an integrating pattern-based approach. An enterprise architecture management pattern (**EAM pattern**) is a general, reusable solution to a common problem in a given context which identifies driving forces, known usages and consequences. It can be specified on different levels of abstraction and detail, e.g. as a framework for enterprise architectures, as a method for enterprise modeling, or as a reference model. EAM patterns address social, technical and economic issues in a balanced manner.

Produkt-Variabilität im gesamten Lebenszyklus (PVLZ 2009)

Klaus Marquardt[1], Dietmar Schütz[2], Markus Völter[3]

[1]Dräger Medical
klaus.marquardt@draeger.com, marquardt@acm.org
[2] Siemens AG Corporate Technology
dietmar.schuetz@siemens.com
[3] voelter Ingenieurbüro
markus@voelter.de

Ansätze zu Produktlinien und Produktfamilien beschäftigen sich meist mit der Architektur, die Erweiterbarkeit und Variabilität ausdrückt und modelliert. Dieser Workshop betrachtet den gesamten Lebenszyklus und bringt Anforderungen, Architektur, Implementierung, Testen, Installation und Wartung miteinander in Beziehung. Für jede Phase in der Software-Entwicklung gibt es eine Reihe von erprobten Techniken und Praktiken. Je nach verwendeter Architektur sind diese Praktiken unterschiedlich sinnvoll – im schlechtesten Fall können Konzepte in späteren Phasen manche Ziele aus früheren Phasen konterkarieren. Jenseits der technischen Lösung im Fokus der Architektur gibt es viel zu beachten:

- wie werden Anforderungen erstellt und verwaltet? Wie wird dort Variabilität modelliert?
- Welcher Teil der Software und Architektur hat welche Lebensdauer und Gültigkeit?
- Was wird wie versioniert und freigegeben?
- Welche Qualitätsmaßnahmen greifen für welche Art von variablen Architekturen?
- Wie wird Kompatibilität sichergestellt – oder unterbunden?
- Wie finden Updates und Migrationen statt?

Ziel dieses Workshops ist es, die Beziehungen zwischen den unterschiedlichen Teilen des Lebenszyklus zu beleuchten. Verschiedene technische Mechanismen der Variabilität können dabei unterschiedliche Praktiken erfordern. Insgesamt sollen zueinander passende Praktiken identifiziert werden, die sich gegenseitig unterstützen und verstärken.

Requirements Engineering und Business Process Management - Konvergenz, Synonym oder doch so wie gehabt? (REBPM 2009)

Kurt Schneider[1], Jörg Dörr[2], Sebastian Stein[3], Sebastian Adam[2], Daniel Lübke[1]

[1]Leibniz Universität Hannover
{kurt.schneider,daniel.luebke}@inf.uni-hannover.de
[2]Fraunhofer IESE
{joerg.doerr,sebastian.adam}@iese.fraunhofer.de
[3]IDS Scheer AG
sebastian.stein@ids-scheer.com

SOA ist nicht auf die IT-Abteilung und die Softwareentwicklung beschränkt, sondern wird Unternehmen in ihrer Gesamtheit betreffen. Requirements Engineering als Schnittstellendisziplin zwischen „Kunde“ und „Entwickler“ muss sich in diesem Kontext neu definieren, da klare Auftraggeber-Auftragnehmer-Situationen in den Hintergrund treten und die Unternehmen integriert von der Strategie bis zur IT ganzheitlich gestaltet werden. Dies beeinflusst die Anforderungserhebungsphase, die nun viel mehr auf die Umsetzung der Geschäftsziele unter Beachtung der Fähigkeiten der IT eingehen muss. So muss u.a. die Softwareentwicklung viel stärker mit den Geschäftsprozessen abgestimmt werden, um die Flexibilität von SOA nutzen zu können. Auf internationalen Konferenzen werden schon länger Diskussionen geführt, inwiefern in diesem Zusammenhang Geschäftsprozessmanagement und Requirements Engineering überhaupt noch zu trennen sind bzw. wo der Unterschied und die Gemeinsamkeiten liegen. Dieser Workshop soll nun auch in der deutschen SE Community die Diskussion über dieses generelle Problem fördern, sowie Methoden, die beide Bereiche verbinden, entwickeln bzw. vorstellen.

Software ENgineering within Social software Environments (SENSE 2009)

Ralf Klamma[1], Volker Wulf[2], Matthias Jarke[1], Anna Glukhova[1]

[1] RWTH Aachen
{klamma,jarke,glukhova}@dbis.rwth-aachen.de
[2] Universität Siegen
volker.wulf@uni-siegen.de

Due to the new business models of the Web 2.0 and the new generation of Web users the development of social software requires a new SE approach. At the same time social software brings new possibilities for the SE process. Building on the positive experiences from the open source community we propose the term "social software engineering" which should include both Web 2.0 style engineering and engineering of the social software. Communication between different stakeholders, developers and end-users is an essential aspect for successful software engineering (SE). SE presents a social process that has to be supported by the communication enabling tools. SE-participants can express their opinion and exchange their ideas. With the globalization of software production, the social software can serve as a good option for socialization of SE processes. The goal of the workshop is to bring together the researchers and practices interested in engineering process of social software, which in turn should/could be applied during software engineering. During the workshop a state-of-the-art discussion on advanced research and open issues on social software engineering will take place. The special focus will be put on design process and design guidelines of social systems created for SE support.

Collaboration and Knowledge Sharing in Software Development Teams (SOFTEAM 2009)

Hans-Jörg Happel[1], Steffen Lohmann[2], Walid Maalej[3]

[1] FZI Forschungszentrum Informatik an der Universität Karlsruhe (TH)
happel@fzi.de
[2] Universität Duisburg Essen
steffen.lohmann@uni-due.de
[3]Technische Universität München
maalej@in.tum.de

Software engineering is often faced with the challenges of large and distributed development teams that need to adhere to frequent changes and short release cycles. Effective collaboration and knowledge sharing are crucial in order to guarantee successful software development. Methods and tools that support development teams must be based on interdisciplinary research efforts that investigate technologies, tools, processes and human factors in a holistic manner. Only if we understand the interplay between the involved disciplines we are able to deliver methods and integrated tool solutions that provide suitable collaboration and knowledge sharing support for software development teams. This workshop aims to bring together researchers and practitioners working on different aspects of collaboration and knowledge sharing to discuss new results and future research challenges with regard to software development from different perspectives. One major goal is to define a research agenda and establish a community around the crosscutting topics of collaboration and knowledge sharing in software engineering.

Software-Qualitätsmodellierung und –bewertung (SQMB 2009)

Stefan Wagner[1], Manfred Broy[1], Florian Deißenböck[1]
Jürgen Münch[2], Peter Liggesmeyer[2]

[1]Technische Universität München
{wagnerst,broy,deissenb}@in.tum.de
[2]Fraunhofer-Institut für Experimentelles Software Engineering (IESE)
{juergen.muench,peter.liggesmeyer}@iese.fraunhofer.de

Software-Qualität ist ein entscheidender Faktor für den Erfolg eines softwareintensiven Systems. Die Beherrschung der Qualität stellt aber immer noch eine große Herausforderung für Praxis und Forschung dar. Problematisch ist auch die Vielschichtigkeit und Komplexität von Qualität, die zu einer Vielzahl von nicht integrierten Insellösungen geführt hat. Eine umfassende Behandlung von Qualität wird typischerweise durch Qualitätsmodelle und darauf aufbauenden Bewertungen erwartet. Leider können Standard-Qualitätsmodelle, wie die ISO 9126 und ISO 25000, in der Praxis nur schwer angewandt werden, was zu einer Vielzahl von individuellen Qualitätsmodellen geführt hat. Dieser Workshop hat das Ziel, Erfahrungen mit Qualitätsmodellierung und -bewertung zu sammeln und gemeinsam neue Forschungsrichtungen zu entwickeln. Diesmal wird ein spezieller Fokus auf eingebettete Softwaresysteme gerichtet.

GI-Edition Lecture Notes in Informatics

P-1 Gregor Engels, Andreas Oberweis, Albert Zündorf (Hrsg.): Modellierung 2001.

P-2 Mikhail Godlevsky, Heinrich C. Mayr (Hrsg.): Information Systems Technology and its Applications, ISTA'2001.

P-3 Ana M. Moreno, Reind P. van de Riet (Hrsg.): Applications of Natural Language to Information Systems, NLDB'2001.

P-4 H. Wörn, J. Mühling, C. Vahl, H.-P. Meinzer (Hrsg.): Rechner- und sensorgestützte Chirurgie; Workshop des SFB 414.

P-5 Andy Schürr (Hg.): OMER – Object-Oriented Modeling of Embedded Real-Time Systems.

P-6 Hans-Jürgen Appelrath, Rolf Beyer, Uwe Marquardt, Heinrich C. Mayr, Claudia Steinberger (Hrsg.): Unternehmen Hochschule, UH'2001.

P-7 Andy Evans, Robert France, Ana Moreira, Bernhard Rumpe (Hrsg.): Practical UML-Based Rigorous Development Methods – Countering or Integrating the extremists, pUML'2001.

P-8 Reinhard Keil-Slawik, Johannes Magenheim (Hrsg.): Informatikunterricht und Medienbildung, INFOS'2001.

P-9 Jan von Knop, Wilhelm Haverkamp (Hrsg.): Innovative Anwendungen in Kommunikationsnetzen, 15. DFN Arbeitstagung.

P-10 Mirjam Minor, Steffen Staab (Hrsg.): 1st German Workshop on Experience Management: Sharing Experiences about the Sharing Experience.

P-11 Michael Weber, Frank Kargl (Hrsg.): Mobile Ad-Hoc Netzwerke, WMAN 2002.

P-12 Martin Glinz, Günther Müller-Luschnat (Hrsg.): Modellierung 2002.

P-13 Jan von Knop, Peter Schirmbacher and Viljan Mahni_ (Hrsg.): The Changing Universities – The Role of Technology.

P-14 Robert Tolksdorf, Rainer Eckstein (Hrsg.): XML-Technologien für das Semantic Web – XSW 2002.

P-15 Hans-Bernd Bludau, Andreas Koop (Hrsg.): Mobile Computing in Medicine.

P-16 J. Felix Hampe, Gerhard Schwabe (Hrsg.): Mobile and Collaborative Busi-ness 2002.

P-17 Jan von Knop, Wilhelm Haverkamp (Hrsg.): Zukunft der Netze –Die Verletzbarkeit meistern, 16. DFN Arbeitstagung.

P-18 Elmar J. Sinz, Markus Plaha (Hrsg.): Modellierung betrieblicher Informationssysteme – MobIS 2002.

P-19 Sigrid Schubert, Bernd Reusch, Norbert Jesse (Hrsg.): Informatik bewegt – Informatik 2002 – 32. Jahrestagung der Gesellschaft für Informatik e.V. (GI) 30.Sept.-3.Okt. 2002 in Dortmund.

P-20 Sigrid Schubert, Bernd Reusch, Norbert Jesse (Hrsg.): Informatik bewegt – Informatik 2002 – 32. Jahrestagung der Gesellschaft für Informatik e.V. (GI) 30.Sept.-3.Okt. 2002 in Dortmund (Ergänzungsband).

P-21 Jörg Desel, Mathias Weske (Hrsg.): Promise 2002: Prozessorientierte Methoden und Werkzeuge für die Entwicklung von Informationssystemen.

P-22 Sigrid Schubert, Johannes Magenheim, Peter Hubwieser, Torsten Brinda (Hrsg.): Forschungsbeiträge zur "Didaktik der Informatik" – Theorie, Praxis, Evaluation.

P-23 Thorsten Spitta, Jens Borchers, Harry M. Sneed (Hrsg.): Software Management 2002 – Fortschritt durch Beständigkeit

P-24 Rainer Eckstein, Robert Tolksdorf (Hrsg.): XMIDX 2003 – XML-Technologien für Middleware – Middleware für XML-Anwendungen

P-25 Key Pousttchi, Klaus Turowski (Hrsg.): Mobile Commerce – Anwendungen und Perspektiven – 3. Workshop Mobile Commerce, Universität Augsburg, 04.02.2003

P-26 Gerhard Weikum, Harald Schöning, Erhard Rahm (Hrsg.): BTW 2003: Datenbanksysteme für Business, Technologie und Web

P-27 Michael Kroll, Hans-Gerd Lipinski, Kay Melzer (Hrsg.): Mobiles Computing in der Medizin

P-28 Ulrich Reimer, Andreas Abecker, Steffen Staab, Gerd Stumme (Hrsg.): WM 2003: Professionelles Wissensmanagement – Erfahrungen und Visionen

P-29 Antje Düsterhöft, Bernhard Thalheim (Eds.): NLDB'2003: Natural Language Processing and Information Systems

P-30 Mikhail Godlevsky, Stephen Liddle, Heinrich C. Mayr (Eds.): Information Systems Technology and its Applications

P-31 Arslan Brömme, Christoph Busch (Eds.): BIOSIG 2003: Biometric and Electronic Signatures

P-32 Peter Hubwieser (Hrsg.): Informatische Fachkonzepte im Unterricht – INFOS 2003

P-33 Andreas Geyer-Schulz, Alfred Taudes (Hrsg.): Informationswirtschaft: Ein Sektor mit Zukunft

P-34 Klaus Dittrich, Wolfgang König, Andreas Oberweis, Kai Rannenberg, Wolfgang Wahlster (Hrsg.): Informatik 2003 – Innovative Informatikanwendungen (Band 1)

P-35 Klaus Dittrich, Wolfgang König, Andreas Oberweis, Kai Rannenberg, Wolfgang Wahlster (Hrsg.): Informatik 2003 – Innovative Informatikanwendungen (Band 2)

P-36 Rüdiger Grimm, Hubert B. Keller, Kai Rannenberg (Hrsg.): Informatik 2003 – Mit Sicherheit Informatik

P-37 Arndt Bode, Jörg Desel, Sabine Rathmayer, Martin Wessner (Hrsg.): DeLFI 2003: e-Learning Fachtagung Informatik

P-38 E.J. Sinz, M. Plaha, P. Neckel (Hrsg.): Modellierung betrieblicher Informationssysteme – MobIS 2003

P-39 Jens Nedon, Sandra Frings, Oliver Göbel (Hrsg.): IT-Incident Management & IT-Forensics – IMF 2003

P-40 Michael Rebstock (Hrsg.): Modellierung betrieblicher Informationssysteme – MobIS 2004

P-41 Uwe Brinkschulte, Jürgen Becker, Dietmar Fey, Karl-Erwin Großpietsch, Christian Hochberger, Erik Maehle, Thomas Runkler (Edts.): ARCS 2004 – Organic and Pervasive Computing

P-42 Key Pousttchi, Klaus Turowski (Hrsg.): Mobile Economy – Transaktionen und Prozesse, Anwendungen und Dienste

P-43 Birgitta König-Ries, Michael Klein, Philipp Obreiter (Hrsg.): Persistance, Scalability, Transactions – Database Mechanisms for Mobile Applications

P-44 Jan von Knop, Wilhelm Haverkamp, Eike Jessen (Hrsg.): Security, E-Learning. E-Services

P-45 Bernhard Rumpe, Wofgang Hesse (Hrsg.): Modellierung 2004

P-46 Ulrich Flegel, Michael Meier (Hrsg.): Detection of Intrusions of Malware & Vulnerability Assessment

P-47 Alexander Prosser, Robert Krimmer (Hrsg.): Electronic Voting in Europe – Technology, Law, Politics and Society

P-48 Anatoly Doroshenko, Terry Halpin, Stephen W. Liddle, Heinrich C. Mayr (Hrsg.): Information Systems Technology and its Applications

P-49 G. Schiefer, P. Wagner, M. Morgenstern, U. Rickert (Hrsg.): Integration und Datensicherheit – Anforderungen, Konflikte und Perspektiven

P-50 Peter Dadam, Manfred Reichert (Hrsg.): INFORMATIK 2004 – Informatik verbindet (Band 1) Beiträge der 34. Jahrestagung der Gesellschaft für Informatik e.V. (GI), 20.-24. September 2004 in Ulm

P-51 Peter Dadam, Manfred Reichert (Hrsg.): INFORMATIK 2004 – Informatik verbindet (Band 2) Beiträge der 34. Jahrestagung der Gesellschaft für Informatik e.V. (GI), 20.-24. September 2004 in Ulm

P-52 Gregor Engels, Silke Seehusen (Hrsg.): DELFI 2004 – Tagungsband der 2. e-Learning Fachtagung Informatik

P-53 Robert Giegerich, Jens Stoye (Hrsg.): German Conference on Bioinformatics – GCB 2004

P-54 Jens Borchers, Ralf Kneuper (Hrsg.): Softwaremanagement 2004 – Outsourcing und Integration

P-55 Jan von Knop, Wilhelm Haverkamp, Eike Jessen (Hrsg.): E-Science und Grid Adhoc-Netze Medienintegration

P-56 Fernand Feltz, Andreas Oberweis, Benoit Otjacques (Hrsg.): EMISA 2004 – Informationssysteme im E-Business und E-Government

P-57 Klaus Turowski (Hrsg.): Architekturen, Komponenten, Anwendungen

P-58 Sami Beydeda, Volker Gruhn, Johannes Mayer, Ralf Reussner, Franz Schweiggert (Hrsg.): Testing of Component-Based Systems and Software Quality

P-59 J. Felix Hampe, Franz Lehner, Key Pousttchi, Kai Ranneberg, Klaus Turowski (Hrsg.): Mobile Business – Processes, Platforms, Payments

P-60 Steffen Friedrich (Hrsg.): Unterrichtskonzepte für inforrmatische Bildung

P-61 Paul Müller, Reinhard Gotzhein, Jens B. Schmitt (Hrsg.): Kommunikation in verteilten Systemen

P-62 Federrath, Hannes (Hrsg.): „Sicherheit 2005“ – Sicherheit – Schutz und Zuverlässigkeit

P-63 Roland Kaschek, Heinrich C. Mayr, Stephen Liddle (Hrsg.): Information Systems – Technology and ist Applications

P-64 Peter Liggesmeyer, Klaus Pohl, Michael Goedicke (Hrsg.): Software Engineering 2005

P-65 Gottfried Vossen, Frank Leymann, Peter Lockemann, Wolffried Stucky (Hrsg.): Datenbanksysteme in Business, Technologie und Web

P-66 Jörg M. Haake, Ulrike Lucke, Djamshid Tavangarian (Hrsg.): DeLFI 2005: 3. deutsche e-Learning Fachtagung Informatik

P-67 Armin B. Cremers, Rainer Manthey, Peter Martini, Volker Steinhage (Hrsg.): INFORMATIK 2005 – Informatik LIVE (Band 1)

P-68 Armin B. Cremers, Rainer Manthey, Peter Martini, Volker Steinhage (Hrsg.): INFORMATIK 2005 – Informatik LIVE (Band 2)

P-69 Robert Hirschfeld, Ryszard Kowalcyk, Andreas Polze, Matthias Weske (Hrsg.): NODe 2005, GSEM 2005

P-70 Klaus Turowski, Johannes-Maria Zaha (Hrsg.): Component-oriented Enterprise Application (COAE 2005)

P-71 Andrew Torda, Stefan Kurz, Matthias Rarey (Hrsg.): German Conference on Bioinformatics 2005

P-72 Klaus P. Jantke, Klaus-Peter Fähnrich, Wolfgang S. Wittig (Hrsg.): Marktplatz Internet: Von e-Learning bis e-Payment

P-73 Jan von Knop, Wilhelm Haverkamp, Eike Jessen (Hrsg.): “Heute schon das Morgen sehen“

P-74 Christopher Wolf, Stefan Lucks, Po-Wah Yau (Hrsg.): WEWoRC 2005 – Western European Workshop on Research in Cryptology

P-75 Jörg Desel, Ulrich Frank (Hrsg.): Enterprise Modelling and Information Systems Architecture

P-76 Thomas Kirste, Birgitta König-Riess, Key Pousttchi, Klaus Turowski (Hrsg.): Mobile Informationssysteme – Potentiale, Hindernisse, Einsatz

P-77 Jana Dittmann (Hrsg.): SICHERHEIT 2006

P-78 K.-O. Wenkel, P. Wagner, M. Morgenstern, K. Luzi, P. Eisermann (Hrsg.): Land- und Ernährungswirtschaft im Wandel

P-79 Bettina Biel, Matthias Book, Volker Gruhn (Hrsg.): Softwareengineering 2006

P-80 Mareike Schoop, Christian Huemer, Michael Rebstock, Martin Bichler (Hrsg.): Service-Oriented Electronic Commerce

P-81 Wolfgang Karl, Jürgen Becker, Karl-Erwin Großpietsch, Christian Hochberger, Erik Maehle (Hrsg.): ARCS´06

P-82 Heinrich C. Mayr, Ruth Breu (Hrsg.): Modellierung 2006

P-83 Daniel Huson, Oliver Kohlbacher, Andrei Lupas, Kay Nieselt and Andreas Zell (eds.): German Conference on Bioinformatics

P-84 Dimitris Karagiannis, Heinrich C. Mayr, (Hrsg.): Information Systems Technology and its Applications

P-85 Witold Abramowicz, Heinrich C. Mayr, (Hrsg.): Business Information Systems

P-86 Robert Krimmer (Ed.): Electronic Voting 2006

P-87 Max Mühlhäuser, Guido Rößling, Ralf Steinmetz (Hrsg.): DELFI 2006: 4. e-Learning Fachtagung Informatik

P-88 Robert Hirschfeld, Andreas Polze, Ryszard Kowalczyk (Hrsg.): NODe 2006, GSEM 2006

P-90 Joachim Schelp, Robert Winter, Ulrich Frank, Bodo Rieger, Klaus Turowski (Hrsg.): Integration, Informationslogistik und Architektur

P-91 Henrik Stormer, Andreas Meier, Michael Schumacher (Eds.): European Conference on eHealth 2006

P-92 Fernand Feltz, Benoît Otjacques, Andreas Oberweis, Nicolas Poussing (Eds.): AIM 2006

P-93 Christian Hochberger, Rüdiger Liskowsky (Eds.): INFORMATIK 2006 – Informatik für Menschen, Band 1

P-94 Christian Hochberger, Rüdiger Liskowsky (Eds.): INFORMATIK 2006 – Informatik für Menschen, Band 2

P-95 Matthias Weske, Markus Nüttgens (Eds.): EMISA 2005: Methoden, Konzepte und Technologien für die Entwicklung von dienstbasierten Informationssystemen

P-96 Saartje Brockmans, Jürgen Jung, York Sure (Eds.): Meta-Modelling and Ontologies

P-97 Oliver Göbel, Dirk Schadt, Sandra Frings, Hardo Hase, Detlef Günther, Jens Nedon (Eds.): IT-Incident Mangament & IT-Forensics – IMF 2006

P-98 Hans Brandt-Pook, Werner Simonsmeier und Thorsten Spitta (Hrsg.): Beratung in der Softwareentwicklung – Modelle, Methoden, Best Practices

P-99 Andreas Schwill, Carsten Schulte, Marco Thomas (Hrsg.): Didaktik der Informatik

P-100 Peter Forbrig, Günter Siegel, Markus Schneider (Hrsg.): HDI 2006: Hochschuldidaktik der Informatik

P-101 Stefan Böttinger, Ludwig Theuvsen, Susanne Rank, Marlies Morgenstern (Hrsg.): Agrarinformatik im Spannungsfeld zwischen Regionalisierung und globalen Wertschöpfungsketten

P-102 Otto Spaniol (Eds.): Mobile Services and Personalized Environments

P-103 Alfons Kemper, Harald Schöning, Thomas Rose, Matthias Jarke, Thomas Seidl, Christoph Quix, Christoph Brochhaus (Hrsg.): Datenbanksysteme in Business, Technologie und Web (BTW 2007)

P-104 Birgitta König-Ries, Franz Lehner, Rainer Malaka, Can Türker (Hrsg.) MMS 2007: Mobilität und mobile Informationssysteme

P-105 Wolf-Gideon Bleek, Jörg Raasch, Heinz Züllighoven (Hrsg.) Software Engineering 2007

P-106 Wolf-Gideon Bleek, Henning Schwentner, Heinz Züllighoven (Hrsg.) Software Engineering 2007 – Beiträge zu den Workshops

P-107 Heinrich C. Mayr, Dimitris Karagiannis (eds.) Information Systems Technology and its Applications

P-108 Arslan Brömme, Christoph Busch, Detlef Hühnlein (eds.) BIOSIG 2007: Biometrics and Electronic Signatures

P-109 Rainer Koschke, Otthein Herzog, Karl-Heinz Rödiger, Marc Ronthaler (Hrsg.) INFORMATIK 2007 Informatik trifft Logistik Band 1

P-110 Rainer Koschke, Otthein Herzog, Karl-Heinz Rödiger, Marc Ronthaler (Hrsg.) INFORMATIK 2007 Informatik trifft Logistik Band 2

P-111 Christian Eibl, Johannes Magenheim, Sigrid Schubert, Martin Wessner (Hrsg.) DeLFI 2007: 5. e-Learning Fachtagung Informatik

P-112 Sigrid Schubert (Hrsg.) Didaktik der Informatik in Theorie und Praxis

P-113 Sören Auer, Christian Bizer, Claudia Müller, Anna V. Zhdanova (Eds.) The Social Semantic Web 2007 Proceedings of the 1st Conference on Social Semantic Web (CSSW)

P-114 Sandra Frings, Oliver Göbel, Detlef Günther, Hardo G. Hase, Jens Nedon, Dirk Schadt, Arslan Brömme (Eds.) IMF2007 IT-incident management & IT-forensics Proceedings of the 3rd International Conference on IT-Incident Management & IT-Forensics

P-115 Claudia Falter, Alexander Schliep, Joachim Selbig, Martin Vingron and Dirk Walther (Eds.) German conference on bioinformatics GCB 2007

P-116 Witold Abramowicz, Leszek Maciszek (Eds.) Business Process and Services Computing 1st International Working Conference on Business Process and Services Computing BPSC 2007

P-117 Ryszard Kowalczyk (Ed.) Grid service engineering and manegement The 4th International Conference on Grid Service Engineering and Management GSEM 2007

P-118 Andreas Hein, Wilfried Thoben, Hans-Jürgen Appelrath, Peter Jensch (Eds.) European Conference on ehealth 2007

P-119 Manfred Reichert, Stefan Strecker, Klaus Turowski (Eds.) Enterprise Modelling and Information Systems Architectures Concepts and Applications

P-120 Adam Pawlak, Kurt Sandkuhl, Wojciech Cholewa, Leandro Soares Indrusiak (Eds.) Coordination of Collaborative Engineering - State of the Art and Future Challenges

P-121 Korbinian Herrmann, Bernd Bruegge (Hrsg.) Software Engineering 2008 Fachtagung des GI-Fachbereichs Softwaretechnik

P-122 Walid Maalej, Bernd Bruegge (Hrsg.) Software Engineering 2008 - Workshopband Fachtagung des GI-Fachbereichs Softwaretechnik

P-123 Michael H. Breitner, Martin Breunig, Elgar Fleisch, Ley Pousttchi, Klaus Turowski (Hrsg.)
Mobile und Ubiquitäre Informationssysteme – Technologien, Prozesse, Marktfähigkeit
Proceedings zur 3. Konferenz Mobile und Ubiquitäre Informationssysteme (MMS 2008)

P-124 Wolfgang E. Nagel, Rolf Hoffmann, Andreas Koch (Eds.)
9th Workshop on Parallel Systems and Algorithms (PASA)
Workshop of the GI/ITG Speciel Interest Groups PARS and PARVA

P-125 Rolf A.E. Müller, Hans-H. Sundermeier, Ludwig Theuvsen, Stephanie Schütze, Marlies Morgenstern (Hrsg.)
Unternehmens-IT: Führungsinstrument oder Verwaltungsbürde
Referate der 28. GIL Jahrestagung

P-126 Rainer Gimnich, Uwe Kaiser, Jochen Quante, Andreas Winter (Hrsg.)
10th Workshop Software Reengineering (WSR 2008)

P-127 Thomas Kühne, Wolfgang Reisig, Friedrich Steimann (Hrsg.)
Modellierung 2008

P-128 Ammar Alkassar, Jörg Siekmann (Hrsg.)
Sicherheit 2008
Sicherheit, Schutz und Zuverlässigkeit
Beiträge der 4. Jahrestagung des Fachbereichs Sicherheit der Gesellschaft für Informatik e.V. (GI)
2.-4. April 2008
Saarbrücken, Germany

P-129 Wolfgang Hesse, Andreas Oberweis (Eds.)
Sigsand-Europe 2008
Proceedings of the Third AIS SIGSAND European Symposium on Analysis, Design, Use and Societal Impact of Information Systems

P-130 Paul Müller, Bernhard Neumair, Gabi Dreo Rodosek (Hrsg.)
1. DFN-Forum Kommunikations-technologien Beiträge der Fachtagung

P-131 Robert Krimmer, Rüdiger Grimm (Eds.)
3rd International Conference on Electronic Voting 2008
Co-organized by Council of Europe, Gesellschaft für Informatik and E-Voting.CC

P-132 Silke Seehusen, Ulrike Lucke, Stefan Fischer (Hrsg.)
DeLFI 2008:
Die 6. e-Learning Fachtagung Informatik

P-133 Heinz-Gerd Hegering, Axel Lehmann, Hans Jürgen Ohlbach, Christian Scheideler (Hrsg.)
INFORMATIK 2008
Beherrschbare Systeme – dank Informatik
Band 1

P-134 Heinz-Gerd Hegering, Axel Lehmann, Hans Jürgen Ohlbach, Christian Scheideler (Hrsg.)
INFORMATIK 2008
Beherrschbare Systeme – dank Informatik
Band 2

P-135 Torsten Brinda, Michael Fothe, Peter Hubwieser, Kirsten Schlüter (Hrsg.)
Didaktik der Informatik – Aktuelle Forschungsergebnisse

P-136 Andreas Beyer, Michael Schroeder (Eds.)
German Conference on Bioinformatics GCB 2008

P-137 Arslan Brömme, Christoph Busch, Detlef Hühnlein (Eds.)
BIOSIG 2008: Biometrics and Electronic Signatures

P-138 Barbara Dinter, Robert Winter, Peter Chamoni, Norbert Gronau, Klaus Turowski (Hrsg.)
Synergien durch Integration und Informationslogistik
Proceedings zur DW2008

P-139 Georg Herzwurm, Martin Mikusz (Hrsg.)
Industrialisierung des Software-Managements
Fachtagung des GI-Fachausschusses Management der Anwendungsentwick-lung und -wartung im Fachbereich Wirtschaftsinformatik

P-140 Oliver Göbel, Sandra Frings, Detlef Günther, Jens Nedon, Dirk Schadt (Eds.)
IMF 2008 - IT Incident Management & IT Forensics

P-141 Peter Loos, Markus Nüttgens, Klaus Turowski, Dirk Werth (Hrsg.)
Modellierung betrieblicher Informations-systeme (MobIS 2008)
Modellierung zwischen SOA und Compliance Management

P-142 R. Bill, P. Korduan, L. Theuvsen, M. Morgenstern (Hrsg.)
Anforderungen an die Agrarinformatik durch Globalisierung und Klimaveränderung

P-143 Peter Liggesmeyer, Gregor Engels, Jürgen Münch, Jörg Dörr, Norman Riegel (Hrsg.)
Software Engineering 2009
Fachtagung des GI-Fachbereichs Softwaretechnik

P-144 Johann-Christoph Freytag, Thomas Ruf, Wolfgang Lehner, Gottfried Vossen (Hrsg.)
Datenbanksysteme in Business, Technologie und Web (BTW)

The titles can be purchased at:

Köllen Druck + Verlag GmbH
Ernst-Robert-Curtius-Str. 14 · D-53117 Bonn
Fax: +49 (0)228/9898222
E-Mail: druckverlag@koellen.de