Gesellschaft für Informatik e.V. (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
• seminars
• proceedings
• dissertations
• thematics
current topics are dealt with from the vantage point of research and development, teaching and further training in theory and practice. The Editorial Committee uses an intensive review process in order to ensure high quality contributions.

The volumes are published in German or English.

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

ISSN 1617-5468
ISBN 978-3-88579-277-2

This volume contains papers from the Software Engineering 2011 conference held in Karlsruhe from February 21st to 25th 2011. The topics covered in the papers range from software requirements, technologies or development strategies to reports that discuss industrial project experience.

Ralf Reussner, Matthias Grund, Andreas Oberweis, Walter Tichy (Hrsg.):
Software Engineering 2011

183

GI-Edition

Lecture Notes in Informatics

**Ralf Reussner, Matthias Grund,
Andreas Oberweis, Walter Tichy (Hrsg.)**

Software Engineering 2011

Fachtagung des GI-Fachbereichs Softwaretechnik

21.–25. Februar 2011 in Karlsruhe

Proceedings

Ralf Reussner, Matthias Grund,
Andreas Oberweis, Walter Tichy (Hrsg.)

Software Engineering 2011

Fachtagung des GI-Fachbereichs Softwaretechnik

21. – 25. Februar 2011
in Karlsruhe

Gesellschaft für Informatik e.V. (GI)

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

Volume P-183

ISBN 978-3-88579-277-2
ISSN 1617-5468

Volume Editors
Ralf Reussner
Andreas Oberweis
Walter Tichy
Karlsruher Institut für Technologie (KIT)
Am Fasanengarten 5
76131 Karlsruhe
Email: {reussner, oberweis, tichy}@kit.edu
Matthias Grund
andrena objects ag
Albert-Nestler-Straße 11
76131 Karlsruhe
Email: matthias.grund@andrena.de

printed by Köllen Druck+Verlag GmbH, Bonn

Willkommen zur SE 2011 in Karlsruhe!

Die Tagung *Software Engineering 2011* (SE 2011) ist die siebte Veranstaltung einer inzwischen etablierten Reihe von Fachtagungen, deren Ziel die Zusammenführung und Stärkung der deutschsprachigen Software-Technik ist. Die SE 2011 bietet ein Forum zum intensiven Austausch über praktische Erfahrungen, wissenschaftliche Erkenntnisse sowie zukünftige Herausforderungen bei der Entwicklung von Softwareprodukten bzw. Software-intensiven Systemen. Sie richtet sich gleichermaßen an Teilnehmer aus Industrie und Wissenschaft.

Die Software Engineering-Tagungsreihe wird vom Fachbereich Software-Technik der Gesellschaft für Informatik e.V. getragen. Die Software Engineering 2011 wird von den Software-Technik-Professoren des Karlsruher Instituts für Technologie (KIT) und des Forschungszentrum Informatik (FZI) veranstaltet.

Die SE 2011 bietet im Hauptprogramm begutachtete Forschungsarbeiten und eingeladene wissenschaftliche Vorträge. Von den 32 Einreichungen für das technisch-wissenschaftliche Programm wurden 12 Beiträge 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 ein vielfältiges Begeltprogramm. Dazu gehört zum wiederholten Male das *SE-FIT*, ein Forum für Informatik-Transferinstitute und ein Doktorandensymposium. Neu dazu gekommen erstmalig bei der SE 2011 ist das Nachwuchswissenschaftlersymposium, bei dem sich junge Leiter und Leiterinnen wissenschaftlicher Nachwuchsgruppen mit ihren Forschungsideen vorstellen, ebenso erstmalig bei der SE 2011 ist der Tag der Informatiklehrer und -lehrerinnen, bei dem sich Informatiklehrer und -lehrerinnen informieren und austauschen über neue Ansätze der Software-Technik und Lehre um Schulunterricht. Vor dem Hauptprogramm der Konferenz finden 11 Workshops sowie sechs Tutorials zu aktuellen, innovativen und praxisrelevanten Themen im Software Engineering statt. Die Durchführung der Tagung Software Engineering 2011 wäre ohne die Mitwirkung der Sponsoren und vieler engagierter Personen nicht möglich gewesen. Ich bedanke mich bei allen Sponsoren der SE 2011. Darüberhinaus gebührt besonderer Dank Matthias Grund (andrena objects AG) und meinen Kollegen Andreas Oberweis für die Planung des Industrietags, Walter Tichy für die Organisation des Nachwuchswissenschaftlerprogrammes, Stefan Jähnichen (TU Berlin) für die Koordination des Workshop- und Tutorialprogramms, Alexander Pretschner für die Organisation des Doktorandensymposiums, Barbara Paech (Univ. HD) und Theo Heußer (Bergstrassen-Gymnasium) für die Organisation des Informatiklehrer/-Innen-Tages und Dr. Mircea Trifu (FZI) für die Planung und Durchführung des SE-FIT. Ganz besonders bedanke ich mich bei meinem Mitarbeiter Jörg Henß für seinen unermüdlichen Einsatz rund um die Organisation der Tagung, sowie bei meinen Mitarbeitern Zoya Durdik, Erik Burger, Qais Noorshams, Andreas Rentschler und Benjamin Klatt, meiner Assistentin Tatiana Rhode und meiner Sekretärin Elena Kienhöfer für die große Hilfe bei der Vorbereitung, sowie bei allen Mitarbeitern und Studierenden meiner Forschungsgruppe für die große Unterstützung während des Ablaufs der Tagung. Ohne diese Unterstützung wäre die SE 2011 nicht möglich.

Karlsruhe, im Februar 2011 — Ralf Reussner, Tagungsleiter

Tagungsleitung
Ralf Reussner, KIT Karlsruhe

Leitung Industrietag
Wilhelm Schäfer, Universität Paderborn

Leitung Workshops und Tutorials
Stefan Jähnichen, TU Berlin

Tagungsorganisation
Matthias Grund, andrena objects AG
Wilfried Juling, KIT Karlsruhe
Andreas Oberweis, KIT Karlsruhe
Alexander Pretschner, KIT Karlsruhe
Ralf Reussner, KIT und FZI Karlsruhe
Stefan Tai, KIT Karlsruhe
Walter Tichy, KIT Karlsruhe

Programmkomitee
Steffen Becker, Universität Paderborn
Klaus Beetz, Siemens AG
Manfred Broy, TU München
Bernd Brügge, TU München
Jürgen Ebert, Universität Koblenz-Landau
Gregor Engels, Universität Paderborn
Martin Glinz, Universität Zürich
Michael Goedicke, Universität Duisburg-Essen
Volker Gruhn, Universität Leipzig
Jens Happe, SAP AG
Wilhelm Hasselbring, Christian-Albrechts-Universität zu Kiel
Stefan Jähnichen, TU Berlin
Matthias Jarke, RWTH Aachen
Gerti Kappel, TU Wien
Udo Kelter, Universität Siegen
Jens Knoop, TU Wien
Heiko Koziolek, ABB
Claus Lewerentz, BTU Cottbus
Horst Lichter, RWTH Aachen
Peter Liggesmeyer, TU Kaiserslautern
Oliver Mäckel, Siemens AG
Florian Matthes, TU München
Oscar Nierstrasz, Universität Bern
Andreas Oberweis, KIT und FZI Karlsruhe
Barbara Paech, Universität Heidelberg
Klaus Pohl, Universität Duisburg-Essen

Alexander Pretschner, TU Kaiserslautern
Ralf Reussner, KIT und FZI Karlsruhe
Matthias Riebisch, TU Ilmenau
Andreas Roth, SAP AG
Bernhard Rumpe, RWTH Aachen
Wilhelm Schäfer, Universität Paderborn
Klaus Schmid, Universität Hildesheim
Kurt Schneider, Leibniz Universität Hannover
Andy Schürr, TU Darmstadt
Rainer Singvogel, msg systems ag, München
Stefan Tai, KIT und FZI Karlsruhe
Walter F. Tichy, KIT und FZI Karlsruhe
Markus Voß, Accso – Accelerated Solutions GmbH, Darmstadt
Andreas Winter, Universität Oldenburg
Mario Winter, Fachhochschule Köln
Uwe Zdun, Universität Wien
Andreas Zeller, Universität des Saarlandes
Heinz Züllighoven, Universität Hamburg
Albert Zündorf, Universität Kassel

Offizieller Veranstalter
Fachbereich Softwaretechnik der Gesellschaft für Informatik (GI)

Mitveranstalter
Karlsruher Institut für Technologie (KIT)
Forschungszentrum Informatik Karlsruhe (FZI)

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

Sponsoren

SE 2011 Platinsponsoren

SE 2011 Goldsponsoren

SE 2011 Silbersponsoren

SE 2011 Bronzesponsoren

Inhaltsverzeichnis

Eingeladene wissenschaftliche Vorträge

Präsentationen des Industrietags

Forschungsarbeiten

Überblicksartikel und Erfahrungsberichte

Workshops

SE | 11
SOFTWARE ENGINEERING

Eingeladene wissenschaftliche Vorträge

Dynamische Analyse mit dem Software-EKG

Johannes Weigend, Johannes Siedersleben, Josef Adersberger

QAware GmbH, Aschauer Str. 32, D-81549 München

{johannes.weigend, johannes.siedersleben, josef.adersberger}@qaware.de

Abstract: Dieses Papier zeigt, wie man komplexe, heterogene Systeme analysiert, wenn die einfachen Methoden (Debugger, Profiler) nicht ausreichen. Wir erläutern die Grundlagen, beschreiben ein Vorgehen mit den nötigen Tools und bringen einige Beispiele aus unserer Praxis. Wir behandeln ferner den präventiven Einsatz des Vorgehens im Entwicklungsprozess und definieren die Diagnostizierbarkeit (Diagnosibility) eines Software-Systems als wichtige nicht-funktionale Eigenschaft.

1 Welches Problem wollen wir lösen?

Die Prüfung auf *funktionale* Korrektheit und Vollständigkeit eines Systems ist zumindest in der Theorie kein Problem: Jedes Vorgehensmodell von RUP bis V-Modell legt genau fest, was von der Anforderungsanalyse bis zum Abnahmetest zu tun ist. Aber bei den *nicht-funktionalen* Eigenschaften (also Performance, Robustheit, Verfügbarkeit usw.) liegen die Dinge anders: Es gibt bis heute keine gesicherten Verfahren, um nichtfunktionale Eigenschaften geplant herbeizuführen, wir können sie nur mit enormen Aufwand testen (z.B. durch Parallelbetrieb), und wirklich sichtbar werden sie erst in der Produktion. Natürlich kann man Hardware und Bandbreite großzügig auslegen, man kann ein System redundant gestalten und Failover-Mechanismen vorsehen. Aber damit trifft man die eigentliche Ursache nicht: Harte Performance-Fehler legen jede noch so leistungsfähige CPU lahm, Hardware-Redundanz hilft nicht bei Software-Fehlern, und auf Failover-Mechanismen ist im Ernstfall nicht immer Verlass.

Gute Performance und hohe Robustheit sind das Ergebnis vermiedener Fehler und einer geeigneten Architektur. Das *Software-EKG* dient der systematischen Überprüfung von Performance und Robustheit auf der Basis von Monitoring und Logging. Man erkennt sowohl Programmierfehler (etwa einen Speicher-Leak) als auch Schwächen der Architektur (etwa einen ungeeigneten Client/Server-Schnitt). Das Software-EKG ist ein Verbund von Werkzeug und Vorgehen zur Unterstützung von Planung und Test nichtfunktionaler Eigenschaften. Das Vorgehen hat eine gewisse Ähnlichkeit mit dem Einsatz von EKGs in der Medizin: Man legt viele Messkurven übereinander; die Charts ähneln entfernt den richtigen EKGs. Und wie in der Medizin gibt es typische Krankheitsbilder, die wir mit Software-Unterstützung erkennen.

Der Anlass für die Entwicklung des Software-EKG war eine Krise: Ein komplexes, un-

ternehmenskritisches System lief unbefriedigend; die Benutzer protestierten massiv. So beschreiben wir in Abschnitt 2 unseren Ansatz zunächst aus der Feuerwehr-Sicht, wenn es darum geht, ein krankes System rasch wieder auf die Beine zu stellen. Abschnitt 3 zeigt, wie sich das Software-EKG in die bestehende Welt von Tools und Verfahren einordnet, Abschnitt 4 beschreibt Aufbau und Funktionsweise. Abschnitt 5 befasst sich mit dem systematischen Einsatz des Software-EKG in Krisensituationen. Die Fehler, die wir dabei entdecken, sind immer dieselben und sind beschrieben in Abschnitt 6. Abschnitt 7 beleuchtet den präventiven Einsatz des Software-EKGs, etwa im Entwicklungsprozess. Aber Prävention betrifft auch die Software-Architektur: Unter der Überschrift *Design for Diagnosibility (DfD)* beschreiben wir in Abschnitt 8 den Weg zum *gläsernen System*.

2 Wie untersucht man kranke Systeme?

Manche Systeme sind krank: Sie werden langsam, verhalten sich eigenartig, produzieren Fehler und reagieren am Ende gar nicht mehr. Andere verhalten sich launisch, laufen tagelang problemlos, um dann plötzlich ohne Warnung zu abzustürzen. Woran kann das liegen? Zuwenig Speicher, zu viele Threads, zu wenig Bandbreite, Programmierfehler, inkompatible Versionen von Softwareprodukten, Fehler in einem Produkt?

Die Werkzeuge zur Analyse des laufenden Systems sind zahlreich: Es gibt Monitore (für die JVM, Betriebssystem, Datenbank, Massenspeicher, Netzwerk), Logger, Profiler und Debugger. Aber obwohl diese Tools große Mengen an Information liefern, erweist sich die Diagnose kranker Systeme immer wieder als die Suche nach der Stecknadel im Heuhaufen, vor allem dann, wenn es um Systeme geht, die auf mehrere Rechner verteilt sind, oft mit verschiedenen Betriebssystemen, Datenbanken und Programmiersprachen, das Ganze verborgen hinter einem Web-Client. Der Grund für diese Misere sind zunächst einmal die handelnden Personen: Da sind die verschiedenen Experten für Betriebssystem, Datenbank oder Web-Anbindung, die sich in der Regel in ihrem Fachgebiet sehr gut auskennen, aber von den anderen Bereichen – vor allem von der Anwendung selbst – zu wenig wissen. Dann gibt es die Projektmitarbeiter, vom Programmierer bis zum Chefarchitekten, die ihre Anwendung kennen, aber oft wenig wissen über die Interna der technischen Infrastruktur. Diese verschiedenen, im Extremfall disjunkten Kompetenzen sind regelmäßig Anlass für ein Kommunikationsproblem, das dadurch verstärkt wird, dass alle Beteiligten Fehler im jeweils eigenen Zuständigkeitsbereich ausschließen: Der Software-Architekt erlaubt nicht, dass man seinen Entwurf in Frage stellt, der Programmierer weist den Vorwurf zurück, er habe seine Threads falsch synchronisiert, und der Microsoft-Berater hält es für abwegig, eine Microsoft-Komponente zu verdächtigen.

Aber disjunkte Kompetenzen und psychologische Empfindlichkeiten sind keine ausreichende Erklärung für die oft beobachtete Hilflosigkeit bei der Analyse kranker Systeme. Der wichtigste Grund lautet nämlich: Wir arbeiten unprofessionell. Die genannten Werkzeuge werden meistens ad hoc eingesetzt, man misst über zufällige Zeiträume und in der Regel viel zu kurz, manchmal in der Produktion, manchmal in der Testumgebung, die Messprotokolle liegen in unterschiedlichen Formaten in diversen Verzeichnissen herum und werden mit trickreichen grep-Aufrufen durchforstet. Weil es für die verschiedenen

Messungen keine gemeinsame Zeitachse gibt, lässt sich kaum feststellen, wie die Benutzer mit ihrem Verhalten das System beeinflussen. Im Übrigen skaliert der manuelle Ansatz nicht: Zehn Rechner mit zehn Prozessen und zehn Messwerten pro Prozess ergeben 1000 Messwerte pro Messzeitpunkt. Kein Mensch kann so etwas manuell verfolgen, geschweige denn auswerten. Aber die korrekte, schnelle Diagnose sollte nicht vom Zufall abhängen, sondern das Ergebnis einer systematischen Analyse sein. Das Software-EKG ist die Vorlage für eine systematische Analyse und lässt sich in sechs Punkten zusammenfassen:

1. Wir beobachten das System über definierte, oft sehr lange Zeiträume. Wir verwenden jede Art von Messung, auf jeder Ebene (Betriebssystem, Datenbank, Application Server, o. a.) und auf verschiedenen Rechnern.

2. Wir verwenden fertige Schnittstellen zu den gängigen Messwerkzeugen; neue oder exotische Umgebungen lassen sich jederzeit anbinden; der dafür nötige Aufwand hängt natürlich von der jeweiligen Umgebung ab. Wir speichern alle Messwerte in einer einzigen zentralen Datenbank ab, die sehr groß werden kann, und deren Rückgrat die gemeinsame Zeitachse aller Messungen ist. Die beteiligten Rechner können irgendwo auf der Welt stehen. Diese Datenbank gestattet zahlreiche Auswertungen und Visualisierungen.

3. Wir messen nicht nur das Verhalten des Systems, sondern auch das der Benutzer über dieselben Zeiträume. Wir speichern alle Protokolle in derselben Datenbank mit derselben Zeitachse.

4. Dadurch sind wir in der Lage, Systemanomalien (z.B. eine drastisch erhöhte CPU-Last) durch vertikale Schnitte über alle Messungen und Protokolle hinweg zu analysieren. Wir sehen genau, was auf den verschiedenen Ebenen in den verschiedenen Rechnern in einem bestimmten Zeitpunkt passiert ist – und das ist genau die Information, die man für die Diagnose braucht. Wir kennen inzwischen eine Reihe typischer Krankheitsbilder, genauso wie die Ärzte bei den EKGs.

5. Ein Werkzeug zur automatischen Trenderkennung unterstützt uns bei der Suche nach Anomalien.

6. Erst jetzt setzen wir – wenn nötig – Profiler und Debugger ein, und zwar nicht aufs Geratewohl, sondern genau am Krankheitsherd.

Die zentrale Datenbank mit vielfältigen Auswertungen ist eine gemeinsame Informationsbasis für die unterschiedlichen Experten, die man natürlich immer noch braucht. Aber jetzt können sie auf der Basis von Fakten sachlich und konstruktiv diskutieren.

3 Was gibt es schon?

Das Software-EKG befasst sich mit dem laufenden System und ist daher ein Werkzeug der *dynamischen Analyse*. Dort gibt es vier Werkzeuggattungen, die sich in ihrer Invasi-

vität und Einsetzbarkeit unterscheiden: Monitore, Logger, Profiler und Debugger. Werkzeuge wie Sonar[1], structure101[2] oder Sotograph[3] betreiben *statische Analyse*: Sie untersuchen auf der Basis der Quellen die statische Struktur des Systems, ermitteln Kennzahlen zur Komplexität und geben Hinweise auf unerwünschte oder unzulässige Abhängigkeiten. Aber das hat mit unserem Problem fast nichts zu tun: Ein Pannenhelfer, der das liegengebliebene Auto nicht repariert, sondern erst einmal die Konstruktion des Motors analysiert, wird wenig Dankbarkeit ernten.

Die Werkzeuge der dynamischen Analyse sind vielfältig: *Monitore* findet man als Bestandteil des Betriebssystems (z.B. Windows Perfmon), der Datenbank, des Plattensystems, des Netzwerks oder von virtuellen Maschinen. Über den Betriebssystem-Monitor sieht man CPU-Auslastung, Prozesse und den belegten Speicherplatz. Der Datenbank-Monitor gibt Auskunft über stattgefundenen SQL-Abfragen, Ausführungsgeschwindigkeit, möglicherweise auch Deadlocks. Plattensystem-Monitore liefern Informationen über Speicherzugriffe, Wartezeiten und die Verwendung von Caches. Netzwerk-Sniffer verraten uns Latenzzeiten und Übertragungsraten. JMX[4] ist die Schnittstelle des Monitors der JVM (Java Virtual Machine). Man sieht z.B., was der Garbage-Collector tut, wie viele Objekte die JVM erzeugt und freigibt, und den Speicherbedarf. JMX ermöglicht auch den Zugriff auf anwendungsspezifische Kennzahlen, sofern die Anwendung dafür vorbereitet ist. Nagios[5] ist ein Monitoring-Werkzeug, das für alle gängigen Unix-Systeme zur Verfügung steht und Daten beliebiger Monitoren einsammelt. JMX und Nagios sind beim typischen Einsatz nicht oder minimal invasiv, lassen sich aber auch für invasive Aktionen missbrauchen.

Auch *Logger* (z.B. Log4J) gehören zur dynamischen Analyse. Über den Log-Level lässt sich der Umfang des Protokolls steuern: Logging auf niedrigem Level ist wenig invasiv; Logging auf hohem Level kann das System drastisch verlangsamen und ist daher im Produktivbetrieb ausgeschlossen. *Profiler* und *Debugger* beeinflussen das Systemverhalten ebenfalls maßgeblich und kommen daher nur in der Entwicklung und im Test zum Einsatz. Beide beobachten in der Regel immer nur einen Prozess.

Das Software-EKG systematisiert den Einsatz von Monitoren und Logger und gibt zuverlässige Hinweise auf kritische Bereiche, die dann mit Profiler und Debugger im Detail zu untersuchen sind. Das Software-EKG verwendet derzeit Windows Performance Counter, JMX und Nagios und ist offen für andere (mehr dazu in Abschnitt 3). Diese Tools sind beschränkt auf eine bestimmte Plattform (Windows, Unix, Java) und funktionieren im Wesentlichen wie der Taskmanager, den viele Windows-Benutzer nur allzu gut kennen: Man beobachtet die Kennzahlen der Prozesse eines Rechners online über einen kurzen Zeitraum. Das ist völlig ausreichend, um zu sehen, welches Windows-Programm gerade die CPU blockiert. Wir aber beobachten viele Rechner, viele Prozesse und Threads über Stunden, Tage und manchmal Wochen, und analysieren die ermittelten Messwerte offline mit einer leistungsfähigen GUI und einem Tool, zur automatischen Trenderkennung.

Unter dem Namen *Application Performance Management (APM)* hat IBM einen Ansatz

[1] `http://www.sonarsource.org`
[2] `http://www.headwaysoftware.com/products/structure101`
[3] `http://www.hello2morrow.com/products/sotograph`
[4] Java Management Extensions
[5] `http://www.nagios.org`

vorgestellt, das denselben Zweck verfolgt wie das Software-EKG [Whi08]. Die bekanntesten APM-Werkzeuge sind dynaTrace[6] und AppDynamics[7], die Monitoring und Profiling verbinden. Sie sind aber beschränkt auf Java bzw. .NET, machen erhebliche Annahmen über das zu untersuchende System und verursachen mehr als 5% Overhead. APM-Werkzeuge sind hilfreich als Datenquelle für das Software-EKG.

Der Forschungszweig *Software Performance Engineering* (SPE, siehe [BM04], [MKBR10]) verfolgt einen ganz anderen Ansatz zum selben Thema: Man versucht, die Performance von Komponenten in Abhängigkeit vom Ressourcenverbrauch und von der Performance importierter Komponenten formal zu beschreiben und daraus per Modellbildung die Performance des künftigen Systems zu ermitteln. Wir halten diesen Weg für interessant und vielversprechend für den Software-Entwurf, aber ungeeignet für Feuerwehreinsätze.

4 Das Software-EKG-Tool

Das Tool ist einfach und leichtgewichtig, weniger als 100 Bearbeitertage stecken darin. Es ist in Java programmiert und enthält etwa 10000 LOC. Wir erläutern zunächst Funktionsumfang, erklären die verwendeten Datenquellen (Monitore und Logger), und zeigen schließlich, wie das Tool tatsächlich gebaut ist und mit den zu untersuchenden Systemen zusammenarbeitet.

4.1 Funktionsumfang

Das Tool sammelt Kennzahlen und wertet sie aus. Die Kennzahlen sind vielfältig, es können neue, heute unbekannte dazukommen, und sie stammen aus verschiedenen Quellen. Die erwarteten Mengen sind enorm; wir erinnern an das Beispiel mit 10 Rechnern, jeweils 10 Prozessen und jeweils 10 Kennzahlen: 6 Messungen pro Minute ergeben 360.000 Messwerte pro Stunde und $8,64$ Millionen pro Tag. Das Problem ist natürlich nicht der Speicherplatz, sondern die Auswertung: Wer soll das alles anschauen?

Per Konfiguration können wir einstellen, welche Kennzahlen in welcher Frequenz von den Zielsystemen gelesen werden. Die Konfiguration erfolgt in der Weise, dass man bei k Rechnern, m Prozessen und n Kennzahlen nur $k + m + n$ Parameter einstellt, sofern alle Rechner und alle Prozesse auf jedem Rechner gleichbehandelt werden sollen. Natürlich kann man jeden Rechner und jeden Prozess auch individuell parametrieren, so dass man maximal $k * m * n$ Parameter setzt. Das Tool unterstützt Kurzzeit-EKGs und Langzeit-EKGs. Kurzzeit-EKGs dienen der Analyse eines bestimmten Anwendungsfalls und dauern nicht länger als eine Stunde. Langzeit-EKGs dauern Tage oder Wochen.

Die Sammlung der Daten kann an verschiedenen Rechnern irgendwo auf der Welt stattfinden, aber die Auswertung erfolgt in einem zentralen Analysesystem. Dieses System erhält

[6] `http://www.dynatrace.com`
[7] `http://www.appdynamics.com`

die eingesammelten Daten als CSV-Dateien in einem festen Format und lädt diese in die zentrale Datenbank. Der Ladevorgang lässt sich parametrieren; insbesondere kann man die Daten bereits beim Laden verdichten, indem man etwa 360 Messungen pro Stunde jeweils durch Mittelwert, Maximum und Minimum ersetzt. Die geladenen Messwerte lassen sich über eine komfortable GUI betrachten: Man kann den Maßstab verändern und verschiedene Messreihen übereinanderlegen. Es ist auch nach dem Ladevorgang möglich, Daten zu verdichten.

4.2 Datenquellen

Ergiebige Datenquellen sind die Programmschnittstellen *JMX* und *Windows Performance Counter*. Auf Unix-Systemen verwenden wir *Nagios* und sind im Übrigen offen für jede Art von Log-Dateien, die viele Systeme produzieren. In allen Fällen beeinflusst die Analyse das untersuchte System nicht: Es läuft während der Messung so gut oder so schlecht wie sonst auch. Aus offensichtlichen Sicherheitsgründen ist es bei *JMX*, *Windows Performance Counter* und *Nagios* nötig, beim Start des zu untersuchenden Systems die Erlaubnis für den Zugriff von außen zu erteilen. In vielen Fällen werden diese Schnittstellen in Produktionsumgebungen bereits eingesetzt.

Viele Systeme haben eigene Protokollmechanismen, die man im Testbetrieb ein- und ausschalten kann. Diese liefern CSV-Dateien entweder direkt oder können in dieses Format umgewandelt werden. Wir verarbeiten jede Art von CSV-Dateien, wenn nur jede Zeile einen Messwert enthält, und wenn man über Position, Offset oder einen regulären Ausdruck herausbekommt, wann der Wert gemessen wurde und was er bedeutet. Schließlich gibt es verschiedene Frameworks, die fachliche Kennzahlen liefern. Ein Beispiel ist das Open-Source-Framework JETM[8] für Java. Mit JETM lassen sich mittels eines AOP[9]-Adapters Messpunkte für die Aufrufe von beliebigen Spring-Beans definieren. Diese Messwerte lassen sich mit JMX abfragen. Der Einbau von AOP-Mechanismen muss natürlich im Rahmen des Systementwurfs vorbereitet sein – mehr dazu in Abschnitt 8 (Design for Diagnosibility).

4.3 Architektur

Das EKG-Tool besteht aus zwei Komponenten: dem Datensammler (*Kollektor*) und dem Analysewerkzeug (*Auswerter*). Zu einem Versuchsaufbau gehören beliebig viele Kollektoren, die ihre Messwerte von beliebig vielen Rechnern über beliebige Quellen einsammeln, und ein Auswerter, der über CSV-Dateien versorgt wird. Die Trennung der beiden Komponenten ist zwingend, denn Sammeln und Analysieren sind zwei verschiedenen Tätigkeiten, die an verschiedenen Orten und zu verschiedenen Zeiten stattfinden (Abbildung 1). Hierin unterscheidet sich das Software-EKG von APM, wo alle Analysen zur Laufzeit stattfinden.

[8] `http://jetm.void.fm`
[9] Aspect Oriented Programming

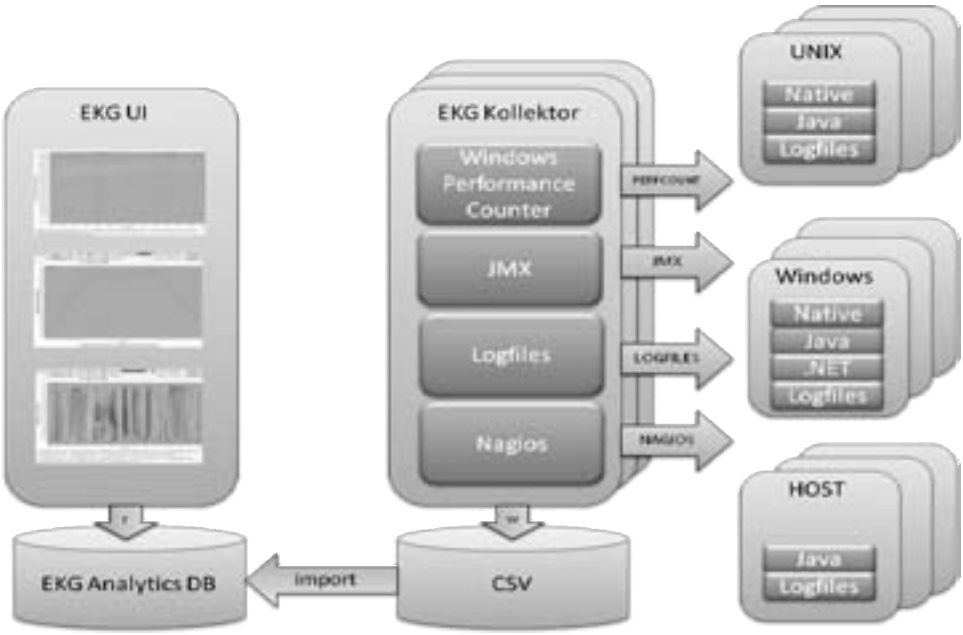


Abbildung 1: Architektur des EKG-Tool

Kollektor und Auswerter sind einfache Java-Anwendungen, die sich problemlos auf beliebigen Rechnern installieren lassen. Der Auswerter hat eine grafische Benutzeroberfläche, die mit Swing, JDesktop und JFreeChart realisiert ist; die angebotenen Funktionen orientieren sich an [UW07]. Die EKG Analytics DB ist nichts anderes als ein Ordner von CSV-Dateien; relationale Datenbanken haben sich wegen langer Importzeiten und umständlichen, langlaufenden Abfragen nicht bewährt. Durch die gewählte Architektur entfallen Importzeiten fast vollständig; einfache Schnittstellen zur Speicherung und zur Ermittlung von Kennzahlen ermöglichen die schnelle Anbindung von anderen Speichermedien und Datenquellen. In der Summe besteht die GUI aus 36 Klassen, 300 Methoden und 3000 Programmzeilen.

4.4 Automatische Trenderkennung

Die Analysedatenbank enthält viele Tausend Messreihen über derselben Zeitachse. Die manuelle Suche nach Anomalien ist auch mit einem komfortablen GUI mühsam, und natürlich besteht die Gefahr, dass selbst ein erfahrener Chefarzt den entscheidenden Punkt übersieht. Deshalb besitzt das Software-EKG ein Tool zur automatischen Trenderkennung mit derzeit drei Detektoren: Der *Trenddetektor* erkennt Monotonie-Eigenschaften, etwa den Anstieg des Ressourcenverbrauchs. Der *Korrelationsdetektor* erkennt Korrelationen

zwischen verschiedenen Messreihen, etwa den parallelen Anstieg von Speicherverbrauch und der Anzahl von Threads. Andere Beispiele: Wachsender Speicherbedarf in Verbindung mit hoher CPU-Auslastung ist ein Hinweis auf ein Memory Leak in Verbindung mit einem heftig arbeitendem Garbage Collector, CPU-Auslastung und I/O-Last nahe Null signalisieren einen Dead-Lock. Die Laufzeit des Korrelationsdetektors wächst quadratisch in der Anzahl der untersuchten Messreihen und kann Stunden betragen. Der *Frequenzdetektor* unterzieht die Messkurven der Fouriertransformation und erkennt so Änderungen der Frequenz.

Aber alle Detektoren liefern nur Anomalie-Kandidaten; die Entscheidung trifft der Chefarzt. Mit der automatischen Trenderkennung stehen wir am Anfang eines vielversprechenden Weges: Die gesamte Theorie der Zeitreihenanalyse wartet auf ihren Einsatz bei der Diagnose von instabilen Software-Systemen.

5 Vorgehen in der Krise

Das Software-EKG liefert enorme Mengen an Messwerten rasch und mit wenig Aufwand. Aber gerade deshalb sollte man sich vorher genau überlegen, was man eigentlich messen will, und was man mit den Messwerten tun wird. Die die schlichte Anwendung des gesunden Menschenverstands und die Übertragung gängiger Teststrategien führen zu einem Vorgehen in sechs Schritten:

1. **Bestandsaufnahme System:** Wir lassen uns erklären, was da überhaupt läuft. Wir müssen wissen, welche Rechner involviert sind, was auf diesen Rechnern alles installiert ist, um welche Anwendungen es geht, und aus welchen Komponenten und Frameworks diese Anwendungen aufgebaut sind. Es geht also um die T- und die TI-Architektur im Sinne von [Sie04]. Diese Bestandsaufnahme ist in der Regel mühsam, denn oft ist einfach nicht klar, was da alles läuft. Dies liegt zum Teil an mangelnder Dokumentation, zum Teil aber auch an der komplizierten Systemsoftware: .NET-Serveranwendung bestehen oft aus einer Unzahl von COM+ Surrogatprozessen (`dll-host.exe`); einige Java-Anwendungen benutzen externe Prozesse (`.cmd`, `.wsh`). Manchmal zeigt eine genaue Betrachtung, dass der Zoo von Hard- und Software so kompliziert oder so ungeeignet ist, dass er einfach nicht funktionieren kann.

2. **Bestandsaufnahme Problem:** Wir lassen uns die Probleme und die bisher eingesetzten Workarounds (z.B. Reboot) erklären. Es gibt Performanceprobleme und Probleme der Stabilität (mehr dazu in Abschnitt 6). Wir klassifizieren beide Typen nach der Lokalität (lokal, systemweit) und der Art, wie sie zeitlich auftreten (permanent, erratisch oder degradierend). *Degradierend* heißt, dass die Probleme erst nach einer gewissen Laufzeit auftreten.

3. **Messplan und Messumgebung:** Wir formulieren Hypothesen über mögliche Fehlerursachen und ermitteln daraus die zu messenden Kennzahlen. Wir legen fest, welche Systeme wir vermessen, und installieren die nötigen Kollektoren und den Auswerter.

Wir legen fest, was an den Zielsystemen zu tun ist, damit die Messungen möglich sind, und wir legen fest, was mit den Zielsystemen während des Messzeitraums geschieht.

4. **Messung und Analyse:** Wir führen die Messungen durch, laden die Messergebnisse in den Auswerter und versuchen zu verstehen, was da alles passiert ist. Dieser Schritt hat drei mögliche Ausgänge: Fehler gefunden, reparieren (Schritt 6), Problem eingekreist, Detailanalyse (Schritt 5), Messdaten zu wenig aussagekräftig, Messplan und Messumgebung anpassen (Schritt 3).

5. **Detailanalyse:** Wir untersuchen die verdächtigen Stellen (Hotspots) mit Profiler, Debugger und Netzwerk-Sniffer. Dieser Schritt hat zwei mögliche Ausgänge: Fehler gefunden, reparieren (Schritt 6), Messdaten zu wenig aussagekräftig, Messplan und Messumgebung anpassen (Schritt 3). Wichtig ist, dass in Schritt 4 die richtigen Hotspots gefunden wurden, denn Profiling an der falschen Stelle kann das Analyseteam über Monate in die falsche Richtung schicken.

6. **Reparatur und Test:** Mindestens ein Fehler wurde gefunden. Er wird behoben oder durch einen Workaround unschädlich gemacht; ein Regressionstest stellt sicher, dass alle Funktionen noch fachlich richtig laufen. Der Erfolg der Reparatur wird überprüft (Schritt 4).

Die Analyse eines kranken Systems ist für die betroffenen Software-Ingenieure und Manager ein schmerzhafter Prozess, der Fingerspitzengefühl und Überzeugungskraft verlangt. Der Leidensdruck ist hoch, und das Projektteam hat natürlich schon alles versucht, um die Fehler zu finden. Nach unserer Erfahrung kann in der Problemanalyse folgende Teamkonstellation erfolgreich sein: Die gesamte Analyse erfolgt im Auftrag und in der Verantwortung eines Managers, der so weit oben angesiedelt ist, dass er im Fall von Schuldzuweisungen oder Blockaden eingreifen kann. Die wichtigste Person ist der Chefarzt, der nicht aus dem Projekt kommen darf, denn sonst wäre er als Übeltäter verdächtig. Der Chefarzt hat die Erfahrung und das technische Wissen für den gesamten Analyseprozess. Er braucht kompetente und kooperationsbereite Ansprechpartner für alle relevanten Gebiete von der GUI-Programmierung bis zu den fachlichen Themen. Das Vorgehen ist in erster Linie iterativ: Der Chefarzt wird sich zu Beginn auf einen kleinen Problembereich konzentrieren. Im besten Fall gelingen schon in den ersten Tagen ein paar schnelle Erfolge – das schafft Vertrauen und beruhigt die Gemüter. Der Chefarzt muss sich auf die Fehlersuche konzentrieren und die Fehlerbehebung dem Projektteam überlassen.

Ein wichtiger Erfolgsfaktor ist die Verfügbarkeit geeigneter Umgebungen: In der Produktion kann man schon aus Sicherheitsgründen nicht alles messen, und der Produktivbetrieb ist nicht reproduzierbar. Daher braucht man zum Nachstellen von Fehlersituationen, zur versuchsweisen Reparatur oder auch zum Vergleich von Reparaturalternativen geeignete Umgebungen in nicht unerheblicher Anzahl. Hier drängen sich virtuelle Umgebungen auf der Basis von VMware oder Xen geradezu auf.

6 Typische Probleme

Es überrascht nicht, dass immer wieder dieselben Probleme auftreten: Leaks, hoher CPU-Verbrauch, Blockaden (Locks) und hohe I/O-Last. In diesem Abschnitt erläutern wir die Standardprobleme und ihre Ursachen, die immer wieder dieselben sind. Dabei ist folgendes wichtig: Die Analyse eines vorliegenden Diagramms ist meistens trivial – auch ein Laie würde z.B. erkennen, dass die CPU ausgelastet ist. Die Kunst besteht darin, im Heuhaufen der zahllosen Messungen die entscheidenden Anomalien durch Spürsinn und mit Unterstützung durch die automatische Trenderkennung zu finden. Von den sechs Schritten des Standardvorgehens können wir hier nur die Detailanalyse vorstellen: Die Bestandsaufnahme liegt hinter uns, die Messungen sind gelaufen, und wir haben das Chart gefunden, das uns den Hotspot zeigt, die Stelle also, wo wir per Profiler und Debugger weitersuchen müssen. Der dornige Weg vom Beginn der Suche bis zum gefundenen Hotspot entzieht sich einer kurzen Darstellung – das muss man einfach mal gemacht haben.

6.1 Leaks

Leaks betreffen endliche Ressourcen, das sind Hauptspeicher, Threads und Handles. Jedes System hat nur endlich viel Speicher, verkraftet nur eine bestimmte Anzahl an Threads und Handles. Der Fehler liegt immer in der unsachgemäßen Allokierung und/oder Deallokierung dieser Ressourcen. Wir zeigen als Beispiel eine Messung über 8 Tage vom 3. bis zum 9. Dezember 2009. Die Y-Achse zeigt die Anzahl der Threads im betrachteten System. Alle fünf Stunden kommen 25 Threads dazu. Das geht eine Weile gut, aber nach vier Tagen ist mit rund 400 aktiven Threads das Maximum erreicht, und das System versucht verzweifelt aber erfolglos, weitere Threads zu allokieren. Es gab im Messzeitraum zwei Neustarts.

Durch Analyse der Logfiles vom 7.12. 0:30 konnte die Ursache ermittelt werden: Das Problem lag in einem selbstgebauten Thread Pool mit einem systematischen Fehler beim Beenden von Threads. Es handelte sich um einen lokalen Fehler, der mit geringem Aufwand beseitigt werden konnte.

6.2 CPU-Verbrauch

Ein notorischer CPU-Fresser ist die Speicherbereinigung (Garbage Collection), die im Falle eines Memory Leak immer häufiger und mit immer weniger Erfolg tätig wird. Systeme sterben also einen doppelten Tod durch Speichermangel und Mangel an CPU-Kapazität – sie erfrieren und verhungern gleichzeitig. Aber die vielleicht wichtigste Ursache von hohem CPU-Verbrauch ist naive Programmierung, der wir immer wieder in immer neuen Formen begegnen. Ein Beispiel: Um festzustellen, welche Objekte sowohl in einer Liste A als auch in einer Liste B vorkommen, kann man eine doppelte Schleife programmieren mit dem Aufwand $O(n * m)$. Wenn man aber beide Listen sortiert, beträgt der Aufwand

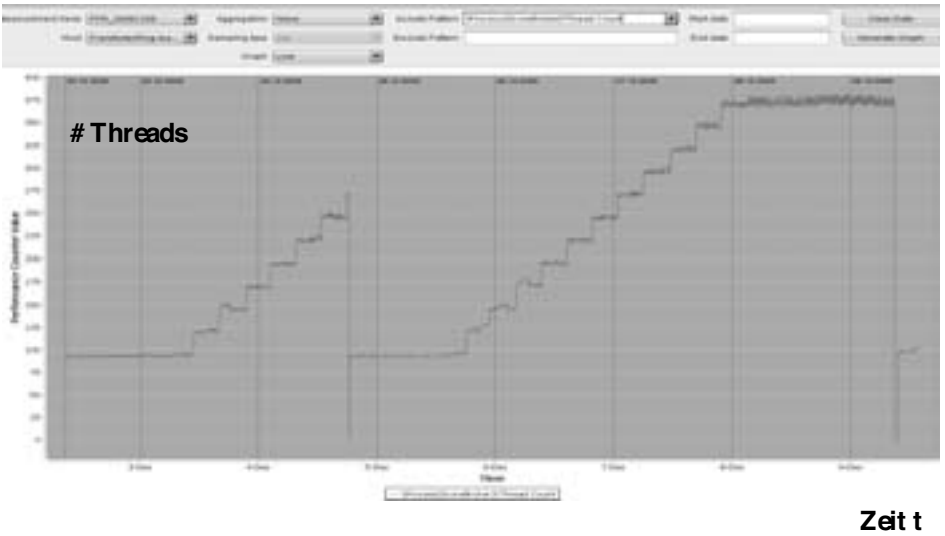


Abbildung 2: Ein Thread Leak

nur $O(n * log(n))$; dabei sei n die Länge der längeren Liste. Solange eine der beiden Liste kurz ist, spielt der Unterschied keine Rolle, und daher kann das naive Programm viele Tests überstehen und vielleicht sogar einige Jahre in der Produktion. Aber eines Tages sind aus Gründen, die ganz woanders liegen, beide Listen sehr lang, und so kommt es, dass ein Programm, das lange Zeit problemlos gelaufen ist, plötzlich den Rechner lahm legt.

Das folgende Beispiel betrifft die oft unterschätzte Serialisierung von Objekten. Das Diagramm zeigt den CPU-Anteil mehrerer Prozesse auf einem Rechner. Jeder farbige Graph entspricht einem Prozess. Im Beobachtungszeitraum von etwa 16 Minuten beansprucht der Prozess mit dem schwarzen Graph die CPU für jeweils ein bis zwei Minuten nahezu vollständig für sich (das sind die roten Kästen) und blockiert alle anderen Prozesse.

Dank der Informationen über den verursachenden Prozess und die genaue Tatzeit war die Ursache mit dem Profiler rasch gefunden: Der CPU-Verbrauch stieg immer kurz nach dem Lesen aus der Datenbank und kurz vor dem Schreiben in die Datenbank auf fast 100%. Der Grund: Nach dem Motto „Doppelt genäht hält besser“ wurde ein umfangreiches Objektgeflecht bei jedem Datenbankzugriff gleich zweimal hintereinander serialisiert. Man kann lange über Sinn und Unsinn von Serialisierung streiten, aber einmal Serialisieren sollte auf alle Fälle langen. Der Ausbau der überflüssigen Serialisierung reduzierte den CPU-Verbrauch auf erwartete 50% und löste damit das akute Problem. Aber auch die einmalige Serialisierung ist teuer.

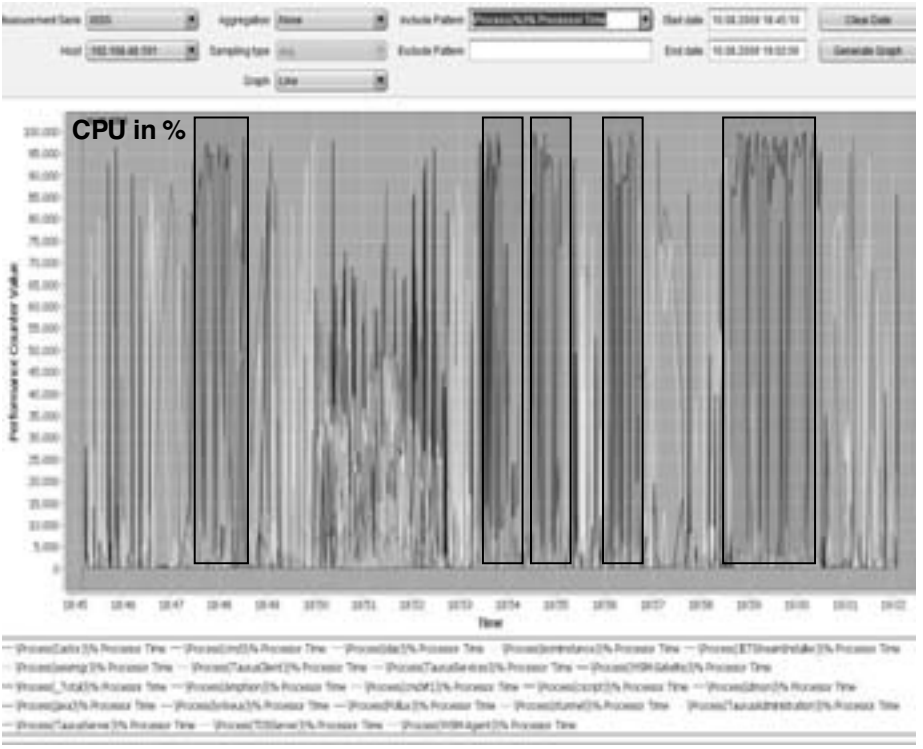


Abbildung 3: Ein Prozess, der die CPU in unregelmäßigen Abständen nahezu blockiert

6.3 Locks

Wir unterscheiden *Shortlocks*, *Livelocks* und *Deadlocks*. Sehr kurze Shortlocks (Millisekunden oder weniger) bleiben oft lange Zeit unbemerkt, aber ein Shortlock, der viele Sekunden oder gar Minuten dauert, ist außerordentlich störend. Livelocks entstehen, wenn zwei oder mehr Threads nach dem Muster „Nein, bitte nach Ihnen“ einen Endlos-Dialog führen: Der CPU-Verbrauch steigt, das System wird langsam. Deadlocks entstehen bei zyklischen Wartesituationen: Der CPU-Verbrauch ist Null, Teilsysteme oder das Gesamtsystem sind eingefroren. Alle drei Lock-Arten sind regelmäßig die Konsequenz naiver Thread-Programmierung. Leider ist es so, dass viele Programmierer ihre *synchronized*-Anweisungen nach Gefühl verteilen; die Konzepte für die Thread-Synchronisierung sind oft mangelhaft und gelegentlich einfach nicht vorhanden. Ein Standardbeispiel für fehlerhafte Thread-Programmierung ist *Double Checked Locking*[RJB+07].

6.4 I/O-Last

Oft ist Logging die Ursache für hohe I/O-Last: Die Log-Datei ist der Flaschenhals, wenn viele Komponenten gleichzeitig ihre Log-Informationen loswerden wollen, denn alle Schreibvorgänge erfolgen seriell. Andere Standardursachen sind ungeschickte Datenbankzugriffe: Joins über eine ungeschickt normalisierte Datenbank oder Lesen/Schreiben riesiger serialisierter Objektklumpen. Hohe I/O-Last ist in der Regel begleitet von einer untätigen CPU.

6.5 Ein gesundes System

Abbildung 4 zeigt die Antwortzeiten eines gesunden Systems währen der ersten neun Minuten nach dem Start. Jede Kurve beschreibt das Verhalten einer Systemkomponente. Nach der ersten Minute, in der noch keine Benutzer da sind, steigt die Last stark an: Komponenten werden gesucht, gefunden und geladen, Caches werden gefüllt, liefern zu Beginn aber nur wenige Treffer. Das System erreicht nach etwa fünf Minuten in einen stabilen Zustand. G. Luck [Luc10] spricht hier von *Elefantenkurven*.

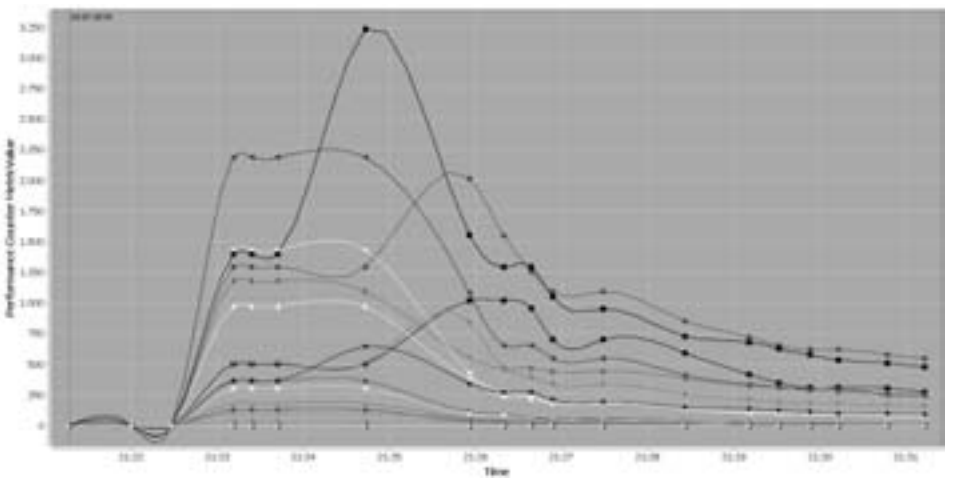


Abbildung 4: Elefantenkurven – Merkmal eines gesunden Systems

7 Vorbeugen ist besser als Heilen

Die genannten Fehler sind alles andere als originell und sollten sich eigentlich herumgesprochen haben. Es gibt im Wesentlichen drei Problemquellen und dementsprechend drei Maßnahmen:

1. Manche Fehler geschehen aus Nachlässigkeit, etwa ein vergessener *release*-Aufruf. Es ist Aufgabe der Qualitätssicherung, solche Fehler zu entdecken.

2. Manche Fehler sind das Ergebnis eines mangelhaften oder fehlenden Konzepts: Jedes System braucht eine klare, wohldokumentierte Thread-Architektur – wer seine Threads nach Gefühl startet, synchronisiert und beendet, wird zwangsläufig Schwierigkeiten haben.

3. Am schlimmsten sind die Architektur-Fehler: Manche Systeme sind einfach zu kompliziert, zu verflochten, haben so viele Abhängigkeiten, dass die Analyse zwar ein Problem nach dem anderen findet, dass aber die nachhaltige Fehlerbehebung mehr oder weniger den Neubau von Systemteilen oder des ganzen Systems bedeutet.

An dieser Stelle ergeht ein Appell an alle, die Software-Engineering unterrichten: Software-Engineering ist 80% Handwerk und 20% Kreativität. Eine solide handwerkliche Ausbildung ist der beste Weg, um Standardfehler zu vermeiden.

Das Software-EKG wurde für die Analyse von instabilen Systemen entwickelt. Es funktioniert aber genauso bei der Optimierung von Systemen in Bezug auf Performanz und Ressourcenverbrauch, sowie bei der schlichten Vermessung der technischen Architektur, um zu verstehen, was das eigene System überhaupt tut. Wir verwenden das Verfahren standardmäßig im Rahmen von Architektur-Reviews und benutzen es im Entwicklungsprozess zur laufenden Qualitätssicherung. Architektur-Reviews legen den Schwerpunkt meist auf die statische Analyse. Aber dynamische Analyse ist genauso wichtig: Auch in rund laufenden Systemen können verborgene Fehler stecken, die sich erst bei erhöhter Last oder geändertem Benutzerverhalten bemerkbar machen. Daher empfiehlt sich der Einsatz des Software-EKGs vor allem dann, wenn ein System neuen Belastungen ausgesetzt oder für neue Benutzergruppen geöffnet wird. Es eignet sich ebenfalls für die Vorbereitung von Optimierungsmaßnahmen. Der More-Power-Ansatz funktioniert nicht immer und ist teuer. Das Software-EKG kann Geld sparen, denn mit seiner Hilfe findet man Performance-Fehler und Hardware-Engpässe.

Der systematische Einsatz des Software-EKG unterstützt auch den Entwicklungsprozess: Die genannten Probleme (Leaks, Locks, Performance, I/O) lassen sich durch Architekturkonzepte, Programmierrichtlinien und statische Codeanalyse niemals vollständig beseitigen. Daher laufen in unserem Haus alle Regressionstests unter Kontrolle des Software-EKGs. So erkennen wir Anomalien mit hoher Wahrscheinlichkeit unmittelbar nach der Entstehung und können die Ursache rasch ermitteln: Wenn eine Anomalie heute zum ersten Mal auftritt, dann muss gestern jemand einen Fehler eingebaut haben. Das spart Geld und Nerven: Im Oktober 2010 ging das größte Online-Portal Deutschlands mit über 11 Millionen Besuchern pro Tag in einer runderneuerten Version in Betrieb. Wir hatten mit dem Software-EKG während der Entwicklung einige harte Performance-Fehler gefunden. In der Produktion lief das Portal von der ersten Sekunde an ohne Probleme, was zahlreiche Elefantenkurven dokumentieren.

8 Design for Diagnosibility (DfD): Das gläserne System

Auch das beste System kann verborgene Fehler enthalten und Anlass für eine Krise sein. Daher ist es sinnvoll, bereits beim Systementwurf Diagnosemöglichkeiten vorzusehen, ähnlich wie man Autos seit vielen Jahren mit Diagnosesteckern ausrüstet. Diagnostizierbarkeit (diagnosibility) ist eine oft schmerzlich vermisste nicht-funktionale Eigenschaft.

Wir unterscheiden *aktive* und *passive* Diagnostizierbarkeit. Passive Diagnostizierbarkeit heißt nur, dass die Anwendung für den Einsatz des Software-EKGs vorbereitet ist: Die EKG-Kollektoren sind installiert, die Sicherheitsfragen beim entfernten Zugriff auf Produktionssysteme sind geklärt. Passive Diagnostizierbarkeit kostet fast nichts und spart im Ernstfall Nerven. Sie ist aber zwangsläufig beschränkt auf die Informationen der verfügbaren Monitore. Anwendungsbezogene Kennzahlen stehen nicht zur Verfügung, denn die müssten von der Anwendung kommen; die Anwendung müsste *aktiv* werden.

Der einzige verbreitete Beitrag zur *aktiven* Diagnostizierbarkeit ist Logging. Wohl jede Anwendung produziert Log-Dateien in Abhängigkeit vom Log-Level. Aber die Erfahrung zeigt, dass Log-Dateien im Ernstfall nicht immer so nützlich sind wie erhofft: Ist der Log-Level niedrig (wie in der Produktion), enthält die Log-Datei meist zu wenige Informationen. Ein hoher Log-Level beeinflusst das Systemverhalten massiv und produziert eine Unmenge von Information, die sich der manuellen Analyse entzieht. Bereits hier greift das Software-EKG: Log-Dateien jeder Größe lassen sich im Rahmen des Software-EKG hervorragend auswerten, sofern ein paar einfache Konventionen beachtet werden (z.B. braucht jeder Log-Eintrag einen Zeitstempel). Aber damit ist das Logging-Dilemma nicht beseitigt: In der Produktion ist Logging nur eingeschränkt möglich, und daher oft wertlos. Aber es gibt eine Alternative zum Logging: eine *Black Box* wie im Flugzeug. Diese speichert zyklisch etwa 100 Flugdaten; damit sind die letzten Flugminuten dokumentiert. Genauso kann man jede Anwendung mit einer Black Box in Form eines Speicherbereichs ausrüsten, wo die wichtigsten Anwendungskennzahlen in einer festgelegten Frequenz (etwa 10 Mal pro Sekunde) zyklisch (etwa die letzte Minute) gespeichert sind. Zyklusdauer, Frequenz und zu speichernden Kennzahlen sind parametrierbar. Bei jeder vernünftigen Auslegung sind der benötigte Speicherplatz und die zusätzliche CPU-Belastung vernachlässigbar gering, der Implementierungsaufwand ebenfalls. Der Punkt ist: Mit einem geeigneten Werkzeug (JMX oder Nagios) kann man die Black Box in einer einstellbaren Frequenz auslesen und die gewonnenen Anwendungskennzahlen dem Software-EKG zuführen! So gelangt man auch in der Produktion an Informationen, die per Logging unerreichbar sind. Anwendungsspezifische Kennzahlen sind z.B. Laufzeit und Häufigkeit bestimmter Anwendungsfälle, Anzahl und Art aufgetretener Sonderfälle, Hit-Raten von Anwendungscaches. Die Übertragung dieser Daten in die Black Box ist kein Problem; vor allem bei nachträglichen Änderungen kann AOP nützlich sein.

DfD bedeutet, dass man sich schon beim Systementwurf klar macht, welche Daten man im Ernstfall sehen möchte, und mit Hilfe von Black Box, Logging und Software-EKG geeignete Diagnosemöglichkeiten schafft. DfD ist erstens eine nützliche Übung für den Software-Architekten. Zweitens macht es bei einem rund laufenden System viel Spaß, die im grünen Bereich liegenden Kennzahlen zu verfolgen – wie in einem Leitstand bei der Formel 1. Und der Ernstfall tritt hoffentlich nie ein.

Literatur

[BM04] Antonia Bertolino und Raffaela Mirandola. CB-SPE Tool: Putting component-based performance engineering into practice. In *Proceedings of the 7th International Symposium on Component-Based Software Engineering (CBSE 2004*, Seiten 233–248. Springer, 2004.

[Luc10] Greg Luck. Introducing the Elephant Curve. `http://gregluck.com/blog/archives/2010/10/introducing-the-elephant-curve`, 2010. abgerufen am 8.11.2010.

[MKBR10] Anne Martens, Heiko Koziolek, Steffen Becker und Ralf Reussner. Automatically improve software architecture models for performance, reliability, and cost using evolutionary algorithms. In *Proceedings of the first joint WOSP/SIPEW international conference on Performance engineering*, WOSP/SIPEW '10, Seiten 105–116, New York, NY, USA, 2010. ACM.

[RJB+07] David Bacon (IBM Research), Joshua Bloch (Javasoft), Jeff Bogda, Cliff Click (Hotspot JVM project) et al. The Double-Checked Locking is Broken. `http://www.cs.umd.edu/~pugh/java/memoryModel/DoubleCheckedLocking.html`, 2007. abgerufen am 8.11.2010.

[Sie04] Johannes Siedersleben, Hrsg. *Moderne Software-Architektur: Umsichtig planen, robust bauen mit Quasar*. dpunkt, Heidelberg, 2004.

[UW07] Antony Unwin und Graham Wills. Exploring Time Series Graphically. Statistical Computing and Graphics Newsletter. 2007.

[Whi08] Nicholas Whitehead. Run-time performance and availability monitoring for Java systems. `http://www.ibm.com/developerworks/library/j-rtm1`, 2008. abgerufen am 8.11.2010.

SE | 11 SOFTWARE ENGINEERING

Präsentationen des Industrietags

Messsystematik zur Steuerung der Produkt- und Prozessqualität in Systemintegrationsprojekten – ein Erfahrungsbericht

Ingo Elsen, Michael Schmalzbauer

T-Systems GEI GmbH
{ingo.elsen, michael.schmalzbauer}@t-systems.com

Der Erfolg eines Softwarenentwicklungsprojektes insbesondere eines Systemintegrationsprojektes wird mit der Erfüllung des „Teufelsdreiecks“, „In-Time“, „In-Budget“, „In-Quality“ gemessen. Hierzu ist die Kenntnis der Software- und Prozessqualität essenziell, um die Einhaltung der Qualitätskriterien festzustellen, aber auch, um eine Vorhersage hinsichtlich Termin- und Budgettreue zu treffen. Zu diesem Zweck wurde in der T-Systems Systems Integration ein System aus verschiedenen Key Performance Indikatoren entworfen und in der Organisation implementiert, das genau das leistet und die Kriterien für CMMI Level 3 erfüllt.

Messsystematik und Erfahrungen im Betrieb

Neben einem Satz von KPIs zur Messung von Produkt- und Prozessqualität ist ein weiteres Ziel, die Qualität der Daten mit denen die KPIs berechnet werden, zu überprüfen. Die Prozessqualität wird dabei im Wesentlichen auf Basis der Erhebung von Fehlern entlang des Softwareentwicklungsprozesses gemessen. Zusätzlich prüfen wir noch die Stabilität der Anforderungen und die Einhaltung der Software-Entwicklungsstandards der T-Systems. Bei der Produktqualität werden die Defektdichte sowie die Rate der vor der Inbetriebnahme einer Software behobenen Fehler in Relation zur Gesamtfehlerzahl erhoben. Hier hat sich gezeigt, dass es wichtig ist, die Fehler auf Plausibilität anhand der Zuordnung zu den SE-Phasen zu prüfen: In welcher Phase wurden wieviele Fehler gefunden und in welcher Phase ist der Fehler entstanden? So lassen sich gezielt Schwächen im SE-Prozess erkennen und Maßnahmen ableiten.

Zur weiteren Beurteilung des Prozesses haben wir zusätzliche Messungen der handwerklichen Softwarequalität eingeführt. Hier prüfen wir ebenfalls deren Signifikanz im Zusammenhang mit der Fehlerdichte mit dem Ziel, die Messsystematik kontinuierlich zu verbessern und an den Bedürfnissen der Organisation auszurichten.

Zukünftig sollen noch Ressourcenmetriken zum Tooleinsatz und der Produktivität eingeführt werden.

Der dritte Weg – von der ingenieurmäßigen Entwicklung flexibler Anwendungen

Konstantin Diener

COINOR AG
konstantin.diener@coinor.de

Anwendungssysteme bilden immer die Prozesse der jeweiligen Fachabteilungen eines Unternehmens ab. In den Anfangstagen der Entwicklung von Geschäftsanwendungen erfolgte die Abbildung dieser fachlichen Aspekte vornehmlich im Quellcode der Anwendung; im Projektalltag ist heute häufig die Rede von „hartkodierten“ Prozessen, Regeln oder Konfigurationen. Mit der Wiederverwendung von Anwendungskomponenten bildete sich später eine Aufteilung der Implementierung in einen Anwendungskern und eine passende Konfiguration heraus, mit der sich der Kern bei jeder Verwendung an das jeweilige fachliche Szenario anpassen ließ.

Man sollte meinen, dass die stetige Weiterentwicklung der Konfigurationstechnologie zu einer schnellen zufriedenstellenden Abbildung neuer fachlicher Anforderungen in den Anwendungssystemen geführt hat. Tatsächlich zeigt die tägliche Projektpraxis ein anderes Bild: Anpassungen an Anwendungssystemen sind aufgrund von Change-, Release- und Anforderungsprozessen meist schwergewichtig und langwierig – bei gleichzeitige steigendem Druck auf die Fachabteilungen, eine immer kürzere Time to Market zu erreichen. Neben der langen Dauer für die Umsetzung einer fachlichen Anforderung existiert meist auch eine Lücke zwischen den fachlichen Anforderungen und der technischen Umsetzung, so dass die ausgelieferten Anwendungssysteme nicht zwingend die Anforderungen der Fachabteilungen abbilden. Aus diesem Grund weicht man dort meist auf Lösungen auf Basis von Excel und Access aus.

Beide beschriebenen Probleme – die Verzögerung und die unzureichende Umsetzung – gehen auf den Übersetzungsprozess zurück, der der Anwendungsentwicklung zu Grunde liegt: Da selbst in konfigurierbaren Anwendungen die Konfiguration in den meisten Fällen technisch orientiert ist, müssen die fachlichen Anforderungen in eine technische Darstellung übersetzt werden. Dieser Prozess ist zweitaufwändig und selbst in den besten Fällen nicht verlustfrei.

Die vorgestellte Präsentation beschreibt eine Lösung, in der durch die Anwendungsentwicklung ein technischer Anwendungskern bereitgestellt wird, der sich durch komplexe fachliche Beschreibungen konfigurieren lässt. Für einen Großteil der Änderungen an der Software wird der Übersetzungsprozess so vorgezogen und standardisiert. Verwendung finden dazu Technologien wie Domain Specific Languages und Business Rules. Neben der Flexibilität für die Fachabteilung bietet der Lösungsansatz den für den Betrieb der IT-Systeme Verantwortlichen eine im Gegensatz zu Excel- und Access-Lösungen wartbare, versionierbare und kontrollierbare Software. Gleichzeitig entfällt die zusätzliche fachliche

Dokumentation des Systems, da die entsprechenden Informationen direkt in der Konfiguration hinterlegt sind. Die Vorstellung des Lösungsansatzes erfolgt anhand eines praktischen Beispiels aus dem Bereich der Verarbeitung von Wertpapierdaten.

Kurzbiographie

Konstantin Diener Konstantin Diener ist Senior Expert Consultant bei der COINOR AG und im Bereich der Finanzdatenversorgung für Kapitalanlagegesellschaften tätig, wo sein Aufgabenfeld von JEE- und OSGi-Anwendungen bis zu Fat-Client-Anwendungen reicht. Er beschäftigt sich seit zehn Jahren mit der Java- Plattform und sein aktuelles Interesse gilt neben regelbasierten Software-Architekturen mit Groovy, Drools & Co den agilen Methodiken wie Scrum oder Kanban. Konstantin Diener ist unter `konstantin.diener@coinor.de` erreichbar.

Systematic Quality Engineering – Lessons Learned

Markus Großmann, Frank Salger

CSD Research
Capgemini
Carl-Wery-Strasse 42
81739 München
{markus.grossmann;frank.salger}@capgemini.com

Abstract: Three main areas determine customer satisfaction in software development projects: Time, Budget and Quality. In contrast to 'time' and 'budget' where achievement can be determined quite effectively, judging 'quality' often remains difficult. In this work, we outline Quasar Analytics®, the quality assurance framework used at Capgemini Germany. We describe best practices and lessons learned of applying Quasar Analytics®.

1 Situation

Time, Budget and Quality are the main aspects that determine customer satisfaction in software development projects. But whereas being in time and budget can be determined quite effectively, determining quality often remains elusive. A systematic engineering-like approach to quality assurance of critical applications is missing:

- Fundamental conceptual decisions miss the original business objectives. The wrong system is built.
- Core artefacts like requirements specifications and software architectures are not rigorously assessed. 'Downstream' artefacts like code, test cases and documentation are affected by very expensive changes.
- Quality Assurance is just a synonym for 'testing' and not comprehensive enough, leaving quality attributes like 'maintainability' unconsidered. A cost effective maintenance is difficult to achieve.
- Quality assurance is missing continuity and only done when the project 'has the time to do it'. Technical debt slows down development and maintenance.

2 Goal

From these observations, requirements for a truly systematic quality assurance technique can be derived. A comprehensive 'quality engineering' approach would: a) Ensure that business objectives are systematically traced down into system development. b) Start early to check maturity of (conceptual) core work products like software requirements specifications and software architectures. c) Take comprehensively all aspects of quality into account, in particular hard ones like 'maintainability'. d) Ensure continuity of quality assurance over the whole development lifecycle and automate quality assurance to a high extend to achieve cost efficient system development.

3 Solution

From these requirements, we derived a systematic quality assurance approach, called 'Quasar Analytics®' [QA]. It is based on three pillars: 'Quality Gates', 'Software Measurement' and 'Testing'. At Software Engineering 2010, we gave an overview Quasar Analytics®. In our SE 2011 talk, we will share some details of the method. More importantly however, we will reports our lessons learned and insights from applying the Quality Gates and Software Measurement in various large-scale and business critical projects.

Quality Gates: Quality gates are systematic evaluations applied at specific points in time assessing the maturity and sustainability of produced artefacts as well as the processes followed to produce them. We pursue the following high-level objectives: Make the maturity of development artefacts and processes transparent, derive effective countermeasures for major problems encountered and 'standardize' audits to a reasonable extent. The main characteristics of applying quality gates are: They are applied early in the respective project phase. They are applied according to a defined quality gate process and end with a decision. They are no formal process checks but evaluate the content of artefacts. They are conducted by project-external experts.

Lessons learned (Quality Gates): In the talk, we will share lessons learned from applying the Quality Gates. For Example: a) Non-functional requirements pose more problems than functional ones. b) Using non-formal notations (e.g., pictures or examples) help to complement more formal software requirements specifications.

Though quality gates play an important role for the validation of conceptual results of a software project, they must be complemented as soon as the scene changes to implementation results: Here, 'Testing' and 'Measurement' become the main actors of systematic quality engineering. Testing focuses on assessment of the external quality attributes of a system (functionality, usability, etc.) whereas measurement has its focal point on the internal quality attributes (maintainability, testability, etc.).

Measurement: The idea of measurement in Quasar Analytics is to use a so called "Software Blood Count", a set of quality metrics that are continuously collected and managed by a Software Cockpit. The Software Blood Count is supplemented with a set of common quality indicators, which describe certain metric value patterns that allow the detection of typical quality problems. This approach is the result from our practical experience that on the one hand it is easier to verify the presence of quality deficits but not their absence, and, on the other hand, the impact estimations of quality indicators can't be automated as they need context-specific knowledge and software engineering experience. Hence, human experts perform the quality rating. Differences between Measurement and Quality Gates are: Quality gates just provide a snapshot assessment of the system whereas measurement provides a continuous quality feedback that is needed to recognize the insidious effects of software erosion. Quality gates just assess samples of the system whereas measurement provides automation features that allow the assessment of a whole code base.

Lessons learned (Measurement): The talk will share some lessons learned from the application of Measurement: a) The TOP 3 important questions and most used metrics with respect to internal software quality, showing the most relevant maintainability problems in the past. b) How to use metric thresholds in a right way and keep a relationship of trust in order to prevent measurement dysfunction: the team concentrates on optimizing metric values but the real goals are lost of sight. c) Weaknesses of measurement tools in large-scale project contexts

Quality gates and Measurement are less well suited to verify that the final system complies with the original requirements specification. This is where the third pillar, namely our ISTQB-based test method comes into play. Together, quality gates, measurement and systematic testing ensure comprehensive quality assurance, aligning business objectives and system development and cover the whole software development lifecycle. The overall benefits of Quasar Analytics® from a company point-of-view are a more standardized quality assurance by reusable tools and methods. Further, Quasar Analytics® can be seen as consolidated software engineering knowledge and improve quality assurance skills of employees ("I will do it differently the next time"). Last but not least, we can provide better services to the Customer as quality gets more visible! Our next steps comprise the development of a measurable quality standard for custom software development within the BMBF-funded research project QUAMOCO (Quality Modelling and Integrated Controlling) [QU]. The collection of best practices and processes for software measurement within a guideline and setting up a quantitative controlling process to supervise the quality of the quality gates is another future goal.

References

[QA] Homepage von Quasar Analytics®, Capgemini (accessed 10.01.2011) : http://www.de.capgemini.com/capgemini/forschung/research/analytics/

[QU] Homepage von QUAMOCO (accessed 10.01.2011) : http://www.quamoco.de

Social BPM: Software Engineering in agilen Business Communities

Frank Schönthaler

PROMATIS software GmbH, Ettlingen/Baden
hq@promatis.de

Veränderungsfähigkeit ist in vielen Unternehmen zum entscheidenden Erfolgsfaktor geworden. Mit veränderten Geschäftsmodellen und Strategien muss laufend auf neue Erfordernisse des Markts reagiert werden. Hierzu müssen die Veränderungen kurzfristig und zu geringen Kosten in den Geschäftsprozessen und Informationssystemen umgesetzt werden. Die IT-Abteilung ist dazu aus Kapazitäts- und Kostengründen, aber auch angesichts ihrer Verfahren und Werkzeuge oft nicht in der Lage. Die gewünschte Veränderungsfähigkeit kann also nur dann erreicht werden, wenn die Business Community sich „emanzipiert“ und eine aktive Rolle in der Gestaltung und Umsetzung der Geschäftsprozesse übernimmt. Dies ist die grundlegende Idee des Social Business Process Managements (Social BPM).

Mit Social BPM werden Verfahren und Technologien des Web 2.0 für BPM nutzbar gemacht. Wesentliche Voraussetzung für den erfolgreichen Einsatz von Social BPM ist eine agile Business Community. Das bedeutet zunächst, dass alle an der Durchführung des Geschäftsprozesses beteiligten Rollen – oder idealerweise sogar alle Individuen – in die Prozessgestaltung eingebunden sind. In der Community wirken nicht nur unternehmensinterne Personen mit, sondern ganz bewusst auch Geschäftspartner und externe Experten. Die Bereitschaft der Business Community, selbst aktiv Veränderung zu gestalten, qualifiziert sie als agile Business Community.

Social BPM stellt der Business Community ein Instrumentarium zur Verfügung, so dass diese in die Lage versetzt wird, neue und geänderte Geschäftsprozesse schnell und in weiten Teilen unabhängig von der IT-Abteilung umzusetzen. Daraus erwächst eine neue Rolle der IT, denn sie muss dem Business einen wohldefinierten Architekturrahmen vorgeben, der die Einhaltung unternehmensweiter Standards sicherstellt und einen zukünftigen informationstechnischen Ausbau der Lösung jederzeit möglich macht. Insofern bedeutet Social BPM keine Abkehr von bewährten Software Engineering- Verfahren, sondern ergänzt diese um eine neue Dimension der Partizipation des Software- Anwenders.

Der Beitrag behandelt Konzepte des Social BPM, aber auch die damit verbundenen soziologischen Aspekte in der Kollaboration von Business Community und IT. Wie praktische Anwendungsfälle zeigen, sind sie gleichermaßen Ursprung von nicht zu unterschätzenden Projektrisiken und innovativen Lösungsideen. Der Beitrag wird durch Vorstellung eines Social BPM-Piloten auf Basis von Horus[1] und der Oracle BPM Suite[2] abgerundet.

[1] Horus® ist ein Produkt der Horus software GmbH, Ettlingen, Deutschland.
[2] Oracle® BPM Suite ist ein Produkt der Oracle Corp., Redwood Shores, CA, USA.

Regelgestützte Maskenvalidierung und -steuerung

Valentino Pola, Jörg Ramser

COINOR AG
{valentino.pola, joerg.ramser}@coinor.de

Komplexe Anwendungen mit kundenspezifischen Prozessen zur Laufzeit mandantenbezogen konfigurieren zu können, ist eine Anforderung mit der sich IT-Anbieter zunehmend konfrontiert sehen (bspw. Whitelabeling).

Die Interpretation von Quellcode zur Laufzeit kann eine Lösung für diese Problemgattung sein. Dynamisch gebundene Sprachen wie Groovy bieten die Möglichkeit im Java-Umfeld Logik auszulagern und somit diese Teile konfigurierbar zu gestalten. Eine spezielle Art von Logik stellen die Navigations- und Validierungslogik dar, da sich diese durch die drei Teile einer regelbasierten Sprache beschreiben lassen.

Der Vortrag präsentiert einen Architekturvorschlag, wie in einer Webanwendung JBoss Drools genutzt werden kann, um konfigurierbare Navigations- und Validierungslogik aufzubauen. Eingegangen wird hierbei insbesondere auf die Validierung von Feldern, komplexen Formularen und die regelgestützte Entscheidung bei der Masken-Navigation. Ferner wird ein Ausblick gegeben, welche möglichkeiten der Einsatz eines regelgestützen Systems in Zukunft bringen kann.

Kurzbiographien

Valentino Pola Valentino Pola ist als Expert Consultant bei der COINOR AG im Bereich der Entwicklung von Anwendungssystemen zur Abbildung von Bankprozessen tätig. Seine Schwerpunkte liegen bei dabei auf der flexiblen Skalierbarkeit und dem anwendernahen Design. Technologisch beschäftigt er sich mit Technologien von JBoss Seam über JSF bis hinzu Verarbeitung von Massendaten mit ETL-Werkzeugen. Valentino Pola ist unter `valentino.pola@coinor.de` erreichbar.

Jörg Ramser Jörg Ramser ist Senior Expert Consultant bei der COINOR AG und betreut sowohl Banken als auch Kapitalanlagegesellschaften als Kunden. Seine technologischen Schwerpunkte reichen vom Aufbau ereignisgesteuerter Systeme zur Verarbeitung von Finanzdaten bis hin zur Konzeption und Umsetzung service-orientierter Integrationslösungen. Jörg Ramser ist unter `joerg.ramser@coinor.de` erreichbar.

Selektionswerkzeug zur Auswahl projektspezifischer Vorgehensstrategien

Marianne Heinemann[1], Markus Palme[1], Andreas Rothmann[1], Frank Salger[1], Jürgen Schönke[1], Gregor Engels[1,2]

[1] Capgemini, Carl-Wery-Straße 42, 81739 München
marianne.heinemann@capgemini.com
markus.palme@capgemini.com
andreas.rothmann@capgemini.com
frank.salger@capgemini.com
jürgen.schönke@capgemini.com

[2] Universität Paderborn, Institut für Informatik, Warburger Str. 100, 33098 Paderborn
engels@uni-paderborn.de

Abstract: Agile Vorgehensmodelle haben in den letzten Jahren eine starke Verbreitung erfahren. Jedoch ist nicht jedes Projekt gleichermaßen für ein agiles Vorgehen geeignet. Die spezifischen Projektanforderungen und -eigenschaften sowie das Umfeld haben ganz entscheidenden Einfluss darauf, ob ein agiles, ein iteratives oder ein klassisches Vorgehensmodell mehr Vorteile aufweist. Diese Einsicht führte bei Capgemini zur Entwicklung einer Selektionsmethode, die für ein spezifisches Individualsoftwareentwicklungsprojekt systematisch die Auswahl der passenden Vorgehensstrategie – agil, iterativ oder klassisch – unterstützt und damit zu Projektbeginn wichtige Weichen für den Projekterfolg stellt. Der Nutzen der Selektionsmethode wurde inzwischen in zahlreichen Projekten bestätigt. Zur vereinfachten Anwendung haben wir sie nun mit einem Selektionswerkzeug ergänzt, welches das erste uns bekannte Werkzeug zur projektspezifischen Auswahl einer Vorgehensstrategie darstellt. Wir möchten es daher einem breiten Nutzerkreis auch außerhalb von Capgemini präsentieren und zur Verfügung stellen.

1 Einleitung

Agiles Vorgehen [Be01] wird in den letzten Jahren verstärkt nachgefragt, um ein Softwareprodukt enger, flexibler und sichtbarer am tatsächlichen Kundenbedarf auszurichten sowie schneller an den Markt bringen zu können [C01]. Es gibt aber auch Stimmen, die einzelne Vorteile differenziert betrachten, vom Projektkontext abhängig machen oder für ein Austarieren der Agilität plädieren [BT05, L09, OW08]. Bei Capgemini führte diese Diskussion zu der Erkenntnis, dass jede Vorgehensstrategie (agil, iterativ, klassisch) Eigenschaften besitzt, die sich je nach Projektkontext als Vor- oder Nachteile auswirken können. Deshalb ist es wichtig, für das konkrete Projekt diejenige Vorgehensstrategie zu identifizieren, die eben in diesem spezifischen Kontext die meisten Vorteile aufweist.

2 Zusammenfassung des methodischen Ansatzes

Die Selektionsmethode [HE10] unterstützt die Auswahl der Vorgehensstrategie (agil, iterativ oder klassisch) für ein Individualsoftwareentwicklungsprojekt. Die Entscheidung wird nicht auf Basis einzelner Vorgehensmodelle, sondern vielmehr auf Basis der Essenz dieser Vorgehensmodelle, d.h. der dahinter liegenden Vorgehensstrategien getroffen. Mit welchem konkreten Vorgehensmodell (z.B. Scrum [S04] für agil, RUP [K03] für iterativ und Stufenmodell [HDE10] für klassisch) die gewählte Vorgehensstrategie dann implementiert wird, hängt häufig von den vorhandenen Erfahrungen ab [WB10].

Die Herleitung der Selektionsmethode fußt auf den Kernelementen der drei Vorgehensstrategien klassisch, iterativ und agil [HE10]. Wir leiteten Projekteigenschaften ab, welche diese Kernelemente zu Vorteilen und nicht zu Nachteilen werden lassen. Die Summe dieser relevanten Projekteigenschaften sind die idealen Projekttypen für jede der drei Vorgehensstrategien. Die Selektionsmethode gleicht die Charakteristika eines spezifischen Projektes mit den idealen Projekttypen ab und bestimmt aufgrund der größten Übereinstimmung die am besten geeignete Vorgehensstrategie.

Sechs Projektattribute sind für alle drei Vorgehensstrategien als relevant identifiziert worden (Projektdimensionen im Polargraphen Abb. 1), allerdings belegt mit verschiedenen Werten (z. B. das Projektattribut „erwartete Kundenbeteiligung“ mit Werten zwischen „intensiv“ bis „weniger intensiv“). Die Selektionsmethode berücksichtigt sowohl diese sechs allseits relevanten Projektattribute als auch weitere spezifische Projekteigenschaften, die nur für jeweils eine Vorgehensstrategie größere Bedeutung besitzen. Das geschieht in zwei aufeinander folgenden Prozessschritten. Zuerst erfolgt der Grobabgleich des konkreten Projektes gegen die sechs gemeinsamen Projektattribute, anschließend werden die zusätzlichen Eigenschaften für die noch in Frage kommenden Vorgehensstrategien überprüft.

Die Selektionsmethode unterscheidet zwischen obligatorischen und optionalen Projekteigenschaften. Erstere müssen für den Erfolg einer Vorgehensstrategie zutreffen und dienen daher methodisch als Ausschlusskriterien. Letztere sind zwar förderlich, aber nicht unbedingt erforderlich. Jede Abweichung des konkreten Projektes von einer optionalen idealen Projekteigenschaft einer Vorgehensstrategie führt zu einem Projektrisiko, das in der Selektionsmethode identifiziert und mit einer Bewältigungsmaßnahme beplant wird. Kommen nach Anwendung des Ausschlussverfahren noch mehr als eine Vorgehensstrategie in Frage, kann der Benutzer die endgültige Entscheidung somit anhand einer Risikominimierung treffen.

3 Selektionswerkzeug

Das Selektionswerkzeug setzt den beschriebenen methodischen Ansatz um. Anwender sind typischerweise die Entscheidungsträger zu Projektbeginn, wie Accountmanager, Projektmanager oder –leiter, funktionale und technische Architekten. Die Eingaben können zusammen mit Kundenvertretern durchgeführt werden. Alternativ können die Ergebnisse mit dem Kunden diskutiert und Eingabewerte bei Bedarf modifiziert werden.

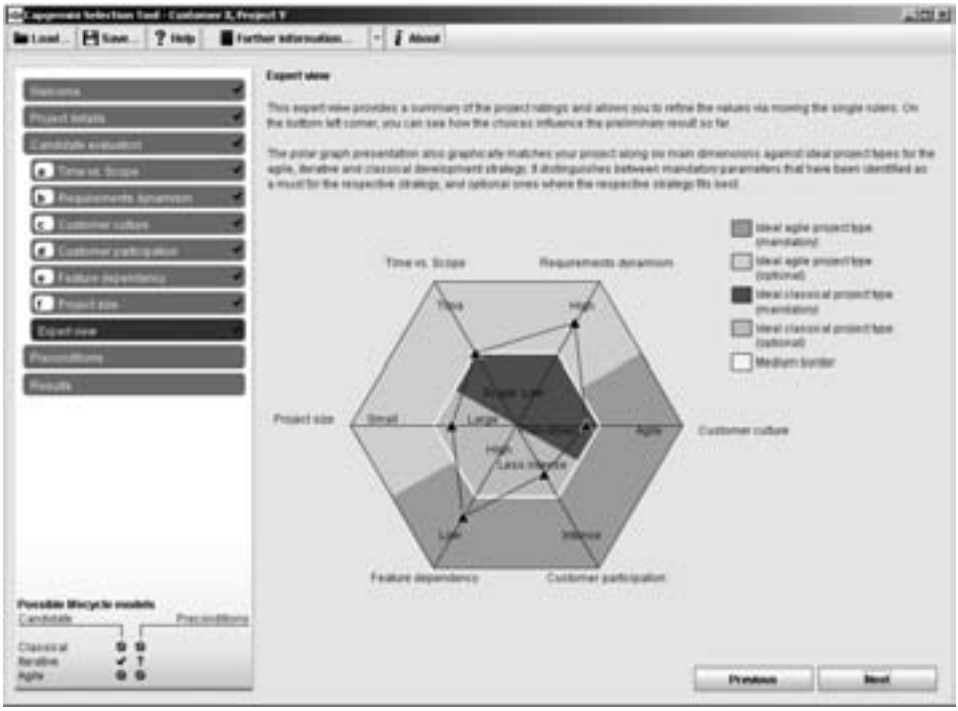


Abbildung 1: Expertenansicht des Selektionswerkzeuges

Der erstmalige Anwender wird schrittweise durch die Screens geführt und erhält für jede vorzunehmende Projektbewertung detaillierte Hilfestellungen. Der erfahrene Anwender kann für den Grobabgleich direkt auf die Expertenansicht (Abb. 1) springen und dort das Projekt entlang der sechs Achsen im Polargraphen bewerten. Im Polargraphen sind die idealen Projekttypen für die klassische und die agile Vorgehensstrategie diametral angeordnet - klassisch innen, agil außen. Der ideale iterative Projekttyp befindet sich dazwischen, lässt sich aber weniger scharf eingrenzen und überlappt mit den anderen beiden. Anhand der Polargraphen-Darstellung erhält der Anwender bereits einen ersten visuellen Eindruck von der Einordnung seines Projektes. Im folgenden Screen „Preconditions“ werden abhängig von den noch möglichen Vorgehensstrategien zusätzliche Projekteigenschaften abgefragt. Das Ergebnis besteht aus der Empfehlung für die Vorgehensstrategie sowie der zugehörigen Risikoliste. Sämtliche Eingaben und Ergebnisse können gespeichert, nach Excel exportiert und gedruckt werden.

4 Erfahrungen und nächste Schritte

Die seit Mai 2009 bei Capgemini eingesetzte ursprüngliche Selektionsmethode wurde von den Anwendern zwar als sehr nutzbringend, aber ebenso als schwierig benutzbar angesehen. Das Selektionswerkzeug sollte die Benutzung vereinfachen. Daher wurde bei der Entwicklung des Selektionswerkzeuges sehr viel Wert auf die Beratung durch einen Usability-Experten und die Durchführung mehrerer Usability-Tests gelegt.

Die Auswertung der Usability-Tests sowie erste Erfahrungen mit realen Nutzern zeigen, dass auch neue Anwender mit Hilfe des Selektionswerkzeuges ohne persönliche Unterstützung die passende Vorgehensstrategie auswählen können. Für die Testpersonen „…macht das Tool einen sehr guten Eindruck." „Ich habe es schon mal kurz für mein bisheriges Projekt ausgeführt, und es kommt Iteratives Vorgehen heraus. Das ist tatsächlich auch das Vorgehen, was wir schon seit 4 Jahren erfolgreich anwenden." Die Testpersonen bemerkten positiv, dass für die Beantwortung einer doch recht komplexen Fragestellung relativ wenig Aufwand anfiel. Ebenso positiv wurde die „gute grafische Darstellung" in der Expertenansicht hervorgehoben, „an der man durch Verschiebung der Bewertungspunkte ein gutes Gefühl für die Auswirkungen auf das Ergebnis (links unten) bekommt". Mit einem quantitativen Fragebogen, der sich an der ISO Norm 9241 Teil 10 orientiert, haben wir die wahrgenommene Qualität des Selektionswerkzeugs gemessen. Die Onlineumfrage haben 30 Teilnehmer komplett abgeschlossen und dabei durchgängig positives Feedback abgegeben: In jeder der sieben Kategorien wurde eine durchschnittliche Wertung erzielt, die deutlich über den Werten bekannter Standardsoftware (bekannt aus Normtabellen) liegt.

Das hier vorgestellte Selektionswerkzeug weist noch methodische Grenzen auf, die wir im Zuge der Weiterentwicklung angehen wollen: Das Werkzeug empfiehlt für ein konkretes Projekt eine (oder zwei) Vorgehensstrategie(n) und zeigt ebenso auf, mit welchen Eigenschaften das konkrete Projekt von den Idealbedingungen für die empfohlene Vorgehensstrategie abweicht. Der erfahrene Anwender ist mit dieser Information in der Lage, die gewählte Vorgehensstrategie anzupassen, indem er beispielsweise Elemente aus einer der beiden anderen Vorgehensstrategien integriert. Für den weniger erfahrenen Anwender soll in Zukunft auch der Schritt der Anpassung der einmal ausgewählten Vorgehensstrategie vom Werkzeug übernommen werden.

Literaturverzeichnis

[Be01] Beck, K. et. al.: Manifesto for agile software development. http://agilemanifesto.org/, 2001.

[BT05] Boehm, B.; Turner, R.: Balancing Agility and Discipline. A Guide for the Perplexed. Addison-Wesley, 2005.

[C01] Coldewey, J.: Beständig ist nur der Wandel – Agile Entwicklung und Änderbarkeit. In: OBJEKTspektrum 06/2001, S. 77-82.

[HDE10] Heinemann, M.; Duwe, B.; Engels, G.: Enriching RUP with key success factors for large-scale custom software development projects. In: Software & Systems Engineering Essentials (SEE) 2010, Köln, 2010, S. 27-46.

[HE10] Heinemann, M.; Engels, G.: Auswahl projektspezifischer Vorgehensstrategien. In Linssen, O.; Greb, T.; Kuhrmann, M.; Lange, D.; Höhn, R. (Hrsg.): Integration von Vorgehensmodellen und Projektmanagement. Shaker, 2010, S. 132-142.

[K03] Kruchten, P.: The Rational Unified Process: An Introduction. Addison-Wesley, 2003.

[L09] Linssen, O.: Agile Vorgehensmodelle aus betriebswirtschaftlicher Sicht. In Höhn, R.; Linssen, O. (Hrsg.): Vorgehensmodelle und Implementierungsfragen. Shaker, 2009.

[OW08] Oestereich, B.; Weiss, C.: APM - Agiles Projektmanagement. Dpunkt, 2008.

[S04] Schwaber, K.: Agile Project Management with Scrum. Microsoft Press Corp., 2004.

Forschungsarbeiten

PQI – Ein Ansatz zur prozess- und projektorientierten Qualitätsintegration

Timea Illes-Seifert, Frank Wiebelt

EnBW Systeme Infrastruktur Support GmbH
Business Solutions - Business Consulting & Partner Management
Durlacher Allee 93
76131 Karlsruhe

t.illes-seifert@enbw.com
f.wiebelt@enbw.com

Abstract: In vielen großen Unternehmen ist eine heterogene Projektlandschaft vorzufinden, die unterschiedliche Vorgehensmodelle für die Softwareentwicklung erfordert. Gleichzeitig stellt sich die Notwendigkeit einer vergleichbaren Erhebung von Prozess- und Produktkennzahlen. In diesem Beitrag wird ein qualitätsmodellorientierter Ansatz vorgestellt, welcher einerseits ein einheitliches Qualitätsmodell für alle Projekte zugrunde legt und andererseits dessen Verankerung in das Unternehmen durch eine mehrstufige, abstrakte Beschreibung von Prozessen vorsieht. Vorteil des Ansatzes ist, dass einerseits jedes Projekt entsprechend seines Kontextes ein geeignetes Vorgehen aus dem abstrakten Prozessmodell nach vordefinierten Regeln instanziieren kann. Durch die Verankerung des Qualitätsmodells in die abstrakten Prozessbeschreibungen andererseits, wird die Möglichkeit einer vergleichbaren Erhebung von Kennzahlen geschaffen.

1 Einleitung und Ausgangssituation

Der Qualitätsanspruch an die zu entwickelnde oder bereits entwickelte Software im Unernehmen EnBW ist sehr hoch. Jedoch müssen heterogene Geschäftsfelder mit ebenso heterogenen Qualitätsansprüchen bedient werden. Um es an 2 Beispielen kurz aufzuzeigen, gibt es auf der einen Seite hoch komplexe Schnittstellen-Entwicklungen, beispielsweise zur Kraftwerkssteuerung, die ohne menschliche Interaktion, hoch performant, aber auch sehr hohen Sicherheitskriterien gerecht werden müssen. Auf der anderen Seite gibt es Entwicklungen für den Endkundenbereich, um Zählerstände über das Internet zurück zu melden. Hier spielen die Kriterien wie Performanz und Sicherheit im Vergleich zur Usability eine geringere Rolle. Der Endkunde muss sich im letzteren Fall schnell auf der grafischen Benutzeroberfläche zurecht finden, vom System gut geführt werden und passendes Feedback bekommen.

Bedingt durch diese heterogene Projektlandschaft, ergibt sich die Herausforderung, ein Qualitätsmodell zu operationalisieren, das zum einen den unterschiedlichsten Projektspezifika gerecht wird, es aber zum andern auch schafft, vergleichbare qualitätsbezogene Kennzahlen, sogenannte Key Performance Indicators (KPIs) zu erheben. Ähnlich zu den heterogenen Qualitätsansprüchen, gilt es auch die unterschiedlichen Vorgehensmodelle zur Softwareentwicklung bei der Definition und Verankerung des Qualitätsmodells zu berücksichtigen. Auch hier reichen die Bandbreiten von hoch-inkrementellen Vorgehensweisen bei Projekten mit unklaren Anforderungen, bis hin zum klassischen Wasserfallmodell, bei risikoarmen, kleinen Projekten.

Um den beiden vorgestellten Anforderungen gerecht zu werden, wird im Unternehmen EnBW Systeme Infrastruktur Support GmbH ein Ansatz zur „Prozess und projektorientierten Qualitätsintegration“ (PQI) verfolgt. Der Grundgedanke dieses Ansatzes besteht in der Definition eines auf der ISO/IEC 9126 [ISO/IEC01] basierenden Qualitätsmodells. Dieses Modell wird in der Projektlandschaft derart verankert, indem jedes Projekt die definierten Qualitätskriterien in Form von projektspezifischen Qualitätszielen instanziiert und an deren Erreichungsgrad gemessen wird.

Was bedeutet aber „projektorientiert“ in einem Entwicklungsumfeld, in dem von agilen bis hin zu strikten Wasserfall-Methoden, die unterschiedlichsten Vorgehensmodelle vorkommen?

Adenin, Cytosin, Guanin, Tyhmin bilden die Grundbausteine einer jeder menschlichen DNA. Wenn die Metapher auf die Softwareentwicklung übertragen wird, stellt sich die Frage nach den Grundbausteinen von Vorgehensmodellen zur Software- und Systementwicklung. In vielen großen Unternehmen kann aufgrund der hohen Heterogenität der durchgeführten Projekte nicht nur ein einziges Vorgehensmodell definiert bzw. eingeführt werden, vielmehr sind Koexistenzen mehrerer Problemlösungsstrategien vorhanden. Die Problematik dieser Konstellation besteht in der fehlenden Möglichkeit zur vergleichbaren Erhebung von KPI zur Steuerung der Prozess- und Produktqualität. Das ist auch die Situation bei der EnBW Systeme Infrastruktur Support GmbH. Letztlich muss eine Integration des gewählten Qualitätsmodells in eine firmenweit gültige Prozessbeschreibung geschaffen werden. Um dieser Herausforderung zu begegnen, integriert der PQI-Ansatz die Definition eines Prozessframeworks, welches unterschiedliche Abstraktionsebenen vorsieht. Hierdurch wird es möglich, auf einer abstrakten Ebene, eine allgemeine Repräsentation für alle Prozesse zu ermöglichen (diese Repräsentanten bilden die Grundbausteine aller Prozesse) und das Qualitätsmodell zu integrieren. Auf unteren Abstraktionsebenen werden konkrete Ausprägungen in Form von Vorgehensmodellen modelliert, die aber alle Eigenschaften des abstrakten Modells aufweisen.

Ziel des PQI-Ansatzes ist es, allen Mitarbeitern, die an IT-Projekten beteiligt sind, genügend Unterstützung zur Erstellung qualitativ hochwertiger Softwareprodukte zu bieten, deren Qualität sich über heterogene Projekte hinweg erheben und steuern lässt.

Zusammenfassend lassen sich folgende Anforderungen an den PQI-Ansatz definieren:

A1: Berücksichtigung der Heterogenität der zu entwickelnden Softwareprodukte
Das heterogene Produktportfolio setzt unterschiedliche Qualitätsanforderungen für die jeweiligen Produkte voraus. Demnach muss ein für den jeweiligen Projektkontext anpassbares Qualitätsmodell definiert werden. Dennoch muss ebenfalls eine unternehmenseinheitliche Mindestqualität definiert sein.

A2: Berücksichtigung der Heterogenität der Entwicklungsprozesse
Unterschiedliche Produkte setzen unterschiedliche Vorgehensweisen bei der Softwareerstellung voraus, die ebenfalls berücksichtigt werden müssen. Die Definition eines vereinheitlichten Vorgehensmodells ist somit nicht zielführend, da viel zu unterschiedliche Vorgehensweisen vorherrschen und sinnvoll sind.

A3: Vergleichbarkeit der KPI
Trotz unterschiedlicher Softwareprodukte und Vorgehensmodelle stellt sich dennoch die Anforderung der Definition vergleichbarer Kennzahlen für Softwareprozesse und -produkte.

2 Begriffe

Zum besseren Verständnis werden nachfolgend in dieser Ausarbeitung verwendete Begrifflichkeiten definiert.

Das **Qualitätsmodell** dient der Definition der Softwarequalität, als Qualitätsvorgabe und zur Qualitätsbewertung. Das in der EnBW entwickelte Qualitätsmodell basiert auf der ISO/IEC 9126 [ISO/IEC01] und definiert Unterbegriffe (Qualitätsmerkmale), die die unterschiedlichen Aspekte der Softwarequalität angeben. Dabei sind die Qualitätsmerkmale hierarchisch angeordnet. Die Qualitätsmerkmale der obersten Stufe werden als Faktoren bezeichnet und werden immer weiter verfeinert. Die untergeordneten Qualitätsmerkmale heißen Qualitätskriterien bzw. Qualitätsindikatoren. Qualitätskriterien werden auf der „untersten Stufe“ über Metriken operationalisiert.

Ein Prozessmodell bezeichnet im Kontext der vorliegenden Arbeit eine Verkettung von Phasen, die in allen gängigen Vorgehensmodellen in unterschiedlichen Ausprägungen auffindbar sind. Eine jede dieser Phasen wird mit einem definierten Quality-Gate abgeschlossen.

Phasen dienen in der vorliegenden Arbeit als ein Sammelbegriff zur Gruppierung von Aktivitäten, die in allen Vorgehensmodellen in unterschiedlicher Ausprägung aufzufinden sind.

Ein Vorgehensmodell bezeichnet die nach „Lehrbuch“ beschriebenen Softwareentwicklungsprozesse, wie beispielsweise das Wasserfallmodell, Rational Unified Process oder auch ein V-Modell bzw. Scrum.

Engineering Pattern dient als Oberbegriff für unterschiedliche Patterns, die in der Software Entwicklung zum Einsatz kommen. Sie beinhalten Patterns, die den Bereichen Design Pattern, Test Pattern, Usability Pattern, Interaction Design Pattern, etc. zugeordnet sind. Dabei behält die Definition, dass ein Pattern ein Vorgehen für bekannte Probleme ist, weiterhin ihre Gültigkeit [AI77].

3 PQI – Prozess- und projektorientierte Qualitätsintegration

3.1 Grundlegender Ansatz

Die Basis für die Integration eines Qualitätsmodells in einem heterogenen Projektumfeld mit jeweils unterschiedlichen Qualitätsansprüchen bildet die Einbettung des Qualitätsmodells in die Entwicklungsprozesse. Dies geschieht dadurch, dass Entwicklungsprozesse auf unterschiedlichen Abstraktionsebenen, dem Prozessframework, beschrieben werden. Die Verankerung des Qualitätsmodells erfolgt auf der obersten Abstraktionsebene, wodurch jedes abgeleitete Vorgehensmodell das Qualitätsmodell erbt.

Wie bereits in der Einleitung angesprochen, besteht die DNA eines jeden Menschen aus gleichen Grundbausteinen. Alle gängigen Vorgehensmodelle bestehen aus einer Ideen-, Anforderungs-, Design-, Test- und Implementierungsphase. Weiterhin müssen Projektmanagementtätigkeiten durchgeführt werden. Diese Prämisse wurde verwendet, um ein allgemeingültiges Prozessframework aufzubauen, wodurch vergleichbare Prozesskennzahlen erhoben werden können. Die Vergleichbarkeit produktorientierter Kennzahlen wird durch die Integration des Qualitätsmodells in die abstrakten Beschreibungen von Prozessmodellen gewährleistet.

Die PQI-Prozessarchitektur besteht aus dem Prozessframework, dem Qualitätsmodell, sowie einem Trigger, welcher die ablauforientierte Prozessbeschreibung und die statische Qualitätssicht operativ verbindet. In den nachfolgenden Kapiteln wird die Prozessarchitektur vorgestellt. Im Abschnitt 3.2 wird das Prozessframework detailliert erklärt, während im Abschnitt 3.3 das Qualitätsmodell erläutert wird. Im Abschnitt 3.4 wird kurz auf den Trigger eingegangen.

3.2 Das PQI Prozessframework

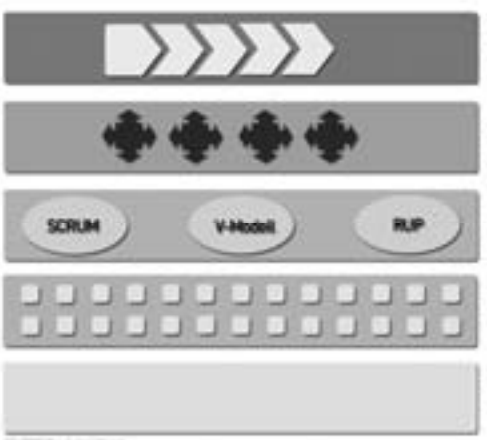
Grundsätzlich besteht das Prozessframework aus unterschiedlichen Ebenen, die sich jeweils an unterschiedliche Stakeholder richten.

Die **Prozessmetamodell-Ebene** enthält das Prozess-Metamodell (M2) und bildet somit eine abstrakte Beschreibung für alle abgeleiteten Vorgehensmodelle. Auf Prozessmetamodell-Ebene werden das Qualitätsmodell sowie Quality Gates integriert. Somit ermöglicht diese Ebene die Erhebung von KPIs, die für das Management als Stakeholder des PQI relevant sind.

Die **Vorgehensmodell-Ebene** beinhaltet Vorgehensmodelle für die Softwareentwicklung (M1) wie beispielsweise Scrum oder V-Modell und richtet sich vor allem an Projektbeteiligte. Diese Ebene beinhaltet eine Entscheidungshilfe für Projektleiter zur Auswahl des geeigneten Vorgehensmodells bezogen auf ihren Projektkontext. Der Kontext ergibt sich aus der Projektgröße, dessen Kritikalität, sowie der Teamzusammensetzung. Je nach Ausprägung dieser Faktoren, wird ein Vorgehensmodell auf Basis einer Entscheidungslogik vorgeschlagen. Abweichungen vom vorgeschlagenen Vorgehensmodell müssen vom Projektleiter begründet und dokumentiert werden. Weiterhin nutzt sie eine Sammlung von Engineering Patterns, um die Vorgehensmodelle in den unterschiedlichen Entwicklungsphasen methodisch anzureichern. Die Engineering Patterns bieten Hilfestellung zur Erreichung angestrebter Qualitätsziele und werden unter anderem durch die Attribute Name, Kontext, Vorbedingung, Problembeschreibung, Problemlösung, Nachbedingung beschrieben.

Die **Mapping-Ebene** dient vor allem der Abbildung unterschiedlicher Vorgehensmodelle aufeinander. Insbesondere dient sie der „Explizit-Machung" der Vererbungshierarchie („instance of" Beziehungen) und damit äquivalenter Entitäten in den unterschiedlichen Vorgehensmodellen. Diese Ebene enthält redundante Informationen die in dieser Form für den Prozessarchitekten aufbereitet wird, um eine schnelle Vergleichbarkeit der Vorgehensmodelle zu gewährleisten.

Ein **Projekt** konkretisiert ein spezifisches Vorgehensmodell und instanziiert das Qualitätsmodell. Abbildung 1a) veranschaulicht die grundsätzlichen Abstraktionsebenen des Prozessframeworks, Abbildung 1b) zeigt die jeweiligen Modelle auf den jeweiligen Abstraktionsebenen.



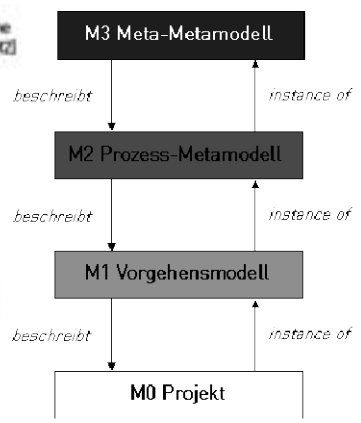


Abbildung 1a - vereinfachte Sich auf Prozessframework Abbildung 1b - Prozessframework Ebenen Ansicht

M3 - Meta-Metamodell Das Meta-Metamodell bildet die Grundlage zur Beschreibung von Prozessen beliebiger Art in einer Organisation, d.h. es stellt die Basis für die Beschreibung beliebiger Workflows dar. Generell betrachtet ordnen Workflows Phasen, in denen Methoden angewendet werden. Grundprämisse einer jeden Aktivität ist, dass

sie zu einem Ergebnis führt. Abbildung 2 zeigt das Meta-Metamodell M3 des Prozessframeworks.

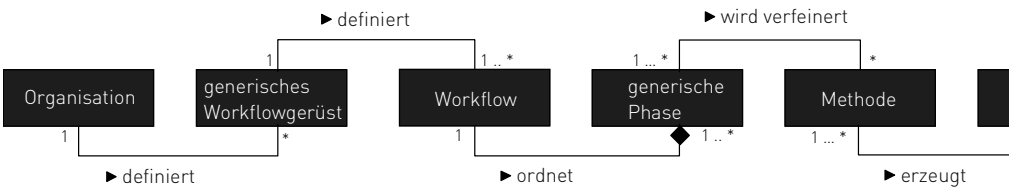


Abbildung 2 - M3 Meta-Metamodell von Workflows

M2 - Prozess-Metamodell Das Prozess-Metamodell definiert die „Grammatik“ zur Beschreibung von Vorgehensmodellen (Abbildung 3). Generell können Vorgehensmodelle iterativ oder sequenziell sein. In beiden Fällen setzen sich Iterationen und Sequenzen aus folgenden Phasen zusammen: Idee, Analyse, Design, Implementierung, Test, Projektmanagement, die je nach Vorgehensmodell spezifisch ausgeprägt sind. In den unterschiedlichen Phasen der Softwareentwicklung werden Methoden angewendet, um Arbeitsergebnisse zu erzeugen. Die Idee dieses Ansatzes ist es, nicht die Phasen über mehrere Aktivitäten zu verfeinern, sondern einen Baukasten an Engineering-Patterns zur Verfügung zu stellen, die in unterschiedlichen Entwicklungsphasen angewendet werden, um Arbeitsergebnisse zu erzeugen.

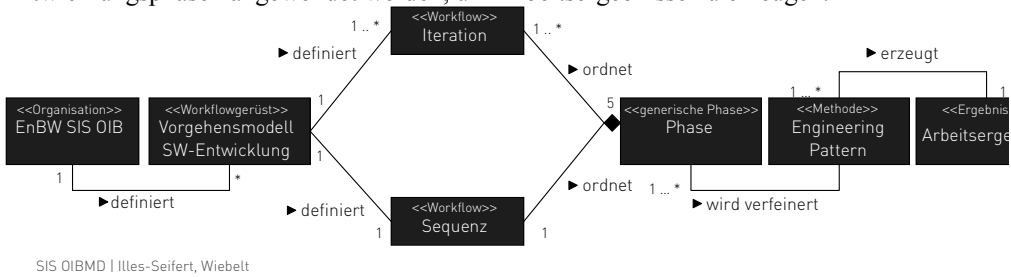


Abbildung 3 - M2 Prozess-Metamodell

M1 - Vorgehensmodelle (Wasserfall) Auf dieser Ebene werden unterschiedliche Vorgehensmodelle definiert. In einem ersten Schritt wurden Wasserfall und Scrum instanziiert. Das Wasserfall-Modell (Abbildung 4) sieht die sequenzielle Durchführung der Entwicklungsphasen vor. Am Beispiel der Phase „Test“ soll verdeutlicht werden, wie Phasen, Engineering-Patterns und Arbeitsergebnisse zusammenhängen. Beispielsweise wird als Arbeitsergebnis das Testkonzept erstellt. Hierfür muss eine Risikoanalyse sowie eine Analyse bisheriger Fehler durchgeführt werden. Zur Erstellung der Testspezifikation können unterschiedliche Testpatterns angewendet werden.

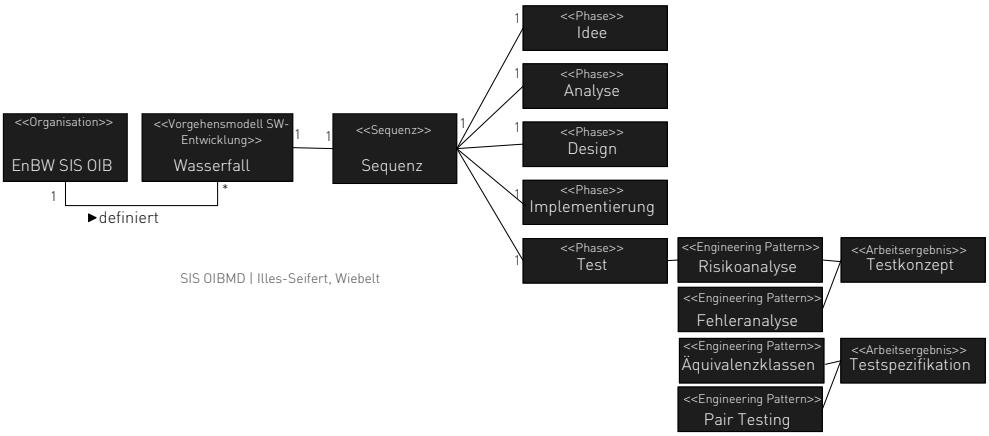


Abbildung 4 - M1 Wasserfall-Modell

M1 - Vorgehensmodelle (Scrum) Abbildung 5 stellt die Instanziierung von Scrum schematisch dar und beschränkt sich auf die in dem Vorgehensmodell enthaltenen Konzepte. Die Instanziierung zeigt die in der ersten Iteration stattfindende Festlegung der Vision sowie die Initialdefinition des „Product Backlog“. Die nächsten Iterationen, die in Scrum als „Sprint“ bezeichnet werden, enthalten jeweils ein „Sprint Planning Meeting“ in dem die Ziele, sowie die umzusetzenden Elemente des Product Backlogs definiert werden. Arbeitsergebnisse sind die „Sprint Goals“ sowie die im aktuellen Sprint umzusetzenden Features, die im „Sprint Backlog“ aufgelistet sind. In einem Sprint werden die Entwicklungsphasen überlappend durchgeführt. Für die Ausgestaltung innerhalb dieser Phasen macht Scrum keine Vorgaben. Der PQI-Ansatz sieht ein Baukasten-System zur methodischen Ausgestaltung dieser Phasen vor, indem je nach Qualitätszielen, entsprechende Engineering-Patterns vorgeschlagen werden. Wie die Instanziierung eines Vorgehensmodells in einem Projektkontext aussieht, wird im nachfolgenden Abschnitt beschrieben

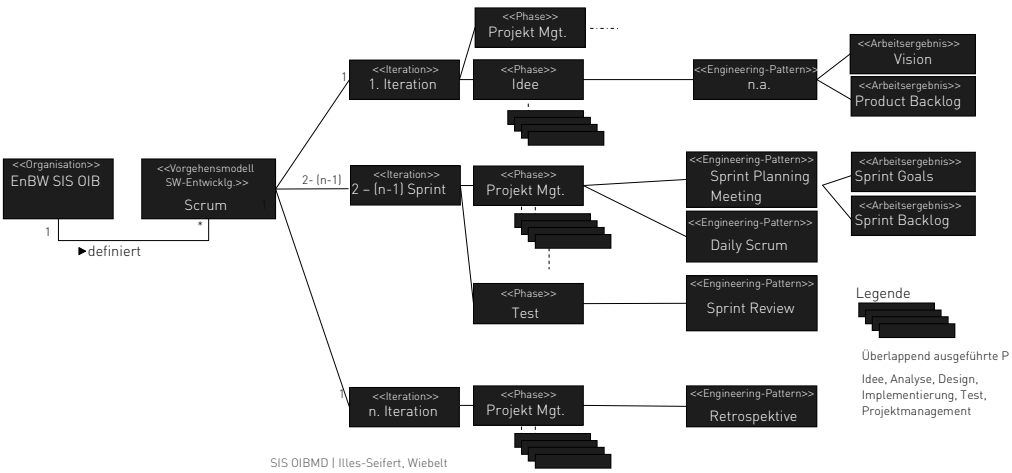


Abbildung 5 - M1 Scrum

M0 - Projektebene Softwareentwicklungsprojekte wählen, je nach Kontext, das für sie geeignete Vorgehensmodell aus und instanziieren dieses. Neben der Projektgröße, dessen Kritikalität, sowie der Teamzusammensetzung spielen Qualitätskriterien eine entscheidende Rolle bei der Instanziierung eines konkreten Vorgehensmodells. Grundsätzlich wird die Entscheidung getroffen, ob eine iterative oder eine sequenzielle Vorgehensweise gewählt wird. Weiterhin werden die Engineering-Patterns ausgewählt, die auf der Grundlage der Projektbedingungen, sowie der konkret ausgeprägten Qualitätskriterien vorgeschlagen werden. Steht beispielsweise das Qualitätskriterium „Usability" im Fokus des Projektes, so werden für die unterschiedlichen Phasen Patterns (z.B. Paper Prototyping, Focus Groups, Fieldstudy, etc.) vorgeschlagen, die eine benutzerzentrierte Projektausrichtung sicherstellen. Die Auswahl der anzuwendenden Engineering-Patterns sowie der zu erzeugenden Arbeitsergebnisse wird durch unterschiedliche Faktoren beeinflusst.

- Faktor firmenweiter Richtlinien: Diese definieren Arbeitsergebnisse, die verpflichtend zu erstellen sind, unabhängig vom gewählten Vorgehensmodell.

- Faktor Vorgehensmodell: Das ausgewählte Vorgehensmodell kann bestimmte Engineering-Patterns oder Arbeitsergebnisse vorschreiben.

- Faktor Qualitätsziele: Die Ausprägung des instanziierten Qualitätsmodells kann dazu führen, dass bestimmte Engineering-Patterns anzuwenden und zugehörige Arbeitsergebnisse zu erstellen sind.

3.3 Das PQI Qualitätsmodell

Im Qualitätsmodell werden die Dimensionen interne und externe Softwarequalität, sowie Prozessqualität betrachtet. Zur Strukturierung der internen und externen Softwarequalität wird der ISO-Standard [ISO/IEC01] zu Grunde gelegt und um weitere Aspekte der Prozessqualität erweitert. Diese Aspekte umfassen die Gebiete: Projektplanung, Projektüberwachung sowie das Konfigurations- und Architekturmanagement. Das Qualitätsmodell ist hierarchisch gegliedert und setzt sich aus Faktoren zusammen, die über Qualitätskriterien verfeinert werden. Die Qualitätskriterien werden über Metriken operationalisiert.

Das Qualitätsmodell wird im Prozess-Metamodell verankert und somit allen abgeleiteten spezifischen Vorgehensmodellen zur Verfügung gestellt. Jedes Softwareentwicklungsprojekt wählt ein geeignetes Vorgehensmodell aus und definiert Qualitätsziele basierend auf dem Qualitätsmodell. Um diese Qualitätsziele zu erreichen werden unterschiedliche Engineering-Patterns angewendet. Bei der Definition der Qualitätsziele müssen Verantwortliche die relative Wichtigkeit der unterschiedlichen Qualitätsfaktoren auf einer dreistufigen Skala bewerten, wobei die erste Stufe der Skala den Standard definiert. Durch die getroffene Auswahl (was das Vorgehensmodell sowie die Qualitätsziele betreffen) werden Engineering-Patterns vorgeschlagen, die sich für den angegebenen Kontext am besten eignen. Die während der Softwareentwicklung erzeugte Ist-Qualität wird gemessen und mit der Soll-Qualität abgeglichen.

Die im PQI-Ansatz definierten Qualitätsstufen sind:

- Stufe 1 - Bronze: Das ist die „Standardqualität“, die für Softwareentwicklungen bezüglich eines angegebenen Qualitätsfaktors angeboten wird. (Der Kunde ist zufrieden, es fällt so wenig wie möglich negativ auf, das Produkt entspricht in weiten Teilen seinen Anforderungen).

- Stufe 2 - Silber: Das ist eine über den Standard hinaus gehende Qualität. (Der Kunde ist positiv überrascht, das Produkt entspricht vollkommen seinen Vorstellungen).

- Stufe 3 - Gold: Wenn Stufe Gold bezüglich eines Qualitätsfaktors gewählt wird, muss die Qualität die Erwartungen des Kunden weit übertreffen. (Kunde ist begeistert).

Wie wird nun das Qualitätsmodell projektspezifisch instanziiert? Auf Basis von Kundenanforderungen werden Prioritäten für die unterschiedlichen Qualitätsfaktoren festgelegt. Hierbei ist darauf zu achten, dass sich manche Qualitätskriterien widersprechen und das die Erreichung einer höheren Qualitäts-Stufe meist mit einem zusätzlichen Aufwand einhergeht.

Abbildung 6 zeigt exemplarisch, wie in einem Projekt die Qualitätsziele definiert wurden. Im gezeigten Beispiel sind vor allem die Qualitätsfaktoren Wartbarkeit und Testbarkeit besonders wichtig. Aufgrund der Ist-Analyse können der aktuelle Status bezüglich der einzelnen Faktoren gemessen und Engineering-Patterns integriert werden, um die Soll-Qualität zu erreichen. (Anmerkung: Die Engineering-Patterns bieten bewährte Praktiken an, die zur Herstellung und Überprüfung der geforderten Qualität eingesetzt werden können).

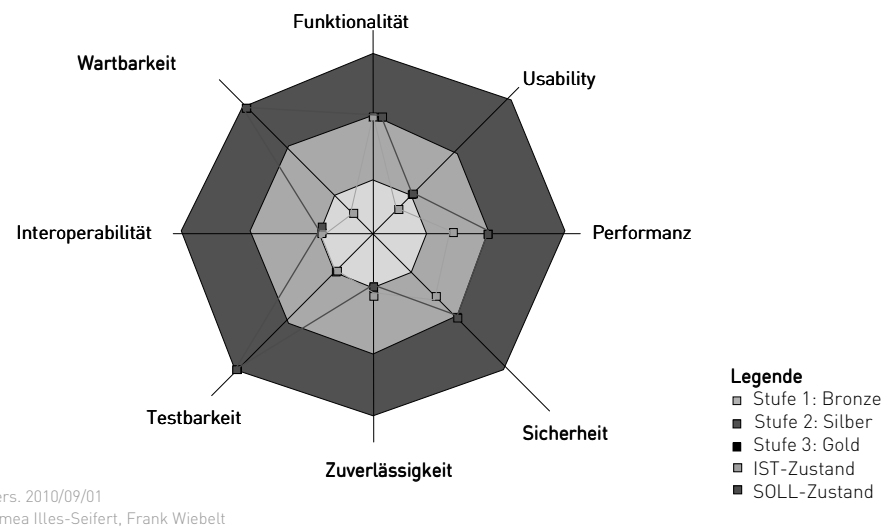


Abbildung 6 - Instanziiertes Qualitätsmodell

3.4 Der PQI Trigger

Die Verknüpfung des ablauforientierten Prozess-Frameworks und des statischen Qualitätsmodells erfolgt über den Trigger [DCF07]. Dieser überwacht den Prozessablauf und meldet die Fertigstellung von Arbeitsergebnissen an definierten Quality Gates. Hierbei werden die Prüfdaten ausgewertet und die Soll- mit der Ist-Qualität verglichen. Die Prüfung kann manuell (Reviews, Test) oder rechnergestützt (statische Analyse) durchgeführt werden.

4 Verwandte Arbeiten

Bisherige Arbeiten konzentrieren sich auf die Vorstellung und Verbesserung einzelner Vorgehensmodelle für die Softwareentwicklung. Die ersten Arbeiten dieser Art wurden von Royce [Ro70] oder Boehm [Bo84], [Bo88] vorgestellt. Weitere Arbeiten, die auf einzelne Vorgehensmodelle eingehen, sind beispielsweise RUP [JBR99], Scrum [Sc01], V-Modell XT [Vm08]. Die Integration unterschiedlicher Vorgehensmodelle, insbesondere die Integration „traditioneller“ und agiler Methoden, wurde bisher nur wenig beleuchtet. Aktuelle Forschungsberichte thematisieren den Übergang von wasserfallorientierten Vorgehensmodellen hin zu agiler Entwicklung wie beispielsweise in [PMS09], [Su07], [Be07], [FG07], [Se07]. Der in diesem Bericht vorgestellte PQI-Ansatz zielt auf die *Integration* unterschiedlicher Vorgehensmodelle ab.

Eine weitere Gruppe verwandter Arbeiten fokussiert sich auf die Definition von Qualitätsmodellen. Hierzu gehören generische Frameworks, wie die ISO [ISO/IEC01], die die Softwarequalität gesamtheitlich betrachten. Andere Arbeiten untersuchen spezielle Qualitätsmerkmale, wie die Design-Qualität beispielsweise in [Bo94], [Co98], [Ka98], [BD02]. Die meisten Arbeiten dieser Gruppe konzentrieren sich auf das Qualitätsmodell selbst und weniger auf dessen Integration in Vorgehensmodelle. Der PQI-Ansatz behandelt die Qualitätsmodellierung und die Prozessmodellierung integrativ, womit erreicht wird, dass Softwarequalität als integraler Bestandteil der Softwareentwicklung verstanden und etabliert wird.

5 Diskussion und Zusammenfassung

In dem vorliegenden Dokument wird der bei der EnBW Systeme Infrastruktur Support GmbH erarbeitete Ansatz zur prozess- und projektorientierten Qualitätsintegration (PQI) vorgestellt. Dieser sieht eine Beschreibung der Softwareentwicklungsprozesse auf unterschiedlichen Abstraktionsebenen vor. Die Verankerung des Qualitätsmodells erfolgt auf der höchsten Abstraktionsebene, wodurch dessen Verwendung in allen abgeleiteten Vorgehensmodellen und Projekten verbindlich wird. Jedes Projekt kann das Qualitätsmodell instanziieren und Qualitätsziele definieren. Hierdurch wird gewährleistet, dass Projekte an der Erreichung der individuell festgelegten Qualitätszielen gemessen werden (Delta-Messung), wobei gleichzeitig eine firmenweit gültige Standardqualität zu Grunde gelegt wird. Hierdurch kann die Heterogenität der zu entwickelnden Produkte, sowie deren individuellen Qualitätsziele Rechnung getragen werden (A1).

Der hier vorgestellte PQI-Ansatz erfüllt weiterhin die Anforderung der Integration unterschiedlicher Vorgehensmodelle (A2) für die Softwareentwicklung. Das Prozessframework erlaubt die Einführung unterschiedlicher Vorgehensmodelle in das Unternehmen ohne das Prozessframework zu verändern. Die Grundprämisse ist, dass jedes neue Vorgehensmodell vom generischen Workflow abgeleitet ist.

Durch das Zugrundelegen des generischen Workflows, auf dem alle Vorgehensmodelle basieren, ist es möglich, vergleichbare Prozesskennzahlen über Projekte hinweg zu erheben (A3). Weiterhin wird durch die Instanziierung des Qualitätsmodells möglich, vergleichbare Kennzahlen über die Software-Qualität zu generieren. Dabei wird für die Berichterstattung über die KPIs nicht der absolut erreichte Wert verwendet, sondern die Soll-Ist-Differenz. Der Soll-Wert wird zum Projektstart in Zusammenarbeit mit dem Auftraggeber definiert.

Ein Projekt ist durch seine Einzigartigkeit definiert. Durch den hier vorgestellten Ansatz wird es möglich, zum einen ein Grundgerüst vorzugeben, um vergleichbare Kennzahlen zu erheben. Andererseits können Projekte individuell ihr Vorgehensmodell, sowie die anzuwendenden Engineering-Patterns definieren.

Der hier vorgestellte Ansatz beruht nicht auf dem Prinzip des Tailorings, der einen Standard Prozess voraussetzt und Anpasst. Vielmehr werden Regeln definiert, die es erlauben, Vorgehensmodelle je nach Kontext auszuprägen. Der Vorteil der hier gewählten Lösung besteht in der *Einfachheit* der Prozessbeschreibung auf abstrakter Ebene und in der *Integration von bewährten Engineering-Patterns,* als Wissensbasis für die Organisation, die das gewählte Vorgehensmodell durch beschrieben Engineering Patterns anreichern.

Die nächsten Schritte sehen den Ausbau der Design-Patterns, die Erweiterung von etablierten Vorgehensmodellen, sowie die Verankerung des Modells in der gesamten Organisation vor.

Literaturverzeichnis

[AI77] Alexander, C., Ishikawa, S.: Murray Silverstein Subject(s) Architecture Publisher Oxford University Press Publication date 1977

[BD02] Bansiya, J.; Davis, C.: A Hierarchical Model for Object-Oriented Design Quality Assessment, IEEE Transactions on Software Engineering, 28(1), 2002, 4-17.

[Be07] Beavers, P. A.: Managing a large “agile” software engineering organization, Proceedings of the AGILE Conference, pp. 296-303, 2007.

[Bo84] Boehm, B.W.: Verifying and Validating Software Requirements and Design Specifications, IEEE Software, vol. 1, no. 1, 75-88, 1984.

[Bo88] Boehm, B.W.: A Spiral Model of Software Development and Enhancement, Computer, v.21 n.5, p.61-72, May 1988.

[Bo94] Booch, G.: Object-Oriented Analysis and Design with Applications (2. Auflage). Benjamin/Cummings, Redwood City, CA, 1994.

[Co98] Cockburn, A.: Object-Oriented Analysis and Design: Part 2. C/C++ Users Journal, 16(6), 1998.

[DCF07] Damiani, E.; Colombo, A.; Frati, F. und Bellettini, C.: A metamodel for modeling and measuring Scrum development process, In (Concas, G.; Damiani, E.; Scotto, M. und Succi, G. Hrsg.); Proceedings of the 8th international Conference on Agile Processes in Software Engineering and Extreme Programming; Lecture Notes In Computer Science. Springer-Verlag, Berlin, Heidelberg, 74-83.

[FG07] Fry, C. und Greene, S.: Large scale agile transformation in an on-demand world. In Proceedings of the AGILE Conference, Seiten 136-142, 2007.

[ISO/IEC01] Standard, 9126. 2001. ISO/IEC Standard 9126: Software Engineering -- Product Quality. Part 1: International Organization for Standarization.

[JBR99] Jacobson, I.; Booch, G.; Rumbaugh, J.: The unified software development process, Addison-Wesley, 1999

[Ka98] Kafura, D.: Object-Oriented Software Design and Construction with C++. Prentice Hall, Upper Saddle River, NJ, 1998.

[PMS09] Pinheiro, C.; Maurer, F.; Sillito, J.; MCIT Solutions, AB, Improving Quality, One Process Change at a Time, 31st International Conference on Software Engineering - Companion Volume, 2009. ICSE-Companion 2009.

[Ro70] Royce, W. W., Managing the Development of Large Software Systems: Concepts and Techniques, Proceedings of WESCON, August 1970.

[Sc01] Schwaber, K.; Beedle, M.: Agile Software Development with Scrum. Prentice Hall, Upper Saddle River 21. Oktober 2001.

[Se07] T. R. Seffernick. Enabling agile in a large organization our journey down the yellow brick road. In Proceedings of the AGILE Conference, pp. 200-206, 2007.

[Su07] Sumrell, M.; From Waterfall to Agile - How does a QA Team Transition? Proceedings of the Agile2007.

[Vm08] www.v-modell-xt.de, V-Modell XT.

Self-Adaptive Software Performance Monitoring

Jens Ehlers, Wilhelm Hasselbring

Software Engineering Group
Christian-Albrechts-University Kiel
24098 Kiel
{jeh,wha}@informatik.uni-kiel.de

Abstract: In addition to studying the construction and evolution of software services, the software engineering discipline needs to address the *operation* of continuously running software services. A requirement for its robust operation are means for effective monitoring of software runtime behavior. In contrast to profiling for construction activities, monitoring of operational services should only impose a small performance overhead. Furthermore, instrumentation should be non-intrusive to the business logic, as far as possible. Monitoring of continuously operating software services is essential for achieving high availability and high performance of these services. A main issue for dynamic analysis techniques is the amount of monitoring data that is collected and processed at runtime. On one hand, more data allows for accurate and precise analyses. On the other hand, probe instrumentation, data collection and analyses may cause significant overheads. Consequently, a trade-off between analysis quality and monitoring coverage has to be reached.

In this paper, we present a method for self-adaptive, rule-based performance monitoring. Our approach aims at a flexible instrumentation to monitor a software system's timing behavior. A performance engineer's task is to specify rules that define the monitoring goals for a specific software system. An inference engine decides at which granularity level a component will be observed. We employ the Object Constraint Language (OCL) to specify the monitoring rules. Our goal-oriented, self-adaptive method is based on the continuous evaluation of these rules. The implementation is based on the Eclipse Modeling Framework and the Kieker monitoring framework. In our evaluation, this implementation is applied to the iBATIS JPetStore and the SPECjEnterprise2010 benchmark.

1 Introduction

While for most software systems performance is a critical requirement, tools that monitor the operation of software systems at *application* level are rarely used in practice. This dichotomy is indicated by a survey carried out at a recent conference among Java practitioners and experts [Sna09]. Garbage collection, concurrency control, remote service calls, database access and legacy integration are identified as typical performance problem areas. Nevertheless, adequate monitoring tools that allow the analysis of these problems and their root causes are seldom known and employed in software engineering projects. The prevalent negligence of continuous operational monitoring is caused by the following symptoms: (1) a posteriori failure analysis, i.e. appropriate monitoring data

is seldom collected and evaluated systematically before a failure occurred, (2) inflexible instrumentation, i.e. probes are placed into a component's source code only at a limited number of fixed points such that recompilation and redeployment are required for future modifications, and (3) inability of request tracing in distributed systems, i.e. tracing of user requests beyond the borders of a single component or its execution container in a distributed system is not supported or not applied.

In this paper, we present a sophisticated method for self-adaptive performance monitoring and evaluate the design of an associated tool to handle the above shortcomings for existing component-based Java EE applications. Means for efficient and flexible observation, analysis, and reporting of a software system's runtime behavior are indispensable to ensure its proper operation. Therefore, two constraints have to be considered for continuous monitoring: (1) acceptable overhead and (2) non-intrusive instrumentation. In contrast to profiling at construction time, the monitoring overhead at operation time has to be kept deliberately small. Secondly, the instrumentation of probes should not pollute the application's business logic.

A main issue for dynamic analysis techniques addressing software behavior comprehension is the amount of information which is collected and processed at runtime. The more detailed monitoring data is the more precise subsequent analysis can be. On the other hand, instrumentation injection, data collection, logging, and online analysis itself cause measurable overhead. Consequently, a trade-off between analysis quality and monitoring coverage has to be reached.

It is not the injection of numerous dummy probes that causes overhead, but the complexity of real probe implementations. We have evaluated and quantified the decisive impact of how the observed data is collected, processed, and logged for subsequent analyses [vHRH+09]. A finding is that it is possible to inject diverse probes at a multitude of relevant join points, as long as not all of them are active at the same time during operation. Thus, our approach aims not at a fixed instrumentation to monitor a software system's timing behavior. Instead, a major objective is a self-adaptive activation of probes and their related join points. For that purpose, an inference engine decides at which granularity level a component will be observed. A performance engineer's task is to specify rules that define the monitoring goals for a specific software system.

Typical goals of software system monitoring include the required evidence of SLA compliance, the localization of faults causing QoS problems, capacity planning support, and mining of usage patterns for interface design or marketing purposes. In this paper, we focus on the goal to detect the origins of QoS problems or other system failures that effect a change in the system's normal behavior as perceived by its users. We employ the Object Constraint Language (OCL) [OMG10] to specify the monitoring rules. The rules refer to performance attributes such as responsiveness metrics and derived anomaly scores that change their values during runtime. Our goal-oriented self-adaptation is based on the continuous evaluation of these rules. Our presented implementation is based on EMF (Eclipse Modeling Framework) [SBPM08] meta-models which allow to evaluate OCL query expressions on object-oriented instance models at runtime. The contributions of this paper are

(1) the design of a generic monitoring process for component-based software systems which is based on our Kieker monitoring framework,[1]

(2) an approach for self-adaptive performance monitoring as an extension to (1),

(3) an implementation and evaluation of this approach.

The remainder of the paper is structured as follows: We briefly describe our Kieker monitoring framework in Section 2. In Section 3, we present our approach for self-adaptive performance monitoring. Its evaluation in lab experiments and industrial systems is summarized in Section 4. Related work is discussed in Section 5. Finally, a conclusion and outlook to future work is given in Section 6.

2 The Kieker Monitoring Framework

This section introduces the monitoring process underlying our Kieker monitoring framework for component-based software systems [vHRH+09]. The monitoring process consists of the following activities, as illustrated in Fig. 1: (1) probe injection, (2) probe activation, (3) data collection, (4) data provision, (5) data processing, (6) control and visualization, and (7) adaptation. The sequence of the activities and their interrelation with the major architectural components of the monitoring framework is annotated to the component diagram shown in Fig. 1. In this paper, we emphasize the (self-)adaptation activity which is presented in detail in Section 3. As the adaptation is closely interconnected with the preceding activities named above, our integrated monitoring process is subsequently explained.

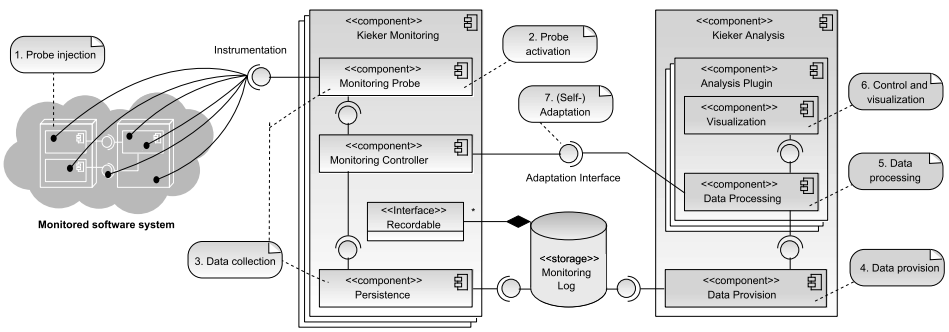


Figure 1: Self-Adaptive Monitoring Process and Architecture

Probe Injection: The software system to be monitored has to be instrumented with probes. A probe is positioned at each join point at which observation of control or data flow could be interesting. To measure service times, probes are placed at

[1] http://kieker.sourceforge.net/

operation entry and exit points. These include particularly the public services provided by a component's interface, but may also cover private methods or code blocks. Aspect-oriented programming (AOP) is an appropriate means to inject application-level probes into class methods in object-oriented software systems [FHRS07]. Thus, our default approach is to utilize the interception framework AspectJ for instrumentation. If method-level instrumentation is not fine-grained enough, it is proposed to use byte-code instrumentation approaches such as Javassist [Chi00]. To impurify the code by placing the probes manually, remains the least valued choice for instrumentation.

Probe Activation: As illustrated in the left part of Fig. 1, Monitoring Probes are triggered from various join points embedded in the monitored software system. The probes are part of a Kieker Monitoring component. In a distributed software system, an instance of this monitoring component is deployed on each execution container and interconnected with the hosted application components, e.g. by load-time aspect weaving. Each Kieker Monitoring component is controlled by a single Monitoring Controller. The controller manages which probes and probe join points are currently activated. Additionally, it provides access to a persistence unit for logging the monitored data and provides an adaptation interface for reconfiguration (see Section 3).

Data Collection: Our monitoring framework Kieker comes along with a set of different aspects that allow the detection of service call entries incoming via different interface technologies, including HTTP Servlets, JAX-WS, JAX-RS, EJB 3.0, Java Remote Method Invocation (RMI), or JMS. These aspects intercept specific framework methods, e.g. javax.servlet.http.HttpServlet.do*(..) for any HTTP operation processed by a Servlet or @javax.jws.WebMethod *.*(..) for JAX-WS annotated interface methods. Furthermore, Kieker provides an aspect to intercept application-specific operations. Typically, this includes all methods as part of a component realization that contribute to the system's application logic and which is in the scope of monitoring.

Each request is tagged with an id when it enters the system. By means of the framework aspects, it is possible to trace a request over any inter-component and inter-server communication protocol throughout a distributed system. To track the relation between a caller and its callee, the probe intercepting the calling operation can either store information about its join point in thread local memory or attach it to the meta-data of the service call. The latter is appropriate, if the call is asynchronous or crossing container borders. The probe linked to the called operation receives the information about its calling context and saves it in a monitoring record. Later, analyses will process the recorded calling context information of all operation calls related to a single request and reconstruct a monitored request trace. Besides calling context information, probes gather performance related metrics such as service times, call frequencies (to determine throughput), or resource utilization.

As monitoring data is collected at different nodes of a distributed system, it has to be collected for system-wide analysis. Fig. 1 illustrates that each monitoring component pushes its records into a central repository called Monitoring Log. Recording with Kieker can be applied to any persistent storage repository like a file system, a database, or a buffered message queue. As indicated in Fig. 1, probes can collect arbitrary data records and push them into the log, on condition that the record class implements the Kieker

interface for Recordable entities. Note that the persistence operations cause a major part of the measurement expense [vHRH+09].

Data Provision: While the Kieker Monitoring component is deployed several times, with one instance at each monitored execution container, it is sufficient to run only one instance of the Kieker Analysis component. The Kieker Analysis component frames a client application with a graphical control panel addressing the system's performance engineer. Our implementation of this component is realized as a tool based on Eclipse RCP and EMF. A screenshot is shown later in Section 3 in Fig. 4. The data stream in the Kieker Analysis component follows the pipes-and-filters pattern [TMD09]. The monitored records serve as input for the piped analysis data stream. The first and lowest-level filter is a Data Provision component. This data provider is configured to consume the records of multiple monitored execution container nodes, each one being equipped with a separate Monitoring Controller. Each controller provides information to the data provider about its Monitoring Log used for record persistence. Due to this information, a data provider can connect to the log(s) and continuously receive and forward the monitoring records to subsequent analysis filters. Thus, a data provider serves as a central consolidation filter that merges observations made in the single nodes of a distributed system.

Data Processing: The proceeding filters of the analysis data stream are joined together via a plugin mechanism. The Kieker Analysis component is constructed to be easily extensible with (third-party) Analysis Plugins containing Data Processing and/or Visualization components. Such data processing or visualization filters can be subscribed to a data provision filter or another supplying data processing filter, provided that the filters' input and output ports match. Visualizations that display the analysis results are usually a sink of the data stream. Each filter can implement any kind of processing based on the records received from its predecessors. The incoming records deliver pieces of information that can be assembled or visualized in different ways. For instance, records are related to tracing, responsiveness, or utilization.

Tracing allows to recap how one or more client requests have been executed by the software system under inspection. Tracing can be applied at different abstraction levels, e.g. studying interaction at the level of components, classes, or methods. In any case, information about the calling dependencies between arbitrary interacting entities is collected. This information can be represented in different ways, e.g. by a dynamic call tree [JSB97], by a call graph [GKM82], or by a calling context tree [ABL97]. These representations differ in their accuracy and efficiency, and hence are suitable for different tasks. A calling context tree (CCT) is an appropriate intermediate representation for continuous monitoring, as it is more accurate than a call graph and less resource consuming than storing every single call trace. The breadth of a CCT is limited by the number of observed entities and not by the number of requests over time. All requests with the same trace are aggregated in a CCT. Though the metrics of identical traces are merged, the stack context of each call is preserved [RvHG+08].

Our adaptive monitoring approach employs analyses that are based on the deviation of observed and expected service time. A derived measure is the anomaly score assigned to a sample of observed operation executions, see Section 3.

Control and Visualization: It is a major requirement of a monitoring tool to provide a control panel that conveniently visualizes the monitored data for analysts. The Kieker Analysis component serves as such a control panel framework and comes along with a set of plugins that implement the graph representations explicated above to picture tracing and compositional dependencies. Different libraries such as Graphviz[2] and Eclipse Zest[3] are used with Kieker for graph visualization. Various other plugins have been developed or are work in progress to be integrated into the framework, e.g. reverse engineered dependency and sequence diagrams, Markov chains, or 3D runtime behavior visualizations. As stated, the analysis component can easily be extended with further plugins if required. Concerning the runtime adaptability of monitoring probes, the related monitoring adaptation plugin described in the following section is of particular interest.

3 Adaptive Monitoring

Considering a software system in productive operation, it is not feasible to record each observed occurrence of any probe at any join point. In fact, this restriction is caused by the great number of expected requests per time slice, not by the number of injected probe join points. Thus, initial monitoring rules have to be configured prescribing which probes are where and when active. Regarding a component-based software system, an appropriate initial monitoring setting can be to activate only join points at operations which serve as part of a component's provided interface (see the activated monitoring points in Fig. 2). At runtime, the monitoring level can be increased to inspect the interior control flow of a component in case it does not behave as expected. The selective activation of a probe join point may depend on its call stack level, the responsiveness of the related operation, the current workload, a random probability, etc.

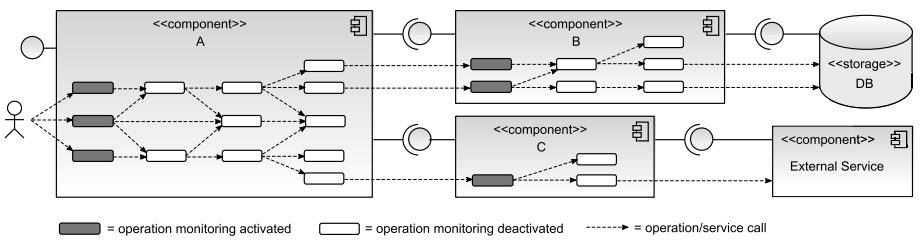


Figure 2: Software system with operation intercepting probes

In this section, we focus on the goal to detect the origins of QoS problems or other system failures that effect a change in the system's normal behavior as perceived by its users. A major part of the failure recovery time is required to locate a failure's root cause. In [KF05], the authors refer to an estimation that 75% of the recovery time is spent just for

[2] http://www.graphviz.org/
[3] http://www.eclipse.org/gef/zest/

fault detection. They report that it often takes days to search for a failure's cause, while it can be fixed quickly once the fault is found.

In case that harmful faults induce anomalous runtime behavior at application level, our proposed solution allows to adapt the monitoring coverage on demand. If the adaptation is conducted manually, the human decision to change the set of active probes or join points is usually based on a previous interpretation of performance visualizations provided by other Kieker plugins. A typical situation is that time series curves indicate a decline of a service's throughput though the load did not increase. A performance engineer who observes this incidence is interested in the cause of the phenomenon and therefore tries to activate more join points in the affected components. Afterwards, it takes a while until enough analysis-relevant records have been collected via the newly activated probe join points. Our self-adaptive monitoring process aims at reducing this potentially business-critical wait time that delays a failure or anomaly diagnosis.

Our proposed Adaptation Plugin is designed to be one of several plugins integrated into the Kieker Analysis component. It allows to (re)configure the set of active probes and probe join points that were previously injected into the system. The proposed adaptation process works like a simple rule-based expert system that employs deductive reasoning to reach a decision. The performance engineer acts as the expert who formulates the inference rules. To adjust the monitoring coverage, the desired conclusions are either to activate a set of currently disabled probe join points, to deactivate a set of currently enabled probe join points, or to leave the current setting unchanged. The set of join points to be changed is addressed in the premise of monitoring rules.

We employ the OCL to specify these rule premises. A premise consists of an OCL context element and an OCL expression. Indeed, the context element specifies the context in which the expression will be evaluated. Often, the adaptation filter itself is an appropriate context element, because it allows navigation to all records that have been supplied by preceding data processing filters.

The OCL expression specifies a set of join points in form of method signatures at which the referenced probes should be activated. The evaluation of the monitoring rules can either take place manually, i.e. released on a click in the control panel, or automatically, i.e. repetitive each time a specified time period has elapsed. As shown in Fig. 1, the Monitoring Controller of each monitored container provides an Adaptation Interface. The Adaptation Plugin contains a derived Data Processing filter that allows to start a concurrent thread for continuous evaluation of the monitoring rules. When a violation of the rule premises is detected, the filter uses the Adaptation Interface of each monitored container to execute the adaptation autonomously. Goal-oriented self-adaptation is based on the possibility to refer to (performance) attributes in the monitoring rules that change their values during runtime, e.g. responsiveness metrics and derived anomaly scores. The time interval between the two succeeding evaluations of the monitoring rules has to be set up. Enough time is required to measure reliable responsiveness and anomaly values in each cycle iteration, but also short enough to react flexibly and without distracting delays to performance degradations. An interval value in the scale of a couple of minutes is appropriate.

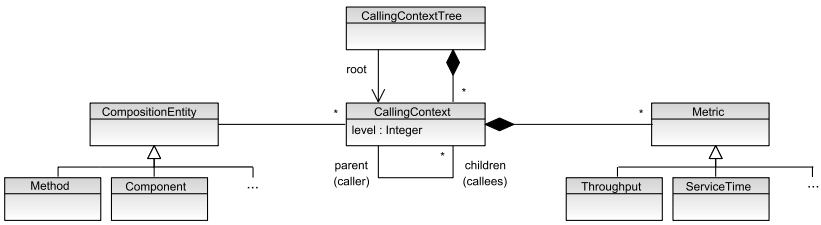


Figure 3: Meta-model of a calling context tree

In the following, example monitoring rules are discussed. The first one starts with a minimal set of active probe join points comprising only class methods (as concrete compositional entities) being placed at the topmost level of a CCT. It is assumed that the context element referenced by the OCL identifier self is set to a CCT instance which was previously delivered to the adaptation filter. A meta-model for a CCT is depicted in Fig. 3.

Example Rule R_1: "If a callee (such as a method) is at the top of an observed call stack (level 1 of the CCT), then enable the probe join points required to intercept and monitor calls to this callee." The corresponding OCL expression to monitor interface operations is:

context CallingContextTree: **self**.callingContexts→*select*(level = 1)→*collect*(method)

OCL provides several built-in collection operations to enable powerful ways of projecting new collections from existing ones. The expression above employs the operations *select* and *collect*. In case of the monitoring rules, the demand is to reduce the set of all join points to a selection of those ones to be activated at the moment. The operation *select* satisfies this demand, as it selects a subset of a collection based on a boolean expression. The OCL simple syntax form is collection→*select*(boolean-expression). The operation *collect* serves to come up to a derived collection that contains different object types than the collection it is originated from, i.e. the new collection is not a subset of the original one. The OCL simple syntax form is collection→*collect*(expression). OCL is mostly known for its purposes to specify invariants on classes, pre- and postconditions on operations, or guards annotated to UML diagrams. Despite that, the first purpose mentioned in the OCL specification suggests OCL to serve as a query language [OMG10]. Indeed, the premises of the monitoring rules can be seen as queries for a set of probe join points to be activated.

The second example rule is more sophisticated. It aims at intensifying the monitoring coverage within a component if it behaves anomalous.

Example Rule R_2: "If the premise of R_1 is fulfilled or if the corresponding caller is already monitored and behaves anomalous, i.e. its anomaly score exceeds a specified threshold t, then enable the probe join points required to intercept and monitor calls to the callee." The OCL expression to monitor interface and anomalous operations is:

context CallingContextTree: **self**.callingContexts→*select*(level = 1 **or** (level > 1 **and** parent.monitoringActivated **and** parent.anomalyRating > t))→*collect*(method)

Imagine that the monitoring rule R_2 is applied to the sample system depicted in Fig. 2. In the initial monitoring coverage only the system's interface operations are observed. If it is discovered that some of these operations do not behave as expected during runtime, the evaluation and appliance of the rule R_2 leads to the join point activation for all callees of the operations indicated as anomalous. Subsequently it can turn out that some of the newly observed callee operations behave anomalous as well. The repeated appliance of R_2 allows to zoom into the control flow of a component in order to seek for the root cause of higher-level anomalies. It is obvious that a manual exploration of such cause-and-effect chains is much more time-consuming and error-prone than an automated processing. A contribution of the self-adaptive monitoring approach is to save this time and effort. Due to space restrictions, the employed anomaly rating procedures are not discussed in this paper. An introduction to a variety of appropriate methods can be found in [CBK09, PP07].

4 Implementation and Evaluation

We have employed the Kieker monitoring component in the productive systems of a telecommunication company and a digital photo service provider, in order to show its practicability [RvHH+10]. Both industrial partners did not observe any perceivable runtime overhead due to the injection of our monitoring probes. In lab experiments, we evaluated the monitoring overhead in more detail and quantified the rates caused by instrumentation, data collection, and logging [vHRH+09].

We implemented the self-adaptive monitoring approach as a Kieker plugin. The implementation of the Kieker Analysis component employs EMF meta-models. For the Adaptation Plugin, we utilize the EMF Model Query[4] sub-project, which allows to construct and run queries on EMF models by means of OCL. Our realization of the Adaptation Interface is based on Java's remote method invocation (RMI) protocol. Probe (de)activation instructions are transmitted through this interface. We evaluated the Adaptation Plugin in lab experiments with the iBATIS JPetStore[5] and the SPECjEnterprise2010[6] benchmark. The sample applications have been stressed with intensity-varying workloads. As long as the system's capacity limits are not exceeded, no anomalies are reported and the monitoring coverage remains constant. In the next step, we manually induced faults into some component-internal operations. We tested successfully that the self-adaptive monitoring is able to locate these faulty operations via anomaly detection as the root cause for the resulting failures perceived by the system users.

Fig. 4 displays a screenshot of the Kieker Analysis component. The left frame (1) in the screenshot contains a project navigator that allows to organize the data processing and visualization filters of a monitoring project. In the frame on the top right (2), the configuration editor of the monitoring adaptation is depicted. It allows to specify a monitoring rule's OCL context element and OCL expression. The middle frame (3) shows a method-level CCT of the SPECjEnterprise2010 benchmark. Based on the failure

[4] http://www.eclipse.org/modeling/emf/?project=query
[5] http://sourceforge.net/projects/ibatisjpetstore/
[6] http://www.spec.org/jEnterprise2010/

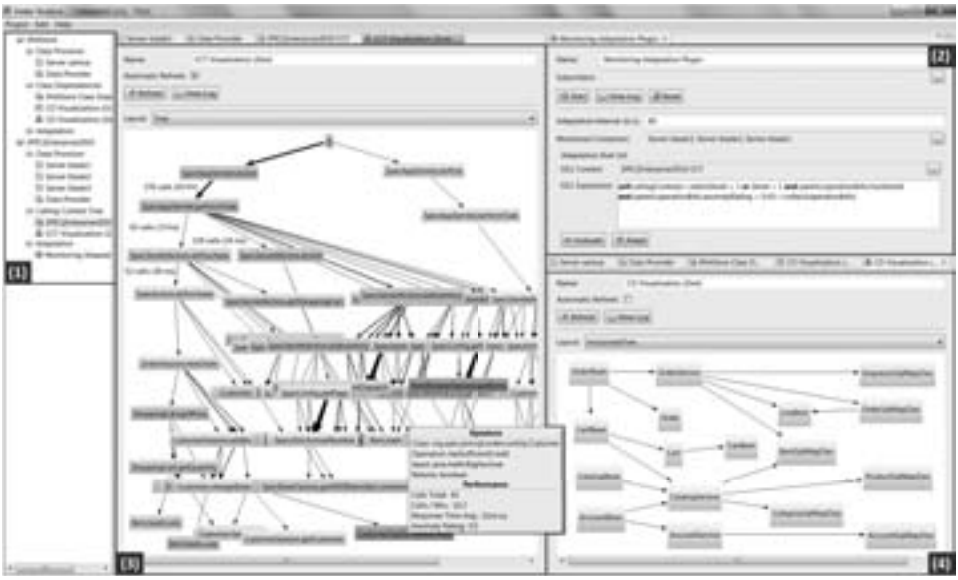


Figure 4: Screenshot of the Kieker analysis tool

diagnosis method of [MRvHH09], the colorings of the method nodes from green to red indicate their current anomaly rating. Call frequencies and average service times of calling contexts are displayed interactively due to clicks on the edges. The depicted tooltip highlights the collected performance metrics for one of the methods. In the frame on the bottom right (4), a class dependency graph of the JPetStore application is displayed. The edges are annotated with aggregated call frequencies among interdependent classes.

5 Related Work

Any integrated monitoring framework addresses two aspects: monitoring (i.e. instrumentation and data collection) and subsequent analysis. In [PMM06], the COMPAS JEEM tool is presented as such an integrated approach. It allows to inject probes as a component-level proxy layer into Java EE systems. Thus, interception is limited to the interface level of Java EE components such as EJBs or Servlets. Runtime adaptation of the monitoring coverage is studied in [MM04]. By observing component-internal operations, Kieker provides a finer-grained analysis of the runtime behavior than COMPAS.

Another approach is Magpie [BDIM04], which monitors resource utilization and component interactions in distributed systems. Magpie is implemented to monitor systems based on Microsoft technology, while Kieker concentrates on Java-based systems. The Pinpoint approach [KF05] utilizes monitoring data to determine anomalous runtime behavior. In contrast to Kieker, Pinpoint does not focus on performance, but applies data mining techniques to detect anomalies in the observed request traces. The Rainbow

project [GCH+04] employs monitoring for architecture-based adaptation of software systems. Magpie, Pinpoint, and Rainbow do not contribute means for self-adaptation concerning the monitoring coverage.

Like Kieker, the open-source projects Glassbox and InfraRED suggest AOP-based instrumentation to introduce monitoring probes. Both projects do not cover request tracing in distributed systems, as well as sophisticated data analyses. The popular open-source tool Nagios is rather intended for infrastructure monitoring than for application-level introspection. Related commercial products like CA Wily Introscope, DynaTrace, or JXInsight do not yet provide rule-based self-adaptation to control the monitoring coverage.

6 Conclusions and Future Work

Predictive support for failure prevention is desired in order to improve a system's fault-tolerance. If alarming system events indicating future performance drops are detected early enough, mechanisms that adapt the system's architectural configuration can be triggered. Therefore, responsiveness and scalability of the system components have to be continuously observed and evaluated. If, for example, a performance degradation is indicated, adaptation decisions and actions concerning the monitoring coverage are derived and the anomalous behavior may be reported or visualized. Filtering the set of active probes and join points on demand allows to zoom into a component if it behaves not as expected. In this case, zooming means to activate more (or less) join points in the control flow aiming at increasing (or decreasing) insight, e.g. into the operation call stack, effective loop iterations, or conditional branches taken. The activation control of probes and join points can either be applied manually or autonomously. For self-adaptive control a set of guiding monitoring rules is required.

In this paper, we presented the design of a generic monitoring process for component-based software systems. Tooling for the monitoring process is provided by our Kieker monitoring framework. Further, we proposed an approach for self-adaptive performance monitoring based on the continuous runtime evaluation of OCL monitoring rules. The feasibility of the self-adaptation has been shown by our implementation of a monitoring adaptation plugin as an extension for Kieker. In lab experiments, we applied our self-adaptive monitoring to benchmark applications such as SPECjEnterprise2010. In our future work, we will report on quantitative evaluation results for different anomaly rating procedures used for self-adaptive monitoring. Besides, we will concentrate on further evidence of the practicability of our self-adaptive monitoring approach by applying it to additional case studies and industrial systems.

References

[ABL97] G. Ammons, T. Ball, and J. R. Larus. Exploiting Hardware Performance Counters with Flow and Context Sensitive Profiling. In *Proc. of the ACM SIGPLAN 1997 Conf. on Programming Language Design and Implementation*, pages 85–96. ACM, 1997.

[BDIM04] P. Barham, A. Donnelly, R. Isaacs, and R. Mortier. Using Magpie for request extraction and workload modelling. In *Proc. of the 6th Conf. on Symposium on Operating Systems Design & Implementation*, pages 259–272. USENIX, 2004.

[CBK09] V. Chandola, A. Banerjee, and V. Kumar. Anomaly Detection: A Survey. *ACM Computing Surveys*, 41(3):1–58, 2009.

[Chi00] S. Chiba. Load-time Structural Reflection in Java. In *Proc. of the 14th European Conf. on OO Programming (ECOOP 2000)*, pages 313–336. Springer, 2000.

[FHRS07] T. Focke, W. Hasselbring, M. Rohr, and J.-G. Schute. Instrumentierung zum Monitoring mittels Aspekt-orientierter Programmierung. In *Tagungsband SE 2007*, volume 105 of *LNI*, pages 55–58. Köllen Druck+Verlag, 2007.

[GCH+04] D. Garlan, S.-W. Cheng, A.-C. Huang, B. Schmerl, and P. Steenkiste. Rainbow: Architecture-Based Self-Adaptation with Reusable Infrastructure. *Computer*, 37(10):46–54, 2004.

[GKM82] S. L. Graham, P. B. Kessler, and M. K. Mckusick. Gprof: A call graph execution profiler. *ACM SIGPLAN Notices*, 17(6):120–126, 1982.

[JSB97] D. F. Jerding, J. T. Stasko, and T. Ball. Visualizing interactions in program executions. In *ICSE '97: Proc. of the 19th Intl. Conf. on Software Engineering*, pages 360–370. ACM, 1997.

[KF05] E. Kiciman and A. Fox. Detecting application-level failures in component-based internet services. *IEEE Trans. on Neural Networks*, 16(5):1027 –1041, 2005.

[MM04] A. Mos and J. Murphy. COMPAS: Adaptive Performance Monitoring of Component-Based Systems. In *2nd ICSE Workshop on Remote Analysis and Measurement of Software Systems (RAMSS 04), 26th Intl. Conf. on Software Engineering (ICSE 2004)*, pages 35–40, 2004.

[MRvHH09] N. Marwede, M. Rohr, A. van Hoorn, and W. Hasselbring. Automatic Failure Diagnosis Support in Distributed Large-Scale Software Systems based on Timing Behavior Anomaly Correlation. In *Proc. of the 2009 European Conf. on Software Maintenance and Reengineering (CSMR'09)*, pages 47–57. IEEE, 2009.

[OMG10] OMG. Object Constraint Language, Version 2.2. `http://www.omg.org/spec/OCL/2.2/`, 2010.

[PMM06] T. Parsons, A. Mos, and J. Murphy. Non-intrusive end-to-end runtime path tracing for J2EE systems. *IEE Proc. – Software*, 153(4):149–161, 2006.

[PP07] A. Patcha and J.-M. Park. An overview of anomaly detection techniques: Existing solutions and latest technological trends. *Computer Networks*, 51(12):3448–3470, 2007.

[RvHG+08] M. Rohr, A. van Hoorn, S. Giesecke, J. Matevska, W. Hasselbring, and S. Alekseev. Trace-context sensitive performance profiling for enterprise software applications. In *Proc. of the SPEC Intl. Performance Evaluation Workshop (SIPEW '08)*, volume 5119 of *LNCS*, pages 283–302. Springer, 2008.

[RvHH+10] M. Rohr, A. van Hoorn, W. Hasselbring, M. Lübcke, and S. Alekseev. Workload-intensity-sensitive timing behavior analysis for distributed multi-user software systems. In *WOSP/SIPEW '10: Proc. of the 1st joint WOSP/SIPEW Intl. Conf. on Performance Engineering*, pages 87–92. ACM, 2010.

[SBPM08] D. Steinberg, F. Budinsky, M. Paternostro, and E. Merks. *EMF: Eclipse Modeling Framework*. Addison-Wesley, 2nd edition, 2008.

[Sna09] R. G. Snatzke. Performance Survey 2008 – Survey by codecentric GmbH. `http://www.codecentric.de/de/m/kompetenzen/publikationen/studien/`, 2009.

[TMD09] R. N. Taylor, N. Medvidovic, and E. Dashofy. *Software Architecture: Foundations, Theory, and Practice*. Wiley, 2009.

[vHRH+09] A. van Hoorn, M. Rohr, W. Hasselbring, J. Waller, J. Ehlers, S. Frey, and D. Kieselhorst. Continuous monitoring of software services: Design and application of the Kieker framework. Technical Report TR-0921, Department of Computer Science, University of Kiel, 2009.

Towards Syntactical Model Quality Assurance in Industrial Software Development: Process Definition and Tool Support *

Thorsten Arendt, Sieglinde Kranz, Florian Mantz, Nikolaus Regnat, Gabriele Taentzer

Philipps-Universität Marburg, Germany
Siemens Corporate Technology, Germany
Høgskolen i Bergen, Norway

{arendt,taentzer}@mathematik.uni-marburg.de
{sieglinde.kranz,nikolaus.regnat}@siemens.com
fma@hib.no

Abstract: The paradigm of model-based software development has become more and more popular, since it promises an increase in the efficiency and quality of software development. Following this paradigm, models become primary artifacts in the software development process where quality assurance of the overall software product considerably relies on the quality assurance of involved software models. In this paper, we concentrate on the syntactical dimension of model quality which is analyzed and improved by model metrics, model smells, and model refactorings. We propose an integration of these model quality assurance techniques in a predefined quality assurance process being motivated by specific industrial needs. By means of a small case study, we illustrate the model quality assurance techniques and discuss Eclipse-based tools which support the main tasks of the proposed model quality assurance process.

1 Introduction

In modern software development, models play an increasingly important role, since they promise more efficient software development of higher quality. It is sensible to address quality issues of artifacts in early software development phases already, for example the quality of the involved software models. Especially in model-driven software development where models are used directly for code generation, high code quality can be reached only if the quality of input models is already high.

In this paper, we consider a model quality assurance process which concentrates on the syntactical dimension of model quality. Syntactical quality aspects are all those which can be checked on the model syntax only. They include of course consistency with the language syntax definition, but also other aspects such as conceptual integrity using the same

*This work has been partially funded by Siemens Corporate Technology, Germany.

patterns and principles in similar situations and conformity with modeling conventions often specifically defined for software projects. In [FHR08], the authors present a taxonomy for software model quality distinguishing between inner and outer quality aspects of models. Inner quality aspects are concerned with a single model only. The consistency of a model, its conceptual integrity, and its conformity to standards are typical examples for inner quality aspects of models. Outer quality aspects are concerned with relations of a model to other artifacts of a software engineering process. The completeness of a design model wrt. to its analysis model on the one hand and to code on the other hand falls in this category of quality aspects. These and further quality aspects are discussed in [FHR08].

In the literature, typical syntactical quality assurance techniques for models are model metrics and refactorings, see e.g. [GPC05, SPLTJ01, MB08, Por03]. They origin from corresponding techniques for software code by lifting them to models. Especially class models are closely related to programmed class structures in object-oriented programming languages such as C++ and Java. For behavior models, the relation between models and code is less obvious. Furthermore, the concept of code smells can be lifted to models leading to model smells. Again code smells for class structures can be easily adapted to models, but smells of behavior models cannot directly deduced from code smells.

In this paper, we present a **quality assurance process for syntactical model quality** which can be adapted to specific project needs. It consists of two phases: First, project- and domain-specific quality checks and refactorings have to be specified which should be done before a software project starts. Model quality checks are formulated by model smells which can be specified by model metrics or anti-patterns. Thereafter, the specified quality assurance process can be applied to concrete software models by reporting all their model smells and applying model refactorings to erase at least some of the model smells found. However, we have to take into account that also new model smells can come in by refactorings. This check-improve cycle should be performed as long as needed to get a reasonable model quality. The process is supported by tools, i.e. model metrics reports, smell detection and the application of refactorings are supported by Eclipse plug-ins being based on the Eclipse Modeling Framework.

This paper is organized as follows: In the next section, the need for syntactical model quality assurance in industrial software development is motivated. In Section 3, we present a two-phase quality assurance process for syntactical model quality which can be adapted to specific project needs. In Section 4, we present Eclipse plug-ins EMF Metrics, EMF Smell, and EMF Refactor at a small example. Finally, related work is discussed and a conclusion is given.

2 The need for syntactic model quality assurance in industrial software development

A typical model-based software development project at Siemens covers between 10 - 100 developers. These developers use models in different ways, e.g. specifying the software architecture and design, using input for code implementation or getting information for

tests and test case generation. Often these developers are not on one site, e.g. architects are located in Germany and the implementers are located at a site in east Europe. In this case, models are an essential part of development communication and their quality influences the quality of the final product to a great extent. In addition, model-based software projects are often part of mechatronic systems with safety relevant parts. In these cases safety aspects must also be observed. Standards like the IEC 61508 requires that for a mechatronic system all intermediate results during the development process including software models must be of an appropriate quality.

At the moment the most commonly used modeling language is the UML. It is a very comprehensive and powerful language, but does not cover any particular method and comes without built-in semantics. On the one hand, this allows a flexible use but includes a high risk of misuse on the other hand. Without tailoring a project specific usage of UML before starting development, the practical experience showed that the created models can be difficult to understand or even misinterpreted. Detailed project-specific guidelines are very important therefore and their compliance must be enforced and controlled.

An essential aspect of modeling is the capability to consider a problem and its solution from different perspectives. The existence of different perspectives increases, however, the risk of inconsistencies within a model. To avoid large models and to deal with different organizational responsibilities, the model information is frequently split into a set of smaller models with relationships in between. Information contained in one model is reused (and probably more detailed) in other models. Consistency is therefore not only needed within a model, but also between a project-specific set of models. Unfortunately, the support of available UML modeling tools to prevent the user from modeling inconsistencies and to easily find contradictions is very limited. Often, parallel changes in one model cannot be avoided in development projects. A frequent reason for that is an established feature oriented software development process. Required subsequent merges can easily result in inconsistencies, especially since the merge functionality of most UML modeling tools is not satisfactory.

Since model-driven software development is not yet a well established method in industry, software development is often faced with the problem that at least a part of its project members have no or limited experience with modeling in general as well as with the modeling language used and its tools. This often leads to misuse such as modeling of unnecessary details on a higher abstraction level and removing of no longer needed model elements only from diagrams instead from the model. Especially at the beginning of a model-based project such problems should be identified as soon as possible to avoid that the misuse is copied by other modelers and to ensure that the effort to correct these problems is still low.

Unfortunately, the existing and established quality assurance methods *document review* and *code inspection* cannot be used one-to-one within model-based development. The manual review of models is very time consuming and error prone. Models cannot be reviewed in a sequential way because of the existence of links between its elements and to other models. To reduce the quality risks of models mentioned above and unburden the review effort, it is essential that each project defines a specific list of guidelines at the beginning of its model-based software development, derives syntactic checks from these guidelines and automates these checks by a tool.

3 The syntactical model quality assurance process

In this section we propose the definition and application of a structured model quality assurance process that can be used to address project-specific needs as described in the previous section. The approach uses already known model quality assurance techniques like model metrics, model smells, and model refactorings which are combined in an overall process for structured model quality assurance concerning syntactical model issues.

3.1 A two-phase model quality assurance process

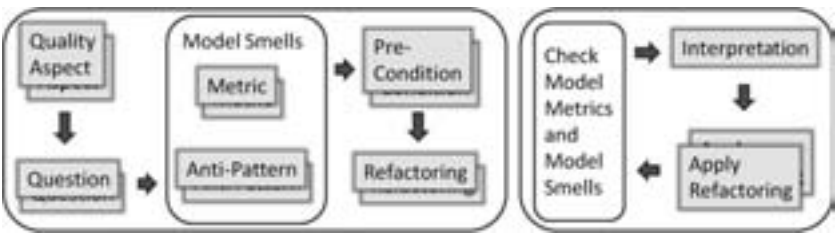


Figure 1: Project-Specific Quality Assurance Process - Specification and Application

As already mentioned, it is essential to define quality-related issues at the beginning of model-based software development. On the left side of Figure 1, we show how to define a model quality assurance process for a specific project. Firstly, it is required to determine those quality aspects which are important for project-specific software models. In the next step, static syntax checks for these quality aspects are defined. This is done by formulating questions that should lead to so-called model smells which hint to model parts that might violate a specific model quality aspect. Here, we adopt the Goal-Question-Metrics approach (GQM) that is widely used for defining measurable goals for quality and has been well established in practice [BCR94]. The formulated questions need answers which can be given by considering the model syntax only. Some of these answers can be based on metrics. Other questions may be better answered by considering patterns. Here, the project-specific process can reuse general metrics and smells as well as special metrics and smells specific for the intended modeling purpose.

Finally, a specified model smell serves as pre-condition of at least one model refactoring that can be used to restructure models in order to improve model quality aspects but appreciably not influence the semantics of the model. Since every model refactoring comes along with initial and final pre-conditions which have to be checked before respectively after user input, a mapping of a certain model smell to some initial pre-conditions might be a hint for using the corresponding refactoring in order to eliminate the smell.

During model-based software development, the defined quality assurance process can be applied as shown on the right side of Figure 1. For a first overview, a report on model

metrics might be helpful. Furthermore, a model has to be checked against the existence (respectively absence) of specified model smells. Each model smell found has to be interpreted in order to evaluate whether it should be eliminated by a suitable model refactoring or not. It is recommended that the process is supported by appropriate tools as presented in the next section.

3.2 Example case

Here, we describe a small example case for the proposed project-specific model quality assurance process. Please note that due to space limitations we concentrate on one quality aspect only. We also do not specify each process implementation issue in detail.

The quality aspect we consider in our example is *consistency*. This quality aspect has several facets. We concentrate on inner consistency wrt. the modeling language used and wrt. modeling guidelines to be used. The modeling language comprises at least class models and state machines. Example questions can be:

- Are there any elements not shown in any diagram of the model?
- Are there any cycles in the element dependency graph?
- Are there any equally named classes in different packages?
- Are there any abstract classes that are not specialized by at least one concrete class?
- Are there any attributes redefining other ones within the same inheritance hierarchy?
- Are there any state diagrams without initial or final state?

These questions can lead to the definition of the following model smells, partially known from literature: (1) *element not shown in diagram*, (2) *dependency cycle* [Mar02], (3) *multiple definitions of classes with equal names* [Lan07], (4) *no specification* [Rie96], (5) *attribute name overridden*, and (6) *missing initial/final state* [RQZ07].

As already mentioned, there are at least two different ways to check a quality aspect by model smells. One alternative is to define a metric-based model smell which can be evaluated on the model. In our case study, we can use metrics *NOCS* (number of concrete subclasses of a given class), *NOIS* (number of initial states of a given state diagram region), and *NOFS* (number of final states of a given state diagram region) to address model smells (4) and (6), for example. If any of these metrics is evaluated to zero in given model contexts, the corresponding smell is identified. The second alternative to define model smells is to specify an anti-pattern representing a pattern which should not occur. It is defined based on the abstract syntax of the modeling language. In our case study we use anti-patterns *Equally named classes*, *No concrete subclass*, and *Redefined attribute* to address model smells (3) - (5). Note that model smell (4) can be specified in both ways: using model metric *NOCS* or anti-pattern *No concrete subclass*, respectively. Figure 4 shows anti-pattern *No specification*.

In order to eliminate specific model smells, corresponding model refactorings have to be specified. To address smells *Multiple definition of classes with equal names* and *No specification* of our example, we can use the well known model refactorings *Rename Class* and *Insert Subclass*, respectively. Of course, it is very important to consider all possible effects of a specific model refactoring ahead of its application.

4 Tool support

Since a manual model review is very time consuming and error prone, it is essential to automate the tasks of the proposed model quality assurance process as effectively as possible. Therefore, we implemented three tools supporting the included techniques metrics, smells, and refactorings (MSR) for models based on the Eclipse Modeling Framework (EMF) [EMF], a common open source technology in model-based software development.

Each tool consists of two independent components. The *generation module* addresses project managers respectively project staff who are responsible for the definition of the project-specific model quality assurance process. Project-specific metrics, smells, and refactorings are defined wrt. a specific modeling language specified by an EMF model. The second component of each tool supports the *application* of the specified MSR tools using the Java code produced by the generation module.

MSR tools can be specified by implementing metrics, smells and refactorings directly in Java or by using a model transformation tool such as Henshin [Hen], a new approach for in-place transformations of EMF models [ABJ+10]. Henshin uses pattern-based rules which can be structured into nested transformation units with well-defined operational semantics. We implemented a number of metrics, smells, and refactorings for EMF-UML models. In the following we shortly present the Eclipse plug-ins *EMF Metrics*, *EMF Smell*, and *EMF Refactor*.

4.1 EMF Metrics

Using the *EMF Metrics* prototype, metrics can be defined and calculated wrt. specific EMF-based models. When defining a new metric, the context of the metric has to be specified, i.e. the meta model given by its *NamespaceUri* as well as the model element type (e.g. *UML::Class*) to which the metric shall be applied. The prototype supports two distinct methods for defining new metrics. They are either defined directly by model transformations or two existing metrics are combined using an arithmetic operation.

Figure 2 shows the Henshin rule defining UML model metric *NOCS* (see previous section) specified on the abstract syntax of UML. *EMF Metrics* uses this rule to find matches in a concrete UML model. Starting from node *selectedEObject* of type *Class* being the context element, the Henshin interpreter returns the total number of matches that can be found in the model. This number represents the value of the corresponding model metric.

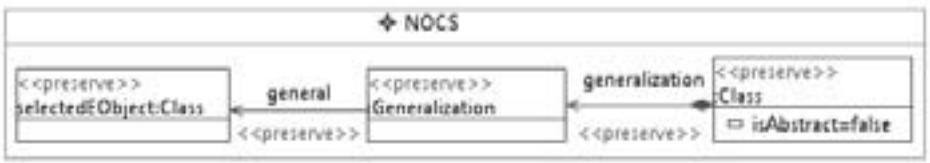


Figure 2: Henshin rule defining UML model metric *NOCS*

Figure 3 shows an example model and the corresponding result view after calculating five different metrics on abstract class *Vehicle*. This class owns four attributes *horsepower*, *seats*, *regNo*, and *owner*. Only the latter attribute has public visibility, so the AHF value of class *Vehicle* is evaluated to 0.75 . Each result contains a time stamp to trace metric values over time. For reporting purposes, *EMF Metrics* provides an XML export of its results.

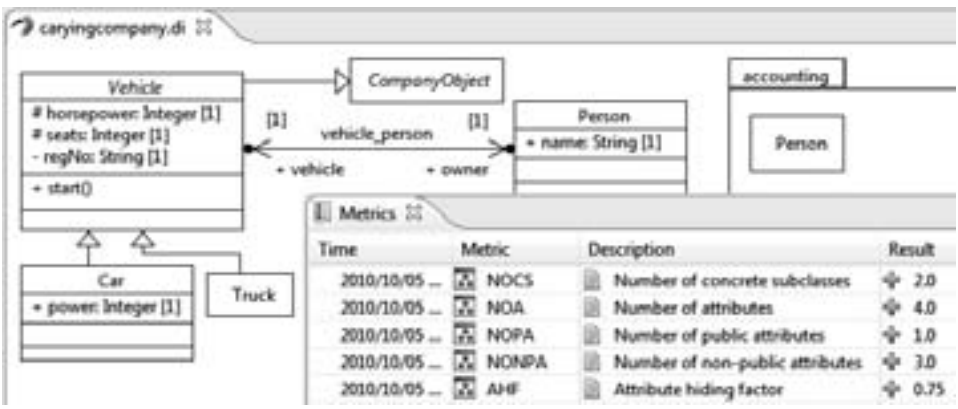


Figure 3: Model metrics result view after calculating five different metrics on abstract class *Vehicle*

4.2 EMF Smell

Similarly to *EMF Metrics*, *EMF Smell* consists of a generation and an application module. In the generation module, model smells can be specified using Henshin rules in order to define anti-patterns. There is no context that has to be specified because the anti-pattern has to be identified along the entire model. Of course, the modeling language has to be referred to. The definition of smells based on metrics is up to future work.

Figure 4 shows the Henshin rule for checking UML model smell *No Specification*. The pattern to be found specifies an abstract UML class (on the left) that is not specialized by a non-abstract UML class (on the right). Please note that parts of the pattern that are not allowed to be found are tagged with *forbid*. Additionally, parameter *modelElementName* is

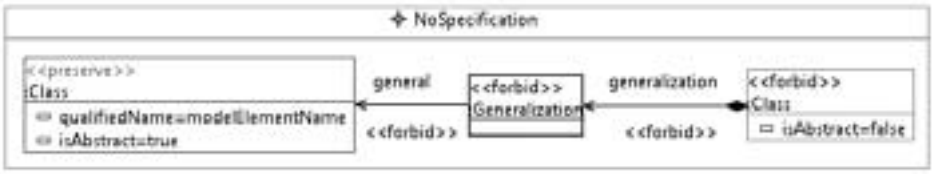


Figure 4: Henshin rule for checking UML model smell *No specification*

set by each pattern match to return model element instances participating in the identified model smell.

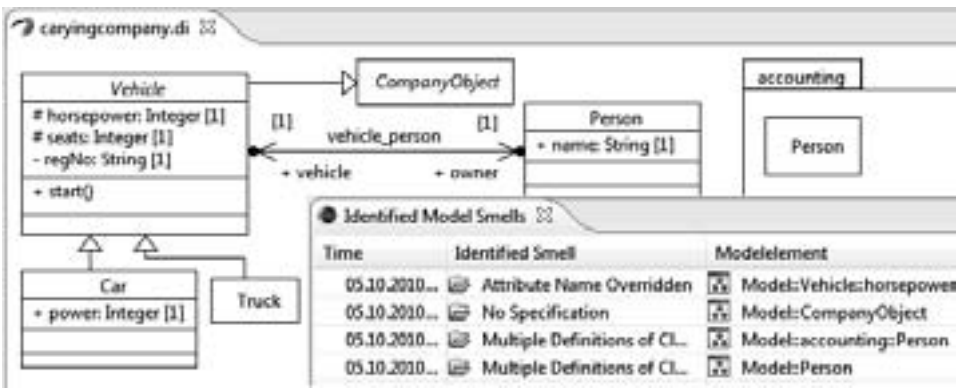


Figure 5: Model smell result view after checking the sample UML model

Figure 5 shows the application of the *EMF Smell* checking component on our example UML class model. The smell checking process can be triggered from within the context menu of the corresponding model file. Four smells have been found in the example model: There are two equally named classes *Person* while abstract class *CompanyObject* has no (direct) concrete subclass. Furthermore, attribute *horsepower* of class *Vehicle* is redefined by attribute *power* of class *Car*[1]. Like in *EMF Metrics*, model smells found are presented in a special view.

4.3 EMF Refactor

The third model quality assurance tool, *EMF Refactor* [Ref], is a new Eclipse incubation project in the Eclipse Modeling Project consisting of three main components. Besides a code generation module and a refactoring application module, it comes along with a suite

[1]This fact is not visible in the graphical view!

of predefined EMF model refactorings for UML and Ecore models. [2]

Since the application module uses the Eclipse Language Toolkit (LTK) technology [LTK], a refactoring specification requires up to three parts, implemented in Java and maybe generated from model transformation specifications, that have to be defined. They reflect a primary application check for a selected refactoring without input parameter, a second one with parameters and the proper refactoring execution.

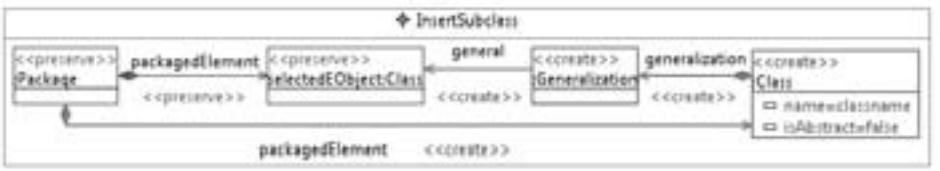


Figure 6: Henshin rule for executing UML model refactoring *Insert Subclass*

Figure 6 shows the Henshin rule for executing UML model refactoring *Insert Subclass*. The invocation context is given by node *selectedEObject* of type *Class*. The rule inserts a new non-abstract class named *classname* to the same package owning the selected class. Furthermore, the newly created class becomes a specialization of the selected class.

EMF Refactor provides two ways to configure the set of model refactorings. First, so-called refactoring groups are set up in order to arrange model refactorings for one modeling language. Second, these groups can be referenced by a specific Eclipse project to address project-specific needs.

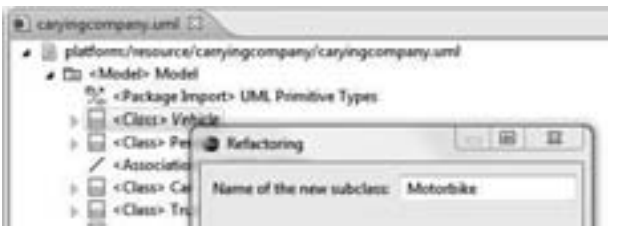


Figure 7: UML model refactoring *Insert Subclass* with parameter input

The application of a model refactoring mirrors the three-fold specification of refactorings based on LTK. After specifying a trigger model element such as class *Vehicle* in Figure 7, refactoring-specific initial conditions are checked. Then, the user has to set all parameters, for example the name of the new subclass in our *Insert Subclass* refactoring. *EMF Refactor* checks whether the user input does not violate further conditions. In case of erroneous parameters a detailed error message is shown. In our concrete example, it is checked whether the package owning the selected class already owns an element with the specified

[2] Of course, *EMF Metrics* and *EMF Smell* shall also come along with suites of predefined metrics and smells, respectively, in future.

name. If the final check has passed, *EMF Refactor* provides a preview of the changes that will be performed by the refactoring using EMF Compare as shown in Figure 8. Last but not least, these changes can be committed and the refactoring can take place.

Currently, *EMF Refactor* supports refactoring invocation from within three different kinds of editors: tree-based EMF instance editors (like in Figure 7), graphical model editors generated by Eclipse GMF (Graphical Modeling Framework), and textual model editors used by Xtext.

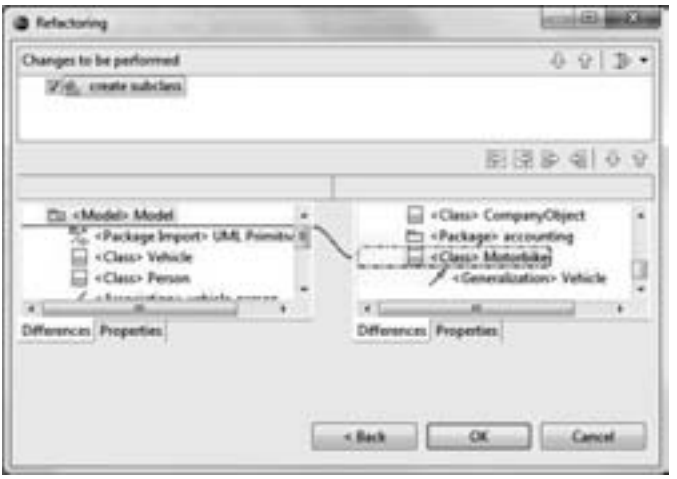


Figure 8: EMF Compare preview dialog of UML model refactoring *Insert Subclass*

5 Related work

Software quality assurance is concerned with software engineering processes and methods used to ensure quality. The quality of the development process itself can be certified by ISO 9000. CMMI [CMM] can help to improve existing processes. Considering the quality of a software product instead, the ISO/IEC 9126 standard lists a number of quality characteristics concerning functionality, reliability, usability, etc. Since models become primary artifacts in model-based software development, the quality of a software product directly leads back to the quality of corresponding software models.

There is already quite some literature on quality assurance techniques for software models available, often lifted from corresponding techniques for code. Most of the model quality assurance techniques have been developed for UML models. Classical techniques such as software metrics, code smells and code refactorings have been lifted to models, especially to class models. Model metrics have been developed in e.g. [GPC05] and model smells as well as model refactorings are presented in e.g. [SPLTJ01, MB08, Por03, MTR07, PP07]. In [Amb02], Ambler transferred the idea of programming guidelines to UML models.

Since EMF has evolved to a well-known and widely used modeling technology, it is worthwhile to provide model quality assurance tools for this technology. There are further tools for the specification of EMF model refactorings around as e.g. the Epsilon Wizard Language [KPPR07]. We compare our approach with other specification tools for EMF model refactorings in [AMST09]. To the best of our knowledge, related tools for metrics calculation and smells detection in EMF models are not yet available.

6 Conclusion

Since models become primary artifacts in model-based software development, model quality assurance is of increasing importance for the development of high quality software. In this paper, we present a quality assurance process for syntactical model quality being based on model metrics, model smells and model refactorings. The process can be adapted to specific project needs by first defining specific metrics, smells and refactorings and applying the tailored process to the actual models thereafter. Smell detection and model refactoring are iterated as long as a reasonable model quality has not reached.

As a next step, we plan to evaluate the proposed process as well as the presented tool-support in a bigger case study by integrating the tools in IBM's Rational Software Architect and applying the process on a large-scale model at Siemens.

There are model smells which are difficult to describe by metrics or patterns: For example, *shotgun surgery* is a code smell which occurs when an application-oriented change requires changes in many different classes. This smell can be formulated also for models, but it is difficult to detect it by analyzing models. It is up to future work to develop an adequate technique for this kind of model smells.

Furthermore, future work shall take complex refactorings into account which need to be performed in the right order depending on inter-dependencies of basic refactorings. Moreover, if several modelers refactor in parallel, it might happen that their refactorings are in conflict. In [MTR07], conflicts and dependencies of basic class model refactorings are considered and execution orders as well as conflict resolution are discussed.

As pointed out in Section 2, model consistency is not only a subject within one model but also between several models. Thus, refactorings have to be coordinated by several concurrent model transformations. A first approach to this kind of model transformations is presented in [JT09]. It is up to future work, to employ it for coordinated refactorings.

References

[ABJ+10] Thorsten Arendt, Enrico Biermann, Stefan Jurack, Christian Krause, and Gabriele Taentzer. Henshin: Advanced Concepts and tools for In-Place EMF Model Transformation. In *Model Driven Engineering Languages and Systems, 13th International Conference, MoDELS 2010. Proceedings*, LNCS, pages 121–135. Springer, 2010.

[Amb02] Scott W. Ambler. *The Elements of UML Style*. Cambridge University Press, 2002.

[AMST09] Thorsten Arendt, Florian Mantz, Lars Schneider, and Gabriele Taentzer. Model Refactoring in Eclipse by LTK, EWL, and EMF Refactor: A Case Study. http://www.modse.fr/modsemccm09/doku.php?id=Proceedings, 2009.

[BCR94] Victor Basili, Gianluigi Caldiera, and Dieter H. Rombach. The goal question metric approach. In J. Marciniak, editor, *Encyclopedia of Software Engineering*. Wiley, 1994.

[CMM] Capability Maturity Model Integration (CMMI). http://www.sei.cmu.edu/cmmi/.

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

[FHR08] Florian Fieber, Michaela Huhn, and Bernhard Rumpe. Modellqualität als Indikator für Softwarequalität: eine Taxonomie. *Informatik Spektrum*, 31(5):408–424, 2008.

[GPC05] M. Genero, M. Piattini, and C. Calero. A Survey of Metrics for UML Class Diagrams. *Journal of Object Technology*, 4(9):59 – 92, 2005.

[Hen] Henshin. http://www.eclipse.org/modeling/emft/henshin.

[JT09] Stefan Jurack and Gabriele Taentzer. Towards Composite Model Transformations using Distributed Graph Transformation Concepts. In Andy Schuerr and Bran Selic, editors, *MoDELS*, volume 5795 of *Lecture Notes in Computer Science*. Springer, 2009.

[KPPR07] Dimitrios S. Kolovos, Richard F. Paige, Fiona Polack, and Louis M. Rose. Update Transformations in the Small with the Epsilon Wizard Language. *Journal of Object Technology*, 6(9):53–69, 2007.

[Lan07] Christian F.J. Lange. *Assessing and Improving the Quality of Modeling: A series of Empirical Studies about the UML*. PhD thesis, Department of Mathematics and Computing Science, Technical University Eindhoven, The Netherlands, 2007.

[LTK] The Language Toolkit (LTK). http://www.eclipse.org/articles/Article-LTK.

[Mar02] Robert C. Martin. *Agile Software Development, Principles, Patterns, and Practices*. Prentice Hall, 1st edition, 2002.

[MB08] Slavisa Markovic and Thomas Baar. Refactoring OCL Annotated UML Class Diagrams. *Software and Systems Modeling*, 7:25–47, 2008.

[MTR07] Tom Mens, Gabriele Taentzer, and Olga Runge. Analysing refactoring dependencies using graph transformation. *Software and System Modeling*, 6(3):269–285, 2007.

[Por03] Ivan Porres. Model Refactorings as Rule-Based Update Transformations. In G. Booch P. Stevens, J. Whittle, editor, *Proc. UML 2003: 6th Intern. Conference on the Unified Modeling Language*, LNCS, pages 159–174. Springer, 2003.

[PP07] Alexander Pretschner and Wolfgang Prenninger. Computing Refactorings of State Machines. *Software and Systems Modeling*, 6(4):381–399, December 2007.

[Ref] EMF Refactor. http://www.eclipse.org/modeling/emft/refactor/.

[Rie96] Arthur J. Riel. *Object-Oriented Design Heuristics*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1996.

[RQZ07] C. Rupp, S. Queins, and B. Zengler. *UML 2 glasklar*. Hanser Fachbuchverlag, 2007.

[SPLTJ01] G. Sunye, D. Pollet, Y. Le Traon, and J. Jezequel. Refactoring UML models. In *Proc. UML 2001*, volume 2185 of *LNCS*, pages 134–148. Springer-Verlag, 2001.

Structural Equivalence Partition and Boundary Testing

Norbert Oster and Michael Philippsen
Computer Science Department, Programming Systems Group
University of Erlangen, Germany
[oster, philippsen]@cs.fau.de

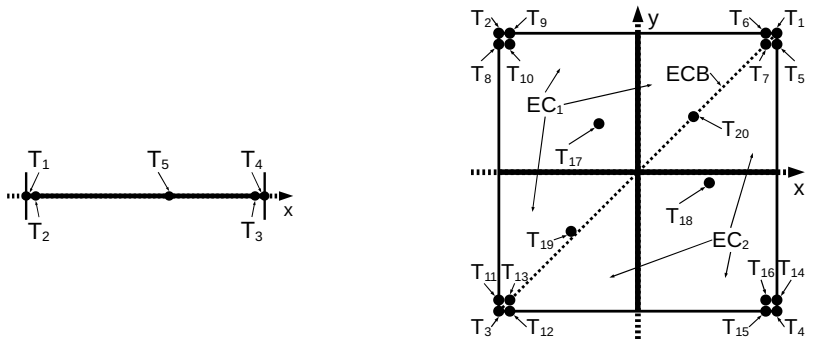
Abstract: Structural (manual or automated) testing today often overlooks typical programming faults because of inherent flaws in the simple criteria applied (e.g. branch or all-uses). Dedicated testing strategies that address such faults (e.g. mutation testing) are not specifically designed for smart automatic test case generation. In this paper we present a new coverage criterion and its implementation that accomplishes both: it detects more faults and integrates easily into automated test case generation. The criterion is targeted towards unveiling faults that originate from shifts in the equivalence classes that are caused by small coding errors (inspired by mutation testing). On benchmark codes from the Java-API and from an open-source project we improve the fault detection capability by up to 41% compared to branch and all-use coverage.

1 Introduction

Testing is still important in modern software engineering. Although remarkable progress has been made in the field of white-box (structural) testing, the average fault detection capability achieved with traditional coverage criteria (e.g. branch or all-uses) is still too low [Bis02, MMB03, Ost07], even if test case generators are used to produce test sets with high coverage ratios. The reason is that although programs often fail at the boundary of processing domains (e.g. due to primitive typo-like faults such as a < instead of a <= in a conditional expression), test cases designed for branch or all-uses coverage typically fail to detect such boundary faults. The test cases are *somewhere* on both sides of the boundary but not *at* the boundary. Even dataflow oriented testing (all-uses coverage) fails because it just checks the flow of information through the program withoug considering the *values* processed.

Mutation analysis (MA) is a better way to tell whether a test set is likely to find many bugs in the system under test (SUT), even *at* the borderline mentioned above [Lig02, OMK04]. For each test case MA compares the behavior of the SUT with that of its mutant, i.e. a copy with an artificially introduced fault. The ratio of mutants that show altered behavior under a test is called *mutation score* (MS) and is an objective indicator of the fault detection capability of this test set. The problem is that MA is not constructive, i.e. so far the idea of MA cannot be directly used to guide the automatic generation of good test cases.

In this paper we present a new coverage criterion called *Structural Equivalence Partition and Boundary Testing* (SEBT) that implements the insights of MA and that *can* be easily used in test case generators to produce better test sets than with traditional criteria.



(a) `x++`: unary, one dimension (b) `x < y`: binary, two dimensions

Figure 1: Unary and binary operator examples.

The paper is structured as follows. Section 2 introduces our new criterion SEBT. Section 3 describes the tool support available for SEBT. Section 4 shows the results we achieve with this approach. Section 5 discusses related approaches before the paper concludes in section 6 and gives an outlook on our future plans.

2 Structural equivalence partition and boundary criterion

In general, languages like JAVA provide unary (e.g. post-increment `a++` or boolean negation `!b`), binary (e.g. division `a/b` or logical "and" `a&&b`), and ternary (e.g. shortcut for value-returning if-then-else `a ? b : c`) operators. We further distinguish four operand classes according to their data type: *enumerable* with constant values (e.g. `boolean` with `true` and `false`); *discrete*, where neighboring elements always have a constant distance of 1 (e.g. `byte`, `char`, `short`, `int`, `long`); *pseudo-real*, where the values are discrete due to limited machine representation but the distances between the elements vary (e.g. `float`, `double`); and finally *references* to objects or `null`.

In the following, we will focus on discrete and pseudo-real operands of unary and binary operators. Since `a ? b : c` is the only ternary operator in JAVA and has special typing requirements, we discuss it on its own. In this paper we do not address reference type operands, as this topic is out of the scope of SEBT yet. Whenever at least one of the operands of an operator under consideration is `enum`erable, we require testing the corresponding statement with all possible values of that operand in combination with the classes of the other operands, as described next.

Fig. 1 shows two examples of the distribution of the equivalence classes and their boundaries for an unary and a binary operator. Since the expression `x++` depends on only one variable, its input domain in terms of testing is the whole domain of `x`'s data type. Its input domain is regarded as one equivalence class. If tested on its own, the expression should be evaluated with different values of `x`, covering both the equivalence class and its boundaries, as shown in Fig. 1(a). Test cases T_1 and T_4 represent the limits of the data type of `x`, similar to the *on-points* in [WC80]. For `double` in JAVA, T_4 means

`Double.MAX_VALUE` $\approx 2^{1024}$. Respectively, T_2 and T_3 are seen as the near-boundaries of the class, similar to the *off-points* in [WC80], and should be tested as well. The values to be chosen here for `x` should be those immediately next to T_1 resp. T_4 in terms of machine arithmetics. For practical applicability, the tester may provide a distance δ such that any value $T_2 \in]T_1, T_1 + \delta]$ and $T_3 \in [T_4 - \delta, T_4[$ would be acceptable. Finally, an arbitrary representant $T_5 \in]T_1 + \delta, T_4 - \delta[$ within the equivalence class should be tested as well.

Two equivalence classes and twenty test cases are reasonable for checking the expression `x < y`, as shown in Fig. 1(b). The dotted diagonal line represents the boundary of the two equivalence classes for the given expression: equivalence class EC_1 comprises all pairs (x, y) such that the condition `x < y` holds, while EC_2 represents the opposite. Test cases $T_1, T_2, \ldots, T_{16}$ represent the boundary or near-boundary values of the data types of both variables. Additionally, arbitrary inner representants (here: T_{17} and T_{18}) from within each class are required, that are not a (near-)boundary input at the same time. The statement `x < y` must be tested with near-boundary values on "both sides" of each equivalence class as well. In Fig. 1(b), the latter is achieved by the test cases T_{19} and T_{20} – where T_{20} represents pairs (x, y) satisfying the condition $x = y$ and thus the boundary of EC_2 towards EC_1 (a generic formal definition of both is given in section 2.1). In contrast to [WC80], we do not distinguish *on-points* and *off-points* for the boundary between classes, since we expect boundary and near-boundary test cases for all classes, thus implicitly always requiring both kinds of "points".

In the case of binary operators, we will distinguish between relational (i.e. `<`, `<=`, `=`, `!=`, `>=`, `>`) and arithmetic (i.e. `+`, `-`, `*`, `/`, `%`, `^`, ...) expressions. For relational operators we require adequate test cases such that the operands are evaluated to values identical to or immediately at the limits imposed by their data type, e.g. $T_1 - T_{16}$ in Fig. 1(b). Arithmetic operators put additional restrictions on the operands, that might be of higher interest for testing: the boundaries of the *result*. As an example, consider a simple addition like `x + y`. In the visualization of this (binary) operator according to Fig. 1(b), test cases T_1 and $T_5 - T_7$ (respectively T_3 and $T_{11} - T_{13}$) will trigger overflows (resp. underflows). Thus, new boundaries for reasonable equivalence classes are imposed by the conditions $x + y < t^{x,y}_{min}$ or $x + y > t^{x,y}_{max}$, where $t^{x,y}_{min}$ and $t^{x,y}_{max}$ are the boundaries of the resulting data type after implicit conversion [GJSB05].

2.1 Generic SEBT criterion

Based on the examples above we can now formally define the SEBT criterion.

Unary operators: Let *op* be an unary operator applied to an expression (e.g. variable) v of type t^v and $\overline{op} \neq op$ a type-compatible operator, which can syntactically replace *op*. Further let $\delta^v > 0$ be the minimum distance between the lowest possible value t^v_{min} of v according to t^v and the smallest value $t^v_{min} + \delta^v$, such that $]t^v_{min}, t^v_{min} + \delta^v[= \emptyset$; respectively let $\epsilon^v > 0$ be the minimum distance between the highest possible value t^v_{max} of v according to t^v and the greatest value $t^v_{max} - \epsilon^v$, such that $]t^v_{max} - \epsilon^v, t^v_{max}[= \emptyset$. For practical feasibility, δ^v and ϵ^v may be relaxed to arbitrary but adequate distances chosen

by the tester in accordance with the SUT.

SEBT requires that for a given statement v *op* or *op* v, the expression must be executed at least once with each of the following test cases: $v_1 = t^v_{min}$, $v_2 \in]t^v_{min}, t^v_{min} + \delta^v]$, $v_3 \in [t^v_{max} - \epsilon^v, t^v_{max}[$, $v_4 = t^v_{max}$, and for each compatible $\overline{op}$ with at least one test case $v_5 \in]t^v_{min} + \delta^v, t^v_{max} - \epsilon[$ such that the expression behaves differently (e.g. for arithmetic expressions: *op* $v_5 \neq \overline{op}\ v_5$). Example: `-1` $\neq$ `+1` (but `-0` vs. `+0` is *not* enough).

Binary operators: Let *op* be a binary operator applied to two expressions (e.g. variables) a of type t^a respectively b of type t^b returning a result of type $t^{a,b}$ and $\overline{op} \neq op$ a type-compatible operator, which can syntactically replace *op*. Further let δ^a, δ^b, $\delta^{a,b}$, t^a_{min}, t^b_{min}, $t^{a,b}_{min}$, ϵ^a, ϵ^b, $\epsilon^{a,b}$, t^a_{max}, t^b_{max}, $t^{a,b}_{max}$ have the same meaning for a, b, and the result as in the definition above for v in unary expressions. Additionally, let $\xi^a_{(a)}$ ($\xi^b_{(b)}$) be the smallest distance between the current value of variable a (b) and the values immediately next to it according to the machine precision with respect to the data type t^a (t^b) – please note that in JAVA, ξ is *not* necessarily constant, e.g. $\xi^{double}_{(x)} \in [5 \cdot 10^{-324}, 10^{292}]$.

SEBT requires that for a given statement a *op* b and each compatible $\overline{op}$, the expression must be executed with at least one test case from each of the generic test classes determined by the conditions in Table 1 (upper and middle part), which is based on abbreviations defined in Table 2, distinguishing between relational and arithmetic operators. For the example `x < y`, twenty test cases satisfying SEBT are depicted graphically in Fig. 1(b).

Ternary operators: The conditional operator `a ? b : c` in JAVA requires `a` to be of type `boolean`. A coding error may only result from interchanging the three operands, if all three are of the same type – or at least the second and third operand, which must be of compatible type anyway. In order to detect such programming faults, we take advantage of the idea of *modified condition/decision coverage (MC/DC)* [Lig02], as described in the following. Let t be an arbitrary test case and $v_t(a)$ (respectively $v_t(b)$ and $v_t(c)$) be the (hypothetical) evaluation result of operand `a` (resp. `b` and `c`) when executing the statement under test with test case t. Please note that due to short-cut evaluation in JAVA, at least one of the operands is not evaluated each time the expression is executed.

SEBT requires that for a given statement `a ? b : c`, the test set must contain at least one pair (t_1, t_2) of test cases for each of the generic test classes in Table 1 (lower part).

2.2 Special consideration: Feasibility

When executing program logics, *feasibility* may prevent certain structural entities (e.g. def/use-pairs) from being covered, although required by the chosen criterion. Often the reason is that no input (no test case) can be found, that reaches certain nodes or edges of the control flow graph. Depending on the SUT, the generic SEBT criterion defined above may also be affected by such infeasibility for certain test classes. For example, consider the pair *op* := "<" and $\overline{op}$:= ">=" of arithmetic operators. No input exists that covers the generic test classes T_{17}, T_{19}, and T_{20}, because the equivalence class EC_1 is empty, i.e. $\nexists x, y : x<y \wedge x>=y$. Any tool implementation of SEBT must account for such combinations to compute a reasonable coverage measure.

Table 1: Generic test classes (See Table 2 for the legend).

	Class	Description	Conditions on operands
for ***binary relational*** *operators*	T_1	limit (I)	$a = t^a_{max} \wedge b = t^b_{max}$
	T_2	limit (II)	$a = t^a_{min} \wedge b = t^b_{max}$
	T_3	limit (III)	$a = t^a_{min} \wedge b = t^b_{min}$
	T_4	limit (IV)	$a = t^a_{max} \wedge b = t^b_{min}$
	T_5	near-limit (I)	$a = t^a_{max} \wedge b \in]t^b_{max}, t^b_{max} - \epsilon^b]$
	T_6	near-limit (I)	$a \in]t^a_{max}, t^a_{max} - \epsilon^a] \wedge b = t^b_{max}$
	T_7	near-limit (I)	$a \in]t^a_{max}, t^a_{max} - \epsilon^a] \wedge b \in]t^b_{max}, t^b_{max} - \epsilon^b]$
	T_8	near-limit (II)	$a = t^a_{min} \wedge b \in]t^b_{max}, t^b_{max} - \epsilon^b]$
	T_9	near-limit (II)	$a \in]t^a_{min}, t^a_{min} + \delta^a] \wedge b = t^b_{max}$
	T_{10}	near-limit (II)	$a \in]t^a_{min}, t^a_{min} + \delta^a] \wedge b \in]t^b_{max}, t^b_{max} - \epsilon^b]$
	T_{11}	near-limit (III)	$a = t^a_{min} \wedge b \in]t^b_{min}, t^b_{min} + \delta^b]$
	T_{12}	near-limit (III)	$a \in]t^a_{min}, t^a_{min} + \delta^a] \wedge b = t^b_{min}$
	T_{13}	near-limit (III)	$a \in]t^a_{min}, t^a_{min} + \delta^a] \wedge b \in]t^b_{min}, t^b_{min} + \delta^b]$
	T_{14}	near-limit (IV)	$a = t^a_{max} \wedge b \in]t^b_{min}, t^b_{min} + \delta^b]$
	T_{15}	near-limit (IV)	$a \in]t^a_{max}, t^a_{max} - \epsilon^a] \wedge b = t^b_{min}$
	T_{16}	near-limit (IV)	$a \in]t^a_{max}, t^a_{max} - \epsilon^a] \wedge b \in]t^b_{min}, t^b_{min} + \delta^b]$
	T_{17}	representant (EC_1)	$C_0 \wedge C_1 \wedge C_2$
	T_{18}	representant (EC_2)	$C_0 \wedge C_3 \wedge C_4$
	T_{19}	boundary (EC_1)	$C_0 \wedge C_1 \wedge \neg C_2$
	T_{20}	boundary (EC_2)	$C_0 \wedge C_3 \wedge \neg C_4$
for ***binary arithmetic*** *operators*	T_1	underflow (I)	$a \; op \; b < t^{a,b}_{min} - \delta^{a,b}$
	T_2	overflow (II)	$a \; op \; b > t^{a,b}_{max} + \epsilon^{a,b}$
	T_3	near-limit (Ia)	$a \; op \; b \in [t^{a,b}_{min} - \delta^{a,b}, t^{a,b}_{min}[$
	T_4	near-limit (IIa)	$a \; op \; b \in]t^{a,b}_{max}, t^{a,b}_{max} + \epsilon^{a,b}]$
	T_5	limit (I)	$a \; op \; b = t^{a,b}_{min}$
	T_6	limit (II)	$a \; op \; b = t^{a,b}_{max}$
	T_7	near-limit (Ib)	$a \; op \; b \in]t^{a,b}_{min}, t^{a,b}_{min} + \delta^{a,b}]$
	T_8	near-limit (IIb)	$a \; op \; b \in [t^{a,b}_{max} - \epsilon^{a,b}, t^{a,b}_{max}[$
	T_9	representant (EC_1)	$C_0 \wedge C_1 \wedge C_2 \wedge C_5$
	T_{10}	representant (EC_2)	$C_0 \wedge C_3 \wedge C_4 \wedge C_5$
	T_{11}	boundary (EC_1)	$C_0 \wedge C_1 \wedge \neg C_2 \wedge C_5$
	T_{12}	boundary (EC_2)	$C_0 \wedge C_3 \wedge \neg C_4 \wedge C_5$
ternary	T_1	a true, vary b only	$a = true \wedge v_{t_1}(b) \neq v_{t_2}(b) \wedge v_{t_1}(c) = v_{t_2}(c)$
	T_2	a false, vary c only	$a = false \wedge v_{t_1}(b) = v_{t_2}(b) \wedge v_{t_1}(c) \neq v_{t_2}(c)$
	T_3	vary a, keep b and c with b ≠ c	$v_{t_1}(a) \neq v_{t_2}(a) \wedge v_{t_1}(b) \neq v_{t_1}(c) \wedge$ $v_{t_1}(b) = v_{t_2}(b) \wedge v_{t_1}(c) = v_{t_2}(c)$

Table 2: Abbreviations of conditions used in Table 1.

Abbrev.	Description	Condition on operands
C_0	non-limit	$a \in]t^a_{min} + \delta^a, t^a_{max} - \epsilon^a[\wedge b \in]t^b_{min} + \delta^b, t^b_{max} - \epsilon^b]$
C_1	same behavior	$a \; op \; b = a \; \overline{op} \; b$
C_2	same behavior	$\forall \psi^a \in \{-\xi^a_{(a)}, 0, \xi^a_{(a)}\}, \forall \psi^b \in \{-\xi^b_{(b)}, 0, \xi^b_{(b)}\}$:
	(at neighborhood)	$(a + \psi^a) \; op \; (b + \psi^b) = (a + \psi^a) \; \overline{op} \; (b + \psi^b)$
C_3	different behavior	$a \; op \; b \neq a \; \overline{op} \; b$
C_4	different behavior	$\forall \psi^a \in \{-\xi^a_{(a)}, 0, \xi^a_{(a)}\}, \forall \psi^b \in \{-\xi^b_{(b)}, 0, \xi^b_{(b)}\}$:
	(at neighborhood)	$(a + \psi^a) \; op \; (b + \psi^b) \neq (a + \psi^a) \; \overline{op} \; (b + \psi^b)$
C_5	non-under/overflow	$a \; op \; b \in]t^{a,b}_{min} + \delta^{a,b}, t^{a,b}_{max} - \epsilon^{a,b}[\wedge$
	(non-limit)	$a \; \overline{op} \; b \in]t^{a,b}_{min} + \delta^{a,b}, t^{a,b}_{max} - \epsilon^{a,b}[$

3 Tool support

Regardless of the testing strategy, software testing today is hardly possible without adequate tool support. There is a broad variety of approaches to automatic test case generation. Random test data generators produce myriads of redundant test cases, covering the same

entities of the control or data flow several times [MMS98, BHJT00]. Other techniques try to remove redundant test cases *after* their random generation, thus giving smaller test sets [OW91, Ton04, McM04]. But the general problem remains NP-complete.

Heuristic test case generators [WBP02, OS06, PBSO08, FZ10] are usually driven by one or more *objective functions*. Those objectives compute quantifiable characteristics such as the structural coverage achieved (e.g. according to the branch criterion) or the "distance" [Bar00] of an individual test case from covering a certain test goal (e.g. a statement not yet executed). In general, such a generator first instruments the SUT once. It then randomly generates a set of test cases. Each test case is executed and evaluated according to each objective. The results of the evaluation are then used to guide the heuristics towards generating new and better test cases (or test sets) based on the old ones. The cycle of generation, execution, and evaluation is repeated until the test set is considered good enough.

An important characteristic of SEBT is that this coverage criterion can be implemented as such an objective function plug-in for arbitrary heuristic test case generators. Hence SEBT makes it possible to generate test cases that achieve better mutation scores than traditional coverage criteria and that hence help build more trust in the test. We have implemented the SEBT criterion for JAVA as such a plug-in. It provides hooks for *instrumentation*, *execution*, and *evaluation*. The three hooks of the SEBT plug-in work as follows.

Instrumentation: We parse the source code and construct an abstract syntax tree. In a transformation we then insert so-called probes into the given source code that do not modify the semantics of the original program. The probes are calls to a logging subsystem. The instrumentation tool logs each transformation in a so-called Static Logging Data (SLD) file that contains information on the instrumented statements: a unique identifier and the kind of the occurring operator. As an example, consider an excerpt from the `Dijkstra` benchmark (section 4):

```
while (nodesToProcess.size() > 0) {
    Node nextNodeToProcess = getNodeWithSmallestDistance();
    nodesToProcess.remove(nextNodeToProcess);
    Vector neighbours = nextNodeToProcess.getNeighbours();
    for (int i = 0; i < neighbours.size(); i++) {
        Node neighbour = (Node)neighbours.get(i);
        if (neighbour.getCostFromRoot() >
                nextNodeToProcess.getCostFromRoot()
                + nextNodeToProcess.getCostToNeighbour(neighbour)) {
            neighbour.setPredecessor(nextNodeToProcess);
```

The automatic instrumentation gives the following (pretty-printed) code:

```
while (logger.my_gt_function("12_1_4", nodesToProcess.size(), 0)) {
    Node nextNodeToProcess = getNodeWithSmallestDistance();
    nodesToProcess.remove(nextNodeToProcess);
    Vector neighbours = nextNodeToProcess.getNeighbours();
    for(int i=0; logger.my_lt_function("12_1_5", i, neighbours.size());
            i = logger.my_post_inc_function ("12_1_6", i)) {
        Node neighbour = (Node)neighbours.get(i);
        if (logger.my_gt_function("12_1_8", neighbour.getCostFromRoot(),
                logger.my_plus_function ("12_1_7" ,
                    nextNodeToProcess.getCostFromRoot(),
                    nextNodeToProcess.getCostToNeighbour(neighbour)))) {
            neighbour.setPredecessor(nextNodeToProcess);
```

Furthermore, it adds corresponding entries to the SLD file, shown in Fig. 2(left).

	12_1_8;GT;double;double;1.797...157E308;1.797...157E308
12_1_4;GT	12_1_8;GT;double;double;1.797...157E308;2.0
12_1_5;LT	12_1_8;GT;double;double;2.0;2.0
12_1_6;POST_INC	12_1_8;GT;double;double;5.0;1.797...157E308
12_1_7;PLUS	12_1_8;GT;double;double;5.0;10.0
12_1_8;GT	12_1_8;GT;double;double;0.0;4.9E-324

Figure 2: Excerpts from the SLD (left) and DLD files of the `Dijkstra` example.

Although we did not execute any test case yet, the plug-in derives from statically analyzing those lines of the SLD, that the code excerpt contains two occurrences of the operator ">" (GT). Since we must cover 20 test case classes (see Table 1) for each feasible pair of $op :=$ ">" and any compatible $\overline{op}$, this leads to well-defined 74 different operand combinations, in order to fully satisfy SEBT for the two probes with the IDs `12_1_4` and `12_1_8`. Of course, the other 41 IDs in the SLD of the entire code must be considered accordingly.

Execution: The transformed code is executed for each test case. This results in a Dynamic Logging Data (DLD) file for each test run. Whenever a probe is executed, the unique ID, the type of the operator, the runtime types, and the actual values of each evaluated operand are logged. An excerpt from a DLD, containing some of the essential entries for the operator ID `12_1_8` is shown in Fig. 2(right).

Evaluation: Finally, all current SLDs and DLDs are considered in order to determine the coverage ratio achieved. Even if some relevant statements remain totally uncovered by the test set (and thus do not show up in the DLD), the SLD provides the necessary information to account for them as well. From the runtime types of the operands in the DLD the tool can determine the limits t_{min} and t_{max} of the data-types of each individual operand (see section 2.1, Table 1). Merging the knowledge from the above SLD and DLD excerpts for ID `12_1_8`, we notice that $t^a_{max} = t^b_{max} = t^{double}_{max} = 1.797 \ldots 157E308$ and $\xi^{double}_{(0)} = 4.9E - 324$. Among others, we covered the test classes T_1 (1^{st} line of DLD); T_{20} (3^{rd} line of DLD), T_{17} (5^{th} line), and T_{19} (6^{th} line) for $op :=$ ">" vs. $\overline{op} :=$ ">="; as well as many other test classes for further compatible $\overline{op}$ at the same time.

From this evaluation, the plug-in reports to the driving heuristic test case generator which SEBT classes T^u_i have not been covered yet. Moreover, for each pair of test case t_j and test class T^u_i, the plug-in also provides a means to compute the "distance" of t_j from covering T^u_i, based on a distance metrics from graph theory applied to the control flow graph plus a set of distance functions for conditional expressions applied to branch predicates (see [Bar00] for details). The test case generator can than choose the test class T^u_s with the smallest distance as the new test target and can iteratively evolve an adequate subset of the current population of test cases t_j towards covering T^u_s. In case of a test generator that is based on a genetic algorithm, the distance objective can serve as the fitness value for the selection operator. If T^u_s can be covered by a test case t^s_j within a predefined number of steps, then t^s_j is added to the set of resulting test cases. This is done for all test targets separately.

In the above example, the DLD does not contain an entry covering T_2, but the nearest test case achieves operand values of `5.0;1.797...157E308`. The heuristics is now guided to evolve the current value of the first operand (`5.0`) towards the smallest possible value for the datatype `double`, thus covering T_2.

Table 3: Benchmark results

Project	*# of Java classes*	*lines of code*	*source size (bytes)*	*# of* mutants	*number of killed mutants and mutation scores in %*						*improvement*	
					branch (B)		*all-uses* (A)		*SEBT* (S)		*B→S*	*A→S*
Hanoi	1	38	1279	227	170	75%	174	77%	197	87%	27	23
JDKsort	1	82	2639	852	524	62%	557	65%	577	68%	53	20
Dijkstra	2	141	4080	220	155	70%	158	72%	206	94%	51	48
Huffman	2	298	8931	623	390	63%	390	63%	391	63%	1	1
BigFloat	3	540	17526	1528	1057	69%	1159	76%	1494	98%	437	335
					provided JUnit tests (PJT)				*PJT+SEBT*		*by SEBT*	
JTopas	44	16112	583546	3219	1518		47.2%		1854	57.6%	336	

improvement : Number of faults detected with SEBT but *missed* with branch/all-uses/provided JUnit test cases

Targeting each test class on its own results in intractably large test sets, since such approaches strive for maximized coverage without considering the validation effort in terms of the resulting total number of test cases. SEBT can also be used to guide test case generators that do take the size of the test set into account, e.g. by adding another global optimization phase to the generation cycle described above. For instance, in its global optimization phase, our tool •gEAr [Ost07] works on a *set of test sets*, i.e. not just on a single test set. Each of these test sets is evaluated with respect to its size and the coverage ratio computed by the SEBT plug-in. On these weighted test sets •gEAr applies a genetic algorithm (with cross-over between test sets and mutation within test sets) to construct smaller test sets with similar (or better) SEBT-coverage. Only if this global optimization of the test sets cannot rationally improve the SEBT-coverage or the test set size anymore, i.e., if for example a certain test class T_s^u remains uncovered within an upper number of iterations, •gEAr uses the mechanism described above to evolve those existing test cases that are close with respect to SEBT towards covering T_s^u. The SEBT-plug-in is hence used in both phases that •gEAr repeats iteratively until a test set is found that not just has a good coverage with respect to SEBT but that is also small.

4 Experimental results

To evaluate the power of SEBT, we conducted several experiments in two different setups. The characteristics of the SUTs used and the results achieved are shown in Table 3. Mutation analysis is more adequate to assess the quality of a test set in terms of its fault detection capability, than the mere coverage measure. Killing a mutant means that the corresponding test case is sensitive to and thus able to detect a certain (potential) fault. Additionally, MA is independent of the coverage criterion actually applied to derive the test set. For each SUT, we automatically generated mutants with MuJava [OMK04], applying *all* mutation operators provided by the tool (including class mutation). These sets of mutants may contain functionally equivalent programs.

Setup A (upper part of Table 3): The SUTs used in this setup are textbook examples, such as Dijkstras shortest path algorithm. `JDKsort` is taken from the JAVA-API. `BigFloat` is a simplified variant of java.math.BigDecimal. We generated three test sets for each SUT,

one achieving 100% branch coverage, the second satisfying the all-uses criterion. The third test set has covered the feasible test classes of SEBT for relational and arithmetic operators. For all three groups of test sets, we determined the number of mutants killed. Although the all-uses criterion is quite demanding, the average fault detection (represented by the mutation score) achieved in setup A is about 70.7%; test sets satisfying branch coverage were even weaker and reach approx. 66.6%. The test sets for SEBT performed best and detected an average of 83.0% of the potential faults.

In our benchmarks, we apply SEBT for arithmetic and relational statements. Thus the criterion performs best on SUTs whose behavior mainly relies on such operators and the pure test values, i.e., control and data flow strongly depend on the actual input, rather than on the internal state with weak or no input dependence. Because the `Huffman` algorithm represents straightforward encoding, the input values themselves are not directly processed by arithmetic or relational operations. Instead of the input values, the number of occurrences of different symbols in the input is computed and used to guide the control flow. Since the establishing of almost all SEBT-relevant data is infeasible (e.g. the total input length must always be a small but positive integer due to memory limitations and the JAVA-API specification), SEBT killed only one additional mutant – branch and all-uses perform equally bad. For `BigFloat` however, SEBT has killed 437 ($+29\%$) mutants, that were missed by branch coverage. This is due to the intense number crunching of `BigFloat`, that applies arithmetic and relational operators immediately to the input values. Since SEBT can execute the relevant statements with almost all operand configurations, the resulting fault detection of 98% is impressive.

Setup B (lower part of Table 3): The SUT in this setup is the entire "Java tokenizer and parser tools" package `JTopas` (`http://jtopas.sourceforge.net/jtopas`), comprising 44 JAVA classes with currently 16112 lines of open source code. The `JTopas` distribution comprises a set of 552 JUnit test cases. We supplemented this given set with 584 additional JUnit test cases, that cover the test classes of SEBT.

The JUnit test cases provided with `JTopas` killed 1518 (47.2%) out of 3219 mutants, while the improved test set achieved a mutation score of 1854 (57.6%) – i.e. a total of 336 ($+10.4\%$) mutants, that were *not* detected by the original JUnit tests, are now killed by the SEBT test cases. Hence, SEBT significantly improves the chance of fault detection in an open source project with existing JUnit test cases.

5 Related work

Many different testing strategies and test automation tools exist, but there are very few approaches to automatic test data generation for more demanding testing criteria, and almost all are academic. Test data for structural coverage is typically derived by applying so-called program slicing [FB97].

Static slicing [WC80, KLPU04] explicitly enumerates all control or data flow paths required (e.g. directly for path coverage or indirectly for statement coverage), i.e. all constraints imposed by the conditions at the branching nodes are collected for each path in

part. The resulting constraint system in terms of the input variables represents the domain partitioning of the program with respect to individual control flow paths. Compared to the correct version of the code, faults in the program are expected to either shift some "domain boundary" of the solved constraint system (follow wrong branch for a certain input) or modify some "domain computation" (give wrong output despite following the correct path). Since statically setting up the corresponding constraint system is very expensive or even infeasible (for loops the number of paths is infinite), those approaches have many restrictions. White and Cohen [WC80] introduce their technique for a simplified Algol-like programming language that lacks many features of modern object-oriented languages. Moreover, they make restrictive assumptions with respect to the types of constraints and the input space (only linear borders between domains are allowed). Kosmatov, Legeard et al. [KLPU04] apply the concept to formal models of the SUT. In case of state machines, the predicate conditions (i.e. guards in state transitions) are collected to define the input domains for each modeled behavior. Their approach applies to discrete domains for (simple) formal model languages only, and must also fail for systems with unknown numbers of loop iterations. They do not address source code testing at all. In contrast to the techniques presented above our SEBT criterion is intended for arbitrary imperative or object-oriented languages, and the prototypical implementation for JAVA does not exclude any language feature. In order to cope with the limits of static analysis and to ease the automatic generation of corresponding test cases, SEBT allows generators to consider individual code statements only instead of reckoning complete program paths. Inspired by the concept of *weak* mutation testing [OMK04], SEBT is satisfied with test cases that execute individual statements with manifold data contexts, able to trigger possibly faulty behavior at each program location (e.g. < vs. <=) in part.

In general, dynamic program slicing [Bar00, WBP02, FZ10] executes the SUT with a random test case first. If the desired path has been traversed, the corresponding input is directly available. Otherwise, some heuristics are applied to push the currently covered path towards the desired one, usually by repeatedly applying gentle modifications to the best input known so far. Many approaches of this kind start with a static control flow analysis in order to identify the target entities to be covered according to the coverage criterion. Such techniques suffer from the same drawbacks as static slicing (e.g. infinite number of paths) and are therefore only applicable to simple criteria based on control flow such as branch coverage. Our new criterion SEBT can serve as a driving plug-in for the approach by Wegener et al. [WBP02], but as mentioned above targeting each entity in part will likely result in intractably huge test sets for average programs.

Other techniques collect the test targets "on the fly" during the heuristic optimization and explicitly target missed entities without prior analysis of the entire control flow. Fraser and Zeller [FZ10] reduce the testing effort by minimizing the length of each test case, i.e. roughly the number of statements in the test sequence. In contrast, with SEBT in •gEAr the tester can select between short but rather many test cases similar to [FZ10] and a minimized number of test cases, i.e. a small test set. In the latter case, each test run covers several combinations of SEBT classes at once. Moreover, [FZ10] uses a sophisticated impact analysis of each mutant execution, in order to assess to which extent a mutant has been covered: The more statements are covered in the original code but not in the mutant

and vice versa and the more methods behave differently (i.e. return different values or throw unexpected exceptions) in the mutant compared to the original code, the higher is the impact of this mutant under an assessed test case. If a test case does not cause enough impact of the mutant under assessment (e.g. due to failure masking in the early testing stages) the test case is quite likely dismissed during the generation of the final test set. Nevertheless, such a test case may become important during regression testing. In contrast, SEBT requires to initially cover all operand-value-combinations expected to cause failures (sooner or later).

6 Conclusion and outlook

In this paper, we presented a new testing technique named *structural equivalence partition and boundary testing (SEBT)* based on the notion of mutation testing. The goal of SEBT is to help the tester to systematically choose test cases sensitive to typical coding errors, such as unintentionally using the wrong operator (e.g. "$<$" instead of "$\leq$") or not accounting for limits (over-/underflow) and limitations (machine precision) of the data-types used. The tool computes the coverage achieved by a given test set for the code portion under test according to SEBT. Experimental results are very promising: SEBT outperforms branch coverage by 41% and all-uses coverage by 30% (w.r.t. the mutation score achieved).

The SEBT criterion is primarily meant for classical operators like arithmetical $(+, -, *, /)$ and logical $(\&\&, ||, !)$ ones. Nevertheless, the key idea can easily be extended to more sophisticated language features. We already examined its applicability to coding errors concerning arrays, general computation failures due to rounding, inheritance, polymorphism, method overloading, and to data flow aspects [Fri09]. Applying SEBT to multi-threading faults is still an open and challenging task.

References

[Bar00] A. Baresel. Automatisierung von Strukturtests mit evolutionären Algorithmen. Diploma thesis, FG Softwaretechnik, Humboldt-Universität Berlin, July 2000.

[BHJT00] B. Baudry, Vu Le Hanh, J.-M. Jézéquel, and Y. Le Traon. Building Trust into OO Components Using a Genetic Analogy. In *ISSRE 2000: Proc. 11th Intl. Symp. Software Reliability Engineering*, pages 4–14, Washington, DC, Oct. 2000.

[Bis02] P. G. Bishop. Estimating Residual Faults from Code Coverage. In *21st Intl. Conf. Computer Safety, Reliability and Security*, volume 2434 of *LNCS*, pages 163–174, Catania, Italy, Sep. 2002.

[FB97] I. Forgács and A. Bertolino. Feasible test path selection by principal slicing. *SIGSOFT Software Engineering Notes*, 22(6):378–394, Nov. 1997.

[Fri09] S. Fritz. Konzeption und Implementierung eines Verfahrens zur strukturellen Äquivalenzklassen- und Grenzwerttestüberdeckung. Diploma thesis, Friedrich-Alexander-Universität Erlangen-Nürnberg, CS Dept., Feb. 2009.

[FZ10] G. Fraser and A. Zeller. Mutation-driven generation of unit tests and oracles. In *ISSTA '10: Proc. 2010 ACM SIGSOFT Intl. Symp. Software Testing and Analysis*, pages 147–158, Trento, Italy, July 2010.

[GJSB05] J. Gosling, B. Joy, G. Steele, and G. Bracha. *Java(TM) Language Specification*, chapter Conversions and Promotions. Addison Wesley, 3 edition, June 2005.

[KLPU04] N. Kosmatov, B. Legeard, F. Peureux, and M. Utting. Boundary Coverage Criteria for Test Generation from Formal Models. In *ISSRE '04: Proc. 15th Intl. Symp. Software Reliability Engineering*, pages 139–150, Saint-Malo, France, Nov. 2004.

[Lig02] P. Liggesmeyer. *Software-Qualität – Testen, Analysieren und Verifizieren von Software*. Spektrum - Akademischer Verlag, Heidelberg; Berlin, 2002.

[McM04] P. McMinn. Search-based software test data generation: A survey. *Software Testing, Verification and Reliability*, 14(2):105–156, June 2004.

[MMB03] M. M. Marré and A. Bertolino. Using spanning sets for coverage testing. *IEEE Trans. Software Engineering*, 29(11):974–984, Nov. 2003.

[MMS98] G. McGraw, C. Michael, and M. Schatz. Generating Software Test Data by Evolution. Technical Report RSTR-018-97-01, RST Corporation, Sterling, VA, Feb. 1998.

[OMK04] J. A. Offutt, YuSeung Ma, and YongRae Kwon. An Experimental Mutation System for Java. *Proc. Workshop Empirical Research in Software Testing / ACM SIGSOFT Software Engineering Notes*, 29(5):1–4, Sep. 2004.

[OS06] N. Oster and F. Saglietti. Automatic Test Data Generation by Multi-Objective Optimisation. In *Computer Safety, Reliability and Security*, volume 4166 of *LNCS*, pages 426–438, Gdansk, Poland, Sep. 2006.

[Ost07] N. Oster. *Automatische Generierung optimaler struktureller Testdaten für objektorientierte Software mittels multi-objektiver Metaheuristiken*. PhD thesis, Friedrich-Alexander-Universität Erlangen-Nürnberg, CS Dept., Feb. 2007.

[OW91] T. J. Ostrand and E. J. Weyuker. Data Flow-Based Test Adequacy Analysis for Languages with Pointers. In *Proc. Symp. Testing, Analysis, and Verification*, pages 74–86, New York, Oct. 1991.

[PBSO08] F. Pinte, G. Baier, F. Saglietti, and N. Oster. Automatische Generierung optimaler modellbasierter Regressionstests. In *INFORMATIK 2008 - Beherrschbare Systeme dank Informatik*, volume P-133 of *LNI*, pages 193–198. Ges. f. Informatik, Sep. 2008.

[Ton04] P. Tonella. Evolutionary testing of classes. In *ISSTA '04: Proc. 2004 ACM SIGSOFT Intl. Symp. Software Testing and Analysis*, pages 119–128, New York, July 2004.

[WBP02] J. Wegener, K. Buhr, and H. Pohlheim. Automatic Test Data Generation For Structural Testing Of Embedded Software Systems By Evolutionary Testing. In *GECCO 2002: Proc. Genetic and Evolutionary Comp. Conf.*, pages 1233–1240, New York, July 2002.

[WC80] L. J. White and E. I. Cohen. A Domain Strategy for Computer Program Testing. *IEEE Trans. Software Engineering*, SE-6(3):247–257, May 1980.

Ein kombinierter Black-Box- und Glass-Box-Test

Rainer Schmidberger

Universität Stuttgart
Institut für Softwaretechnologie
Universitätsstr. 38
70569 Stuttgart
rainer.schmidberger@informatik.uni-stuttgart.de

Abstract: Beim Testen kommt der Wahl der Testfälle eine entscheidende Bedeutung zu, denn mit der Festlegung der Testfälle wird über die Chancen zur Fehlerentdeckung entschieden. Viele Untersuchungen gehen der Frage nach, ob beim Black-Box-Test oder beim Glass-Box-Test effektivere Testfälle entstehen. Heute ist sich die Literatur weitgehend einig, dass die beiden Testverfahren keine Alternativen bilden, sondern sich sinnvoll ergänzen. In diesem Artikel wird ein werkzeuggestütztes kombiniertes Black-Box-/Glass-Box-Testverfahren vorgestellt. Als Resultat erhält der Tester konkrete Empfehlungen für neue Testfälle. Der besondere Vorteil dieser Empfehlungen ist der Bezug zu bestehenden Black-Box-Testfällen. Das Verfahren wird anhand der Open-Source Werkzeuge CodeCover und Justus vorgestellt und in einer Fallstudie evaluiert.

1 Einführung

In der industriellen Praxis ist Testen unstrittig die am meisten eingesetzte Technik, um Fehler in Programmen zu finden oder einen Nachweis über eine bestimmte Qualität eines Programms zu liefern. Dabei kommt der Wahl der Testfälle eine entscheidende Bedeutung zu, denn mit der Festlegung der Testfälle wird über die Chancen zur Fehlerentdeckung entschieden. Ein Testfall gilt als effektiv, wenn er eine hohe Wahrscheinlichkeit hat, einen Fehler anzuzeigen. Viele Untersuchungen gehen der Frage nach, ob beim Black-Box-Test oder beim Glass-Box-Test (GBT) effektivere Testfälle entstehen [Ra85], [Ka95], [La05]. Beim Black-Box-Test (oder Funktionstest) prüft der Tester, ob das Programm der vorgegebenen Spezifikation entspricht. Er entwirft für die einzelnen Anforderungen der Spezifikation gezielt Testfälle, die prüfen, ob die Anforderung durch das Programm korrekt umgesetzt ist. Beim Glass-Box-Test (oder White-Box-Test, Strukturtest) prüft der Tester, ob alle vom Programmierer implementierten Anweisungen oder andere Programmierkonstrukte angemessen durch Testfälle ausgeführt werden und das – im Bezug zur Spezifikation – korrekte Resultat für die Eingaben ergeben. Die Literatur ist sich heute einig (z. B. [Yu09]), dass Black-Box- und Glass-Box-Test keine Alternativen bilden, sondern sich sinnvoll ergänzen. Allerdings gibt es in der Literatur kaum Vorschläge, wie diese Kombination praktisch durchgeführt werden soll. In diesem Artikel wird ein solches kombiniertes Black-Box-/Glass-Box-Testverfahren vorgestellt.

Mit den Testfällen des Black-Box-Tests wird mit einem Glass-Box-Test-Werkzeug die Programmcode-Überdeckung erhoben, und aus der Auswertung dieser Überdeckung werden dem Tester Empfehlungen für Eingabedaten neuer Testfälle vorgeschlagen. Der besondere Vorteil dieser Vorschläge ist der Bezug zu den bestehenden Black-Box-Testfällen, sodass der Tester den neuen Testfall ausgehend von einem – ihm bekannten – Black-Box-Testfall entwickeln kann.

Eine Werkzeugunterstützung ist beim Glass-Box-Test Voraussetzung. Am Markt sind auch viele Glass-Box-Test-Werkzeuge für viele der kommerziell wichtigen Programmiersprachen verfügbar [Ya06]. Die Produkte sind zum Teil kommerziell, zum Teil Open-Source, und als Resultat des Glass-Box-Tests liefern die Werkzeuge typischerweise Angaben über Anweisungs- und Zweigüberdeckung. In diesem Artikel wird das Glass-Box-Test-Werkzeug CodeCover [CoCov] verwendet, das in Kombination mit dem Testsuite-Management-Werkzeug Justus [Justus] den kombinierten Testansatz unterstützt. Beide Werkzeuge wurden am Institut für Softwaretechnologie der Universität Stuttgart entwickelt. Die Weiterentwicklung findet für beide Werkzeuge unter einer Open-Source-Lizenz statt.

2 Grundlagen: Glass-Box-Test und Black-Box-Test

Der Glass-Box-Test ist keinesfalls neu: Bereits 1975 beschreibt Huang [Hu75] den Glass-Box-Test prinzipiell in der Form, wie er auch heute in den Lehrbüchern (wie z. B. [Li02]) behandelt wird[1]. Huang definiert Anweisungs- und Zweigüberdeckung auf Grundlage des Kontrollflussgraphen und beschreibt, wie Zähler in ein Programm eingefügt werden und wie die Auswertung des GBT erfolgt. Er zeigt auch die Grenzen des GBT, indem er Beispiele nennt, bei denen auch Testfälle mit vollständiger Anweisungsüberdeckung einen Fehler in einem Ausdruck nicht erkennen.

2.1 Das GBT-Modell

Beim GBT wird in der Regel nicht der gesamte Programmcode betrachtet, sondern es wird ein GBT-Modell des Programmcodes gebildet, auf dessen Grundlage die Entwicklung der Testfälle und die Bewertung der Vollständigkeit des Tests stattfindet. In vielen Publikationen wird der Kontrollflussgraph als GBT-Modell verwendet [Hu75] [Li02], es werden aber auch andere Modelle wie z. B. Datenflusskriterien, Wahrheitstabellen der Bedingungsterme [Ch94] [Lu10] oder der Ableitungsbaum des Programms [CoCov] eingesetzt. Prinzipiell wird im GBT-Modell das Programm auf einzelne GBT-Elemente verkürzt, auf deren Grundlage die Überdeckungsmetriken definiert werden. Beispiele für GBT-Elemente sind Anweisungen, Zweige oder Bedingungsterme.

[1] Soweit man es heute zurückverfolgen kann, wurde der GBT unabhängig von mehreren Autoren in der Zeit ab 1965 bis 1975 publiziert. Der Artikel von Huang hebt sich aber deswegen ab, weil er den GBT weitgehend in der heute bekannten Form beschreibt.

Um die Ausführung der GBT-Elemente zu erheben, wird in der Regel der Programmcode so mit Zählern versehen, dass bei Ausführung des zu einem GBT-Element korrespondierenden Programmcodes ein Zähler inkrementiert wird. Dieses Einfügen von Zählern in das Programm wird in der Literatur in der Regel als Instrumentierung bezeichnet.

2.2 CodeCover

Beim „klassischen" GBT [Li02] werden die Zähler der GBT-Elemente bei Programmstart mit Null initialisiert und bei Programmende ausgelesen und abgespeichert. In der Auswertung ergibt sich ein Überdeckungsbericht für die gesamte Testausführung. Beim Glass-Box-Test mit CodeCover können die Zählerstände auch während der laufenden Testausführung ausgelesen und neu initialisiert werden. Auf diese Weise können mit Beginn der Ausführung eines Testfalls die Zähler neu initialisiert und mit Ende der Ausführung die Überdeckungswerte für genau den ausgeführten Testfall ausgelesen werden. Neben der Auswertung wie beim „klassischen" GBT sind so Aussagen zum Überdeckungsverhalten einzelner Testfälle möglich. Eine ähnliche GBT-Funktion beschreiben Lyu, Horgan und London in [Ly93]. Ihr Werkzeug ATAC erhebt Datenflussüberdeckung und enthält Funktionen, um die Ausführung einzelner Testfälle zu vergleichen. Allerdings können Testfallbeginn und -ende nicht während eines Programmlaufs bestimmt werden, sondern jeder Programmlauf – von Programmstart bis Programmende – entspricht der Ausführung für einen Testfall. Für den Test einzelner kleiner Funktionen ist diese Einschränkung akzeptabel, für den Test großer Systeme ist dieses Vorgehen aber ungeeignet. CodeCover hat gegenüber den anderen GBT-Werkzeugen am Markt [Ya06] noch den Vorteil, dass neben der Anweisungs- und Zweigüberdeckung auch weitere GBT-Überdeckungen erhoben werden können [CoCov]:

- die Termüberdeckung [Lu10] einschließlich der Modified Condition/Decision-Überdeckung (MC/DC) [Ch94]
- die Schleifenüberdeckung, wobei für die Vollständigkeit kein, genau ein und mehr als ein Schleifendurchlauf gefordert werden
- die Bedingter-Ausdrucks-Überdeckung (?-Überdeckung), die für den bedingten Ausdrucks-Operator (?:) fordert, dass beide Alternativen wirksam werden
- die Synchronisationsüberdeckung, die fordert, dass jede synchronized-Anweisung wirksam wird, also sowohl den Thread des Testfalls anhält und eine Warteschlange bildet, als auch den Thread des Testfalls direkt passieren lässt

Die verschiedenen Überdeckungen zielen auf die Entdeckung unterschiedlicher Fehler: Während die Synchronisationsüberdeckung beim Last-Test auf Fehler wie Flaschenhälse oder Verklemmung abzielt, sind es bei der ?-Überdeckung Fehler in bedingten Ausdrücken und bei der Schleifenüberdeckung Fehler, die erst bei mehrfacher Ausführung des Schleifenkörpers wirksam werden. Die Termüberdeckung fordert das gründliche Testen von Bedingungstermen bei if-Prädikaten. Das GBT-Modell von CodeCover sieht für alle genannten Überdeckungen entsprechende GBT-Elemente vor, für die im Verlauf einer Testdurchführung durch eine entsprechende Instrumentierung die Ausführung mitgezählt wird.

2.3 Black-Box-Test

Beim Black-Box-Test (oder Funktionstest) bildet die Anforderungsspezifikation die Grundlage der Testfälle. In der Literatur finden sich zahlreiche Verfahren, wie aus der Spezifikation die Testfälle entwickelt und dokumentiert werden [IE829] [My79] [Li02] [Lu10]. Da auf diese Weise die Testfälle zu den Anforderungen der Spezifikation geschrieben werden – also in der Regel die korrekte Umsetzung einzelner Anforderungen gezielt getestet wird – haben die Testfälle des Funktionstests einen sehr engen Anforderungs- und damit Domänenbezug. Die Testfälle sind für Domänenexperten gut verständlich und können gut auf Korrektheit und Relevanz geprüft werden. Die Testfälle werden in der Regel in einer Testsuite verwaltet. Nach [IE829] [My79] [Li02] [Lu10] wird die Dokumentation der Testfälle in einer einheitlichen Struktur empfohlen: Eindeutige Testfall-Nummer oder Testfallbezeichner, Vorbedingung, die zur Ausführung des Tests erfüllt sein muss, ggf. zuvor ausgeführte Testfälle, Eingabedaten oder Benutzeraktion, und ein Soll-Resultat, das nach Abschluss der Eingaben ausgegeben werden soll. An dieser Struktur orientiert sich auch das Werkzeug Justus. Zudem können in Justus (wie bei vielen anderen Werkzeugen des Black-Box-Tests) Testfälle in einer Ordnerstruktur abgelegt werden. In der Regel werden inhaltlich ähnliche Testfälle in gemeinsamen Ordnern abgelegt. Diese Ordner werden in Justus als Testsequenzen bezeichnet. Als weiteres Attribut eines Testfalls wird von vielen Autoren (z. B. [Am00]) die Priorität empfohlen. Die Priorität eines Testfalls spielt speziell beim Regressionstest eine große Rolle. Nach Amland [Am00] ergibt sich die Priorität eines Testfalls aus dessen Chance, einen schwerwiegenden Fehler anzuzeigen. Hierzu bewertet er die Wahrscheinlichkeit, dass der Fehler eintritt, und den dann entstehenden Schaden. Bei begrenztem Testaufwand empfiehlt Amland die risikobasierte Ausführung der Testfälle, also in der Reihenfolge der Priorität mit dem Ziel, die schweren Fehler früh aufzudecken.

3 Verwandte Arbeiten

Piwowarski, Ohba und Caruso berichten in [Pi93] über den Testprozess beim Test großer Systeme bei der IBM. In ihren Untersuchungen stellen sie fest, dass auch beim gründlichen Black-Box-Test nur eine Anweisungsüberdeckung von etwa 60% erreicht wird. Im Glass-Box-Test können sie diese Überdeckung um 10% erhöhen und dabei eine um ca. 8% höhere Fehlerentdeckungsrate erzielen. Damit sind die Testfälle des GBT nur unwesentlich schlechter im Bezug auf die Fehlerentdeckungsrate als die Testfälle des Black-Box-Tests. Eine weitere Erhöhung der Anweisungsüberdeckung über 70% erweist sich in ihren Untersuchungen als nicht wirtschaftlich, weil der Aufwand zur Herleitung und Ausführung der Testfälle zu groß wird. Piwowarski, Ohba und Caruso beschreiben aber kein Verfahren, wie aus den Resultaten des Glass-Box-Tests neue Testfälle hergeleitet werden sollen. Dupuy und Leveson untersuchen in [Du00] einen kombinierten Black-Box- und Glass-Box-Test anhand einer Satellitensteuerung (der Programmcode dieser Satellitensteuerung ist allerdings wesentlich kleiner als die von Piwowarski et al. untersuchten Systeme). Als Glass-Box-Test-Überdeckung verwenden sie die für sicherheitskritische Software in der Raumfahrt vorgeschriebene MC/DC-Überdeckung [Ch94]. Sie führen dabei zuerst den Black-Box-Test durch.

Bei den so gefundenen Fehlern stellen sie speziell bei „off-nominal“-Testfällen, also Testfällen mit Eingabedaten, die im spezifizierten Einsatz des Systems nicht auftreten können, den überwiegenden Teil an Fehlern fest. Mit Black-Box-Testfällen entdecken sie aber auch Fehler bei speziellen Parameterkombinationen. Trotz des sehr gründlichen Black-Box-Tests stellen Dupuy und Leveson beim Glass-Box-Test Programmteile fest, die nicht überdeckt wurden. Speziell werden hier Programmteile zur Fehlerbehandlung genannt, in denen schließlich auch Fehler gefunden werden.

Yu, Chan und Poon untersuchen in [Yu09] 56 verschiedene Programme, die von Studierenden für eine vorgegebene Spezifikation einer Kreditkarten-Funktion implementiert werden. Sie verwenden dabei die Klassifikationsbaum-Methode zur Herleitung der Black-Box-Testfälle und als Überdeckungsmetrik die Pfadüberdeckung (es handelt sich um sehr kleine Programme ohne Schleifen). In ihrer Untersuchung gehen sie der Frage nach, wie hoch die Gesamtüberdeckung der Testsuite für die einzelnen Programme ist und wie sie durch neue Testfälle, die beim Glass-Box-Test entwickelt werden sollen, erhöht werden kann. Sie stellen dabei fest, dass nicht ausgeführte Programmpfade hauptsächlich Implementierungsdetails behandeln. Zur Erweiterung der Testsuite sprechen sie von „Implementations-inspirierten“ Testfällen; der Tester lässt sich also vom Programmcode inspirieren und ergänzt aus seiner Domänenkenntnis heraus die Testsuite um die neuen Testfälle. Alle genannten Arbeiten zeigen den Nutzen eines kombinierten Black-Box-/Glass-Box-Tests, allerdings ohne darauf einzugehen, wie der Tester konkret aus den Resultaten des GBT neue Testfälle entwirft.

Wegener et al. beschreiben in [Weg01] ein werkzeuggestütztes Verfahren zur Testsuite-Erweiterung. Sie verwenden den Kontrollflussgraphen als GBT-Modell und wählen die Ausführung eines GBT-Elements (eine Anweisung oder eine festgelegte Anweisungsabfolge) als Testziel. Ausgehend von den Eingabedaten bestehender, zu Beginn zufällig gewählter Testfälle variieren sie diese Eingabedaten so, dass der daraus resultierende Kontrollfluss das Testziel ausführt oder ihm möglichst nahe kommt. Hierfür definieren sie eine sogenannte Annäherungsstufe, die die minimale Anzahl an Verzweigungen angibt, die zwischen den von einem Testfall überdeckten GBT-Elementen und dem Testziel liegen. Aus dem Prädikat dieser Verzweigungsstellen ermitteln sie den sogenannten lokalen Abstand, der angibt, wie nahe die Eingabedaten an der gewünschten Änderung des booleschen Prädikatwertes der Verzweigungsstelle liegen. Die Annäherungsstufe und die lokale Nähe fassen sie in einer sogenannten Fitness-Funktion eines Testfalls für ein Testziel zusammen, die darüber entscheidet, ob der Testfall weiter variiert oder verworfen wird. Ähnlich wie bei Wegener et al. generiert Oster [Ost07] Eingabedaten für Testfälle, die vorgegebene Überdeckungskriterien erfüllen sollen. Oster behandelt dabei insbesondere Datenflusskriterien und verfolgt eine Optimierung der Testsuite, sodass möglichst wenige Testfälle die beabsichtige GBT-Überdeckung erzielen. Die Soll-Resultate für die so ermittelten Eingabedaten müssen – wenn kein Orakel für das Prüfobjekt vorliegt – manuell aus der Spezifikation ermittelt werden. Die Anwendbarkeit beider Verfahren sehen die Autoren in der Teststufe des Unit- oder Modultests.

4 Definitionen

Eine Testsuite enthält viele Testfälle, ein Programm wird im GBT-Modell in eine Menge an GBT-Elementen abgebildet. Gegeben sei eine Testsuite *T* und ein Programm *P*. Für einen Testfall $t \in T$ ist *exe(t)* die Menge an GBT-Elementen, die von *t* ausgeführt werden. *dom(e)* ist der Dominator von e $\Leftrightarrow$

$$\forall t \in T: e \in exe(t) \Rightarrow dom(e) \in exe(t) \wedge dom(e) \notin exe(t) \Rightarrow e \notin exe(t)$$

D. h. jeder Kontrollfluss zu *e* führt immer durch *dom(e)*. *ddom(e)* ist der direkte Dominator von *e:*

$$e_d = ddom(e) \Leftrightarrow e_d = dom(e) \wedge \neg\exists\, e_x \in P : \; e_d = dom(e_x) \wedge e_x = dom(e)$$

D. h. es liegt kein GBT-Element „zwischen" *e* und *ddom(e)*. Für ein GBT-Element $e \in P$ ist *testcases(e)* die Menge an Testfällen, die *e* ausführen. Ein Testfall t tangiert ein GBT-Element e:

$$t \in T \text{ tangiert } e \in P \Leftrightarrow t \notin testcases(e) \wedge t \in testcases(ddom(e))$$

D. h. ein Testfall t tangiert ein GBT-Element e, wenn e von t nicht ausgeführt wird, der direkte Dominator von e aber ausgeführt wird. Solche Testfälle haben nach Wegener et al. [Weg01] die maximale Annäherungsstufe und sind für die Herleitung neuer Testfälle im Folgenden von großer Bedeutung.

5 Der kombinierte Black-Box-/Glass-Box-Test

Das im Folgenden beschriebene Vorgehen bezieht sich in großen Teilen auf die beiden Werkzeuge Justus und CodeCover. Justus umfasst dabei den Bereich des Black-Box-Tests und bietet neben der Erfassung und Verwaltung von Testfällen auch eine systematische Testdurchführung mit Test-Berichtserstellung. Die Testdurchführung findet manuell statt, d. h. der Tester führt der Reihe nach die ausgewählten Testfälle von Hand aus und erfasst ggf. Abweichungen zwischen Ist- und Soll-Resultat. Für den GBT wird CodeCover verwendet. Eine wichtige Funktion von CodeCover ist die bereits beschriebene Erfassung der Überdeckung für einzelne Testfälle sowie die Interaktion mit Justus, von wo aus Beginn und Ende der Testfälle signalisiert werden.

5.1 Test-Vorbereitung

Die Black-Box-Testfälle werden, wie in Abbildung 1 dargestellt, mit dem Werkzeug Justus mit Vorbedingung, Eingabedaten und Soll-Resultat erfasst. Das „System under Test" (SUT) wird mit dem Glass-Box-Test-Werkzeug CodeCover für den Glass-Box-Test eingerichtet: Das SUT wird als GBT-Modell, also als eine Menge von GBT-Elementen betrachtet; jedes GBT-Element hat einen Zähler, der die Ausführungen zählt.

Die Besonderheit beim GBT mit CodeCover ist, dass das SUT zwei zusätzliche öffentliche Schnittstellen erhält: Die Zähler der GBT-Elemente können mit Null initialisiert und die aktuellen Zählerstände können abgespeichert werden. Technisch wird diese Schnittstelle für Java-Anwendungen mit JMX (Java Management Extension) realisiert. Die beiden Schnittstellen werden im Rahmen der Instrumentierung automatisch eingefügt, der Tester muss dazu nichts aktiv beitragen.

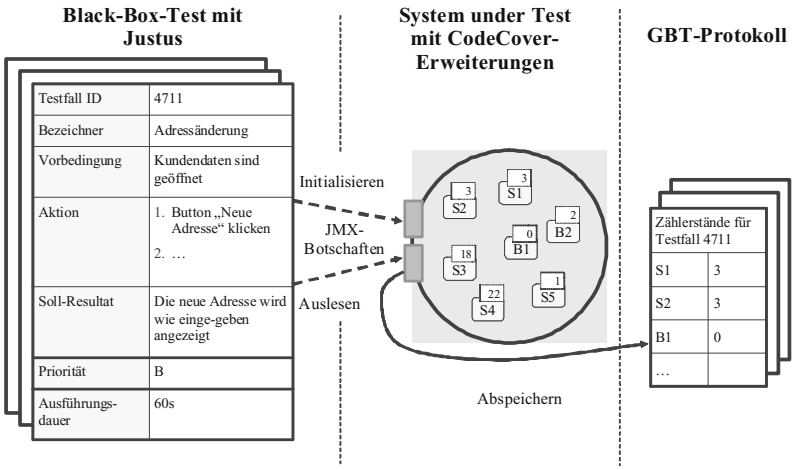


Abbildung 1: Testaufbau

5.2 Test-Durchführung

Aus Sicht des Testers findet zunächst ein „normaler“ Black-Box-Test statt. Mit dem Start der Testdurchführung verbindet sich das Werkzeug Justus mit den im SUT bereitgestellten Schnittstellen und übermittelt in der Abfolge der Testfälle die Initialisierung und das Abspeichern der Zählerstände. Auf diese Weise wird ohne weiteres Zutun des Testers das Überdeckungsverhalten der einzelnen Testfälle erhoben und abgespeichert und kann nach Abschluss der Testdurchführung ausgewertet werden. Alternativ (für einen Funktionstest von Informationssystemen allerdings eher ungewöhnlich) können auch Unit-Tests mit dem Werkzeug JUnit ausgeführt werden. Auch hierfür bietet CodeCover eine entsprechende Unterstützung.

5.3 Anleitung zur Entwicklung neuer Testfälle

Nachdem der Test in der beschriebenen Weise durchgeführt wurde, werden die abgespeicherten GBT-Protokolle ausgewertet. Mit der Erhöhung der GBT-Überdeckung wird das Ziel verfolgt, Eingabedaten für möglichst effektive neue Testfälle zu empfehlen. Den Ausgangspunkt bilden hier die GBT-Elemente, die nicht ausgeführt, aber tangiert werden. Gesucht werden somit GBT-Elemente *e* mit

$$| \text{testcases(e)} | = 0 \ \wedge \ | \ \textit{testcases(ddom(e))} \ | > 0 \qquad (1)$$

d. h. die selbst nicht ausgeführt werden, deren direkter Dominator aber ausgeführt wird. Zur Herleitung eines neuen Testfalls hat das Tupel (*e, testcases(ddom(e))*) für GBT-Elemente nach (1) einen großen praktischen Nutzen: Aus dem direkten Dominator von *e* – z. B. einer if-Verzweigung oder einem Schleifenkopf – kann das Prädikat zusammen mit einem der Testfälle dem Tester als Anhaltspunkt für einen neuen Testfall vorgeschlagen werden. Als Resultat der Auswertung ergibt sich eine Tabelle, die pro Zeile eine Testfall-Empfehlung enthält und dem GBT-Element die Testfälle des direkten Dominators gegenüberstellt. Diese Tabelle könnte beispielsweise wie Tabelle 1 aussehen.

Ungetestetes GBT-Element e	*testcases(ddom(e))*
If-Zweig Codezeile: 111 Prädikat: umsatz > 200	Testfall 4711: Kunde bearbeiten Testfall 4712: Rechnung drucken
Schleifenkörper Codezeile 222 Prädikat: artikelIterator.hasNext()	Testfall 4712: Rechnung drucken
…	

Tabelle 1: Testauswertung

Der Tester wird nun ausgehend von den genannten tangierenden Testfällen Eingabedaten suchen, die das Prädikat des tangierten GBT-Elements zu TRUE werden lassen. Im Beispiel von Tabelle 1 würde der Tester zum Testfall „4711“ Eingabedaten für den Fall „umsatz > 200“ suchen. Das Soll-Resultat des neuen Testfalls muss aus der Spezifikation – oder der Domänenkenntnis des Testers – ermittelt werden.

Bei diesen Überlegungen wird davon ausgegangen, dass das tangierte GBT-Element ein Prädikat enthält, das vom Tester über die Eingabedaten beeinflusst werden kann. Besteht zwischen dem Prädikat und den praktisch möglichen Eingabedaten kein erkennbarer Zusammenhang, ist die Testfall-Empfehlung nutzlos. Ein weiteres Problem sind technisch formulierte oder andere, für den Tester unverständliche Prädikate. In diesen Fällen muss der Tester die Entwickler bei seinen Überlegungen hinzuziehen.

Der „konventionelle“ Glass-Box-Test, also ohne die beschriebene Kombination mit dem Black-Box-Test, könnte die Spalte mit den GBT-Elementen von Tabelle 1 auch ermitteln. Der Vorteil des kombinierten Verfahrens ist die rechte Spalte, also die Angabe der tangierenden Testfälle, auf deren Grundlage der Tester den neuen Testfall entwerfen kann. Dabei lassen sich zwei Ausprägungen der *testcases(ddom(e))*-Menge unterscheiden: Erstens, die Testfälle dieser Menge sind inhaltlich sehr ähnlich, z. B. bei geringfügiger Abweichung der jeweiligen Eingaben. Die Testfälle werden dann als spezifisch für das betrachtete *ddom(e)* bezeichnet. Speziell wenn die *testcases(ddom(e))*-Menge klein ist, kann in der Praxis von spezifischen Testfällen für *ddom(e)* ausgegangen werden. Und zweitens, die Testfälle haben keine erkennbare inhaltliche Überlappung. Das ist insbesondere dann der Fall, wenn sehr viele oder alle Testfälle in der *testcases(ddom(e))*-Menge enthalten sind. Diese Testfälle werden dann für *ddom(e)* als unspezifisch bezeichnet.

Bei spezifischen Testfall-Mengen ist der praktische Nutzen besonders groß, da der Tester einen besonders konkreten Anhaltspunkt für den fehlenden Testfall hat. Er kann so die Spezifikation oder Fachexperten zu einem konkreten Thema heranziehen. Bei unspezifischen Testfall-Mengen, insbesondere wenn bei den Empfehlungen alle Testfälle der Testsuite tangieren, besteht kein praktischer Vorteil dieses Verfahrens gegenüber dem konventionellen Glass-Box-Test, da der Tester einen beliebigen Testfall aus Ausgangspunkt verwenden kann.

5.4 Priorisierung der Empfehlungen

Für Programme der industriellen Praxis ist mit sehr vielen Testfall-Empfehlungen durch die beschriebenen Auswertungen zu rechnen. Obwohl prinzipiell jede Empfehlung geeignet ist, dass auf ihrer Grundlage ein neuer Testfall entsteht, zeigen sich in der Praxis erhebliche Aufwands- und Effizienzunterschiede. Damit kommt einer Priorisierung der Empfehlungen eine große praktische Bedeutung zu. Folgende Kriterien können zur Priorisierung herangezogen werden:

- Priorität der tangierenden Testfälle: Unter der Annahme, dass der neue Testfall inhaltlich eine Variante des tangierenden Testfalls bildet, könnten tangierende Testfälle mit hoher Priorität bevorzugt zur Entwicklung neuer Testfälle herangezogen werden. Man nimmt dabei an, dass die Variante eines wichtigen Testfalls eher auch wichtig, die Variante eines unwichtigen Testfalls eher auch unwichtig wäre.
- Aufwand zur Ausführung: Unter der Annahme, zwei Testfall-Empfehlungen führen zu zwei gleich effektiven neuen Testfällen, würde der tangierende Testfall bevorzugt als Ausgangspunkt verwendet werden, der mit geringerem Aufwand ausgeführt wird.
- Bevorzugung der spezifischen Testfall-Mengen: Man nimmt an, dass Empfehlungen, die auf spezifischen Testfällen basieren, den Tester gezielter auf den neuen Testfall hinweisen als Empfehlungen mit unspezifischen Testfällen.
- Nicht alle GBT-Elemente sind gleich gewichtet: Offensichtlich haben ungetestete then- oder else-Blöcke mehr Gewicht als unwirksame Bedingungsterme oder Schleifenwiederholungen. Diese Bewertung hängt aber vom Testziel ab. Wenn bei einem Last-Test neue Testfälle für unwirksame synchronized-Anweisungen gesucht werden, werden natürlich die synchronized-GBT-Elemente gezielt betrachtet.
- Black-Box-Fehlerprognose: Aus Expertenwissen oder Fehlerstatistik lassen Testfälle ableiten, die eine höhere Fehlerprognose haben. Empfehlungen auf Basis dieser Testfälle können bevorzugt werden.
- Programmcode-Fehlerprognose: Die GBT-Elemente sollten bevorzugt werden, die in Modulen mit hoher Fehlerdichte liegen.

5.5 Iteratives Vorgehen

Die neu gewonnenen Testfälle werden zur Testsuite hinzugefügt. Eine vollständige Wiederholung der Testdurchführung ist aber nicht erforderlich, es werden lediglich die neuen Testfälle ausgeführt. Solange das Programm nicht geändert wird, lassen sich die Überdeckungsprotokolle der ersten und der nun folgenden Testdurchführung zusammenführen. Auf dieser Grundlage findet dann wieder eine Auswertung nach tangierten GBT-Elementen statt, und weitere Testfälle werden auf dieser Grundlage entwickelt. Dieses Vorgehen wird dann solange wiederholt, bis das Testendekriterium (z. B. eine bestimmte Zweigüberdeckung) erreicht ist. Ein Zusammenführen von GBT-Protokollen nach Programmänderung ist von Schumm [Sch09] ausführlich behandelt worden und – abhängig von der Art der Programmänderung – mit den CodeCover-Werkzeugen möglich.

6 Fallstudie

Für zwei Systeme wurde der beschriebene kombinierte Test durchgeführt: Für das Open-Source-Konferenzsystem Confiss [Confiss] und das kommerzielle Informationssystem Ecadia [Ecadia]. In beiden Fällen war es den Testern auf Basis der Spezifikation oder anderer Dokumente (z. B. des Benutzungshandbuchs) nicht mehr möglich, weitere Black-Box-Testfälle zu entwerfen.

6.1 confiss

Confiss ist eine in Java programmierte Web-Applikation und etwa 6300 Anweisungen groß (die grafische Oberfläche ist dabei ausgenommen). Die Black-Box-Testsuite enthält 154 Testfälle, die in 11 Testsequenzen untergliedert sind. Die Testsequenzen entsprechen den Hauptprozessen von Confiss (mit der Ausnahme der Testsequenz „Sonstiges“, die auch nur 4 Testfälle enthält). Mit dieser Testsuite ergibt sich eine Anweisungsüberdeckung von 76%; ein Wert, der deutlich über dem von Piwowarski et al. genannten liegt. Der Grund dafür ist vermutlich, dass Confiss relativ klein, neu und gut spezifiziert ist. Die Anforderungen sind dann noch recht „kompakt“ und gut zu testen.

Trotz der für eine Black-Box-Testsuite relativ hohen Überdeckung ergeben sich beim kombinierten Black-Box-/Glass-Box-Test für Zweig- und Schleifenüberdeckung 592 Empfehlungen für neue Testfälle, 108 entfallen dabei auf if-Anweisungen, bei denen der then-Block nicht ausgeführt wird. Unter diesen 108 Testfall-Empfehlungen sind 38 Empfehlungen auf Basis von spezifischen Testfällen (die Testfälle liegen in derselben Testsequenz); 30 Empfehlungen mit Testfällen aus bis zu fünf verschiedenen Testsequenzen und 40 Empfehlungen mit Testfällen aus mehr als fünf Testsequenzen, also unspezifischen Testfällen. Die 38 Testfall-Empfehlungen mit spezifischen Testfällen werden von durchschnittlich 2,2 Testfällen tangiert. In der Bewertung der Tester werden die Testfall-Empfehlungen auf Basis spezifischer Testfälle durchweg als nützlicher bezeichnet als die Empfehlungen auf Basis der unspezifischen Testfälle. Auch waren die Prädikate der if-Anweisungen im Wesentlichen verständlich und durch die Eingabedaten auch zu beeinflussen.

6.2 Ecadia

Ecadia ist ebenso wie Confiss eine in Java programmierte Web-Applikation, allerdings deutlich größer und älter. Ecadia ist etwa 160.000 Anweisungen groß, die Black-Box-Testsuite enthält 211 Testfälle in 21 Testsequenzen. Mit dieser Testsuite ergibt sich eine Anweisungsüberdeckung von 51%. Mit Zweig- und Schleifenüberdeckung ergeben sich über 6.000 Empfehlungen für neue Testfälle, etwa 3.000 entfallen dabei auf if-Anweisungen, bei denen der if-Block nicht ausgeführt wird. Unter diesen 3.000 Testfall-Empfehlungen sind immer noch etwa 1.000 Empfehlungen auf Basis von spezifischen Testfällen. Da die Testfälle der Testsuite mit Prioritäten versehen sind, erfolgt eine weitere Filterung auf die Testfall-Empfehlungen, die nur von Testfällen mit höchster Priorität tangiert werden. So ergibt sich eine praktikable Menge von etwa 50 Empfehlungen, die einzeln von den Testern ausgewertet werden: Sie entwickeln auf dieser Grundlage 14 neue Testfälle, die etwa 700 Anweisungen zusätzlich ausführen (das entspricht etwa einer um 0,5% höheren Anweisungsüberdeckung). Von den 14 Testfällen wird ein bislang unbekannter Fehler angezeigt. Diese Fallstudie ist zunächst auf einen Durchgang des in Kapitel 5.5 beschriebenen iterativen Vorgehens begrenzt. Eine weitere Auswertung der Testfall-Empfehlungen, die auf Testfällen mit geringerer Priorität basieren, wird aber noch folgen.

7 Zusammenfassung

Das in diesem Artikel beschriebene kombinierte Black-Box- und Glass-Box-Test-Verfahren hat erkennbare Vorteile gegenüber isoliert durchgeführten Tests. In der Fallstudie konnten den Testern nützliche Vorschläge für neue Testfälle geliefert werden, obwohl die Spezifikationen bereits vollständig ausgewertet waren. Der Aufwand des Verfahrens ist bedingt durch den Werkzeugeinsatz gering; der Tester hat außer einem einmaligen Aufwand zur Einrichtung der Werkzeuge Justus und CodeCover keinen Aufwand, der von der Zahl der Testfälle abhängt. In der Praxis müssen bei großen Systemen Priorisierungen der Testfall-Empfehlungen eingeplant werden. Die Zahl der Empfehlungen wird sonst unhandlich groß. Eine Priorisierung der Empfehlungen mit spezifischen Testfällen sowie auf Basis der Priorität der tangierenden Testfälle kann hierzu empfohlen werden. Dabei liefert die Priorisierung dem Tester eine methodische Hilfestellung bei der Abarbeitung; eine Sicherheit, dass sich unter den gering priorisierten Empfehlungen kein Testfall für einen schweren Fehler verbirgt, gibt es nicht.

CodeCover steht unter EPL-Lizenz und ist unter www.codecover.org frei verfügbar. Justus steht unter GPL-Lizenz und ist unter tigris.justus.org ebenfalls frei verfügbar.

Literatur

[Am00] Amland, S., 2000, Risk-based testing: risk analysis fundamentals and metrics for software testing including a financial application case study. J. Syst. Softw. 53, 3 (Sep. 2000), 287-295.

[Ch94] Chilenski, J., Miller, S., 1994, Applicability of modified condition/decision coverage to software testing, Software Engineering Journal
[CoCov] CodeCover Web-Seite: www.codecover.org
[Confiss] Confiss Web-Seite: www.confiss.org
[Du00] Dupuy, A., Leveson, N., 2000, An empirical evaluation of the MC/DC coverage criterion on the HETE-2 satellite software. Proc. Digital Aviation Systems Conference (DASC'00). Philadelphia
[Ecadia] Ecadia Web-Seite: www.ecadia.de
[IE829] IEEE 1998 Standard for Software Test Documentation IEEE Std 829-1998
[Hu75] Huang, J. C., 1975, An Approach to Program Testing. ACM Comput. Surv. 7, 3 (Sep. 1975), 113-128.
[Justus] Justus Web-Seite: justus.tigris.org
[Ka95] Kamsties, E., Lott, C. M., 1995, An empirical evaluation of three defect-detection techniques. Proceedings of the Fifth European Software Engineering Conference. Sitges, Spain.
[My79] Myers, G. J., 1979, Art of Software Testing, John Wiley & Sons, Inc., New York
[La05] Lawrance, J. et al., 2005, How Well Do Professional Developers Test with Code Coverage Visualizations? An Empirical Study, Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing
[Li02] Liggesmeyer, P., 2002, Software-Qualität: Testen, Analysieren und Verifizieren von Software. Spektrum Akademischer Verlag, Heidelberg, Berlin
[Lu10] Ludewig, J., Lichter, H., 2010, Software Engineering, dpunkt Verlag, 2. Auflage
[Ly93] Lyu, M.R., J.R. Horgan, S. London, 1993, A Coverage Analysis Tool for the Effectiveness of Software Testing, Proceedings of ISSRE'93, Denver, CO, pp. 25-34
[Ost07] Oster, N., 2007, Automatische Generierung optimaler struktureller Testdaten für objektorientierte Software mittels multi-objektiver Metaheuristiken, Dissertation, in Arbeitsberichte des Instituts für Informatik, Vol. 40, Nr. 2, Universität Erlangen-Nürnberg
[Pi93] Piwowarski, P.; Ohba, M.; Caruso, J., 1993, Coverage Measurement Experience During Function Test, Proceedings of the 15th International Conference on Software Engineering, ICSE'93
[Ra85] Ramsey, J., Basili, V. R., 1985, Analyzing the test process using structural coverage. In Proceedings of the 8th international Conference on Software Engineering (London, England, August 28 - 30, 1985). International Conference on Software Engineering. IEEE Computer Society Press, Los Alamitos, CA, 306-312.
[Sch09] Schumm, S., 2009, Praxistaugliche Unterstützung beim selektiven Regressionstest, Stuttgart, Univ., Studiengang Softwaretechnik, Diplomarbeit Nr. 2923 http://elib.uni-stuttgart.de/opus/volltexte/2009/4762/pdf/DIP_2923.pdf
[Weg01] Wegener, J. et al., 2001, Evolutionary test environment for automatic structural testing. Information and Software Technology, 43(1):841–854
[Ya06] Yang, Q., Li, J. J., Weiss, D., 2006, A survey of coverage based testing tools. In Proceedings of the 2006 International Workshop on Automation of Software Test (Shanghai, China, May 23 - 23, 2006). AST '06. ACM, New York, NY, 99-103.
[Yu09] Yu, Y. T., Chan, E. Y., Poon, P., 2009, On the Coverage of Program Code by Specification-Based Tests. In Proceedings of the 2009 Ninth international Conference on Quality Software. QSIC. IEEE Computer Society, Washington, DC

A formal and pragmatic approach to engineering safety-critical rail vehicle control software

Michael Wasilewski[1], Wilhelm Hasselbring[2]
[1]Vossloh Locomotives GmbH
24152 Kiel
http://www.vossloh-locomotives.com/
[2]Universität Kiel
Institut für Informatik, Software Engineering Group, 24118 Kiel
http://se.informatik.uni-kiel.de/

Abstract: The engineering processes for safety-critical systems, for instance in the health care or transportation domains, are regulated by law. For software in the railroad industry in Europe the certification procedures have to obey the norm EN50128.

This paper presents the method that was introduced and employed for the development and the successful certification of the software for the vehicle control unit (VCU) of the Vossloh Locomotives' G6 shunting locomotives. The primary goal in the development of the software was conformity to EN50128, the secondary goal is a cost-efficient process without sacrificing safety. To achieve these goals our method is based on formal techniques, but also designed to be easily applicable in our context (pragmatics). Central to our method are functional trees as a design specification mechanism. The outcome of employing this method was the successful certification of the locomotive G6 without any software-related problems.

1 Introduction

As for other means of transportation, the use of computer-based control systems in rail vehicles increased significantly over the last years and is still growing. Due to the fact that by rail heavy weights are moved with high velocities, faults in the control systems' software can have catastrophic consequences for human life and material goods. This potential risk leads to safety and certification requirements that control software for rail vehicles has to fulfill. At the same time manufacturers of rail vehicles have to develop software with limited resources and budgets at high quality and within tight project schedules.

At Vossloh Locomotives we therefore focus on the most time and resource consuming activities of the software development process. These are also the activities, where most faults occur: the specification, verification, implementation and validation of the software modules and their algorithms and the documentation of these activities.

The contribution of this paper is the presentation of a formal and pragmatic method to engineer software components of safety-critical systems, together with an industrial evaluation of this method.

In Section 2, the project context for engineering and certifying the locomotive G6 Vehicle Control Unit, is briefly introduced. Section 3 presents the employed method for developing the software modules, which is based on Binary Decision Diagrams. The guiding principle of this approach is simplicity, both for the engineers and for the certification process. So far, the method is applied manually without dedicated tool support. Such tool support is subject to future work, as will be discussed in Section 4 and indicated in the concluding Section 5. However, the presented method was already successfully employed for developing and certifying the VCU of the locomotive G6 [VL110].

2 Project Context

The development of software for safety-critical systems for rail vehicles starts with the definition of the control architecture.

An example control architecture is displayed in Figure 1. The components of this architecture are the Vehicle Control Unit (VCU), a Drive Control Unit (DCU) and an I/O control module. The DCU is connected with high-voltage power lines to the 3-phase traction motors. The I/O module provides an interface to additional devices, in our example a brake lever. The architectural elements VCU, DCU and I/O module are connected via a data communication bus.

For this paper the development of the safety-critical software for the VCU of the Vossloh shunting locomotive G6 [VL110] is presented. This includes a discussion of legal certification requirements and the design to achieve these requirements.

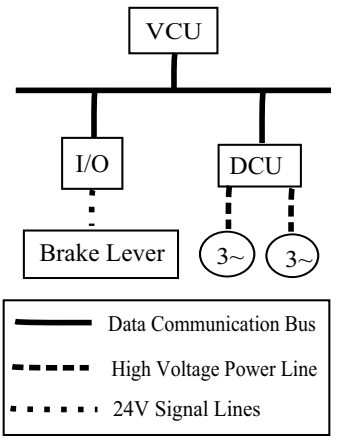


Figure 1: Typical Rail Vehicle Control Architecture

Scope: The scope of the following considerations will be a signal processing system that receives input signals from its environment and provides output signals which cause a change in the environment of the signal processing system. Out of our method's scope are the concrete signal sources and sinks of the signal processing system; thus, we abstract from the concrete signals. Here, it is only relevant that the behavior of this signal processing system may cause a safety risk.

Legal Certification Requirements: Whenever the use of a technology, such as Nuclear Power Plants, Railroads, or Airplanes, implies a potential risk to other legal assets, the use of these technologies should be regulated by law. For software in the European railroad industry, the norm EN50128 [CEN09] is relevant, which requires a software development process based on the V-Model [RHB+07]. In Germany the software is certified for use in rail verhicles by the Eisenbahnbundesamt (EBA) based upon an expert's report of an assessment agency, e.g. TüV. Subject to the assessment are the process planning, the suitability of methods and tools and the documentation of the activities.

Documentation effort: The V-Model software development process is partitioned into activities for requirements engineering, software architecture and design, and software module implementation. For these activities, Table 1 lists some quantities of the documentation effort for certifying the locomotive G6 to fulfill the documentation requirements

Nr.	Phase	Effort	Functional Test Cases
1	Software Requirements	4 documents 1000 pages	1300
2	Architecture and Design	6 documents 1500 pages	none
3	Software Modules	270 Modules 1350 documents 60000 pages	22000

Table 1: Documentation effort – Locomotive Vossloh G6

Focus on the software module documentation activities: The documentation of the software modules was identified at Vossloh Locomotives as the most resource consuming activity of a software development process to meet the requirements of EN50128.
The focus in the whole software development process for a rail vehicle is therefore put on a time and cost effective approach to describe the software modules fulfilling the requirements of EN50128 and covering about 22000 functional points.

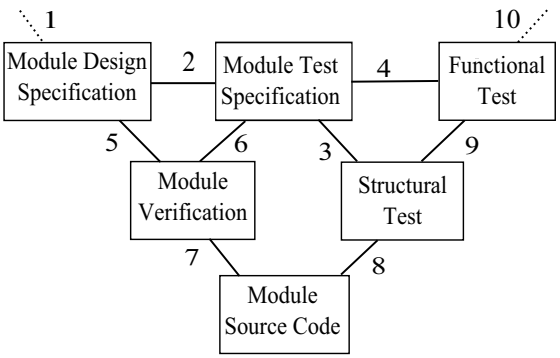


Figure 2: Module Development according to V-Modell

The development activities for software modules are shown in Figure 2. First activity is the specification of the interfaces and the functionalities (1) in a module design specification. All functional requirements have to be tested by procedures (2) defined in the module test specification.
The test procedures are divided into structural tests (white box) (3) and functional tests (black box) (4). A first verification step asserts whether the functional requirements are met (5) and testable by the defined procedure (6). The verification is documented in a module verification report. Next the source code is implemented (7). The source code verification is based on the structural sests (8) from the module test specification (3). After compilation, the executable code is validated (9) according to the functional tests from the module test specification (4). Finally, the module test report asserts the correct development according to EN50128 (10). Five documents are created per module.
The software development strategy follows a product line architecture [PBvdL05]. The software module development is part of domain engineering. Realizing a required function of a concrete product is part of application engineering. The software engineer specifies an architecture, where high-level concrete application requirements are mapped to module requirements. This allows the re-use of the modules in multiple software projects.

3 Software Module Development

This section describes the software module implementation method as it has been introduced and performed for the development of the Vehicle Control Unit for the Vossloh G6 locomotive. The method is based on an extension of Binary Decision Diagrams [Ake78] and Binary Functions [Bry86]. We present the formal basis for the extension of these ideas to describe functions that use discrete and continuous signals (Subsection 3.1), the graph-based representation (Subsection 3.2), the employed development process for an example (Subsection 3.3), and the compliance with high-level requirements (Subsection 3.4).

3.1 Formal Basis

A VCU is a part of a signal-processing system. We formally specify the various types of signals by means of set theory. For our method, the *uniqueness* and *completeness* of the defined sets is an important property, i.e. we define disjoint partitions of signal value sets.

In the following, n is defined as the number of valid values of a signal. A signal (such as a brake lever) is represented by the set of possible signal values (such as the positions of a brake lever).

Discrete Signals: Discrete Signals are used to represent well-defined states of the environment. Signals are represented as integers. An example can be the position of a Brake System Control Lever as shown in Table 2 with $n = 3$ valid positions.

Signal value	Code	Command
1	$C_1 = \{1\}$	Brake Cylinder Pressure Increase
2	$C_2 = \{2\}$	Brake Cylinder Pressure Constant
3	$C_3 = \{3\}$	Brake Cylinder Pressure Decrease
all others	$C_\Omega = \mathbb{Z}\backslash\{1, 2, 3\}$	Brake Cylinder Full Pressure

Table 2: Example for values of discrete signals

Definition of a Discrete Signal:
Any discrete signal value is an element of one of the following disjoint subsets of $\mathbb{Z}$:

$$C_1, C_2, ..., C_n, C_\Omega \subseteq \mathbb{Z}$$

The set C_Ω represents the "unknown values."

Completeness of a Discrete Signal:
The completeness of a discrete signal is given if for its subsets the following holds:

$$\bigcup_{i=1}^{n} C_i \cup C_\Omega = \mathbb{Z}$$

Uniqueness of a Discrete Signal:
The uniqueness of a discrete signal is given if for its subsets the following holds:

$$\forall i, j \in \mathbb{N} | i \leq n \wedge j \leq n \wedge i \neq j \Rightarrow C_i \cap C_j = \{\}$$

$$\forall i \in \mathbb{N} | i \leq n \Rightarrow C_i \cap C_\Omega = \{\}$$

Continuous Signals: Continuous Signals are used to represent physical values such as voltages, temperatures etc. and are represented by real numbers.

Continuous signals are discretized into ranges. With this approach the same mechanisms as for discrete signals can be used. An example can be the temperature of an engine coolant as shown in Table 3.

Value	Range	Description
$< 10°C$	R_{LO}	Engine Under-Temperature
$10°C \geq T < 70°C$	R_1	Engine Cold
$70°C \geq T < 110°C$	R_2	Engine Nominal Temperature
$> 110°C$	R_{HI}	Engine Over-Temperature

Table 3: Example for values of continuous signals

Definition of a Continuous Signal:
Any continuous signal value is an element of one of the following disjoint subsets of $\mathbb{R}$:

$$R_{LO}, R_1, R_2, ..., R_n, R_{HI} \subseteq \mathbb{R}$$

The set R_{LO} represents the "underrun range" and R_{HI} represents the "overrun range."

Completeness of a Continuous Signal:
The completeness of a continuous signal is given if for its subsets the following holds:

$$R_{LO} \cup \bigcup_{i=1}^{n} R_i \cup R_{HI} = \mathbb{R}$$

Uniqueness of a Continuous Signal:
The uniqueness of a continuous signal is given if for its subsets the following holds:

$$\forall i, j \in \mathbb{N} | i \leq n \wedge j \leq n \wedge i \neq j) \Rightarrow R_i \cap R_j = \{\}$$

$$\forall i \in \mathbb{N} | i \leq n \Rightarrow R_i \cap (R_{HI} \cup R_{LO}) = \{\}$$

$$R_{HI} \cap R_{LO} = \{\}$$

Binary Signals Binary signals are used to represent YES/NO decisions. They are expressed by a single bit. The representation as a single bit is important as it excludes any wrong values. The completeness and uniqueness of binary signals are, thus, obvious.

Definition of a Binary Signal:
Any binary signal value is an element of a set with exactly two different elements. Example: {0,1} or {TRUE,FALSE}.

3.2 Graph-Based Representation

Through the formal definition of signals we obtain a basis for specifying the functions by means of so-called *functional trees*. We employ three basic patterns as shown in Figure 3 which we can be used for synthesizing complete functions represented as functional trees.

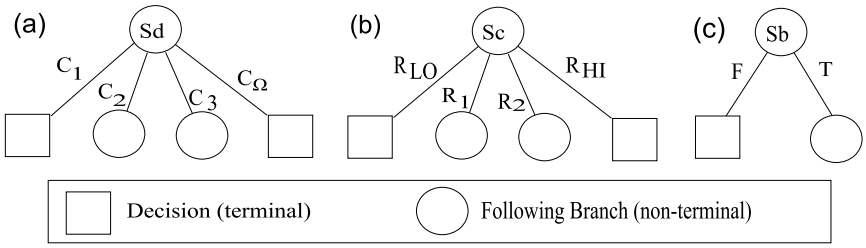


Figure 3: Basic elements for our functional specification

The patterns describe whether a decision for a result is reached or an additional, following branch is taken:

- Discrete Signals (Figure 3a): Results (C_1,C_Ω) and additional branches (C_2,C_3)
- Continuous Signals (Figure 3b): Results (R_{HI},R_{LO}) and additional branches (R_1,R_2)
- Binary Signals (Figure 3c): Results (F) and additional branches (T)

3.3 Module development process activities

In the following, we discuss the module development process activities by means of an example, the brake control.

Signal Name	Type	Description	Range/Code	Order
Sd	Discrete	Brake Lever Position	C_1: Apply Brake C_2: Constant Brake C_3: Release Brake C_Ω: Unknown Position	1
Sb1	Binary	Driver Vital Signal	T: Driver vital F: Driver non-vital	2
Sb2	Binary	Pressure Reservoir Switch	T: Reservoir Air available F: Reservoir Air exhausted	3

Table 4: Example functional signals

Construction of the functional tree (design specification): To specify a functional tree, we use the signals in Table 4. This is an example for the computation of a brake system action.

The functional tree is constructed by joining the specification patterns in Figure 3 according to the order of the signals in Table 4. Essential for the signal order is not the specific order itself, but the fact that there exists an order. By recursively connecting following specification patterns to the branches of a previous specification patterns we obtain a functional tree. The construction of the functional tree terminates if all paths end in a leaf (terminal) that represents the result of the corresponding function.

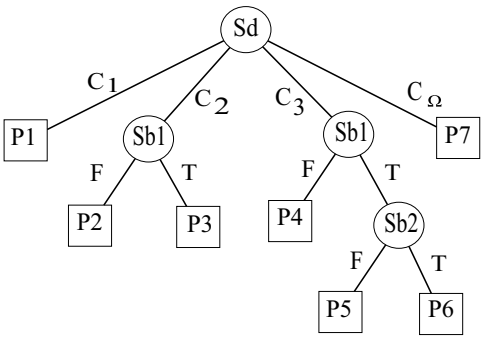


Figure 4: Functional tree for the functional signals in Table 4.

The functional tree in Figure 4 is equivalent to a logical function in disjunctive normal form (DNF) where one path in the functional tree corresponds to one logical AND connected element in a DNF functional equation.

Specification patterns are connected until all paths end in a decision. The decisions are the result of the computation that is represented as a path through the functional tree. This activity complies to step (1) in Figure 2. These trees can be expressed graphically as well as in a table. One specification item is a path through the whole functional tree. As we can see in this example, the Table 5 contains 7 possible paths through the whole functional tree while the approach to cover all possible combinations would result in 16 possibilities (Sd:4 x Sb1:2 x Sb2:2).

Specification of Tests: The specification of tests is divided into structural tests and functional tests. It complies to step (2) in Figure 2. After structuring the design into a functional tree, we have to verify that the function's code is structured accordingly.

The basis for the source code verification is shown in Table 5. The verification strategy will be to find an execution path through the source code that corresponds to the path in the functional tree (Design Specification). Therefore we have the same number of verification points as we have identified paths. The statement of the verification points complies to step (3) in Figure 2.

Path ID	Sd	Sb1	Sb2	Result
P1	C_1	N/A	N/A	Apply Brake (x=0)
P2	C_2	F	N/A	Apply Brake (x=0)
P3	C_2	T	N/A	Constant Brake (x=1)
P4	C_3	F	N/A	Apply Brake (x=0)
P5	C_3	T	F	Apply Brake (x=0)
P6	C_3	T	T	Release Brake (x=2)
P7	C_Ω	N/A	N/A	Apply Brake (x=0)

Table 5: Functional table

Listing 1: Source code example

```
1  Trace = 0;
2  if(Sd == 1)
3     Trace |= 0x1;
4     x = 0;}                    /* P1 */
5  else if(Sd == 2)
6     Trace |= 0x2;
7     if(Sb1 == TRUE)
8        Trace |= 0x10;
9        x = 1;                  /* P3 */
10    else
11       x = 0;                  /* P2 */
12 else if(Sd == 3)
13    Trace |= 0x4;
14    if(Sb1 == TRUE)
15       Trace |= 0x10;
16       if(Sb2 == TRUE)
17          Trace |= 0x20;
18          x = 2;               /* P6 */
19       else
20          x = 0;               /* P5 */
21    else
22       x = 0;                  /* P4 */
23 else
24    x = 0;                     /* P7 */
```

After defining the source code verification points, we can define a test case for each path by just setting the input values as specified and performing a test for the specified result. The selection of the test cases complies to step (4) in Figure 2.

Verification of Design and Test Coverage: The module verification proves that the functional design specification of the module corresponds to a unique and complete set of paths of a functional tree. This step is mandatory for the compliance with EN50128. It uses equivalence classes, which are a highly recommended verification technique in EN50128. Our selection of paths guarantees the uniqueness and completeness of the signal's value sets. Therefore the module verification can be done recursively by the proof of completeness of the tree at the leafs (for instance, Sb1 to P2 and P3, resp. Sb2 to P5 and P6 in Figure 4). Based upon this first step, the completeness can be proven on the next level (Sb1 with precondition $Sd = C_3$) and is finished if the root node is reached (Sd). This activity complies to step (5) in Figure 2. Another issue is the proof that the test cases cover the whole functionality. This is given by the 1-to-1 mapping of functional requirements to test cases. This activity complies to step (6) in Figure 2.

Generation/Implementation of the Source Code: The source code is written manually as a transformation of the functional tree into conditional execution paths. There is no conversion into any Boolean logic. This eliminates a source of faults and simplifies the task of programming significantly. This activity complies to step (7) in Figure 2. The code fragment in Listing 1 presents an example.

The `Trace` variable has been introduced to identify the execution path. This trace variable is computed by setting it to 0 at the beginning of the computation and setting one specific bit depending of the executed path.

Verification of the Generated Source Code: The source code verification is based upon the verification points selected in the test specification of the function.

This activity complies to step (8) in Figure 2. In our example we can map the verification points to the source code lines according to Table 6. The main purpose of the source code verification is, to prove that the execution paths of the source code have an equivalent structure as the paths through the functional tree. This proof is the base of the statement that the specified tests provide a complete condition and path coverage of the source code. The source code verification is a requirement of EN50128. This verification technique also uses equivalence classes, which are highly recommended by EN50128.

Verification Point	Line	Result
P1	04	OK
P2	11	OK
P3	09	OK
P4	22	OK
P5	20	OK
P6	18	OK
P7	24	OK

Table 6: Source Code Verification Results

Module Testing: The module testing is performed by setting the input variables and executing the code. This activity complies to step (9) in Figure 2. After each computation step both the result and the trace information are available. This is shown for our example in Table 7.

By recording the trace, we not only validate the result of a test case but additionally assert that the result is achieved by executing the paths which represents the requirement for this particular test case. In our example the result x=0 appears several times, but can always be traced back to the line of code at which it was set. After completing the tests, we are able to make a final statement on the usability of the module in the software project that has to be developed according to EN50128. This complies to step (10) in Figure 2.

Test Case	Tested Path	Result	hex Trace	binary Trace
Tc1	P1	0	0x1	00 0001
Tc2	P2	0	0x2	01 0010
Tc3	P3	1	0x12	01 0010
Tc4	P4	0	0x4	00 0100
Tc5	P5	0	0x14	01 0100
Tc6	P6	2	0x34	11 0100
Tc7	P7	0	0x0	00 0000

Table 7: Module Test Results

ID	Requirement Text	Module Paths
Rq1	An invalid position of the Brake Command Lever shall always apply full pressure on the Brake Cylinders	P1 to P6: Valid Positions, not applicable P2: Brake applied
Rq2	The Brakes shall never be released with an inactive Driver Vital Signal	P1 to P4,P7: Brake applied P5 and P6: Brake released when Driver Vital Sign active

Table 8: Module Test Results

3.4 Compliance with High Level Requirements

The compliance of the module design to fulfill the functional software requirements is asserted by comparing the informally written high-level requirements of a software requirements specification to the formally written design elements of the module design specification. This is shown in Table 8.

4 Related Work

Advanced model checking techniques can be seen as related work [BBB+04]. The idea behind model checking is to avoid having humans construct proofs. Many important programs, such as vehicle control units, have ongoing behavior and ideally run forever; they don't just start and stop. Temporal logic has been established as a way to describe and reason about these programs. Then, if a program can be specified in temporal logic, it can be realized as a finite state program. A model checker checks whether a finite state graph is a model of a temporal logic specification. A great challenge with model checking is the state-explosion problem, which means that the number of the states may go exponentially high with the number of components of the system. Techniques such as symbolic and bounded model checking achieved significant results to overcome the state-explosion problem [CES09]. Our approach is based on binary decision diagrams which are an enabling technology for model checking [BBB+04, CES09]. So far, we do not employ advanced tools, because our primary motivation is to use the method both for implementation and certification. For automatic model checking, it would be necessary to translate the requirements texts of Table 8 into temporal logic formulas. We envision several areas for tool support as will be indicated in the following section.

A comparable project is the SACEM software [GH90] for train control of the RER Line A in Paris. It consists of 21000 line of ADA code while the Vossloh G6 locomotive has 22000 functional points. Comparability is limited by the fact that the software validation included the whole specification process in the B language, while our approach is restricted to the software modules. The effort for the SACEM project was 100 men years [WLBF09].

5 Conclusions and Outlook

Our presented method strives for both formality and pragmatics. The primary goal is to engineer safe vehicle control units with a method that facilitates building safe vehicle control units and that is approachable to both our engineers and the certification inspectors. Formality is achieved by the mathematical foundation as introduced in Section 3. Pragmatics is achieved by deliberately neglecting advanced tool support. The method is simple such that the certification inspectors can easily retrace the activities of our engineers.

Based upon certification requirements, the software controlling rail vehicles in Europe has to be developed according to EN50128. In the development process, the documentation of the software modules has been identified as the most cost and time consuming part. To standardize this process at Vossloh Locomotives, the specification of the functionality of modules has been based upon a formal definition of the terms completeness and uniqueness. The process employs a constructive model building, based upon the functional requirements to describe single computation steps. This allows to use a manual verification technique. This formal method fulfills the requirements of the EN50128 as follows:

- Specification and proof of uniqueness and completeness requirements
- Simple extraction of test cases with traceability to requirements
- Proof of complete test coverage of requirements and test cases
- Simple creation of readable and maintainable source code with traceability to the requirements
- Proof of structural correspondence of source code and requirements
- Proof of complete condition and path coverage of test cases

To achieve the compliance of the development process to EN50128, we avoid the transformation of functional requirements into Boolean logic. Instead, we employ functional trees. This eliminates potential sources of errors and significantly simplifies the coding process. Additionally full traceability of test results to requirements by execution tracing validates the correctness of the algorithms.

The practical use of this approach is already evaluated in the development of the Vossloh Class G6 shunting locomotive. The whole documentation of over 60000 pages has been generated manually with a three person team within less than two years. For further locomotive projects a re-use rate of over 70% of the software modules is expected, thus making the process long-term productive. The certification authorities have not requested any further extensions of the software documentation or changes of the software development process to certify the locomotive. The development of the software is considered as a critical issue in many other rail vehicle projects.

Future work will address tool support for routine tasks in our method. As discussed in Section 4, the use of model checking tools could become an option. Particularly, tool support for model-based and model-driven code instrumentation [BH09], model-driven

testing [BH08] and trace analysis [RvHG+08] is on our agenda. Another topic for future work addresses the scalability of our methods toward more complex systems. Appropriate tool support will be an important factor. One concrete idea is to define a domain-specific language for our functional trees and to generate the instrumented code automatically from this representation. A coupled transformation could generate test cases and input to some model checker from the same or from some extended representation.

References

[Ake78] S.B. Akers. Binary Decision Diagrams. *IEEE Transactions on Computers*, 27:509–516, 1978.

[BBB+04] R. Buschermöhle, M. Brörkens, I. Brückner, W. Damm, W. Hasselbring, B. Josko, C. Schulte, and T. Wolf. Model Checking - Grundlagen und Praxiserfahrungen. *Informatik-Spektrum*, 27(2):146–158, April 2004.

[BH08] Stefan Bärisch and Wilhelm Hasselbring. Model-Driven Test Case Construction by Domain Experts. In *Proc. 1st Workshop on Model-based Testing in Practice (MoTiP 2008)*, pages 9–18, 2008.

[BH09] Marko Boskovic and Wilhelm Hasselbring. Model Driven Performance Measurement and Assessment with MoDePeMART. In *Proc. MODELS 2009*, volume 5795 of *LNCS*, pages 62–76. Springer-Verlag, 2009.

[Bry86] Randal E. Bryant. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Trans. Computers*, 35(8):677–691, 1986.

[CEN09] CENELEC. *EN50128 - Railway Applications: Software for Railway Control and Protection Systems*. CENELEC, 2009.

[CES09] Edmund M. Clarke, E. Allen Emerson, and Joseph Sifakis. Model checking: algorithmic verification and debugging. *Commun. ACM*, 52(11):74–84, 2009.

[GH90] G. Guiho and C. Hennebert. SACEM software validation. In *Proceedings of the 12th international conference on Software engineering*, ICSE '90, pages 186–191, Los Alamitos, CA, USA, 1990. IEEE Computer Society Press.

[PBvdL05] Klaus Pohl, Günter Böckle, and Frank van der Linden, editors. *Software Product Line Engineering*. Springer, Berlin Heidelberg New York, August 2005.

[RHB+07] A. Rausch, R. Höhn, M. Broy, K. Bergner, and S. Höppner. *Das V-Modell XT: Grundlagen, Methodik und Anwendungen*. dpunkt.verlag, Heidelberg, 2007.

[RvHG+08] Matthias Rohr, André van Hoorn, Simon Giesecke, Jasminka Matevska, and Wilhelm Hasselbring. Trace-Context Sensitive Performance Models from Monitoring Data of Software Systems. In *Proc. TIMERS 2008*, pages 37–44, 2008.

[VL110] Vossloh Locomotives GmbH, Kiel. *Diesel-hydraulische Lokomotive G6*, 2010. `http://www.vossloh-locomotives.com/cms/de/products_and_services/diesel-hydraulic_locomotives/g6/g6_1.html`.

[WLBF09] Jim Woodcock, Peter Gorm Larsen, Juan Bicarregui, and John Fitzgerald. Formal methods: Practice and experience. *ACM Comput. Surv.*, 41:19:1–19:36, October 2009.

Werttypen in objektorientierten Programmiersprachen: Anforderungen an eine Sprachunterstützung

Beate Ritterbach
Axel Schmolitzky

Universität Hamburg, Department Informatik,
Vogt-Koelln-Str. 30, D-22527 Hamburg, Germany
beate.ritterbach@studium.uni-hamburg.de
schmolit@informatik.uni-hamburg.de

Abstract: In der objektorientierten Modellierung von Anwendungssystemen werden *Werte* und *Objekte* häufig als unterschiedliche Abstraktionen aufgefasst. Durch die im softwaretechnischen Umfeld dominierenden objektorientierten Programmiersprachen fällt die Abbildung von Objekten eines Anwendungsbereichs auf Objektklassen dieser Sprachen inhärent leicht, während wertartige Abstraktionen umständlich repräsentiert werden müssen. Eine Programmiersprache, die neben *Objekttypen* auch *Werttypen* durch explizite Mechanismen unterstützt, könnte Abstraktionen, die konzeptionell als Werte einzustufen sind, klarer, knapper und sicherer abbilden. In diesem Artikel geben wir eine Definition von Werttypen und diskutieren, welchen Anforderungen eine Sprache genügen sollte, die dieses Konzept von Werttypen geeignet unterstützt.

1 Einführung

Etliche Veröffentlichungen im Bereich der softwaretechnischen Modellierung, beispielsweise [Zü04], [Bä98], [He00], [We98], [EK95] und [Ho07], weisen darauf hin, dass *Werte* (teilweise auch bezeichnet als „value objects“ oder „Fachwerte“) ein eigenständiges und für die Modellierung relevantes Konzept darstellen. Offensichtlich ist nicht „alles ein Objekt“, häufige fachliche Abstraktionen wie *Datum, Geldbetrag, kartesischer Punkt, Zeitraum* oder *IP-Adresse* weisen Eigenschaften auf, die sie näher an die primitiven Typen objektorientierter Programmiersprachen heranrücken als an deren Objekttypen. Letztere bilden typischerweise zustandsbehaftete und veränderbare Objekte ab, wie etwa *Kunde*, *Konto*, *Bauteil* oder *Vertrag*, die auch einen eindeutigen Erzeugungszeitpunkt aufweisen.

Obwohl es etliche Hinweise auf die Relevanz von Werten in der Modellierung gibt, gehört deren explizites Erkennen und Modellieren nicht zum Standardrepertoire. Dies kann daran liegen, dass oft keine klare Trennlinie zwischen Werten und Objekten gezogen werden kann. Es kann aber auch sein, dass die dominierende Sicht der Objektorientierung in der Modellierung zu einer gewissen „Wertblindheit“ geführt hat.

Doch selbst wenn bei den Architekten eines Softwareprojektes ein klares Verständnis für wertartige Abstraktionen im Anwendungsbereich vorhanden ist und sie geübt und gezielt fachliche Konzepte als Werte modellieren, besteht in der technischen Realisierung das Problem, dass üblicherweise objektorientierte Programmiersprachen wie Java, C++ oder C# zum Einsatz kommen. In diesen Sprachen lassen sich Objekttypen (der Modellierung) direkt in *Objektklassen* (der Programmiersprache) abbilden; Werttypen hingegen passen nicht in das Grundschema. Deshalb wurden, um Werttypen zu programmieren, spezielle Regeln und Konventionen empfohlen, z. B. das Value object pattern [Ri06]. Es kommt beispielsweise in [Ev04] und [Fo08] zum Einsatz.

Diese Ansätze stimmen darin überein, *wie* ein Werttyp abgebildet werden sollte: durch eine Klasse, die auf verändernde Methoden verzichtet (Code-Beispiele sind u. a. In [Fo08] zu finden.) Sie legen allerdings nicht klar fest, *was* unter Werten zu verstehen ist. Es bleibt offen, ob Werte dasselbe sind wie unveränderbare Objekte, d. h. Ob Unveränderbarkeit genügt, um Werte zu charakterisieren. Auch Arbeiten, die sich aus einer anderen Perspektive mit Werten befassen (z.B. ihrer effizienten Implementierung [Ba03], als Spezialfall eines allgemeineren Konzeptes „relation types“ [Va07], oder als Teil des Programmiermodells von X10, einer Sprache für Non-Uniform Cluster Computing [Ch05]) nennen nur Unveränderbarkeit als ihre kennzeichnende Eigenschaft.

Mit diesem Artikel möchten wir zwei Beiträge leisten: Zum einen geben wir in Abschnitt 2 eine Definition von Werttypen, die klarer und umfassender ist als alle informellen Beschreibungen, die wir bisher zu diesem Thema finden konnten. Diese Definition kann bereits für sich genommen hilfreich sein, ein klareres Verständnis in der Modellierung zu entwickeln. Darüber hinaus stellen wir in Abschnitt 3 dar, welche Probleme sich ergeben, wenn Werttypen durch Objektklassen abgebildet werden müssen. Aus diesen Problemen leiten wir in Abschnitt 4 Anforderungen an ein Sprachmodell ab, das Werttypen als gleichberechtigte Abstraktionen ansieht. Abschnitt 5 diskutiert verwandte Ansätze und Abschnitt 6 fasst die Arbeit zusammen.

2 Werte als explizites Konzept

Die u. a. in [Ev04], [Fo08] und [Ri06] aufgeführten Beispiele - *Zahl*, *Zeichenkette*, *Datum*, *Punkt*, *Geldbetrag*, etc. - und die Art ihrer Verwendung legen nahe, dass Werte mehr gemeinsam haben als Unveränderbarkeit. Motiviert durch die *Fachwerte* im *Werkzeug&Material-Ansatz* [Zü04], befassen wir uns am Arbeitsbereich Softwaretechnik der Universität Hamburg seit geraumer Zeit systematisch mit Werttypen ([Wi10], [Ra07], [Ra06], [He05], [Ri04], [FS02], [Sa01], [Mü99]). Dabei wurde immer deutlicher, dass eine präzise Definition des Wertbegriffs notwendig ist, um eine technische Unterstützung (per Framework, per Sprache oder nur per Konvention) für sie anbieten zu können.

2.1 Definition: Wert und Werttyp

Die Erkenntnisse aus den genannten Vorarbeiten führten zu folgender Definition:

Definition:

Werte sind die Elemente eines **Werttyp**s. Ein Werttyp beschreibt die zum Typ gehörenden **Elemente** und die für sie aufrufbaren Operationen.

1. Werte sind **unveränderbar**.
2. Werte werden **nicht erzeugt** und nicht vernichtet.
3. Wert-Operationen sind **seiteneffekt-frei**.
4. Wert-Operationen sind **referentiell transparent**, d. h. wenn sie mehrmals mit denselben Parametern aufgerufen werden, liefern sie immer dasselbe Ergebnis.

Mit diesem Begriff von Werten und Werttypen stützen wir uns insbesondere auf die grundlegende Arbeit von MacLennan [Ma82], der Werte als *zeitlose*, *zustandslose Abstraktionen* beschreibt, die unabhängig von Raum und Zeit existieren. Die Definition baut auf einem abstrakten Typbegriff auf, d. h. einer Menge von Elementen, die durch die für sie verfügbaren Operationen charakterisiert sind [LZ74]. Dementsprechend sind die Eigenschaften 1. bis 4. als *konzeptionelle Eigenschaften* zu verstehen; sie beschreiben das Verhalten von Werten und Wertoperationen aus der Sicht von Klienten, unabhängig von der zugrunde liegenden Implementation. Diese Definition gründet nicht mehr ausschließlich auf Unveränderbarkeit. Unerzeugbarkeit ist nach dieser Definition ebenfalls eine wesentliche, in den bisherigen Arbeiten vernachlässigte Eigenschaft von Werten. Sie impliziert u. a., dass auch die Elementmenge eines Werttyps unveränderlich ist.

2.2 Rationale

Die Wahl der genannten Eigenschaften als charakterisierend für Werte wurde insbesondere durch die beiden folgenden Gesichtspunkte motiviert:

a.) Primitive Datentypen wie Ganzzahlen oder Gleitkommazahlen können als Prototypen für Werttypen angesehen werden. Sie weisen die genannten Eigenschaften auf: Ihre Elemente sind unveränderbar und werden nicht erzeugt, ihre Operationen (syntaktisch oft in der Form von Infix-Operatoren wie +, -, <, ...) sind seiteneffektfrei und referentiell transparent. Wenn unsere Definition von Werttypen eine Klasse von Typen beschreibt, dann sind die primitiven Typen objektorientierter Programmiersprachen Exemplare dieser (Meta-)Klasse.

b.) Zeitlosigkeit und Zustandslosigkeit sind eher vage Begriffe, die präzisiert werden müssen. Erzeugung und Veränderung sind Vorgänge, die sich in der Zeit abspielen - Unerzeugbarkeit und Unveränderbarkeit beschreiben die Zeit-Losigkeit von Werten. Seiteneffekte und *Referentielle Opakheit* (das Gegenteil von Referentieller Transparenz) beruhen auf änderbarem Zustand. Die jeweiligen Gegenstücke, Seiteneffektfreiheit und Referentielle Transparenz, beschreiben zusammen die Zustands-Losigkeit von Werten: Eine Operation ohne Seiteneffekte *bewirkt* keine Zustandsänderungen, eine referentiell transparente Operation macht keine Zustandsänderungen *sichtbar*. Zustandslosigkeit ist essentiell in einem Umfeld, das grundsätzlich änderbaren Zustand zulässt, z. B. in objektorientierten Sprachen.

3 Implementierung von Wert-Typen mit Objekt-Klassen

Die vordefinierten primitiven Datentypen, die von gängigen objektorientierten Sprachen wie Java, C++ oder C# zur Verfügung gestellt werden (int, float, char, ...), genügen unserer Definition von Werttypen. Doch die jeweilige Sprachdefinition gibt Art und Anzahl der primitiven Datentypen fest vor, es können nicht problemlos weitere derartige Typen definiert werden. Wenn in einem Projekt *fachlich motivierte* Werttypen (*Datum*, *Uhrzeit*, *Punkt*, *KomplexeZahl* oder *Geldbetrag*) benötigt werden, ist man in den genannten Sprachen gezwungen, sie durch Objektklassen abzubilden, mit Hilfe von Mustern und Codierungs-Konventionen. Um einige Konsequenzen dieses Vorgehens aufzuzeigen, wird im folgenden die Programmierung eines Typs „Date“ in Java mit Hilfe des Value object patterns ([Ri06]) skizziert. Er soll eine simple Abbildung eines Datums, festgelegt durch Tag, Monat und Jahr, darstellen, Datumsarithmetik ermöglichen, ungültige Werte (wie den 31. Februar 2012) ausschließen, etc. Die Klasse „Date“ definiert ausschließlich Operationen, die ihre Elemente nicht ändern, z. B. liefert `addDays` ein anderes Datum statt das bestehende zu ändern. Die Operation `equals` implementiert die fachliche Gleichheit von Datumswerten, zusammen mit `equals` ist auch `hashcode` zu implementieren.

Code-Beispiel: Definition einer Java-Klasse „Date“

```
class Date {
  private int day, month, year;
  public Date(int year, int month, int day) {
    this.year = year;
    this.month = month;
    this.day = day;
  }
  public Date(int year, int dayOfYear) {
    this.year = year;
    this.month = ...;
    this.day = ...;
  }
  public int getDay() { return day; }
  public int getMonth() { return month; }
  public int getYear() { return year; }
  public int dayOfYear() { return ...; }
  Date addDays(int days) {
    int day = this.day + days;
    if (day > 31) ...
  }
  public boolean equals(Object obj) {
    if (this == obj) {return true;}
    if (obj == null) ...
  }
  public int hashCode() { ... }
  }
}
```

Eine mögliche Benutzung der Klasse Date kann z. B. folgende Anweisungen beinhalten:

Code-Beispiel: Benutzung der oben definierten Klasse „Date“

```
1   Date d1 = new Date(2010,1,30);     // Jan. 30th 2010
2   d1.addDays(7);                     // no effect
3   Date d2 = d1.addDays(7);           // Feb. 6th 2010
4   Date d3 = new Date(2010, 37);      // Feb. 6th 2010
5   if (d2 == d3) {}                   // false
6   if (d2.equals(d3)) {}              // true
```

Der isolierte Aufruf der Methode `addDays` in Zeile 2 ist wirkungslos und darum sinnlos; er führt aber nicht zu einer Fehlermeldung. In Zeile 5 liefert der Vergleich mittels „==“ false; es ist anzunehmen, dass der Klient prüfen wollte, ob sich `d2` und `d3` auf das gleiche Datum beziehen. Die Benutzung von „==“ ist demnach fachlich nicht korrekt, der Code müsste stattdessen wie in Zeile 6 aussehen. Die Konstruktor-Aufrufe in Zeile 1 und 4 suggerieren, dass Elemente von `Date` erzeugt werden; das aber steht im Widerspruch zum Konzept von Werten.

Eine Ursache dieser Schwierigkeiten ist die konzeptionelle Unerzeugbarkeit von Werten. Sie kann im Code nicht zum Ausdruck gebracht werden, da Elemente von Objektklassen immer erzeugt werden müssen. Ein weiteres Problem ist, dass die „Wert-Artigkeit“ durch Kombination mehrerer Einzel-Maßnahmen erreicht wird (Verzicht auf ändernde Methoden, Überschreiben der equals-Methode, ...). So ist sie schwer erkennbar und bei Code-Überarbeitung leicht zerstörbar. Und schließlich gibt es keine Sprachunterstützung für referentielle Transparenz und Seiteneffektfreiheit, da die Sprache Seiteneffekte grundsätzlich zulässt.

Aus diesen Gründen ergeben sich auch bei Einsatz anderer Patterns (z. B. flyweight [Ga95]) oder bei Verwendung anderer objektorientierter Sprachen wie C++ oder C# die gleichen Nachteile: umfangreicher Code, fehleranfällige Benutzung, keine Sicherheit bezüglich der Werteigenschaften. Untermauert wird diese Erkenntnis u. a. durch die Arbeit von Winkler, welche die Umsetzung von Wertypen mit den Sprachmitteln von C++ untersucht [Wi10]. Sie macht deutlich, dass die Programmierung von Klassen, die gemäß der obigen Kriterien als Werttypen anzusehen sind, möglich ist, aber durch die Sprachmechanismen von C++ stark erschwert wird. Der entsprechende Code ist umständlich, nicht als Werttyp erkennbar und in vielen Fällen ineffizient.

Zur Vermeidung von Mißverständissen weisen wir darauf hin, dass C# über den Typkonstruktor „`struct`“ verfügt und die damit definierten Typen im C#-Umfeld als „value types“ bezeichnet werden. Doch der Typkonstruktor `struct` gewährleistet keine der o. g. Eigenschaften von Werten; er bietet lediglich eine Unterstützung für Wertsemantik anstatt – wie bei reference types üblich – für Referenzsemantik.

4 Anforderungen an eine Sprachunterstützung für Werttypen

Werte nach unserer Definition sind rein funktionale Abstraktionen. Wie eben gezeigt, sind objektorientierte Sprachen per se schwach darin, Werte zu unterstützen. Ihre Stärke besteht in der Abbildung von Objekten – veränderbaren, erzeugbaren Abstraktionen und

ihren Zuständen. Objekte und Werte werden gleichermaßen benötigt; *beide* sollten durch geeignete Abstraktionen in einer Programmiersprache unterstützt werden. Aus softwaretechnischer Sicht fordern wir dabei:

1. *Klarheit*: Werttypen sollen im Quelltext deutlich erkennbar und von Objekttypen unterscheidbar sein.
2. *Sicherheit*: Die charakteristischen Eigenschaften von Werten sollen durch Sprachregeln garantiert werden (und möglichst bereits vom Compiler überprüfbar sein).
3. *Einfachheit*: Einfache Werttypen sollen mit geringem Aufwand definiert und benutzt werden können.

Wenn man das Klassenkonstrukt objektorientierter Sprachen (üblicherweise durch das Schlüsselwort `class` eingeleitet) als einen *Typkonstruktor* für Objekttypen bezeichnet (denn es wird u.a. ein Typ konstruiert), dann sollten Werttypen durch einen eigenen, zusätzlichen Typkonstruktor unterstützt werden. Klassen, die durch den postulierten Typkonstruktor für Werttypen definiert werden, bezeichnen wir als *Wertklassen*. Wir setzen voraus, dass Wertklassen ebenso wie Objektklassen nach dem Prinzip der Kapselung aufgebaut werden: durch Festlegung von *Datenfeldern*, welche die interne Repräsentation bilden, und von Operationen, die auf die Repräsentation zugreifen dürfen und für Klienten aufrufbar sind. (In speziellen Fällen ist ein anderes Konstruktionsprinzip möglich: die Definition eines Werttyps mittels Aufzählung. Sie wird im Rahmen dieser Arbeit nicht betrachtet.)

[Ra07] beschreibt Entwurf und Implementation der Sprache VJ, einer Erweiterung von Java um Wertklassen. VJ zeigt, dass und wie ein Werttypkonstruktor in eine objektorientierte Sprache eingebunden werden kann. Ein VJ-Compiler ist unter [VJ] zu finden.

Die Anforderungen an einen Werttypkonstruktor werden im folgenden anhand eines Code-Beispiels illustriert. Wie in Abschnitt 3 bildet es einen Werttyp „Date“ ab. Für Wertklassen gibt es eine Vielzahl möglicher syntaktischer Gestaltungen; deren Pro und Kontra soll im Rahmen dieser Arbeit nicht diskutiert werden. Der Verständlichkeit halber kommt für das Code-Beispiel eine an Java orientierte Syntax zum Einsatz, ähnlich der in VJ verwendeten. (Der Deutlichkeit halber sind andere Schlüsselworte gewählt als in VJ, dort waren aus Kompatibilitätsgründen pragmatische Kompromisse eingegangen worden.) Das Beispiel setzt voraus, dass ein Werttyp Int bereits existiert. Dabei ist irrelevant, ob es sich um einen vordefinierten Typ handelt oder ob er ebenfalls als Wertklasse definiert wurde.

Code-Beispiel: Definition einer Wertklasse „Date“

```
valueclass Date {
  Int _day, _month, _year;
  Int day() {return _day;}
  Int month() {return _month;}
  Int year() {return _year;}

  static Date dateYMD (Int year, Int month, Int day) {
    return select(day, month, year)
  }
```

```
  static Date dateYearDayOfYear (Int year, Int ddd) {
    Int month = ...
    Int day = ...
    return dateYMD(year, month, day)
  }
  Date addDays(Int days) {
    Int day = _day + days
    if (day > 31) ...
    return select(day, month, year)
  }
  Int dayOfYear() {return ... }
  ...
}
```

Code-Beispiel: Benutzung der o.g. Wertklasse „Date“

```
1 Date d1 = Date.dateYMD(2010,1,30);          // Jan. 30th 2010
2 d1.addDays(7);                              // Compile-Fehler!
3 Date d2 = d1.addDays(7);                    // Feb. 6th 2010
4 Date d3 = Date.dateYearDayOfYear(2010, 37); // Feb. 6th 2010
5 if (d2 == d3) ...                           // true
```

Um hervorzuheben, dass Werte und Objekte verschiedene Abstraktionen darstellen und keine ein Spezialfall der anderen ist, sollten jeweils dedizierte Schlüsselworte verwendet werden, z. B. `objectclass` und `valueclass`. Der Suffix „class“ deutet darauf hin, sowohl ein Typ als auch seine zugehörige Implementation definiert wird, eine pragmatische, in objektorientierten Sprachen übliche Verschmelzung.

Wir stellen einen Satz von Sprachregeln auf, welche die charakteristischen Eigenschaften von Werten sicherstellen:

- Restriktionen für die Datenfelder (4.1)
- Selektoren statt Konstruktoren (4.2)
- Gleichheit basierend auf den Datenfeldern (4.3)
- Kein Zugriff auf Objekte (4.4)

4.1 Restriktionen für die Datenfelder

Wir fordern, dass die Datenfelder einer Wertklasse unveränderbar und ihre Typen ebenfalls Werttypen sind. Zusammen bewirken diese beiden Bedingungen, dass Elemente einer Wertklasse unveränderbar sind.

4.2 Selektoren statt Konstruktoren

Die Unerzeugbarkeit von Werten und die Forderung nach Klarheit der Sprachunterstützung implizieren, dass eine Wertklasse keine Konstruktoren haben sollte. Eine Anweisung wie `...new Date(2010,5,3)...` wäre verwirrend und würde die Bedeutung des Typs verschleiern. Es besteht aber auch für Werttypen der Bedarf, auf einen Wert (der konzeptionell bereits vorhanden ist) gezielt zuzugreifen. Das wird mit

Operationen erreicht, die einen Wert des eigenen Typs liefern, im Beispiel etwa `dateYMD` oder `dateYearDayOfYear`. Wir nennen eine derartige Operation *Selektor*, denn sie dient dazu, einen spezifischen Wert aus dem „Werte-Universum" dieses Typs zu selektieren; ihre Parameter legen fest, welchen.

Selektoren unterscheiden sich grundlegend von Konstruktoren: Mehrere Aufrufe eines Selektors mit denselben Parametern liefern immer denselben Wert. Denn Selektoren sind Operationen eines Werttyps und müssen deshalb referentiell transparent sein. Im Gegensatz dazu liefern mehrere Aufrufe eines Konstruktors, auch mit denselben Parametern, immer verschiedene Objekte.

Der Code eines Selektors muss ein Element der eigenen Wertklasse liefern. Gemäß den bisher genannten Voraussetzungen gibt es kein Sprachmittel, womit das bewerkstelligt werden kann; ein solches muss postuliert werden. Wir schlagen dafür eine spezielle, für jede Wertklasse automatisch verfügbare Operation vor: sie wird durch ein eigenes Schlüsselwort aufgerufen („`select`"), ihre Parameter entsprechen in Anzahl und Reihenfolge den Datenfeldern der Wertklasse, und sie liefert dasjenige Element der Wertklasse, dessen Datenfelder genau diese Werte annehmen. Diese spezielle Operation bezeichnen wir als *primären Selektor*. Er sollte nur innerhalb der Wertklasse aufrufbar sein, weil seine Parameter Aufschluss über die interne Repräsentation der Wertklasse geben. Im Beispiel wird der primäre Selektor von `dateYMD` und `addDays` aufgerufen.

4.3 Gleichheit basierend auf den Datenfeldern

Bei einer Wertklasse muss die Gleichheit auf ihren Datenfeldern beruhen. Wenn alle Datenfelder übereinstimmen, muss es sich um denselben Wert handeln - anderenfalls wäre der primäre Selektor nicht referentiell transparent. Im Gegensatz dazu beruht der Vergleich von Objekten auf dem Konzept der Identität, zwei verschiedene Objekte können denselben Zustand haben.

(Wert-)Gleichheit und (Objekt-)Identität können als ein einheitliches Konzept aufgefasst werden: Hinter beiden Bezeichnungen steht die Aussage, dass zwei Ausdrücke dasselbe Element bezeichnen. Dementsprechend fordern wir, dass die Sprache nur einen einzigen Operator (oder eine Operation) für den Vergleich anbietet; er sollte bei Wertklassen die Wertgleichheit, bei Objektklassen die Identität repräsentieren. Bei einer solchen Gestaltung gibt es – anders als im Java-Beispiel aus Abschnitt 3 – keine Möglichkeit, zu prüfen, ob für einen konzeptionellen Wert mehrere Speicherrepräsentationen vorliegen – eine Aussage, die fachlich nicht relevant ist, sondern die zugrunde liegende technische Implementation auf der Ebene der Anwendungsprogrammierung durchscheinen lässt. Das Verhalten von Wertklassen in Bezug auf den Vergleich von Ausdrücken würde damit dem von primitiven Datentypen gleichgestellt, die ebenfalls nur einen Vergleichsoperator kennen. Und es verhindert potentielle Programmierfehler, die aus der versehentlichen Benutzung des „falschen" Vergleichs resultieren – z. B. in Java beim Vergleich von String-Ausdrücken mit „==" statt mit „`equals`".

4.4 Kein Zugriff auf Objekte

Die formalen Parameter und das Ergebnis einer Wertoperation sollten ebenfalls Werttypen sein. Denn wären Objekte als Parameter zulässig, dann könnte die Wertoperation Methoden dieser Objekte aufrufen, die aber können Seiteneffekte hervorrufen oder referentiell opak sein. Objekttypen scheiden somit als mögliche Parametertypen aus. Ein Objekt als Ergebnis der Wertoperation könnte dann nur durch einen Konstruktoraufruf produziert werden, was bei jedem Aufruf ein neues Objekt ergibt und damit die referentielle Transparenz zerstört. Auf ähnliche Art und Weise kann begründet werden, dass auch die Typen lokaler Variablen keine Objekttypen sein sollten.

Die vorgeschlagenen Regeln für den Werttypkonstruktor erfüllen zusammengenommen die geforderten softwaretechnischen Kriterien. Sie bewirken, dass die vorgegebenen Eigenschaften von Werttypen garantiert werden und sind zur Übersetzungszeit überprüfbar („Sicherheit“). Aufgrund des dedizierten Typkonstruktors sind Werttypen im Quelltext als solche erkennbar und ihre intendierte Benutzung für Klienten festgelegt („Klarheit“). Die Regeln sind so gewählt, dass sie für den Anwendungsprogrammierer leicht nachvollziehbar sind („Einfachheit“).

Um die Eigenschaften von Werttypen sicherzustellen, sind sie *hinreichend,* in dieser Hinsicht aber nicht notwendig: Beispielsweise wäre es möglich, das Verbot des Zugriffs auf Objekttypen zu lockern. Dann aber müsste ein komplexeres Regelwerk erstellt werden, welches verhindert, dass Wertoperationen durch Aufruf von Objektoperationen Seiteneffekte hervorrufen oder nicht mehr referentiell transparent sind.

Auch weitere wert-spezifische Konsistenzprüfungen sind möglich, so z. B. das Abfangen von isolierten und damit wirkungslosen Aufrufen von Wertoperationen, wie im Beispiel in Zeile 2.

Die Regeln, konsequent in einer Programmiersprache umgesetzt, führen zu einem asymmetrischen Abhängigkeitsverhältnis zwischen Werten und Objekten: Werte haben keinen Zugriff auf Objekte. Objekte können umgekehrt Werte benutzen, z. B. als Parameter oder Rückgabe-Typen, als Datenfelder oder lokale Variablen. Im Ergebnis wird dadurch jedes System unterteilt in einen „*funktionalen Kern*“ und eine „*objekt-orientierte Schale*“. Der funktionale Kern besteht aus den Werten und ihren zugehörigen Operationen (=Funktionen), er hat keinerlei Kenntnis von oder Zugriff auf die objektorientierte Schale. Diese wiederum umfasst Objekte und deren Operationen und kann beliebig Werte benutzen und Wert-Operationen aufrufen.

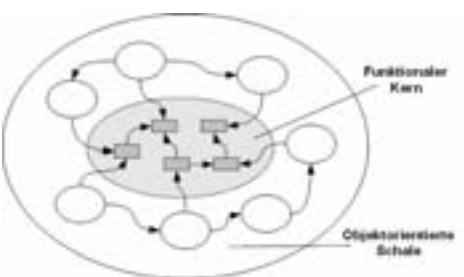


Fig. 1. Funktionaler Kern und objektorientierte Schale

Die Trennung von Objekttypen und Werttypen und ihre konsequente Unterstützung kann so die Ausgangsbasis für den Entwurf einer objekt-funktionalen Sprache bilden. Innerhalb des funktionalen Kerns ist rein funktionale Verarbeitung gewährleistet, was Programmverifikation, Parallelisierung und Optimierung erleichtert. In der objektorientierten Schale wird die Simulation von Objekten und ihren Zustandsänderungen einfach und geradlinig durch passende Abstraktionen unterstützt.

5 Verwandte Arbeiten

Eine objekt-funktionale Sprache, die auf der getrennten Unterstützung von Objekttypen und Werttypen aufbaut, wäre gemäss der in [C2] vorgenommenen Einteilung als *„pure-hybride“* zu klassifizieren. Anders als in einer „hybriden“ objekt-funktionalen Sprache (wie Ocaml [OC] oder Python [Py]) oder einer „puren“ objekt-funktionalen Sprache (wie O'Haskell [OH]) ist darin nicht eines der beiden Paradigmen das führende, und das jeweils andere durch zusätzliche Konstruktionen aufgesetzt, sondern beide Paradigmen stehen gleichberechtigt nebeneinander. Werte und Objekte - respektive Funktionen und Objekt-Operationen - bleiben jeweils als solche erkennbar, die spezifischen Eigenheiten beider Paradigmen bleiben erhalten.

Scala und Fortress sind zwei objekt-funktionale Forschungssprachen, die einige Ähnlichkeiten mit den in diesem Artikel vorgeschlagenen Konzepten aufweisen.

Scala [Od08] wurde an der EPFL von einer Arbeitsgruppe um Martin Odersky entwickelt. Es ist das explizite Ziel dieser Sprache, objektorientierte und funktionale Programmierung zu vereinen. Scala läuft auf der Java Virtual Machine, mittlerweile (seit Juli 2010) ist sie in Release 2.8 veröffentlicht. Scalas Typsystem kennt keine Trennung von Werten und Objekten. In Scala ist jedes Element ein Objekt, d. h. die Instanz einer (Objekt-)Klasse, und jede Operation eine Methode. Damit gibt es in Scala keinen funktionalen Kern, in dem pur funktionale Verarbeitung garantiert ist. Im Prinzip kann jede Operation Seiteneffekte haben oder sichtbar machen.

Es gibt in Scala Parallelen zu der hier vorgeschlagenen Unterstützung von Werttypen. Bei Case Classes erhalten die drei Methoden `toString`, `equals` und `hashcode` automatisch eine Implementation, welche die Struktur der Klassen-Member berücksichtigt, und ihr Konstruktor kann von Klienten ohne das Schlüsselwort „`new`“ aufgerufen werden. Damit weist ihre Benutzung äußerlich Ähnlichkeit mit Werttypen auf, allerdings ohne ihre charakteristischen Eigenschaften zu garantieren.

In Scala delegiert der Operator „==“ immer an die Methode `equals`. Wenn `equals` bei wert-artigen Abstraktionen passend implementiert ist, verhält sich „==“ so, wie oben vorgeschlagen: bei wert-artigen Klassen als Wertgleichheit und bei Objektklassen als Objektidentität. Allerdings gibt es neben „==“ bzw. „`equals`“ noch den Vergleich mit „`eq`“. Damit kann ein Referenzvergleich erzwungen werden, was bei wert-artigen Abstraktionen nicht angebracht ist und die zugrunde liegende Implementation durchscheinen lassen würde.

Fortress [Al08] ist eine unter der Leitung von Guy Steele in den SUN Labs entwickelte Sprache für das High Performance Computing. Sie ist ausgerichtet auf Erweiterbarkeit - viele ihrer Konstrukte sind in Bibliotheken statt im Sprachkern verankert - und die direkte Darstellung mathematischer Notationen. Fortress wurde sowohl von objektorientierten Sprachen wie Java und Eiffel als auch von funktionalen Sprachen wie ML und Haskell inspiriert.

Das Typsystem von Fortress ähnelt dem von Scala, es kennt Objekte, Traits und Vererbung. Es bietet keine dedizierte Unterstützung für Werttypen, sondern betrachtet alle Abstraktionen als Objekte - auch z. B. die Elemente primitiver Typen wie Zahlen und Zeichen, die nicht durch die Sprache selbst, sondern in Bibliotheken definiert werden. Es gibt kein Mittel, um wert-artige Abstraktionen als Werte kenntlich zu machen und die charakteristischen Werteigenschaften für sie zu garantieren.

In Fortress werden Ausdrücke üblicherweise mit dem Operator „=“ verglichen. Anders als z. B. in Java, C#, C++ oder Eiffel gibt es keine weiteren Vergleichsmethoden; der Operator „=“ kann in jeder Klasse nach Bedarf überschrieben werden kann. Damit gibt es nur einen einzigen Vergleich, was die Sprache in dieser Hinsicht einfach macht und unserer oben aufgestellten Forderung nach einem einheitlichen Vergleichsverfahren entspricht.

6 Zusammenfassung

Werttypen spielen eine wichtige Rolle in der Modellierung. Die Programmierung von Werttypen mit den Mechanismen bestehender objektorientierter Sprachen ist möglich, doch der entstehende Code ist umständlich und schwer wartbar. In dieser Arbeit haben wir, basierend auf einer ausführlich reflektierten Definition von Werttypen, einen Vorschlag entwickelt, wie solche Werttypen explizit in objektorientierten Sprachen unterstützt werden können.

Literaturverzeichnis

[Al08] Allen, E. et. al.: The Fortress Language Specification, Version 1.0. Sun Microsystems Inc. 2008, http://labs.oracle.com/projects/plrg/fortress.pdf

[Ba03] Bacon, D.F.: Kava: A Java dialect with a uniform object model for lightweight classes. Concurrency—Practice and Experience 15(3–5), 185–206, 2003

[Bä98] Bäumer, D., et al.: Values in Object Systems. Ubilab Technical Report 98.10.1, UBS AG, Zurich, Switzerland, 1998.

[Ch05] Charles, P., et al.: X10: an object-oriented approach to non-uniform cluster computing. In: Proceedings of the 20th OOPSLA, San Diego, CA, USA, pp. 519–538. ACM Press, New York, 2005

[C2] Cunningham & Cunningham, Inc., Wiki, ObjectFunctional languages. URL: http://www.c2.com/cgi/wiki?ObjectFunctional.

[EK95] Eckert, G., Kempe, M.: Modeling with Objects and Values: Issues and Perspectives. In: ROAD (Report on Object Analysis & Design), vol. 1, no. 5, pp. 20-27, Jan-Feb 1995.

[Ev04] Evans, E.: Domain-driven design: tackling complexity in the heart of software. Addison-Wesley, Boston, 2004
[FS02] Fürter, M., Spreckelmeyer, A.: Konzepte und Ansätze zur Unterstützung der Implementierung fachlicher Werte in objektorientierten Programmiersprachen. Diplomarbeit, Universität Hamburg, Arbeitsbereich Softwaretechnik, 23.08.2002
[Fo08] Fowler, M.: Patterns of enterprise application architecture. Addison-Wesley, Boston, 2008
[Ga95] Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, Reading, MA (1995)
[He00] Henney, K.: Patterns in Java: Value Added. Java Report 5(4), April 2000
[He05] Heiden, M.: Generierung von Fachwerten aus einem abstrakten Sprachmodell. Studienarbeit, Universität Hamburg, Arbeitsbereich Softwaretechnik, 14.07.2005
[Ho07] Hoogendoorn, S.: Die Implementierung von Value-Objekten. In: ObjektSpektrum, pp. 52-56, November 2007
[Lo93] Louden, K.: Programming Languages: Principles and Practice. Wadsworth Publ. Co., 1993
[LZ74] Liskov, B., Zilles, S.: Programming with Abstract Data Types. In: SIGPLAN Notices, 9:4, pp. 50-59, 1974.
[Ma82] MacLennan, B.J.: Values and Objects in Programming Languages. ACM SIGPLAN Notices 17,12, pp. 70-79, 1982
[Mü99] Müller, K.: Konzeption und Umsetzung eines Fachwertkonzeptes, Studienarbeit, Universität Hamburg, Arbeitsbereich Softwaretechnik, 30.08.1999
[OC] OCaml, URL: http://caml.inria.fr/ocaml/
[Od08] Odersky, M. ; Spoon, L., Venners, B.: Programming in Scala. Mountain View, Calif., 2008.
[OH] O'Haskell, URL: http://www.haskell.org/haskellwiki/O'Haskell
[Py] Python. URL: http://www.python.org
[Ra06] Rathlev, J.: Ein Werttyp-Konstruktor für Java, Diplomarbeit, Universität Hamburg, Arbeitsbereich Softwaretechnik, 11.08.2006
[Ra07] Rathlev, J., Ritterbach, B., Schmolitzky, A.: Auf der Suche nach Werten in der Softwaretechnik (Kurzbeitrag), In: Lecture Notes in Informatics - Proceedings SE 2007, Hamburg, pp. 261-262, 2007
[Ri04] Ritterbach, B.: Support for Value Types in an Object-Oriented Programming Language. In: Net.ObjectDays 2004, Erfurt, Germany, September 27-30, Proceedings, Volume 3263 of Lecture Notes in Computer Science, pp. 9-23, Springer, 2004
[Ri06] Riehle, D.: Value object. Proceedings of the 2006 conference on Pattern languages of programs, pp. 1-6, 2006
[Sa01] Sauer, J.: Generierung von Fachwerten aus XML-Beschreibungen. Studienarbeit, Universität Hamburg, Arbeitsbereich Softwaretechnik, 09.07.2001
[Va07] Vaziri, M., Tip, F., Fink, S., Dolby, J.: Declarative Object Identity Using Relation Types. ECOOP 2007: pp. 54-78, 2007
[VJ] VJ Research Language. URL: http://sourceforge.net/projects/vj-lang/
[We98] Werf, P. v. d.: Values and Objects Revisited. In: Journal Of Object Oriented Programming, pp. 25-34, 7/1998
[Wi10] Winkler, F.: Benutzerdefinierte Werttypen in C++. Diplomarbeit. Universität Hamburg, Arbeitsbereich Softwaretechnik, 08.09.2010
[Zü04] Züllighoven, H.: Object-Oriented Construction Handbook. Dpunkt-Verlag, Copublication with Morgan-Kaufmann, 2004

Abgleich von Teilmodellen in den frühen Entwicklungsphasen

Guy Gorek, Udo Kelter
Fachbereich Elektrotechnik und Informatik
Universität Siegen
{gorek,kelter}@informatik.uni-siegen.de

2010-11-29

Zusammenfassung

Teilmodelle sind Daten-, Zustands- oder andere Modelle, die die individuellen, initialen Anforderungen eines einzelnen Stakeholders repräsentieren. Dieses Papier adressiert das Problem, wie autark entstandene Teilmodelle zu einem Gesamtmodell abgeglichen werden können. Zunächst analysieren wir die Eignung bisher vorhandener Mischverfahren für Modelle und zeigen, daß sie wesentliche Defizite aufweisen, i.w. weil sie die typischen Verhältnisse der späten Entwicklungsphasen unterstellen. Zur Behebung dieser Defizite schlagen wir modifizierte Matching-Algorithmen, die die relativ große Unsicherheit der Teilmodelle berücksichtigen, sowie eine Methode zur interaktiven Korrektur von Korrespondenzen vor.

1 Einführung

In frühen Phasen einer Anforderungsanalyse müssen in mittleren bis großen Projekten viele Stakeholder befragt werden, die unterschiedliche Wissenstände, subjektive Bedarfe, Grade an Betroffenheit und Interessen haben. Es ist sinnvoll, die Anforderungen individuell pro Stakeholder in Form von Modellen passenden Typs (Daten-, Zustands- o.a. Modelle) zu erfassen. Solche Modelle bezeichnen wir hier als **Teilmodelle**, da sie nur einen Teil der Anforderungen repräsentieren[1]. Teilmodelle müssen oft autark erfaßt werden, z.B. bei global verteilter Softwareentwicklung, aber oft auch wegen des überfüllten Terminkalenders wichtiger Knowhow-Träger. Teilmodelle ermöglichen es, Wahrnehmungsdifferenzen der Stakeholder explizit und zum Gegenstand von Diskussionen zu machen. Als Beispiel zeigt Bild 1 zwei Teilmodelle, die bei der initialen Modellierung eines Aktionshauses vorhanden könnten, nachdem ein Anbieter (Teilmodell 1) und eine Auktionator (Teilmodell 2) befragt wurden.

[1] Andere Bezeichnungen hierfür sind **Viewpoints** [3] oder Modelle in **Sichten** [6].

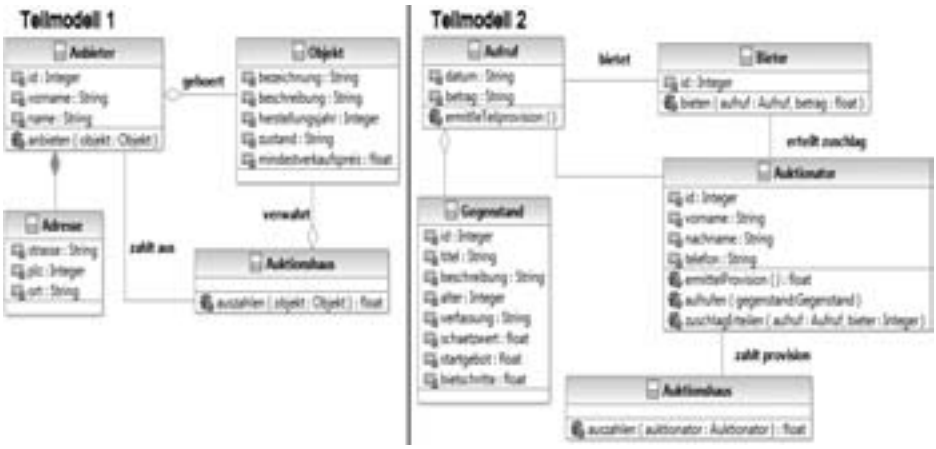


Abbildung 1: Beispiel für zwei Teilmodelle

Diskrepanzen zwischen den Anforderungen verschiedener Stakeholder müssen bereinigt werden, bevor die entsprechende Funktionalität konkret realisiert wird. Hierzu müssen die Diskrepanzen identifiziert und den Stakeholdern geeignet präsentiert werden. Technisch gesehen läuft dies auf den Vergleich und das Mischen der Teilmodelle hinaus.

Das Vergleichen und Mischen von textuellen Entwicklungsdokumenten, die überwiegend in den späten Entwicklungsphasen auftreten, ist tägliche Praxis. Für Modelle werden seit einigen Jahren analoge Algorithmen und Werkzeuge entwickelt. Der Gedanke liegt nahe, die mittlerweile verfügbaren Verfahren für Modelle auch in den sehr frühen Entwicklungsphasen auf Teilmodelle anzuwenden. Technisch ist das durchaus möglich, wie z.B. in [3] gezeigt wird. Es stellt sich allerdings heraus, daß bisherige Verfahren nicht wirklich den Anforderungen genügen, die an Mischwerkzeuge für die sehr frühen Phasen gestellt werden. Bisherige Verfahren unterstellen nämlich stillschweigend analoge Arbeitsprozesse und Randbedingungen, die beim Mischen von (graphischen oder textuellen) Dokumenten in den späten Entwicklungsphasen vorliegen; dies trifft auch auf bisher publizierte Listen von Anforderungen an Mischwerkzeuge für Modelle (z.B. [2]) zu.

Die Anforderungen, die Mischwerkzeuge für die sehr frühen Phasen erfüllen müssen, sind bisher nur oberflächlich analysiert worden [1, 11][2], was nicht zuletzt damit erklärbar ist, daß Mischwerkzeuge für Modelle erst seit kurzem realisierbar sind. Abschnitt 2 analysiert daher zunächst, inwiefern sich die Arbeitsprozesse, in denen Modelle verglichen und gemischt werden, in den späten bzw. sehr frühen Phasen unterscheiden und inwiefern wesentlich andere Randbedingungen vorliegen.

[2]Wenn in den o.g. Quellen vom Vergleich oder Mischung von Modellen die Rede ist, ist meist die Integration von Modellen unterschiedlichen Typs gemeint, die kein Gegenstand dieses Papiers ist.

Wir kommen zugleich zur Erkenntnis, daß bisherige Algorithmen und Mischwerkzeuge diese Arbeitsprozesse nur sehr unzureichend unterstützen.

Die folgenden Abschnitte beschreiben neue oder geänderte Funktionen, die Mischwerkzeuge anbieten sollten, um diese Defizite zu beheben. Eine neue Funktion ist die interaktive Korrektur von Korrespondenzen (Abschnitt 4). Abschnitt 5 beschreibt Lösungen für das Problem, wie die Matching-Algorithmen der Mischwerkzeuge die Unvollständigkeit bzw. Ungenauigkeit der Teilmodelle berücksichtigen können.

2 Vergleichs- und Mischprozesse in den frühen Entwicklungsphasen

2.1 Hintergrund: Hauptfunktionen in Mischwerkzeugen

Um die Besonderheiten der Vergleichs- und Mischprozesse in den frühen Entwicklungsphasen diskutieren zu können, führt dieser Abschnitt zunächst die Hauptfunktionen ein, die Mischwerkzeuge beinhalten.

Eine Mischung von zwei Dokumenten basiert stets auf einem Vergleich, in dem die korrespondierenden Dokumentelemente ermittelt werden. Ein Paar korrespondierender Dokumentelemente wird als **Korrespondenz** bezeichnet, eine Menge von Korrespondenzen als **Matching**. Mischwerkzeuge haben daher zwei Hauptphasen, die in weitere Unterphasen gegliedert sind:

1. Vergleich
   1.1 Berechnung eines Matchings
   1.2 Bestimmung der Differenz, Vorbereitung der Differenzanzeige
2. Mischung
   2.1 Berechnung von Konflikten
   2.2 Anzeige der Differenz und der Konflikte
   2.3 Erfassung von Mischentscheidungen
   2.4 Erzeugung einer neuen Dokumentversion

Paare korrespondierender Dokumentelemente werden nur *einmal* in das Mischergebnis übernommen. Die Wahl der Korrespondenzen hat daher einen ganz *erheblichen Einfluß auf das Mischergebnis*.

Dokumentelemente, die kein korrespondierendes Element im anderen Modell haben, nennen wir **spezielle** Elemente. Zwei spezielle Elemente, die aus verschiedenen Modellen stammen, können unverträglich sein, weil sie nicht zugleich in das Mischergebnis übernommen werden können. Potentiell sind alle Paare spezieller Elemente unverträglich; die Konflikterkennung versucht, möglichst viele dieser Paare als unkritisch zu erkennen. Die restlichen Paare werden als **Konflikte** bezeichnet, hier ist jeweils eine manuelle **Mischentscheidung** durch den Entwickler notwendig.

Beim 3-Wege-Mischen ist zusätzlich eine Basisversion vorhanden; statt spezieller Elemente werden hier Änderungen gegenüber der Basisversion betrachtet, wobei es sich auch um Löschungen handeln kann.

2.2 Unsicherheit der Teilmodelle

Eine erste wesentliche Besonderheit von Teilmodellen in den frühen Phasen ist deren Unsicherheit: Am Anfang eines Analyseprozesses ist das Problemverständnis vieler Stakeholder noch unvollkommen, und die Anforderungen sind teilweise unsicher oder Verhandlungssache. Hierdurch wird vor allem die Korrektheit der berechneten Matchings beeinträchtigt. Wir verstehen hier den Begriff unsicher in einem sehr weiten Sinn: Teilmodelle sind oft *unvollständig* und enthalten ggf. nur 30 - 50% der späteren vollständigen Spezifikationen. Teilmodelle sind ferner oft *unpräzise bzw. unzuverlässig*: die Wahrscheinlichkeit ist relativ hoch, daß vorhandene Angaben noch geändert werden müssen, weil z.B. anfänglich zufällig andere Begriffe verwendet wurden - Bild 1 enthält viele Beispiele hierzu.

Bisherige Matching-Algorithmen berechnen in derartigen Situationen Matchings mit vielen Fehlern, also fehlenden bzw. unzutreffenden Korrespondenzen, die das Mischergebnis völlig wertlos machen können.

2.3 Explizite Ungenauigkeitsangaben

Potentiell sind alle Modellelemente von Teilmodellen unsicher im Sinne des vorigen Abschnitts. Daneben ist oft für *einzelne Modellelemente* bekannt, daß diese unfertig sind und noch überarbeitet werden müssen. Es ist sinnvoll, Informationen hierüber explizit zu erfassen. Beispiele für entsprechende Sprachkonstrukte finden sich in der Modellierungssprache SeeMe [3, 5]. SeeMe enthält gegenüber der UML vereinfachte Daten-, Funktions- und weitere Modelle. SeeMe ist für die frühen Phasen optimiert und definiert mehrere Sprachelemente, mit denen explizit angegeben werden kann, daß ein Modellelement unvollständig oder ungenau ist. Diese Sprachelemente können auf die üblichen (UML-) Modelltypen übertragen werden.

Bisherige Matching-Algorithmen können derartige explizite Ungenauigkeitsangaben in Modellen überhaupt nicht verarbeiten.

2.4 2- vs. 3-Wege-Mischen

Die Vergleichs- und Mischverfahren, die man von Systemen wie SVN oder CVS kennt und die man in den späten Entwicklungsphasen benutzt, unterstützen das parallele Editieren von Dokumenten durch ein nichtinteraktives 3-Wege-Mischen. Teilmodelle entstehen hingegen anfänglich z.B. durch Befragung einzelner Stakeholder weitgehend unabhängig voneinander. Das 3-Wege-Mischen ist daher praktisch nicht anwendbar, weil entweder gar keine Basisversion existiert oder diese noch sehr unvollständig ist und überwiegend neue Teile der Modelle unabhängig voneinander entstehen.

Es liegt i.w. die Situation des 2-Wege-Mischens vor, für die bisherige Matching-Algorithmen kaum Unterstützung bieten.

2.5 Unterstützung von Mischentscheidungen

Bei Mischungen spätphasiger Dokumente spielt die Reduktion der manuell zu behandelnden Konflikte und Automatisierung möglichst aller Mischentscheidungen eine wichtige Rolle, zumal die Dokumente groß und die Änderungen zahlreich sein können. Möglich ist diese Automatisierung aber nur mit Hilfe einer Basisversion, also beim 3-Wege-Mischen. Beim 2-Wege-Mischen können prinzipiell keine Mischentscheidungen automatisiert werden. Man kann allenfalls passend zum jeweiligen Modelltyp die grundlegende syntaktische Verträglichkeit von speziellen Modellelementen prüfen. Im Endeffekt muß der Entwickler alle Entscheidungen selber treffen.

Die Automatisierung von Mischentscheidungen ist daher für Teilmodelle praktisch irrelevant. Wichtiger ist es, die manuellen Mischentscheidungen zu erleichtern, indem den Entwicklern unterstützende Information geliefert wird. Bisherige Matching-Algorithmen liefern keine derartigen Informationen.

3 Hintergrund: Verfahren zur Berechnung von Matchings

In den folgenden Abschnitten analysieren wir im Detail, warum bisherige Matching-Algorithmen und Mischwerkzeuge bei Teilmodellen unzureichend arbeiten.

Als Vorbereitung skizziert dieser Abschnitt die bekannten wesentlichen Verfahren zur Berechnung von Matchings. Diese können in mehrere Gruppen eingeteilt werden [2, 9], die aber jeweils nur unter bestimmten Voraussetzungen, die hier meist nicht erfüllt sind, nutzbar sind:

- Verfahren auf Basis von Editorprotokollen und Verfahren auf Basis von persistenten Identifizierern [10]: diese eignen sich prinzipiell nicht für *neu erzeugte* Modelle oder Modellfragmente.
- Verfahren auf Basis von Signaturen und semantik-basierte Verfahren: diese sind nicht geeignet für *unvollständige* Modelle.

Im Endeffekt eignen sich nur ähnlichkeitsbasierte Verfahren in der hier vorliegenden Situation. Ähnlichkeitsbasierte Verfahren [8, 12] zielen darauf, möglichst ähnliche Elemente der beiden Modelle als korrespondierend zu bestimmen.

Hierzu werden für jeden einzelnen Modellelementtyp ähnlichkeitsrelevante Merkmale und deren Gewichtung definiert. Bei Modellelementtypen, die einen Namen haben, ist z.B. der Name ein ähnlichkeitsrelevantes Merkmal mit einer sehr hohen Gewichtung (25 - 50 %), da Elemente mit gleichem Namen sehr wahrscheinlich korrespondieren. Die Merkmale und deren Gewichtung müssen manuell definiert werden, wobei die typischen Strukturen und Editiervorgänge eines Modelltyps zu berücksichtigen sind.

Beim Vergleich zweier Modelle wird zunächst für jedes Element $e1$ des ersten Modells die Ähnlichkeit zu jedem Element $e2$ des zweiten Modells mit gleichem Typ berechnet. Sofern die Ähnlichkeit von $e1$ und $e2$ einen Mindestwert erreicht, wird $e2$ in die **Präferenzliste** von $e1$ aufgenommen und umgekehrt $e1$

in die Präferenzliste von $e2$ (Ähnlichkeiten sind symmetrisch). Ein Matching-Algorithmus berechnet auf Basis der Präferenzlisten unter Berücksichtigung der Dokumentstruktur ein Matching, wobei versucht wird, anhand von Heuristiken die Ähnlichkeiten korrespondierender Modellelemente zu optimieren.

4 Interaktive Korrektur von Korrespondenzen

4.1 Unsicherheit von Korrespondenzen

Je nach der Struktur der Ähnlichkeiten können die Präferenzlisten suspekt und die daraus abgeleiteten Korrespondenzen unsicher sein. Als Beispiel betrachten wir zwei Elemente e und f aus dem ersten Modell und die jeweils ähnlichsten Elemente, also "Kandidaten" für eine Korrespondenzbildung, aus dem zweiten Modell:

Ähnlichkeiten zu ...	e	f
ähnlichster Kandidat	0.98	0.65
2.-ähnlichster Kandidat	0.68	0.63
3.-ähnlichster Kandidat	0.61	0.63

Bei e ist der ähnlichste Kandidat eine sehr sichere Wahl, weil der zweitähnlichste erheblich unähnlicher ist und, wenn wir die Mindestähnlichkeit mit 0.6 annehmen, nur knapp über dieser Schwelle liegt. Bei f ist der ähnlichste Kandidat eine unsichere Wahl, mit einer signifikanten Wahrscheinlichkeit ist die mit ihm gebildete Korrespondenz falsch: alle Kandidaten liegen nur knapp über der Mindestähnlichkeit, und die geringen Ähnlichkeitsunterschiede können durch Zufälligkeiten verursacht sein (z.B. zufällig ähnliche Bezeichnungen). Im Extremfall weisen zwei Kandidaten die gleiche Ähnlichkeit auf und man muß den Zufall entscheiden lassen.

Suspekte Präferenzlisten können auch beim Vergleich von Modellen aus den späten Phasen auftreten, sind dort aber selten, weil die Modelle detailreich und weitgehend fehlerfrei sind, und machen keine besonderen Maßnahmen nötig. Bei Modellen aus den sehr frühen Phasen sind suspekte Präferenzlisten hingegen häufig: durch die Unvollständigkeit der Modelle ist die Datenbasis, auf der Modelle verglichen werden, wesentlich verkleinert.

Das Mischwerkzeug sollte auf unsichere Korrespondenzen durch entsprechende Warnungen hinweisen, z.B. wie in Bild 2 vorgeschlagen. Unsicher sind Korrespondenzen, wenn eines oder mehrere der folgenden Indizien vorliegen (weitere Indizien werden später vorgestellt):

- die Ähnlichkeit der beiden Modellelemente liegt nur knapp über der Mindestschwelle
- eines der beiden Modellelemente hat nur eine geringfügig bessere Ähnlichkeit als der nächstplazierte Kandidat
- es gibt deutlich ähnlichere Kandidaten, die der Matching-Algorithmus schon vorher für die Bildung von anderen Korrespondenzen verbraucht hat

4.2 Vorgaben für Matchings

Schon aus dem obigen simplen Beispiel wird klar, daß es zu falsch berechneten Korrespondenzen kommen kann und Entwickler imstande sein sollten, Korrespondenzen zu korrigieren.

Konzeptuell sind solche Korrekturen Vorgaben für die Korrespondenzbildung. Eine **positive** Vorgabe legt genau eine Korrespondenz fest, eine **negative** Vorgabe verbietet eine Korrespondenz.

Das Mischwerkzeug muß geeignete Bedienschnittstellen anbieten, um diese Vorgaben bequem erfassen zu können. Nach der Erfassung neuer Vorgaben muß das Matching neu berechnet werden, da i.a. die Darstellung der Differenz komplett neu aufgebaut werden muß.

Die Erfassung von Vorgaben ist nur sinnvoll für Modellelemente, die in der Sichtwelt von Entwicklern einzeln identifizierbar sind und i.d.R. autark angelegt und gelöscht werden können. Nicht sinnvoll ist es, Korrespondenzen zwischen sehr kleinen Modellelementen, z.B. Typen von Parametern von Operationen, vorzugeben. Die Modellelementtypen, für die Vorgaben erfaßt werden sollen, müssen daher für jeden Modelltyp individuell bestimmt werden.

Konzeptuell sind die positiven bzw. negativen Vorgaben einfache Listen von Korrespondenzen, die neben den Modellen weitere Eingabeparameter für den Matching-Algorithmus darstellen. Bisherige Matching-Algorithmen unterstützen positive bzw. negative Vorgaben überhaupt nicht, sind aber prinzipiell dahingehend erweiterbar; Details dieser Erweitungen hängen stark von den jeweiligen Implementierungsstrukturen ab.

4.3 Ein GUI zur Korrektur von Korrespondenzen

Abb. 2 zeigt GUI-Elemente für die manuelle Korrektur von Korrespondenzen. Links ist ein Baum dargestellt, in dem alle Elemente der beiden verglichenen Modelle repräsentiert sind. Korrespondierende Elemente werden nur durch einen gemeinsamen Knoten dargestellt[3].

Eine falsche Korrespondenz kann aufgelöst werden, indem der Knoten, der das Paar korrespondierender Elemente repräsentiert, selektiert wird und über das Kontextmenü die Funktion “Korrespondenz auflösen” aufgerufen wird. Danach muß das Matching neu berechnet werden, da die beiden zuvor an der Korrespondenz beteiligten Elemente anschließend jeweils durch einen eigenen Knoten repräsentiert werden.

Für ein Element, *das aktuell keinen Korrespondenzpartner hat*, kann ein Korrespondenzpartner nach Aufruf der Funktion “Korrespondenz festlegen” manuell bestimmt werden. Abb. 2 zeigt rechts ein entsprechendes Fenster. Dieses zeigt die Namen der möglichen Korrespondenzpartner an, von denen einer zu selektieren ist, um die Korrespondenz festzulegen. Elemente, welche bereits an einer Korrespondenz beteiligt sind, werden ausgegraut dargestellt und sind nicht mehr selektierbar.

[3] Diese Darstellungsform wird auch als Vereinigungsdokument bezeichnet, s. [7]. Stattdessen könnten auch andere Darstellungsformen für Differenzen genutzt werden, s. [7].

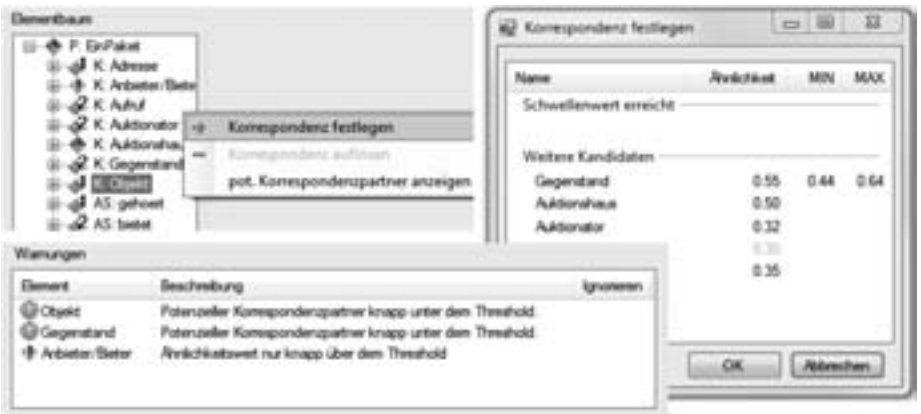


Abbildung 2: GUI-Elemente

Die Liste ist in zwei Bereiche unterteilt. Im oberen Bereich ("Schwellenwert erreicht") wird die Präferenzliste für dieses Element angezeigt, also alle Elemente des anderen Modells, die die Mindestähnlichkeit aufweisen. Im Beispiel in Abb. 2 ist die Präferenzliste leer. Für alle in der Präferenzliste angezeigten Elemente tritt einer der folgenden Fälle zu:

- das Element wurde für eine andere Korrespondenz benutzt (grau dargestellt)
- die zunächst berechnete Korrespondenz mit diesem Element wurde manuell aufgelöst

Der untere Bereich enthält Elemente, deren Ähnlichkeitswert unter der Mindestähnlichkeit liegt. Die Spalten MIN und MAX werden später erklärt.

5 Matching-Algorithmen für die frühen Phasen

Im folgenden stellen wir Modifikationen an ähnlichkeitsbasierten Matching-Algorithmen vor, mit denen das Problem der Unvollständigkeit und Unsicherheit der Teilmodelle adressiert werden kann.

5.1 Behandlung fehlender Angaben

In den frühen Projektphasen interessieren oft bestimmte Details noch nicht und werden daher noch nicht modelliert, z.B. die Multiplizität einer Rolle eines Beziehungstyps. Herkömmliche Vorgehensweisen führen bei solchen fehlenden Angaben zu einer 100%igen Ähnlich- oder Unähnlichkeit der betroffenen Modellelemente; beides ist nicht angemessen.

Im Prinzip müssen solche nicht erfaßten Merkmale als Nullwerte behandelt werden, d.h. solange nichts explizit definiert wurde, gilt die Eigenschaft als undefiniert. Editoren für Teilmodelle müssen daher so konfigurierbar sein, daß

Nullwerte erfaßt und in den Modellrepräsentationen erkennbar dargestellt werden können.

Zur Behandlung von nicht erfaßten Merkmalen bzw. Nullwerten schlagen wir die folgende Strategie vor, die in einer Variante des SiDiff-Systems [12] implementiert und getestet wurde. Angenommen, zwei Modellelemente werden verglichen und für ein Merkmal M tritt ein Nullwert auf:

- Wenn bei *beiden* Modellelementen ein Nullwert vorliegt, hat M keinen Einfluß auf die Ähnlichkeit dieser beiden Elemente. Die Gewichtung von M wird individuell für dieses Elementpaar auf Null reduziert[4].
- Wenn nur bei *einem* Modellelement ein Nullwert vorliegt, werden die beiden Modellelemente bzgl. M als völlig unähnlich behandelt.

Sei G das Gesamtgewicht aller Merkmale, deren Gewicht bei den gegebenen Modellelementen auf Null reduziert wurde. Sei ferner S_{min} die gewichtete Summe der Ähnlichkeiten der restlichen Merkmale. Dann ist

$$S_{min} \leq 1 - G$$

Sofern alle in S_{min} berücksichtigten Merkmale identisch sind, gilt $S_{min} = 1 - G$. Wenn S_{min} zur Sortierung der Präferenzlisten benutzt wird, führt dies mit hoher Wahrscheinlichkeit zu unpassend sortierten Präferenzlisten. Wenn G groß ist, kann sogar die Mindestähnlichkeit unterschritten werden. Daher muß S_{min} geeignet korrigiert werden. Im Endeffekt sind derartige Korrekturen stets Schätzungen, wie ähnlich die fehlenden Merkmale sind.

- Im günstigsten Fall sind alle undefinierten Merkmale 100% ähnlich, die maximale Ähnlichkeit ist daher $S_{max} = S_{min} + G$. Die korrigierte Ähnlichkeit muß also im Intervall $[S_{min}, S_{max}]$ liegen.
- Sofern man für die undefinierten Merkmale die gleiche Ähnlichkeit wie für die definierten annimmt, ist $S_{korr} = S_{min}/(1 - G)$. Dies ist der gewichtete Durchschnitt der in S_{min} berücksichtigten Ähnlichkeiten.

Wenn z.B. für zwei Elemente $G = 0.2$ und $S_{min} = 0.6$ ist, dann ist $S_{max} = 0.8$ und $S_{korr} = 0.75$.

Als Sortierkriterium für die Präferenzlisten schlagen wir die obige Schätzung S_{korr} vor. Das Intervall $[S_{min}, S_{max}]$ ist der Unsicherheitsbereich für diesen geschätzten Ähnlichkeitswert. Bei der manuellen Korrektur von Korrespondenzen sollte dieser Unsicherheitsbereich bekannt sein. Daher wird in dem in Bild 2 gezeigten GUI der Unsicherheitsbereich, sofern vorhanden, in den Spalten MIN und MAX angezeigt.

Die Unsicherheitsbereiche der Ähnlichkeiten können Korrespondenzen (noch) unsicherer machen, hierauf gehen wir später ein.

[4] Normalerweise ist die Gewichtung der ähnlichkeitsrelevanten Merkmale für alle Instanzen eines Modellelementtyps einheitlich.

5.2 Behandlung expliziter Ungenauigkeitsangaben

Die schon in Abschnitt 2.3 erwähnten Sprache SeeMe enthält einen umfangreichen Katalog an Konstrukten, mit denen das Ausmaß der Ungenauigkeit einzelner Modellelemente und weitere, hier nicht relevante Aspekte notiert werden können. Für den Modellvergleich fassen wir die Ungenauigkeit eines Modellelements e als eine Schätzung der Ähnlichkeit zwischen der aktuell vorhandenen Version von e und der zur Zeit noch unbekannten Endversion von e auf und bezeichnen diese Schätzgröße mit $U(e)$.

Offensichtlich sind Ähnlichkeiten, in denen eines der beteiligten Modellelemente ungenau ist, unsicher. Angenommen, zwei Modellelemente $e1$ und $e2$ haben die grundlegende Ähnlichkeit $S(e1, e2)$ und die Ungenauigkeit von $e1$ wird mit $U(e1)$ geschätzt. Wenn man nun die Gültigkeit der Dreiecksungleichung wie in einem metrischen Raum unterstellt[5], dann ist der Unsicherheitsbereich für die Ähnlichkeit

$$[S(e1, e2) - U(e1),\ S(e1, e2) + U(e1)]\ \cap\ [0, 1]$$

Sofern auch für $e2$ eine Ungenauigkeit $U(e2)$ angegeben ist, vergrößert sich der Unsicherheitsbereich an beiden Enden um $U(e2)$ unter Beachtung der Grenzen [0,1].

Offen blieb bisher die Frage, welche konkreten Werte für $U(e)$ sinnvoll sind. In [4] wird anhand eines größeren Experiments gezeigt, daß Werte zwischen 0.05 und 0.15 sinnvoll sind, um intuitive Begriffe wie "etwas ungenau" oder "ziemlich ungenau" im hier vorliegenden Kontext zu quantifizieren. Größere Werte führen zu großen, stark überlappenden Unsicherheitsbereichen und zu einer zu starken Aufblähung der Präferenzlisten.

5.3 Einflüsse auf Präferenzlisten

Wir haben oben zwei Arten der Entstehung von Unsicherheitsbereichen von Ähnlichkeiten kennengelernt: implizit durch Nullwerte und explizit durch Ungenauigkeitsangaben. In diesem Abschnitt diskutieren wir die Konsequenzen für Präferenzlisten.

Sofern bei einer Ähnlichkeit beide Ursachen für Unsicherheit auftreten, ist es in diesem Kontext am sinnvollsten, die Unsicherheitsbereiche im Sinne von Intervallen zu vereinigen.

Sortierung der Präferenzlisten. Die Unsicherheitsbereiche der Ähnlichkeiten können die Reihenfolge innerhalb einer Präferenzliste nicht beeinflussen, weil keine nutzbaren statistischen Informationen darüber vorliegen, wie sich die möglichen Endversionen der Modellelemente von den aktuell vorhandenen unterscheiden. Allerdings kann die Reihung in Einzelfällen unsicher werden. Betrachten wir als Beispiel eine Korrespondenz zwischen den Elementen $e1$ und $e2$. Angenommen, $e2$ stand in der Präferenzliste von $e1$ auf Platz 3. Ferner habe:

[5] Die Dreiecksungleichung gilt für Ähnlichkeiten nur unter bestimmten Voraussetzungen, ist aber als Schätzung des Unsicherheitsbereichs durchaus brauchbar.

– Platz 1 die Ähnlichkeit 0.71 und Unsicherheitsbereich [0.51, 0.76]
– Platz 2 die Ähnlichkeit 0.58 und Unsicherheitsbereich [0.45, 0.65].

Die Anordnung der Plätze 3 und 4 ist unsicher, weil ihre Unsicherheitsbereiche überlappen.

Eine Korrespondenz, bei der wenigstens ein Element auf einem unsicher sortierten Platz steht, ist unsicher. Diese Form der Unsicherheit kommt zu den anderen in Abschnitt 4.1 aufgelisteten Indizien hinzu. Um eine Informationsüberflutung zu vermeiden, sollte ein binäres Gesamturteil über die Unsicherheit einer Korrespondenz gebildet und durch ein Symbol im Elementbaum angezeigt werden. Zusätzliche Informationen über die Ursachen der Unsicherheit können in Form von Warnungen ausgegeben werden (s. Bild 1).

Mindestähnlichkeit. Von Unsicherheitsbereichen betroffen ist ferner die Mindestähnlichkeit: wenn eine Ähnlichkeit $S(e1, e2)$ unter der Mindestähnlichkeit liegt – also normalerweise $e1$ und $e2$ keine Korrespondenz mehr bilden können –, kann trotzdem die *Obergrenze* des Unsicherheitsbereichs deutlich darüber liegen, d.h. unter günstigen Umständen können die beiden Elemente doch noch korrespondieren.

Die offensichtliche Lösung besteht darin, zusätzlich diejenigen Elemente in die Präferenzlisten aufzunehmen, deren Unsicherheitsbereich die Mindestähnlichkeit beinhaltet. In einer umfangreichen Fallstudie [4] wurden die Auswirkungen dieser Lösung anhand mehrerer Teilmodelle (Klassendiagramme) evaluiert; hierbei wurden Standardeinstellungen für den Vergleich von (Entwurfs-) Klassendiagrammen benutzt. Es konnten durch die Erweiterung der Präferenzlisten in der Tat weitere Korrespondenzen gewonnen werden, allerdings nur in wenigen Fällen.

Praktisch der gleiche Nutzeffekt konnte indessen auch erreicht werden, indem die Merkmalsgewichtungen gegenüber den Standardeinstellungen modifiziert wurden. Konkret wurden die hohen Gewichtungen von Namen um ca. ein Viertel reduziert und die Gewichtungen anderer Merkmale, insb. der "Umgebung" der Elemente, entsprechend erhöht. Im Endeffekt war die hierdurch implementierte Heuristik besser an die Editiervorgänge der frühen Phasen adaptiert; nach dieser Optimierung der Merkmalskonfiguration konnten durch die erweiterten Präferenzlisten keine zusätzlichen korrekten Korrespondenzen mehr gewonnen werden.

Letztlich konnten in der Fallstudie [4] keine überzeugenden Argumente gefunden werden, die Unsicherheitsbereiche für die Ausweitung der Präferenzlisten zu nutzen. Als wesentlich wichtiger erwies sich die Anpassung der Ähnlichkeitsdefinition an die Verhältnisse in den frühen Phasen.

6 Resümee

Der Abgleich von Teilmodellen in den frühen Entwicklungsphasen unterliegt deutlich anderen Randbedingungen als das Mischen von Modellen in den späte-

ren Phasen. Hieraus ergeben sich neue Anforderungen an Vergleichs- und Mischwerkzeuge. Aus mehreren Gründen ist es wichtig, vom System berechnete ("vorgeschlagene") Matchings korrigieren zu können. Durch erweiterte Matching-Algorithmen können relevante Informationen gewonnen werden, welche Korrespondenzen unsicher sind und welche alternativen Korrespondenzen in die engere Wahl kommen. Die These, durch explizite Angaben zur Ungenauigkeit von Modellelementen ansonsten "übersehene" Korrespondenzen finden zu können, wurde in einer Fallstudie evaluiert. Der erzielte Nutzen war beschränkt, und als sinnvollere Maßnahme erwies sich, die Ähnlichkeitskriterien anzupassen, d.h. wenn der gleiche Modelltyp auch in den späteren Phasen verwendet wird, können die Ähnlichkeitskriterien nicht unverändert übernommen werden.

Danksagung. Die Motivation zu diesem Papier entstand im Rahmen einer Kooperation mit dem Lehrstuhl Informations- und Technikmanagement (Thomas Herrmann), U. Bochum, in der das Differenzframework SiDiff [12] an SeeMe [5] angepaßt wurde. Beteiligt an diesem Vorhaben war ferner Michael Goedicke, U. Duisburg-Essen. Allen Beteiligten sei für ihre Unterstützung gedankt.

Literatur

[1] Cheng, B.; Atlee, J.: Research Directions in Requirements Engineering; p.285-303 in; Proc. Future of Software Engineering; ACM; 2007; ISBN 0-7695-2829-5;

[2] Förtsch, S.; Westfechtel, B.: Differencing and Merging of Software Diagrams - State of the Art and Challenges; p.90-99 in: Proc. 2nd Intl. Conf. Software and Data Technologies (ICSOFT 2007), Barcelona; INSTICC Press; 2007

[3] Goedicke, M.; Herrmann, Th.: A Case for ViewPoints and Documents; p.62-84 in: Proc. 14th Monterey Workshop; LNiCS 5320, Springer; 2008

[4] Gorek, G.: Untersuchungen zum Abgleich vager Modelle in der Systemanalyse; Diplomarbeit, Fachgruppe Praktische Informatik, Universität Siegen; 2010; http://pi.informatik.uni-siegen.de/CVSM/Go2010DA.html

[5] Herrmann, Th.; Loser, K.: Vagueness in models of socio-technical systems; Behaviour and Information Technology 18:5, p.313-323; 1999

[6] IEEE Std. 1471-2000 Recommended Practice for Architectural Description of Software-intensive Systems; IEEE; 2000 (entspricht ISO/IEC 42010:2007)

[7] Kelter, U.; Schmidt, M.; Wenzel, S.: Architekturen von Differenzwerkzeugen für Modelle; p.155-168 in: Proc. Software Engineering 2008; LNI 121, GI e.V.; 2008

[8] Kelter, U.; Wehren, J.; Niere, J.: A Generic Difference Algorithm for UML Models; p.105-116 in: Proc. Software Engineering 2005; LNI 64, GI; 2005

[9] Kolovos, D. S.; Ruscio, D.; et al.: Different Models for Model Matching: An Analysis Of Approaches To Support Model Differencing; p.1-6 in: Proc. 2009 ICSE Workshop on Comparison and Versioning of Software Models; IEEE; 2009

[10] Ohst, D.; Welle, M.; Kelter, U.: Differences between Versions of UML Diagrams; p.227-236 in: Proc. ESEC/FSE 2003, Helsinki; ACM; 2003

[11] Perrouin, G.; Brottier, E.; Baudry, B.; Le, Y.: Traon: Composing Models for Detecting Inconsistencies: A Requirements Engineering Perspective; in: Proc. REFSQ2009; 2009

[12] SiDiff Differenzwerkzeuge; http://www.sidiff.org; 2010

Zwei Metriken zum Messen des Umgangs mit Zugriffsmodifikatoren in Java

Christian Zoller
Axel Schmolitzky

Arbeitsbereich Softwaretechnik
Universität Hamburg
Vogt-Kölln-Str. 30
22527 Hamburg
christian.zoller@informatik.uni-hamburg.de
schmolit@informatik.uni-hamburg.de

Abstract: Wie viele objektorientierte Programmiersprachen bietet Java die Möglichkeit, über Modifikatoren die Zugreifbarkeit von Typen, Methoden und Feldern in mehreren Stufen einzuschränken. So können für unterschiedliche Gruppen von Klienten differenzierte Schnittstellen definiert werden. Es zeigt sich jedoch, dass in der Praxis die gebotenen Möglichkeiten nicht voll ausgeschöpft werden. Wir beschreiben zwei neue Metriken, mit denen sich der angemessene Umgang mit Zugriffsmodifikatoren in Java messen lässt, sowie ein Werkzeug, das diese Metriken berechnet und beim Einschränken von Schnittstellen hilfreich sein kann. Wir haben unseren Ansatz in zwei kommerziellen Projekten und zwölf Open-Source-Projekten erprobt. Dabei wurde deutlich, dass Zugriffsmodifikatoren oft großzügiger gewählt werden als notwendig.

1 Einleitung

Das Definieren von Schnittstellen und die Kapselung von Implementationsdetails sind zentrale Bestandteile eines jeden Softwareentwurfs. Eine hohe Kapselung erleichtert nicht nur die Wiederverwendbarkeit und Änderbarkeit von Software, da unabhängige, klar abgegrenzte Module sich leichter austauschen lassen, sondern auch die Verständlichkeit, denn je weniger Informationen Module untereinander austauschen, umso leichter ist ihr Zusammenspiel zu verstehen. Moderne objektorientierte Sprachen unterstützen diese Prinzipien durch die Mechanismen der Zugriffskontrolle. Bei der Deklaration von Elementen wie Klassen, Methoden oder Variablen kann die Menge der Klienten festgelegt werden, aus denen auf das deklarierte Element zugegriffen werden darf. Je kleiner diese Menge ist, desto höher ist die Kapselung [Sny86], [Mey97, S. 47: „Few Interfaces Rule“].

1.1 Typen-Schnittstellen in Java

Typen werden in Java durch Klassen und Interfaces definiert. Die Zugriffskontrolle wird über die Zugriffsmodifikatoren `private`, `protected` und `public` geregelt [GJSB05, § 6.6]. Auf ein `private` deklariertes Klassen-Member darf nur innerhalb der eigenen Top-Level-Klasse zugegriffen werden. `protected` erlaubt den Zugriff innerhalb des eigenen Pakets und zusätzlich durch Unterklassen außerhalb des Pakets. `public` deklarierte Elemente dürfen von allen Klienten der Klasse verwendet werden. Lässt man den Zugriffsmodifikator ganz weg, darf das deklarierte Member nur im eigenen Paket verwendet werden. Diese Zugriffsebene wird auch default oder *package-private* genannt.
Die zugreifbaren Member einer Klasse bilden deren Schnittstelle. Aufgrund der Möglichkeit die Zugreifbarkeit von Membern in den genannten Stufen einzustellen, besitzen Klassen nicht nur jeweils *eine* Schnittstelle, sondern verschiedene für unterschiedliche Mengen von Klienten: eine für Klienten innerhalb des eigenen Pakets, eine für Unterklassen außerhalb des eigenen Pakets und eine für alle restlichen Klienten.
Der Begriff der Schnittstelle ist somit von dem Java-Konstrukt Interface zu unterscheiden. Doch auch Interfaces definieren Schnittstellen. Diese bestehen aus allen ihren Membern, da Interface-Member immer `public` sind.

1.2 Paket-Schnittstellen in Java

In Java verfügen nicht nur Typen, also Klassen und Interfaces, über Schnittstellen, sondern auch Pakete. Diese bestehen aus den enthaltenen, zugreifbaren Top-Level-Typen und deren zugreifbaren Membern. Für Top-Level-Typen gibt es jedoch nur zwei mögliche Zugriffsstufen: `public` und default. Die Zugriffsmodifikatoren `private` und `protected` sind Klassen-Membern vorbehalten.
Verfolgt man das Prinzip größtmöglicher Kapselung, sollten sowohl die Schnittstellen von Typen als auch die von Paketen so klein wie möglich sein [Mey97, S. 48: „Small Interfaces Rule“].

1.3 Beobachtungen aus der Praxis

In der Praxis lässt sich jedoch feststellen, dass der Umgang mit Zugriffsmodifikatoren nicht so restriktiv gehandhabt wird, wie es vielleicht möglich wäre. Während die Regel, dass Exemplarvariablen grundsätzlich `private` sein sollten, allgemein anerkannt ist, wird z.B. das Kapseln von paketinternen Klassen oder Methoden deutlich seltener angewendet. Insbesondere bei Top-Level-Klassen scheint das einleitende `public`-Schlüsselwort ein Quasi-Standard zu sein.
Dies hat verschiedene Gründe. Schon ein Blick in einschlägige Lehrbücher (z.B. [BK09], [HC07], [RSSW10], [Sav09]) zeigt, dass das Schnittstellen-Konzept, das auf Klassenebene noch ausführlich mit der Unterscheidung zwischen `public` und `private` erläutert

wird, nur selten auf Paketebene übertragen und die default-Zugreifbarkeit oft nur am Rande erwähnt wird. In dem verbreiteten Quelltextanalyse-Werkzeug *PMD*[1] findet sich sogar eine Regel, die von der Verwendung der default-Zugreifbarkeit abrät – warum diese Empfehlung ausgesprochen wird, lässt sich dabei jedoch nicht nachvollziehen. Darüber hinaus scheuen sich Entwicklerinnen und Entwickler möglicherweise davor, Zugreifbarkeiten zu restriktiv zu wählen, um sich nicht in den eigenen Möglichkeiten zu beschneiden. Dadurch können jedoch ungewollte Abhängigkeiten entstehen, worunter letztlich die Qualität der Software leidet.

1.4 Unser Ansatz

Softwaremetriken sind ein Weg die Qualität von Software automatisiert und objektiv zu bewerten. Es liegt nahe, auch den Umgang mit Zugriffsmodifikatoren einer solchen Bewertung zugänglich zu machen. Zu diesem Zweck haben wir zwei Java-Metriken entwickelt, die den Anteil derjenigen Typen und Methoden messen, denen ein unnötig großzügiger Zugriffsmodifikator zugewiesen ist. Um zu großzügige Zugriffsmodifikatoren zu ermitteln und die Metriken zu berechnen, haben wir ein Werkzeug entwickelt und dieses an zwei kommerziellen und zwölf Open-Source-Projekten erprobt. Dabei stellten wir fest, dass ein großer Teil der verwendeten Zugriffsmodifikatoren großzügiger ist als eigentlich notwendig.
Die Zugreifbarkeit von Feldern lassen wir in unseren Betrachtungen außen vor, da die Regel, dass diese immer `private` sein sollten, hinreichend einfach ist und bereits mit Hilfe bestehender Werkzeuge überprüft werden kann (z.B. *PMD*, *FindBugs*[2]).// Im folgenden Abschnitt 2 führen wir zunächst einige Begriffe ein und definieren dann die beiden Metriken. Außerdem stellen wir das Werkzeug *AccessAnalysis* vor, ein Plug-In für die Entwicklungsumgebung *Eclipse*. Anschließend folgen unter 3. die Ergebnisse unserer exemplarischen Untersuchungen. In Abschnitt 4 diskutieren wir Einsatzmöglichkeiten und Grenzen der Metriken. In Abschnitt 5 gehen wir kurz auf ähnliche Arbeiten in diesem Bereich und mögliche Anschlussarbeiten ein. Am Ende folgt eine Zusammenfassung.

2 Zwei neue Metriken für Java

2.1 Begriffe

Das Nichtvorhandensein eines Zugriffsmodifikators, also die default-Zugreifbarkeit, wird im Folgenden zur sprachlichen Vereinfachung ebenfalls als Zugriffsmodifikator bezeichnet.
Ein Zugriffsmodifikator heißt *strenger* als ein anderer, wenn er weniger Klienten den Zugriff gewährt. Andersherum heißt er *großzügiger*. Da in Java die Zugriffsebenen klar hier-

[1] http://pmd.sourceforge.net
[2] http://findbugs.sourceforge.net

archisch gegliedert sind, ergibt sich die Reihenfolge `private` → **default** → `protected` → `public`, in der die Zugriffsmodifikatoren großzügiger und entgegengesetzt strenger werden.
Legt man die tatsächliche Verwendung der Typen (Klassen und Interfaces) und Methoden innerhalb eines Java-Systems zu Grunde, ergibt sich auf Basis der Sprachdefinition für jedes dieser Elemente ein strengster Zugriffsmodifikator, der ausreicht, um alle Benutzungen des Elements innerhalb des Systems zu gewährleisten. Diesen Zugriffsmodifikator nennen wir *minimalen Zugriffsmodifikator* des Typs bzw. der Methode. Der Zugriffsmodifikator, der einem Element im Quelltext tatsächlich zugeordnet ist, nennen wir *tatsächlicher Zugriffsmodifikator*.
Ist der tatsächliche Zugriffsmodifikator eines Elements großzügiger als sein minimaler, bezeichnen wir diesen in diesem Zusammenhang als *zu großzügig*.

2.2 Inappropriate Generosity with Accessibility of Types (IGAT)

Die Metrik „Inappropriate Generosity with Accessibility of Types“ (IGAT) sei folgendermaßen definiert:

$$IGAT(U,P) = \begin{cases} 0, & \text{wenn } |T(U)| = 0 \\ \dfrac{|T^*(U,P)|}{|T(U)|}, & \text{sonst} \end{cases} \tag{1}$$

wobei

- P der Quelltext aller Übersetzungseinheiten eines Java-Programms ist,
- U mit $U \subseteq P$ eine Quelltext-Teilmenge aus P (z.B. der gesamte Quelltext, ein Paket oder eine Typdeklaration),
- $T(U)$ die Menge aller in U deklarierten Typen und $|T(U)|$ deren Anzahl,
- $T^*(U,P)$ mit $T^*(U,P) \subseteq T(U)$ die Menge aller in U deklarierten Typen mit zu großzügigem Zugriffsmodifikator und $|T^*(U,P)|$ deren Anzahl. Zu großzügig bedeutet hier im Vergleich zu den minimalen Zugriffsmodifikatoren, die auf Basis der Benutzung der Typen innerhalb von P zu bestimmen sind.

Am Anfang der Berechnung steht also die Bestimmung der minimalen Zugriffsmodifikatoren auf Basis des gesamten Quelltextes P. Danach wird dann für die zu vermessende Quelltext-Teilmenge U der Anteil der Typen berechnet, deren Zugriffsmodifikator zu großzügig ist.

Beispiel:

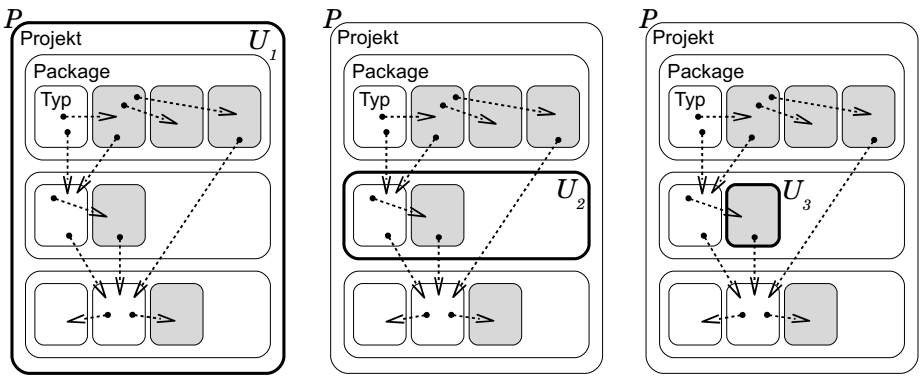


Abbildung 1: IGAT-Berechnung

Abb. 1 zeigt dreimal die schematische Darstellung eines Java-Projekts P, das in drei Pakete mit insgesamt neun Typdeklarationen aufgeteilt ist. Die grau hinterlegten Typen sind diejenigen, deren Zugriffsmodifikator nach Analyse der Benutztbeziehungen (Pfeile) innerhalb von P als zu großzügig erkannt wurde. Je nachdem für welche Untermenge U_i von P der IGAT-Wert berechnet werden soll, ergibt sich die Gesamtzahl der Typen $|T(U_i)|$ und die Anzahl $|T(U_i, P)|$ derjenigen Typen mit zu großzügigem Zugriffsmodifikator. Die Mengen P ist in allen drei Fällen dieselbe.

2.3 Inappropriate Generosity with Accessibility of Methods (IGAM)

Analog zur Typen-Metrik IGAT sei für Methoden die Metrik „Inappropriate Generosity with Accessibility of Methods“ (IGAM) definiert:

$$IGAM(V,P) = \begin{cases} 0, & \text{wenn } |M(V)| = 0 \\ \dfrac{|M^*(V,P)|}{|M(V)|}, & \text{sonst} \end{cases} \tag{2}$$

wobei

- P auch hier der Quelltext aller Übersetzungseinheiten eines Java-Programms ist,
- V mit $V \subseteq P$ eine Quelltext-Teilmenge aus P (z.B. der gesamte Quelltext, ein Paket, eine Typ- oder Methodendeklaration),

- $M(V)$ die Menge aller in V deklarierten Methoden und $|M(V)|$ deren Anzahl,
- $M^*(V,P)$ mit $M^*(V,P) \subseteq M(V)$ die Menge aller in V deklarierten Methoden mit zu großzügigem Zugriffsmodifikator und $|M^*(V,P)|$ deren Anzahl. Zu großzügig bedeutet auch hier im Vergleich zu den minimalen Zugriffsmodifikatoren, die auf Basis der Benutzung der Methoden innerhalb von P zu bestimmen sind.

2.4 Das Werkzeug *AccessAnalysis*

Um die vorgestellten Metriken berechnen zu können, haben wir *AccessAnalysis* [3] entwickelt, ein Plug-In für die Entwicklungsumgebung *Eclipse* [4]. Es kann auf Grundlage eines oder mehrerer Projekte die minimalen Zugriffsmodifikatoren aller enthaltenen Typen und Methoden ermitteln und anhand dieser IGAT und IGAM berechnen. In einer baumartigen Tabelle (Abb. 2) werden die Metriken sowohl auf Projektebene als auch für jedes Paket, jeden Typ und jede Methode (nur IGAM) ausgegeben. Zudem werden die tatsächlichen und minimalen Zugriffsmodifikatoren der Typen und Methoden gegenübergestellt.
Für den Fall, dass ein Typ oder eine Methode innerhalb der analysierten Projekte gar nicht benutzt wird, wird der Pseudo-Zugriffsmodifikator not used als minimaler angegeben.

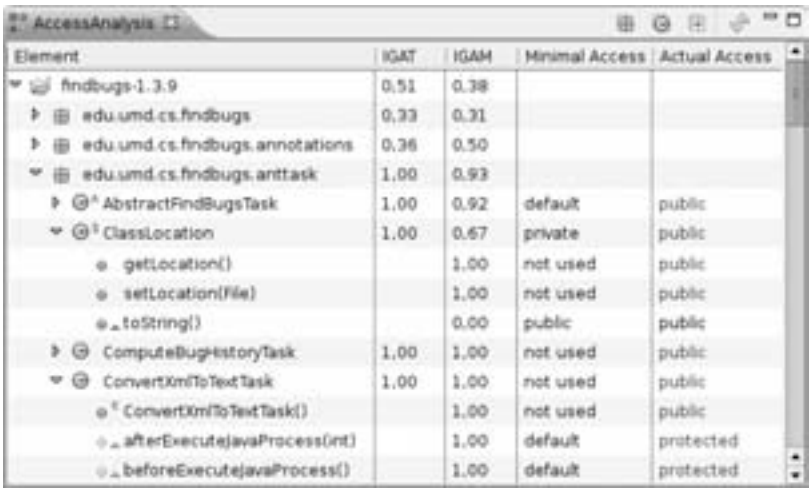


Abbildung 2: Ergebnisausgabe des *Eclipse*-Plug-Ins *AccessAnalysis*

[3] http://accessanalysis.sourceforge.net
[4] http://eclipse.org

3 Erprobung

3.1 Einsatz in kommerziellen Projekten

Die ersten Messungen unter realen Bedingungen haben wir in zwei Softwareunternehmen durchgeführt, bei denen wir jeweils ein Projekt analysieren durften. Die Ergebnisse waren noch nicht sehr aussagekräftig, da wir feststellen mussten, dass durch den Einsatz verschiedenster Frameworks, wie *JUnit*, *Hibernate*, *Spring*, *EJB* etc., der Anteil der Typen und Methoden, die per Reflection aufgerufen werden, extrem hoch ist. Hinzu kommen viele Elemente, die nur für die Benutzung in JSPs bestimmt sind. Da *AccessAnalysis* bisher jedoch nur einfachen Java-Quelltext analysieren kann und so viele Benutzungen nicht registriert werden konnten, führte dies zu einer großen Anzahl falscher Meldungen von zu großzügigen Zugriffsmodifikatoren.
Doch bestärkten diese ersten Versuche unsere These, dass die default-Zugreifbarkeit so gut wie nie eingesetzt wird. So waren im ersten Projekt von insgesamt 2.613 Typen nur sechs default deklariert und damit im eigenen Paket gekapselt; von 22.210 Methoden waren es lediglich 29. Im zweiten, deutlich kleineren Projekt sah das Bild ähnlich aus. Hier waren von 355 Typen nur zwei default und von 4.143 Methoden lediglich eine.

3.2 Vermessung von Open-Source-Software

Um trotz der in den ersten Testläufen aufgezeigten Schwierigkeiten zu verwertbaren Ergebnissen zu kommen, haben wir zwölf Open-Source-Projekte ausgewählt, bei denen Falschmeldungen von zu großzügigen Zugriffsmodifikatoren weitestgehend vermieden werden konnten. Bedingung war auch hier, dass es sich um eigenständige Anwendungen handelt und nicht um Bibliotheken oder Frameworks. Darüber hinaus sollten die Projekte keine JSPs enthalten und Reflection, sei es im eigenen Code oder durch eingesetzte Frameworks, durfte keine große Rolle spielen. Weiterhin war gefordert, dass sich der Quelltext nach dem Import in *Eclipse* und dem Einbinden aller benötigten Bibliotheken ohne weitere Modifikation übersetzen ließ. Die in Tab. 1 aufgeführten, von uns ausgewählten Anwendungen erfüllten weitestgehend diese Forderungen.
Die Größe der Projekte reicht von 29 Typen in 4 Paketen (*JDepend*) bis zu 1.319 Typen in 53 Paketen (*FindBugs*). Nach dem Import der Quelltexte in *Eclipse* haben wir die enthaltenen *JUnit*-Testklassen entfernt. Da diese sowie die in ihnen deklarierten Testmethoden immer `public` sein müssen, obwohl sie anderweitig im Quelltext nicht verwendet werden, hätten auch sie das Ergebnis verfälscht. Zwar fielen so auch die Benutztbeziehungen zwischen den Testklassen und den zu testenden Elementen weg, doch zeigte ein Vergleich, dass sich das Entfernen der Tests nur wenig auf die minimalen Zugriffsmodifikatoren der anderen Typen und Methoden auswirkte. Bei einigen Projekten enthielt der verfügbare Quelltext von vornherein keine Tests. Neben den *JUnit*-Testklassen haben wir ebenso Beispielcode, der bei Quelltextanalyse-Tools, wie z.B. *PMD*, mitgeliefert wurde, entfernt.

BlueJ	3.0.0	Java-IDE	http://bluej.org
Cobertura	1.9.4.1	Java-Test-Coverage-Tool	http://cobertura.sourceforge.net
DoctorJ	5.1.2	Javadoc-Analysetool	http://www.incava.org/projects/java/doctorj
FindBugs	1.3.9	Java-Fehleranalysetool	http://findbugs.sourceforge.net
FreeCol	0.9.3	Strategiespiel	http://www.freecol.org
FreeMind	0.8.1	Illustrationsprogramm	http://freemind.sourceforge.net
JabRef	2.6	Literaturverwaltung	http://jabref.sourceforge.net
JDepend	2.9	Java-Metriktool	http://www.clarkware.com/software/JDepend.html
jrDesktop	0.3.1.0	Fernwartungstool	http://jrdesktop.sourceforge.net
PDFsam	2.2.0	PDF-Werkzeug	http://www.pdfsam.org
PMD	4.2.5	Java-Fehleranalysetool	http://pmd.sourceforge.net
Sweet Home 3D	2.5	Illustrationsprogramm	http://www.sweethome3d.com

Tabelle 1: Analysierte Open-Source-Projekte.

3.3 Ergebnisse

Abb. 3 zeigt die IGAT- und IGAM-Werte für die gesamten Quelltexte der einzelnen, vermessenen Projekte. Man sieht, dass bei den meisten ein nicht unerheblicher Teil der Zugriffsmodifikatoren zu großzügig ist. Der Anteil der Methoden mit zu großzügigem Zugriffsmodifikator bewegt sich dabei relativ eng in einem Bereich von 25 bis 44 %. Nur das Projekt *DoctorJ* liegt mit einem IGAM-Wert von 71 % deutlich außerhalb dieses Bereichs. Die IGAT-Werte zeigen eine deutlich stärkere Schwankung und insgesamt höhere Werte. Die auffälligsten Ergebnisse haben hier *JabRef* und *JDepend*, bei denen jeweils 59 % der Typen einen zu großzügigen Zugriffsmodifikator haben, sowie auf der anderen Seite *PDFsam*, bei dem es nur 16 % sind.
Tab. 2 stellt die Verteilung der tatsächlichen und minimalen Zugriffsmodifikatoren gegenüber. Auch hier zeigt sich, dass die default-Zugreifbarkeit so gut wie nie eingesetzt wird. Durch die Aufschlüsselung der Zahlen der Typen in Top-Level- und Membertypen wird deutlich, dass dies besonders die Top-Level-Typen betrifft. Hier sind lediglich 104 von 4.570 Typen default deklariert, obwohl es 1.259 sein könnten. Doch auch bei den Membertypen, wo der default-Zugriffsmodifikator einen deutlich höheren Anteil hat, scheint dieser eher zufällig zum Einsatz zu kommen. Denn beim größten Teil der default-Membertypen ist `private` der minimale Zugriffsmodifikator, stattdessen könnten auch hier einige der `public`-Typen default sein.

4 Bewertung

Unsere Grundannahme ist, dass Zugriffsmodifikatoren, die sich an der tatsächlichen Benutzung orientieren, die Benutzungsebenen explizit machen und so die Verständlichkeit

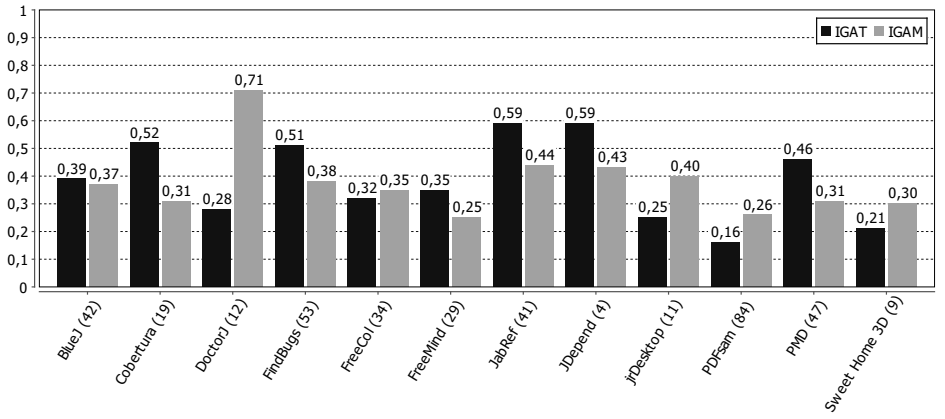


Abbildung 3: IGAT- und IGAM-Gesamtergebnisse der untersuchten Open-Source-Projekte. In Klammern: Anzahl der Pakete.

	Alle Typen		Nur Top-Level-Typen		Nur Membertypen		Methoden	
	tats.	min.	tats.	min.	tats.	min.	tats.	min.
`public`	4.966	2.926	4.466	2.770	500	156	39.060	22.751
`protected`	40	7			40	7	2.347	1.646
default	556	1.391	104	1.259	452	132	1.084	7.325
`private`	592	1.251			592	1.251	8.704	13.438
not used		579		541		38		6.035
gesamt	6.154		4.570		1.584		51.195	

Tabelle 2: Verteilung der Zugriffsmodifikatoren in den analysierten Projekten.

von Software erhöhen. Ein strenger Umgang mit Zugriffsmodifikatoren schützt vor unnötigen Abhängigkeiten und erleichtert so die Änderung einzelner Teile einer Software. Negative Auswirkungen könnten minimale Schnittstellen auf Erweiterbarkeit, Wiederverwendbarkeit und Testbarkeit haben, wenn auf vorhandene Funktionalität ohne Änderung am Quelltext nicht zugegriffen werden kann.
In realen Softwareprojekten kann es etliche Gründe geben, weshalb für einen Typ oder eine Methode nicht immer der minimale Zugriffsmodifikator gewählt wird. Ein naheliegender Grund, weshalb ein Element einen zu großzügigen Zugriffsmodifikator haben kann, ist beispielsweise, dass ein Klient erst noch geschrieben werden muss. Auch wenn ein Projekt nicht als eigenständige Anwendung, sondern als Bibliothek oder Framework entwickelt wird, werden mit großer Wahrscheinlichkeit viele Schnittstellen Elemente enthalten, die innerhalb des Projekts selbst gar nicht benutzt werden. Ebenso spielen bei der Entwicklung eines stark komponentenbasierten Systems, in dem einzelne Komponenten in anderen Pro-

jekten wiederverwendet werden sollen, beim Entwurf der Schnittstellen sicher nicht nur Minimierungsaspekte eine Rolle.
Desweiteren ist auch denkbar, dass der minimale Zugriffsmodifikator eines Elements dessen aus softwaretechnischer Sicht angemessene Zugriffsebene sogar überschreitet. Dies ist dann der Fall, wenn mangelhafte Kapselung bereits durch Klienten ausgenutzt wird.
Der minimale Zugriffsmodifikator kann daher nur als Annäherung für den „richtigen" Zugriffsmodifikator angesehen werden. Seine Bestimmung auf Basis einer abgeschlossenen Quelltextmenge (P) ergibt nur Sinn, wenn sich die Benutzung des zugehörigen Elements auf diese Quelltextmenge beschränkt. In einigen Fällen kann es sinnvoll sein, P auf den Quelltext verwandter Anwendungen zu erweitern.
Der Einsatz der Metriken IGAT und IGAM bietet sich daher vor allem in Projekten an, in denen solch eine abgeschlossene Quelltextmenge angenommen werden kann. Dies haben wir bei der Wahl der von uns untersuchten Anwendungen entsprechend berücksichtigt. In heutigen Systemen müssen neben dem reinen Java-Quelltext auch andere Dokumente bei der Analyse berücksichtigt werden, etwa JSPs oder Konfigurationsdateien von eingesetzten Frameworks. Sind diese Voraussetzungen gegeben, könnten die Metriken in individuellen Qualitätsmodellen als Maß für die Kapselung eingesetzt werden. Ob ein strengerer Umgang mit Zugriffsmodifikatoren tatsächlich Einfluss auf übergeordnete Qualitätsmerkmale wie Änderbarkeit oder Verständlichkeit hat, müsste jedoch empirisch noch nachgewiesen werden.
Im Rahmen des Refactorings können Quelltexteinheiten, die durch hohe IGAT- oder IGAM-Werte auffallen, einer gesonderten Überprüfung ihrer Schnittstellen unterzogen werden. Eine automatische Minimierung der Zugreifbarkeiten erscheint zwar grundsätzlich möglich, jedoch nicht angebracht. Wie dargestellt kann es viele Gründe für einen großzügigeren Zugriffsmodifikator als den minimalen geben. Der minimale Zugriffsmodifikator kann somit als nützlicher Orientierungspunkt dienen, letztlich sollte aber immer eine Entwicklerin oder ein Entwickler über die tatsächliche Zugreifbarkeit entscheiden.
Die Ergebnisse der Untersuchung zeigen, dass solch eine Überarbeitung der Schnittstellen in vielen Projekten angebracht erscheint. Dabei könnten vor allem Pakete als Abstraktionseinheit ein höheres Gewicht bekommen. Die Betrachtung von Paket-Schnittstellen ist nur dann nützlich, wenn Typen bewusst hinzugefügt oder ausgelassen werden. Solange alle Top-Level-Typen wahllos `public` deklariert werden, ist das Schnittstellen-Konzept auf Paketebene wertlos.

5 Verwandte Arbeiten und Ausblick

Der MOOD-Metriken-Katalog von Brito e Abreu [AC94] enthält zwei Metriken, die die Kapselung von Methoden und Feldern zum Gegenstand haben: „Method Hiding Factor" und „Attribute Hiding Factor". Diese geben den Anteil der nicht-öffentlichen Methoden bzw. Felder aller Klassen an. Cao und Zhu [CZ08] erweitern diese Metriken, indem sie die Anzahl der Klassen, die auf eine Methode oder Attribut Zugriff haben, mit einrechnen.
Bouillon et al. beschreiben in [BGS08] ein *Eclipse*-Plug-In, das ebenfalls minimale Zu-

griffsmodifikatoren ermitteln kann, allerdings nur von Methoden. Dieses berechnet keine Metriken, sondern bietet Werkzeug-Unterstützung für die Wahl der Zugriffsmodifikatoren während der Entwicklung.
Eine detailliertere Darstellung der hier vorgestellten Metriken IGAT und IGAM, ihrer Berechnung und der Bestimmung der minimalen Zugriffsmodifikatoren sowie dem *Access-Analysis*-Plug-In und der durchgeführten Fallstudien findet sich in [Zol10]. Darin wird auch ein alternativer Ansatz zur Berechnung der Metriken vorgestellt, bei dem der Anteil der Typen und Methoden mit zu großzügigem Zugriffsmodifikator nicht im Verhältnis zu allen Typen bzw. Methoden angegeben wird, sondern nur zu denjenigen, bei denen sich die Zugreifbarkeit überhaupt einschränken lässt, also zur Menge der Typen und Methoden, deren minimaler Zugriffsmodifikator strenger ist als `public`.

Um z.B. auch JSPs und Konfigurationsdateien von Frameworks untersuchen zu können, müsste *AccessAnalysis* um einen Erweiterungsmöglichkeit ergänzt werden, so dass weitere Analyse-Module als Plug-In hinzugefügt werden könnten.
Eine Übertragung der hier vorgestellten Konzepte auf andere objektorientierte Programmiersprachen wäre wünschenswert, ist aber nur unter bestimmten Voraussetzungen möglich. Zunächst muss das Typsystem der Sprache die vollständige statische Analyse der Benutztbeziehungen zulassen. Darüber hinaus beruht das Prinzip des minimalen Zugriffsmodifikators auf der klaren hierarchischen Gliederung der Zugriffsebenen in Java und der sich daraus ergebenden Ordnung der Zugriffsmodifikatoren. Solch eine Ordnung lässt sich nicht in allen objektorientierten Sprachen ableiten. So gibt es beispielsweise in C# die orthogonalen Zugriffsebenen `internal` und `protected`. `internal` erlaubt den Zugriff innerhalb der gleichen Assembly und `protected` durch alle Unterklassen [ECM06, § 10.5]. Es bedürfte somit einer geeigneten Konvention, ob für Elemente, die nur durch Unterklassen innerhalb der gleichen Assembly benutzt werden, `internal` oder `protected` als minimaler Zugriffsmodifikator gilt.

6 Zusammenfassung

Wir haben in diesem Artikel das Konzept des minimalen Zugriffsmodifikators für Java vorgestellt und diskutiert. Der minimale Zugriffsmodifikator eines Typs oder einer Methode entspricht dem strengsten Zugriffsmodifikator, der ausreicht, um alle Benutzungen des Typs bzw. der Methode innerhalb des Quelltextes eines Systems zu erlauben. Darauf aufbauend haben wir die Metriken IGAT und IGAM als Maße der Übereinstimmung von minimalen und tatsächlichen Zugriffsmodifikatoren definiert. Die Bestimmung der minimalen Zugriffsmodifikatoren und die Berechnung der Metriken haben wir mit dem *Eclipse*-Plug-In *AccessAnalysis* realisiert.
Die Analyse von zwölf Open-Source-Projekten zeigte, dass oft großzügigere Zugreifbarkeiten gewährt werden als eigentlich notwendig. Der Einsatz in zwei kommerziellen Projekten machte allerdings deutlich, dass die Bestimmung der minimalen Zugriffsmodifikatoren in heutigen Systemen nur dann zuverlässig stattfinden kann, wenn neben dem eigentlich Java-Quelltext auch andere Dokumente wie JSPs oder Konfigurationsdateien von Frameworks berücksichtigt werden. Dennoch ließ sich auch hier erkennen, dass besonders

von der Kapselung von Typen innerhalb von Paketen, also der default-Zugreifbarkeit für Java-Klassen und -Interfaces, so gut wie nie Gebrauch gemacht wird.
Wir haben die Vor- und Nachteile einer hohen Kapselung abgewogen und festgestellt, dass das Konzept der Schnittstelle nur dann wertvoll ist, wenn mit Zugreifbarkeiten maßvoll umgegangen wird. Das Ziel der Minimalität ist dabei nicht in allen Fällen zu verfolgen, dennoch kann der minimale Zugriffsmodifikator als Orientierungspunkt für die Wahl des tatsächlichen Zugriffsmodifikators dienen.

Literatur

[AC94] Fernando Brito e Abreu und Rogério Carapuça. Object-Oriented Software Engineering: Measuring and Controlling the Development Process. In *Proceedings of the 4th International Conference on Software Quality (QSIC)*, McLean, VA, USA, 1994.

[BGS08] Philipp Bouillon, Eric Großkinsky und Friedrich Steimann. Controlling Accessibility in Agile Projects with the Access Modifier Modifier. In *Proceedings of TOOLS (46)*, Seiten 41–59, 2008.

[BK09] David J. Barnes und Michael Kölling. *Java lernen mit BlueJ – Eine Einführung in die objektorientierte Programmierung*. Pearson Studium, München, 4. Auflage, 2009.

[CZ08] Yong Cao und Qingxin Zhu. Improved Metrics for Encapsulation Based on Information Hiding. In *The 9th International Conference for Young Computer Scientists (ICYCS 2008)*, Seiten 742–747, Los Alamitos, USA, 2008. IEEE Computer Society.

[ECM06] ECMA International. *Standard ECMA-334 – C# Language Specification*. Genf, Schweiz, 4th edition. Auflage, June 2006.

[GJSB05] James Gosling, Bill Joy, Guy Steele und Gilad Bracha. *The Java Language Specification*. Addison-Wesley, Upper Saddle River, USA, 3. Auflage, 2005.

[HC07] Cay S. Horstmann und Gary Cornell. *Core Java – Volume I: Fundamentals*. Prentice Hall, Upper Saddle River, USA, 8. Auflage, 2007.

[Mey97] Bertrand Meyer. *Object-oriented Software Construction*. Prentice Hall, Upper Saddle River, NJ, USA, 2nd edition. Auflage, 1997.

[RSSW10] Dietmar Ratz, Jens Scheffler, Detlef Seese und Jan Wiesenberger. *Grundkurs Programmieren in Java*. Carl Hanser Verlag, München, 5. Auflage, 2010.

[Sav09] Walter Savitch. *Absolute Java*. Pearson Education International, Boston, USA, 4. Auflage, 2009.

[Sny86] Alan Snyder. Encapsulation and Inheritance in Object-Oriented Programming Languages. In *OOPLSA '86: Conference proceedings on Object-oriented programming systems, languages and applications*, Seiten 38–45, New York, USA, 1986. ACM.

[Zol10] Christian Zoller. Ein Ansatz zur Messung der Kapselung in Java-Systemen. Diplomarbeit, Universität Hamburg, 2010.

Hallway: ein Erweiterbares Digitales Soziales Netzwerk

Leif Singer, Maximilian Peters

Fachgebiet Software Engineering
Leibniz Universität Hannover
Welfengarten 1
30167 Hannover
leif.singer@inf.uni-hannover.de
maximilian.peters@se.uni-hannover.de

Abstract: Social Software verwendet soziale Mechanismen, um beispielsweise Wissen besser in Organisationen verbreiten zu können, Kollaboration zu erleichtern oder das Engagement von Anwendern zu steigern. Durch die Wahl der Mechanismen können daher die Entwickler von Social Software entscheidenden Einfluss auf die Wirkungsweise der Gesamtanwendung nehmen. Bisher gibt es jedoch nur wenige wissenschaftliche Arbeiten, die soziale Mechanismen ihren Effekten zuordnen. Durch diesen Mangel an fundierten Richtlinien zum Entwurf von Social Software fließen Erfahrungen einzelner Entwickler, Spekulationen und Annahmen in die Entwicklung ein. Dadurch ist die die Erfüllung der Anforderungen an die Anwendung schlecht kontrollierbar.
Dieser Beitrag stellt ein digitales soziales Netzwerk (DSN) vor, das mit einem besonderen Fokus auf Erweiterbarkeit und Austauschbarkeit seiner Mechanismen entwickelt wurde. Ebenso wurden bereits einige Mechanismen in alternativen Ausführungen implementiert, so dass diese nun leicht ausgetauscht und Unterschiede im Nutzerverhalten sowie in der Ausbreitung von Informationen im DSN untersucht werden können. Diese Untersuchungen sollen zu Richtlinien für den Entwurf von Social Software führen, die Entwickler auf konkrete Softwareprojekte anwenden können sollen.

1 Einführung

Social Software verknüpft die Mitglieder ihrer Nutzergemeinde, wobei oft Effekte, die aus der impliziten oder expliziten Zusammenarbeit entstehen, nutzbar werden. Sie ist im Rahmen des „Web 2.0“ in den letzten Jahren sehr populär geworden. Die Idee selbst ist jedoch bereits seit Jahrzehnten in Gebrauch: die aktuellen Entwicklungen können als Weiterführung der Arbeiten aus der CSCW-Gemeinde (Computer Supported Cooperative Work) gesehen werden. Ein Beispiel für die implizite Zusammenarbeit ist das kollaborative Bewerten von Inhalten, das die Qualität von Empfehlungen verbessern kann. Explizite Zusammenarbeit bietet beispielsweise Google Docs – verschiedene Nutzer können gleichzeitig ein Dokument bearbeiten.

Ein digitales soziales Netzwerk (DSN) ist eine spezielle Form von Social Software, deren Fokus auf den Verbindungen zwischen den Benutzern liegt [BE08]. Dadurch können Informationen und Wissen durch das soziale Netz der Nutzer als Infrastruktur zur Verbreitung nutzen.

DSN werden beispielsweise zur Awareness-Steigerung in gemeinsamen Projekten, zur Expertensuche oder zum Wissensmanagement verwendet. Zum Software Engineering haben DSN zwei wichtige Bezüge.

1.1 Unterstützung der Softwareentwicklung durch Social Software

Einerseits kann Social Software eingesetzt werden, um Tätigkeiten im Software Engineering zu unterstützen. Dies ist besonders in geographisch oder auch organisatorisch verteilten Projekten interessant, da das wirkliche soziale Netzwerk zwischen den Projektteilnehmern schwächer ausgeprägt ist.

So wurden bereits spezielle anforderungsbezogene DSN eingesetzt, um die Kommunikation zwischen Entwicklern und Testern an verschiedenen Standorten zu verbessern [DM07]. Andere Arbeiten nutzen die in Versionsverwaltungssystemen implizit enthaltenen Informationen über das soziale Netzwerk der Entwickler, um durch Aufbereitung einen Überblick über Projektteilnehmer und -fortschritt zu generieren [BZ10a]. Abschnitt 3 nennt weitere Ansätze in dieser Richtung.

1.2 Ingenieursmäßige Verwendung sozialer Mechanismen

Für den vorliegenden Beitrag interessanter ist die Frage, wie man Mechanismen aus der Social Software systematisch und verlässlich in zu entwickelnde Systeme integrieren kann. Von Fortschritten in dieser Frage würden auch die oben genannten Ansätze profitieren. Populäre Beispiele wie Socialwok[1] zeigen, dass selbst klassische Anwendungen wie eine Textverarbeitung oder Tabellenkalkulation von der Integration sozialer Mechanismen profitieren können.

Um diese Mechanismen gezielt anwenden zu können ist es notwendig, ihre Voraussetzungen und Effekte zu kennen. So mag ein bestimmter Mechanismus nur in Organisationen mit besonders flachen Hierarchien den Wissensfluss zwischen den Mitarbeitern verbessern, starke Hierarchien könnten den Effekt ins Gegenteil umkehren. Wissen um diese Eigenschaften der Mechanismen wäre eine wertvolle Bereicherung des Software Engineering, da es ihren systematischen Einsatz erleichtern würde. Daher stellt dieser Beitrag ein DSN vor, das auf die Evaluierung solcher Mechanismen ausgelegt ist.

[1] http://socialwok.com/

Dieser Beitrag ist wie folgt aufgebaut: Im Anschluss an diese Einführung stellt Abschnitt 2 verschiedene soziale Mechanismen vor. Abschnitt 3 fasst verwandte Arbeiten aus den beiden erwähnten Kombinationsmöglichkeiten von Social Software und Software Engineering zusammen. Abschnitt 4 stellt schließlich „Hallway“ vor, das von uns entwickelte erweiterbare DSN. Der Beitrag schließt mit einer Zusammenfassung und einem Ausblick auf unsere weiteren Vorhaben.

2 Mechanismen in Digitalen Sozialen Netzwerken

Als charakteristische Funktionen eines DSN haben Boyd und Ellison Benutzerprofile, Verbindungen zwischen Benutzern sowie die Möglichkeit zur Anzeige der Verbindungen eines Benutzers identifiziert [BE08]. Meist können Benutzer auch kurze öffentliche Nachrichten und andere Inhalte wie Dokumente veröffentlichen. Auf einem *Activity Stream* erhält jeder Benutzer die Veröffentlichungen seiner Kontakte angezeigt und kann so dem aktuellen Geschehen in seinem sozialen Netzwerk folgen.

Das DSN wirkt dabei als Empfehlungssystem und Filter: Inhalte sind für einige Teilnehmer interessant, die sie dann an ihre eigenen Kontakte weiterreichen können. Auf diese Weise fließt Wissen durch das DSN bzw. die Organisation, bei der es eingesetzt wird. Um Struktur und Wirkungsweise des sozialen Netzwerk eines Benutzers zu beeinflussen, können einige Mechanismen ins DSN eingebracht oder vorhandene angepasst werden.

2.1 Kollaborative Empfehlungssysteme

Einige soziale Mechanismen zielen darauf ab, Informationen auch über die Kontakte eines einzelnen Benutzers hinaus zu transportieren. Ein populäres Beispiel hierfür ist die „Like“-Funktion von Facebook[2]: findet ein Benutzer einen für ihn interessanten Inhalt, so kann er dies durch einen Klick auf die „Like“-Schaltfläche ausdrücken. Dies erzeugt für diesen Nutzer eine neue, implizite Aktivität, die seinen Kontakten angezeigt wird. Auf diese Weise erreicht der eigentlich Inhalt auch Personen, die nicht zu den direkten Kontakten des Autors zählen. Mit den sog. „Retweets“ implementiert Twitter[3] einen Mechanismus mit einem sehr ähnlichen Effekt. Interessante Inhalte werden weitergeleitet, uninteressante werden herausgefiltert.

Zu ähnlichen Zwecken gibt es einige unter anderem aus Onlineshops bekannte Empfehlungsmechanismen. So werden bspw. die Produktbewertungen aller Käufer genutzt, um einzelnen Käufern interessante Produkte zu empfehlen.

Der folgende Abschnitt skizziert, wie nicht nur Inhalte, sondern auch Verbindungen selbst durch Empfehlungen und andere Mechanismen beeinflusst werden können.

[2] http://facebook.com
[3] http://twitter.com

2.2 Verbindungsmodelle

Die beiden klassischen Möglichkeiten, Nutzer eines DSN miteinander zu verbinden, sind das synchrone und das asynchrone Verbindungsmodell. Beim synchronen Modell beantragt ein Nutzer den Kontakt beim Anderen, der diesen dann bestätigen muss. Erst dann sind beide Nutzer miteinander verbunden – als populäres Beispiel verwendet Facebook dieses Modell.

Das asynchrone Modell wird beispielsweise durch Twitter genutzt. Verbindungen bestehen zunächst nur einseitig und müssen per Voreinstellung nicht bestätigt werden. Durch einen Kontakt abonniert ein Nutzer die Inhalte des anderen Nutzers. Oftmals wird dieser von diesem neuen Abonnement informiert, so dass er bei Interesse einen Kontakt in die Gegenrichtung herstellen kann.

Über diese beiden einfachsten Verbindungsmodelle sind noch viele weitere denkbar. Einige Beispiele skizziert die folgende Liste.

- Ein Kontakt zwischen Nutzern wird automatisch hergestellt, falls diese in ihrem Profil ähnliche Interessen angegeben haben oder ähnliche Inhalte als „interessant“ markieren.

- Kontakte werden automatisch hergestellt oder empfohlen, wenn Nutzer einen Schwellwert für räumliche oder organisatorische Nähe unterschreiten.

- Die Frequenz von ausgetauschten privaten Nachrichten zwischen Nutzern fließt in Kontaktempfehlungen oder die automatische Herstellung von Verbindungen ein.

- Verbindungen „verblassen“, falls diese nicht regelmäßig durch Informationsaustausch oder thematische Nähe verstärkt werden.

Diese verschiedenen Methoden, eine Verbindung zwischen Nutzern herzustellen, beeinflussen die Struktur des im DSN abgebildeten sozialen Netzwerks. Die Struktur hingegen beeinflusst, wie Wissen sich verbreiten kann – schnell oder langsam, in der Breite oder in der Tiefe, von Ort oder Thema beeinflusst.

Die Wahl des Verbindungsmodells ist so betrachtet ein Parameter, der die kollaborativen Filterungen und Empfehlungen beeinflusst. Synchrone Verbindungen findet man bspw. eher in identitätsbasierten Gemeinschaften wie Facebook oder XING[4]. Asynchrone Verbindungen sind hingegen sinnvoller, um thematische Gruppierungen zu fördern.

[4] http://xing.com

2.3 Evaluierung alternativer Mechanismen und ihrer Effekte

Wie in den vorangegangenen beiden Abschnitten angedeutet, beeinflusst die Wahl des Verbindungsmodells und der Empfehlungsmechanismen auch die Funktionsweise oder gar den Erfolg der zu entwickelnden Software. Daher wäre es nützlich zu wissen, welche Effekte aus welchen Mechanismen entstehen.

Interessante Fragen, die durch Evaluierungen beantwortet werden könnten, sind beispielsweise die folgenden.

- Wissen weiterzugeben kann für manche Personen bedeuten, dass sie etwas von ihrem Wert aufgeben und deshalb ihren Stand in der Organisation – bspw. bei ihrem Arbeitgeber – verschlechtern. Kann man diesen Effekt abschwächen, indem man in allen Inhalten ihre Autoren als Urheber nennt? Hilft es, Ranglisten von Mitwirkenden zu erstellen oder zur Belohnung Auszeichnungen zu verteilen, die dann im Profil des Nutzers angezeigt werden?

- Die verschiedenen Verbindungsmodelle sind interessant, weil sie für Organisationen verschiedener Größe und mit abweichenden Kulturen womöglich unterschiedliche Effekte haben. Welche Eigenschaften von Organisationen führen zu welchen Effekten? Durch welche Verbindungsmodelle kann man diese ausnutzen oder ihnen entgegenwirken?

- Sind Gruppen mit synchronen Verbindungen „näher“, weil ja alle Verbindungen aus dem Willen beider jeweiligen Parteien entstanden sind? Welche Rolle spielt das Gefühl, eine Kontaktanfrage annehmen zu müssen? Welche Rolle spielt die organisatorische Stellung der Beteiligten hierbei?

- Welches Verbindungsmodell verbreitet Informationen schneller im Netzwerk: das synchrone oder das asynchrone? Da das asynchrone Modell eher thematische als persönliche Beziehungen fördert wirkt es intuitiv wahrscheinlich, dass es Inhalte schneller verteilt. Trifft das wirklich zu?

Das in diesem Beitrag beschriebene erweiterbare DSN soll bei Experimenten unterstützen, die diese und ähnliche Fragen beantworten könnten.

3 Verwandte Arbeiten

Dieser Abschnitt stellt Arbeiten vor, deren Ziel es ist, die Auswirkungen sozialer Mechanismen besser zu verstehen und diese dadurch gezielter und kontrollierter einsetzen zu können.

Nach Cetina bestehen Verbindungen zwischen Menschen vor allem über die Objekte, an denen sie gemeinsames Interesse haben [Ce97]. Dies wurde von der Forschergemeinschaft um DSN aufgegriffen und stellt eine wichtige Entwurfsrichtlinie für DSN dar (siehe bspw. Breslin et. al. [BPD09]).

Ogata et. al stellen ein System zur Expertensuche vor, das das soziale Netzwerk der Anwender ausnutzt [OYF01]. Ein Vergleich mit alternativen Strategien zur Expertensuche findet nicht statt, es erfolgt lediglich eine Evaluierung des einen, eigenen Ansatzes. Die Arbeit schließt mit der Feststellung, dass die CSCW-Literatur zwar immer mehr soziologische Studien über Arbeit und Zusammenarbeit enthält. Technische Unterstützung, die auf sozialen Aspekten basiert, habe jedoch bisher nur wenig Aufmerksamkeit erhalten.

McDonald untersucht zwei verschiedene Strategien zur Expertensuche [Mc03]. Die eher grundlagenorientierte Arbeit stellt allgemeine Probleme bei der Konstruktion von Social Software dar. Konkrete Handlungs- oder Entwurfsempfehlungen finden sich jedoch leider nicht.

Ähnlich wie Ogata et. al. kommt auch McDonald zu dem Schluss, dass soziale Netzwerke als Entwurfselement von Groupware bisher zu wenig untersucht wurden und weitere Evaluationen und Studien notwendig sind.

Im Projekt „Community Lab“[5] sind einige relevante Arbeiten entstanden. In [RLT06] präsentieren Rashid et. al. eine Studie, in der Nutzern der Wert ihres Beitrags angezeigt wurde. Dabei vergleichen sie verschiedene Methoden, um diesen Wert zu berechnen. Ähnliche Studien sind [BLW04] von Beenen et. al. sowie [LCF04] von Ludford et. al.. In [RKK07] untersuchen Ren et. al. die Unterschiede zwischen Gemeinschaften, die auf persönlichen Verbindungen bzw. einer gemeinsamen Gruppenidentität beruhen.

Kraut vergleicht gemeinsam mit Farzan et. al. in [FDK] verschiedene Strategien, um das Engagement der Nutzer von DSN positiv zu beeinflussen. In [RK] entwickeln Ren und Kraut ein Modell, das beim Entwurf von DSN die Auswahl des geeignetsten Moderationslevels erleichtern soll. In [KR] stellen Kraut und Resnick eine Sammlung von Richtlinien zum Entwurf von DSN zusammen. Diese drei Arbeiten befinden sich jedoch noch in der Begutachtung bzw. sind noch nicht fertiggestellt.

Im Gegensatz zu den wenigen Arbeiten zur Nutzung sozialer Mechanismen bei der Konstruktion von Software gibt es verhältnismäßig viele Arbeiten dazu, wie Social Software zur Verbesserung des Software Engineering *eingesetzt* werden könnte.

Treudes Arbeiten zeigen Ansätze, wie die Awareness von Softwareentwicklern mit Hilfe von Social Software verbessert werden kann [Tr10], [TS10]. Black präsentiert einige Fallstudien, in denen „Web 2.0“-Technologien verwendet wurden, um den Softwareentwicklungsprozess zu verbessern [BJ10]. Begel zeigt einen Activity Stream, der auf der Codebook-Plattform zur Untersuchung von Versionsverwaltungssystemen aufsetzt. [BZ10a]. Damian et. al. nutzen DSN zur Unterstützung der Anforderungserhebung [DM07]. Einen Überblick über Entwicklungswerkzeuge mit sozialen Mechanismen bieten Storey et. al. in [STD10] und leiten offene Forschungsfragen ab.

[5] http://communitylab.org

Die vorgestellten Arbeiten zeigen, dass durchaus Interesse und Bedarf an der Nutzung sozialer Mechanismen besteht, ihre Wirkungen aber noch besser untersucht werden müssen. Der folgende Abschnitt stellt nun ein erweiterbares DSN vor, das für solche Untersuchungen konzipiert wurde. So soll nicht nur den Einsatz sozialer Mechanismen verlässlicher gestaltet, sondern auch das Software Engineering selbst bereichert werden.

4 Hallway: Konzeption und Entwurf

Um die Effekte von Social Software bei der Softwareentwicklung kontrollierter nutzen zu können, müssen diese empirisch überprüft werden können. Zu diesem Zweck wurde ein DSN entwickelt, in welches zu überprüfende Mechanismen leicht integriert werden können. Das DSN hat den Arbeitstitel „Hallway“.

4.1 Architektur

Nach Boyd und Ellison kennzeichnet ein Digitales Soziales Netzwerk, das Benutzer in einem solchen webbasierten Dienst Profile besitzen, Verbindungen zueinander herstellen können und sich Verbindungen anderer Benutzer ansehen können [BE07]. In der Praxis bieten sie zudem häufig eine Ansicht, in der Benutzer über die Neuigkeiten ihrer Kontakte informiert werden.

Um ein solches DSN zu konstruieren, wurde als Basisarchitektur das Play!-Framework gewählt. Es forciert eine Web-MVC-Architektur, ist gleichzeitig einfach gehalten und erlaubt schnelle Iterationen, was unserem Entwicklungsprozess entgegenkam.

Das DSN sollte zudem die Möglichkeit bieten, leicht Module mit unterschiedlichen Mechanismen austauschen zu können und diese Mechanismen im Parallelbetrieb evaluieren zu können. Hierzu wurde der von Koziolek vorgestellte Mehrmandanten-Architekturstil SPOSAD (Shared, Polymorphic, Scalable Application and Data) gewählt. SPOSAD ist eine Erweiterung des n-Schichten-Modells und folgt dem Ziel, die Kosten für den Betrieb zusätzlicher Mandanten – Kunden des Softwareanbieters – zu reduzieren.

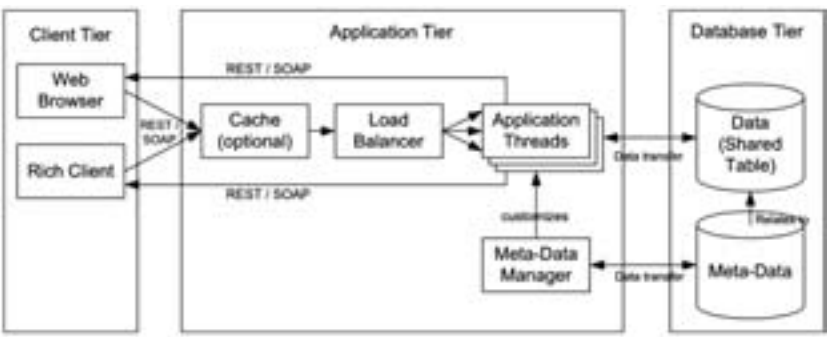


Abbildung 1: Mehrmandanten-Architekturstil SPOSAD, aus Koziolek [Ko10]

Eine SPOSAD-Anwendung wird dadurch gekennzeichnet, dass sie eine einzige Code-Basis für alle Mandanten bereitstellt und durch Ressourcenteilung in der Datenbankschicht sowohl mandantenspezifische Daten, als auch mandantenspezifische Anpassungen (bspw. durch Metadaten) ermöglicht (vgl. Abbildung 1). Bei SPOSAD benötigt die Anwendung lediglich eine Hardware, ein Betriebsystem und eine Datenbank, wobei hiervon beliebig viele Mandanten bedient werden können – die Anwendungen sind hoch skalierbar und günstig zu warten [Ko10].

Der SPOSAD Architekturstil wurde bei Hallway umgesetzt (vgl. Abbildung 2). Auch nach Bezemer und Zaidman, die Multi-Tenant-, Multi-Instance- sowie Multi-User-Anwendungen gegeneinander abgrenzen [BZ10], ist Hallway eine Multi-Tenant- bzw. Mehrmandanten-Anwendung.

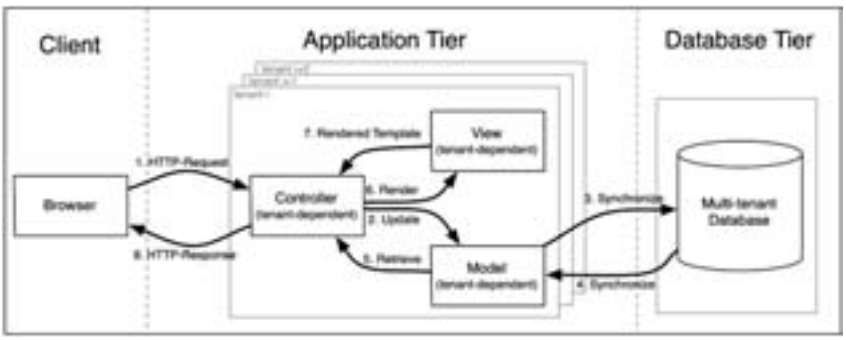


Abbildung 2: Mehrmandanten-Architektur von Hallway im Play!-Framework

Die Mehrmandantenarchitektur von „Hallway“ wurde inzwischen soweit verfeinert, dass mit einer einzigen HTTP-Anfrage ein neuer Mandant angelegt werden kann, sofern dieser noch nicht existiert. Bspw. legt eine Anfrage auf *http://se2011.example.org* den Mandanten „se2011“ in der unter *example.org* laufenden Hallway-Instanz an.

4.2 Module

Mit der umgesetzten Mehrmandantenarchitektur können nun aufgrund des speziell auf Erweiterbarkeit ausgerichteten Entwurfs leicht bspw. Verbindungsmodelle ausgetauscht werden oder zwei unterschiedliche Verbindungsmodelle im Parallelbetrieb von zwei Mandanten verglichen werden. Diese Austauschbarkeit wurde über ein Modulsystem (vgl. Abbildung 3) umgesetzt.

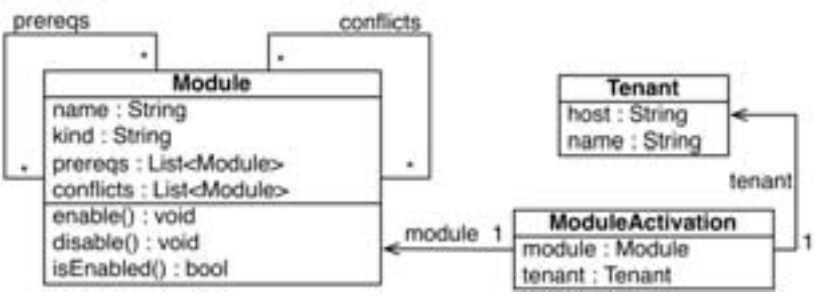


Abbildung 3: Modul-System zum Austausch unterschiedlicher zu untersuchender Mechanismen

Beim Starten der Anwendung werden alle Module registriert – hierfür gibt es für jedes Modul eine Klasse, welche das Interface *RegisterableModule* implementiert. Sobald diese Klasse beim Starten der Anwendung geladen wird, registriert sie ihr entsprechendes Modul.

Diese *RegisterableModule*-Klassen enthalten zudem setup()- und destroy()-Methoden, die bei Aktivierung bzw. Deaktivierung des Moduls ausgeführt werden und dann benötigte Daten in die Datenbank laden oder bspw. ein Mapping für ein Alternativmodul vornehmen können. Dies kann etwa für die Umstellung von „Likes“ auf „Retweets“ notwendig sein. Jedes Modul kann zwei Listen anderer Module referenzieren: Voraussetzungen (prereqs) und Konflikte (conflicts). So können bei Aktivierung des Moduls andere benötigte Module aktiviert oder die Aktivierung verweigert werden, falls bereits ein Modul aktiv ist, welches ähnliche Funktionalität bereitstellt.

Durch das beschriebene Modul-System können nun für jeden Mandanten spezielle Module ein- bzw. ausgeschaltet werden. Dabei wurde bewusst auf zwar sehr leistungsstarke, aber auch komplizierte Komponentenmodelle wie OSGI verzichtet. Die Module sollen in zeitlich begrenzten Experimenten genutzt werden, so dass eine einfache und schnelle Erweiterbarkeit beim Entwurf Vorrang hatte.

4.3 Mechanismen

Hallway soll auch an verschiedene Domänen anpassbar sein, um Experimente bei unterschiedlichen Partnern mit verschiedenen Geschäftsmodellen durchführen zu können.

Hierzu wurden im Anschluss an die Implementierung verschiedene Arten von Dokumententypen entwickelt, um die Erweiterbarkeit auch in dieser Hinsicht zu demonstrieren. Als Beispieltypen können nun Fotoalben mit Fotos angelegt, Spreadsheets von Google Docs importiert und versionierte Dateien angelegt werden. Diese Typen von Dokumenten besitzen alle jeweils einen Activity Stream und Benutzer können sich zu ihnen verbinden (vgl. Abbildung 4).

Beispielsweise könnten durch die Integration von SVN-Commit-Nachrichten und Meldungen vom Bug-Tracker in den Activity Stream die Arbeitsabläufe eines Software Entwicklers unterstützt werden.

Abbildung 4: ActivityStream eines Benutzers, der zu verschiedenen Gruppen, Dokumenten und Personen verbunden ist

Einige der eingangs erwähnten Mechanismen wurden bereits in Hallway umgesetzt und lassen sich leicht austauschen. Insbesondere für Verbreitungsmechanismen, die den Wissensfluss begünstigen, liegen schon Implementierungen vor. So ist bspw. ein Mechanismus an „Likes“ von Facebook angelehnt, während ein anderer wie die „Retweets“ von Twitter funktioniert.

Auch das verwendete Verbindungsmodell lässt sich austauschen. Implementiert sind das synchrone sowie das asynchrone Modell. Die Verbindungen lassen sich hierbei jeweils mit einem Mapping übertragen.

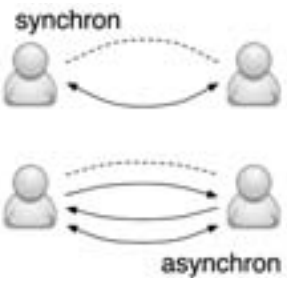


Abbildung 5: Synchrone und asynchrone Verbindungen zwischen Benutzern

Abbildung 5 illustriert den bereits o.g. Unterschied zwischen den beiden Modellen: im synchronen Modell beruht eine Verbindung immer auf Gegenseitigkeit (zwei Möglichkeiten der Verbindung: verbunden oder nicht verbunden), im asynchronen Modell ist dies nicht der Fall (vier Möglichkeiten).

Diese sowie die bereits o.g. möglichen Konzepte für die Verbindungen zwischen Benutzern lassen sich in Hallway leicht über das beschriebene Modul-System austauschen. Durch die Mehrmandanten-Architektur können sie schnell parallel aufgesetzt werden und stehen dann für Evaluationen bereit.

Weiterhin sind die Empfehlungsmechanismen austauschbar. Derzeit ist eine einfache Methode für die Empfehlung zuständig, die hierfür Tags der Benutzer mit denen anderer vergleicht. hybreed[6] ist ein Framework, das die Komposition von Empfehlungs-Workflows unterstützt, die auf erprobten Algorithmen basieren und diese neu kombinieren. Durch eine Integration in Hallway könnten so leicht alternative Empfehlungsstrategien eingebunden werden, um diese miteinander zu vergleichen.

5 Zusammenfassung und Ausblick

Dieser Beitrag hat den im Software Engineering bestehenden Bedarf nach Evaluierungen sozialer Mechanismen motiviert und ein erweiterbares DSN vorgestellt, das durch ein leichtgewichtiges Modulsystem zur Durchführung dieser Evaluierungen geeignet ist. Das DSN unterstützt die in Literatur und Praxis gängigen Grundfunktionen und beinhaltet unterschiedliche soziale Mechanismen, die teilweise auch in alternativen Ausführungen vorliegen. Mittels einer Mehrmandantenarchitektur können nun parallele Experimente mit wenig Aufwand aufgesetzt werden.

Das DSN inklusive seiner Module ist für Evaluierungen, nicht für den längerfristigen, produktiven Einsatz ausgelegt. Daher wurde beim Entwurf des Modul-Systems ein sehr leichtgewichtiger Ansatz mit einigen, für Produktivsysteme unerwünschten Abhängigkeiten gewählt. So sind die Module nicht logisch voneinander getrennt und ziehen sich durch Modell-, Controller- und Präsentationsschicht. Dieser Kompromiss steht jedoch in Einklang mit den Zielen der Plattform: die Umsetzung neuer Module kann mit wenig Aufwand und in kurzer Zeit geschehen.

Eine erste Möglichkeit zur Evaluierung unseres Ansatzes in der Industrie befindet sich zur Zeit in der Anbahnung. Eines der nächsten Ziele ist eine klarere und vollständigere Kategorisierung der sozialen Mechanismen. Aktuelle Arbeiten am Fachgebiet beschäftigen sich außerdem mit der Herleitung angemessener Metriken und Visualisierungen, um die Effekte der Mechanismen erkennen und bewerten zu können.

Literaturverzeichnis

[BE08] Boyd, D.M.; Ellison, N.B.: Social network sites: Definition, history, and scholarship. In: Journal of Computer-Mediated Communication, Wiley Online Library, 2008; Vol. 13, Nr. 1, S. 210-230.

[BJ10] Black, S.; Jacobs, J.: Using Web 2.0 to Improve Software Quality. In: Proceedings of the 1st Workshop on Web 2.0 for Software Engineering, ACM, 2010, S. 6-11.

[6] http://hybreed.org

[BLW04] Beenen, G.; Ling, K.; Wang, X.; Chang, K.; Frankowski, D.; Resnick, P.; Kraut, R.E.: Using Social Psychology to Motivate Contributions to Online Communities. In: Proceedings of the 2004 ACM conference on Computer supported cooperative work, ACM, 2004, S. 212-221.

[BPD09] Breslin, J.G.; Passant, A.; Decker, S.: The Social Semantic Web. Springer Verlag, Heidelberg, 2009.

[BZ10] Bezemer, C.; Zaidman, A.: Multi-Tenant SaaS Applications: Maintenance Dream or Nightmare?. In: ERCIM Workshop on Software Evolution (EVOL) and International Workshop on Principles of Software Evolution (IWPSE), ACM, Antwerp, 2010

[BZ10a] Begel, A.; Zimmermann, T.: Keeping Up With Your Friends: Function Foo, Library Bar.DLL, and Work Item 24. In: Proceedings of the 1st Workshop on Web 2.0 for Software Engineering, ACM, 2010, S. 20-23.

[Ce97] Cetina, K.K.: Sociality with Objects : Social Relations in Postsocial Knowledge Societies. In: Theory, Culture & Society, SAGE, Thousand Oaks, CA, USA, 1997, Vol. 14, Nr. 4, S. 1-30.

[DM07] Damian, D.; Marczak, S.; Kwan, I.: Collaboration patterns and the impact of distance on awareness in requirements-centred social networks. In: 15th IEEE International Requirements Engineering Conference, 2007. RE'07. IEEE, 2007; S. 59-68.

[FDK] Farzan, R., Dabbish, L., Kraut, R. E., & Postmes, T. (eingereicht). Increasing Commitment in Online Communities via Building Social Attachment.

[Ko10] Koziolek, H.: Towards an Architectural Style for Multi-tenant Software Applications. In (Engels, G.; Luckey, M.; Schäfer, W.): Software Engineering 2010, Bonner Köllen Verlag, Paderborn, 2010.

[KR] Kraut, R. E. & Resnick, P. (unter Vertrag). Evidence-based social design: Mining the social sciences to build online communities. MIT Press, Cambridge, MA:

[LCF04] Ludford, P.J.; Cosley, D.; Frankowski, D.; Terveen, L.: Think Different: Increasing Online Community Participation Using Uniqueness and Group Dissimilarity. In: Proceedings of the SIGCHI conference on Human factors in computing systems, ACM, 2004, S. 631-638.

[Mc03] McDonald, D.W.: Recommending Collaboration with Social Networks: A Comparative Evaluation. In: Proceedings of the SIGCHI conference on Human factors in computing systems, ACM, 2003, S. 593-600.

[OYF01] Ogata, H.; Yano, Y.; Furugori, N.; Jin, Q.: Computer supported social networking for augmenting cooperation. In: Computer Supported Cooperative Work (CSCW), Springer Verlag, Heidelberg, 2001, Vol. 10, Nr. 2, S. 189-209.

[RK] Ren, Y., & Kraut, R. E. (eingereicht). A simulation for designing online community: Member motivation, contribution, and discussion moderation.

[RKK07] Ren, Y.; Kraut, R.; Kiesler, S.: Applying Common Identity and Bond Theory to Design of Online Communities. In: Organization Studies, SAGE, Thousand Oaks, CA, USA, 2007, Vol. 28, Nr. 3, S. 377-408.

[RLT06] Rashid, A.M.; Ling, K.; Tassone, R.D.; Resnick, P.; Kraut, R.; Riedl, J.: Motivating Participation by Displaying the Value of Contribution. In: Proceedings of the SIGCHI conference on Human Factors in computing systems, ACM, 2006, S. 955-958.

[STD10] Storey, M.-A.; Treude, C.; van Deursen, Arie; Cheng, L.-T.: The Impact of Social Media on Software Engineering Practices and Tools. Wird erscheinen in: Proceedings of the FSE/SDP Workshop on the Future of Software Engineering Research. ACM, 2010.

[Tr10] Treude, C.: The Role of Emergent Knowledge Structures in Collaborative Software Development. In: Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 2, ACM, 2010, S. 389-392.

[TS10] Treude, C.; Storey, M.A.: Awareness 2.0: Staying Aware of Projects, Developers and Tasks using Dashboards and Feeds. In: Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1, ACM, 2010, 365-374.

Methods to Secure Services in an Untrusted Environment

Matthias Huber, Jörn Müller-Quade

matthias.huber@kit.edu, muellerq@kit.edu

Abstract: Software services offer many opportunities like reduced cost for IT infrastructure. They also introduce new risks, for example the clients lose control over their data. While data can be secured against external threats using standard techniques, the service providers themselves have to be trusted to ensure privacy. In this paper, we examine methods that can increase the level of privacy a service offers without the need to fully trust the service provider.
Keywords: services, cloud computing, security, privacy

1 Introduction

Due to advances in networking and virtualization technology, new paradigms of providing IT infrastructure and software have emerged – among them the so-called *Cloud Computing*. The National Institute of Standards and Technology [NIS09] defines Cloud Computing as "a model for enabling convenient, on-demand network access to a shared pool of computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction". Cloud Computing enables clients to use *Software as a Service* and to outsource their data, thus cutting the cost of maintaining an own computing center.

Inherent to Software as a Service, however, are privacy problems [Pea09, HHK+09]: By using services, clients lose control over their data. The threat copy or misuse of the clients data on the server exists and keeps many potential customers from using Cloud Computing in critical or sensitive scenarios (e.g., scenarios comprising business secrets or customer data). Current security mechanisms focus on protecting the data transfer to and from the service provider.

Protecting a pure storage service against insider attacks is easy. Encrypting all data on the client before uploading it to the server provides a sufficient level of protection based on the used encryption [Bla93]. This, however, prevents the server from performing any meaningful operation on the data. Hence, more complex services require advanced techniques for providing privacy.

In this paper, we will examine techniques and methods that potentially can increase the security of services. In Section 2, we will describe the required properties of solutions. We will examine different methods in Section 3 and evaluate them according to the requirements described in Section 2. We will examine and evaluate applications of these methods in secure database outsourcing scenarios in Section 3.3. In Section 4, we will examine

potential candidates for formal security properties for services. Section 5 concludes.

2 Requirements

In this chapter we discuss the required properties of solutions for the privacy problems inherent to most services. Basically, solutions for the presented problem should provide *provable security* and *practicability*.

2.1 Requirement 1: Provable Security

Formal notions and proofs are provided for cryptographic methods. While providing security in the sense of classical cryptography is infeasible for nontrivial services (e.g. services more complex than pure storage services), solutions for the privacy problem described in Section 1 should provide provable security. This means that we need a formalization of the level of privacy provided. Furthermore, we need to prove that a solution provides this level of privacy (fulfills the formal notion). For a database service such a weaker, yet probably sufficient level of privacy may be that an adversary cannot learn the relations between the attribute values.

2.2 Requirement 2: Practicability

There are cryptographic solutions for problems involving parties that do not fully trust each other. In most cases, however, they are inapplicable due to their complexity. We strive for solutions that are practical and do not cancel the benefits of outsourcing. Consider a solution providing a high level of privacy, but due to its complexity it is more efficient to execute the service on the client. It will be hard to convince customers to use such a service even it provides privacy. Also the provided level of privacy has to be easy to understand, since this is vital to the acceptance of the solution, too.

3 Methods to Enhance the Security of Services

There are methods from different fields, that potentially can enhance the security of services. In this section, we will present methods from the field of software engineering and cryptography. In the field of secure database outsourcing there are applications of these methods. We will also present these applications in this section.

3.1 Software Architecture

There are approaches that enhance the security of services by separating it, and deploying the parts of different servers [HILM02, ABG+], assuming the servers do not cooperate maliciously. There are two different ways to separate a service, namely *serial* and *parallel* separation [Hub10]. For concrete services these separations have different performance implications depending on the usage profile. In most cases, however, a particular separation for itself does not yield security guarantees. Nevertheless, we believe a separation of a service in combination with cryptographic methods can yield provable security with feasible performance overhead.

3.2 Cryptographic Methods

Aside from *encryption* which is uses in many scenarios to prevent eavesdroppers from learning the anything about the information sent over a channel, cryptography offers additional methods and protocols to engage privacy and security issues. In the remainder of this section, we will discuss these methods.

There are encryption schemes that produce ciphertexts with homomorphic properties: Consider for example Textbook-RSA [RSA78]. Multiplying two ciphertexts and decrypting the result yields the same result as decrypting the two ciphertexts and multiplying the plaintexts. However, Textbook-RSA is not considered secure [BJN00]. In 2009, Craig Gentry discovered a *fully homomorphic encryption* that supports multiplication as well as addition [Gen09], and theoretically solves our privacy problem for scenarios involvin only one client: The client could simply use the proposed encryption scheme, and the service provider could adapt its service to work on encrypted data using this scheme. However, this is not feasible since the size of the key scales with the size of the circuit of the algorithm which the service calculates. It is more efficient to execute the service on the client than to use homomorphic encryption.

Secure Multiparty Computations [GMW87, CCD88] are cryptographic solutions for problems where two or more parties cooperatively want to compute a certain function over a set of data without any party learning anything about the input of other parties except what is learned by the output. The problem is that for each party, the computation cost is higher than computing the whole function on the complete input without any other party. This makes the concept of multiparty computation for outsourcing services too expensive and in fact pointless if the client is the only one with private input.

There are methods to retrieve information from an outsourced database without the server learning anything about the information queried [Gas04]. However, these methods are also infeasible in most cases: If the database server must not learn anything about a query, the query issued to the database must contain every cell. Otherwise the server learns which cells do not contribute to the result of the query, and thus learns something about the result set, if no special-purpose hardware is involved [KC04]. The need to iterate over every cell for every query execution makes private information retrieval impractical. If for every

query, the database service has to touch every cell, the service does not scale.

For many scenarios pure software solutions do not provide enough protection. It is believed, that hardware can be made more tamper resistant than software. Therefore many solutions to security problems involving secrets incorporate dedicated hardware [DL07, BS03]. Assuming the hardware can be trusted, there are solutions to security problems, that are otherwise infeasible or impossible. In the context of secure service outsourcing, the *Trusted Computing* approach [TCG] is very interesting. In theory a lightweight tamper resistant piece of hardware the so-called *trusted computing module* can be used for ensuring that the software running on the machine is certified. This, in principle, can solve many problems such as software protection and digital rights management. This, however, does not solve the problem of building a service that provides privacy for the clients data, but the problem of verifying that the desired software is actually running on the server.

3.3 Approaches for Secure Database Outsourcing

Since many services rely on databases, the problem of secure database outsourcing emerged early in 2002. Classical cryptographic notions are not applicable to encrypted databases under practical constraints because in general it is infeasible to realize a database that complies with these notions [KC04]. However, there are approaches that try to provide some level of privacy. We believe that the methods developed in this context are the most advanced ones that try to solve the privacy problem of outsourcing. Most of them combine cryptography with special architectures in order to achieve a higher level of privacy. They implicitly introduce an adapter deployed on the client, that encapsulates en- and decryption as well as distribution of data according to the proposed approach. From an architectural point of view, you can distinguish between two classes of approaches, which we term *coarse indices approach* and *distribution approach*.

The idea of the coarse indices approach [HILM02, DVJ+03, CDV+05, HMT04, BBO07] is to encrypt the database on a tuple level. In order to support efficient query processing indices are created. These indices are coarse-grained. They contain keywords, ranges, or other identifiers and encrypted row ids where the keywords occur in the database. In most approaches, these indices are kept in the adapter. An adapter deployed on the client handles query transformation and sorts out false positives. Figure 1 depicts the architecture of the coarse indices approach. When issued a query, the adapter first queries the coarse-index tables and decrypts the results. Then, it queries the encrypted database with the results from the index tables and recieves encrypted tuples. These tuples potentially contain false positives due to the coarse indices. The adapter decrypts these rows, sorts out the false positives, and retuns an exact anwser to the clients query. Since the adapter handels (en- and) decryption and query transformation, this is transparent to the client application. This approach supports efficient execution of exact match and range queries. It is unclear, however, what level of privacy this approach provide for realistic scenarios. Since we need solutions that provide provable security (c.f. Section 2), these approaches for itself are insufficient.

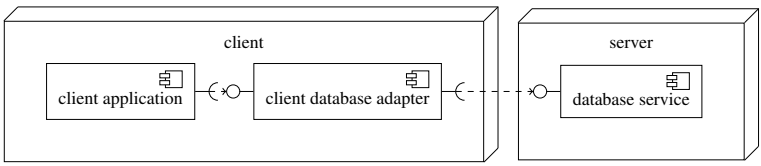


Figure 1: The architecture of a secure outsourced database proposed for example in [HILM02, DVJ+03]. The adapter provides and requires a standard SQL interface. The overall database service consists of the database service on the server and the adapter on the client.

The idea of the *distribution approach* [ABG+, CdVF+] is to distribute the database to several servers in order to fulfill previously defined *privacy constraints*. When these contraints cannot be fulfilled, encryption can be used. A database is called secure, of all privacy constraints for this individual database are met. Consequently this approach does not provide a general but rather a individual security guarantee. The classes of queries this approach supports to execute efficiently depend on the chosen distribution scheme and encryption methods used. In the worst case, it is either not possible to execute any class of queries efficiently, or the approach does not provide provable security. Therefore this approach is not a solution that fulfills out requirements (c.f. Section 2).

4 Anonymity Properties and Notions for Databases

In the previous section, we have seen different approaches that try to solve the secure database outsourcing problem. They, however, either do not provide security guarantees, or have strong assumptions about the input data or hardware, and are therefore not applicable in most cases. In order to provably provide a service that provides privacy for its clients data, we need a formalization of the level of privacy provided. In this section, we will examine different anonymity and security notions for databases with respect to their security and applicability to service outsourcing scenarios.

The anonymity properties k-*anonymity*, l-*diversity*, t-*closeness* and *differential privacy* are motivated from privacy preserving database disclosure. In this scenario, a trusted instance wants to disclose a database that contains data about people whose privacy must be protected. Therefore the database has to be transformed into a database fulfilling a particular anonymity property (offline). In the case of differential privacy, the results of queries are altered (online). Figure 2 depicts the architecture of the offline scenario of privacy preserving database disclosure, where an anonymizer transforms a database d into d'. Although, these properties are originated from privacy preserving database disclosure outsourcing, they might be applicable to secure database outsourcing.

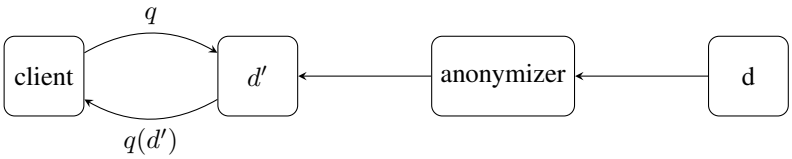


Figure 2: Offline scenario for privacy preserving database disclosure. An anonymizer transforms a database d, and discloses the result d'. Clients then can query d'

4.1 k-anonymity, l-diversity, and t-closeness

In 2002, Latanya Sweeney [Swe02] defined the property k-anonymity. The intention of k-anonymity is that any individual in the database can only be mapped to at least k rows. Therefore attributes of datasets are classified into *identifiers*, so-called *quasi-identifiers*, and *sensitive information*. A anonymized table must not contain attributes classified as identifiers. The property k-anonymity is defined as follows:

Definition 1 *k-anonymity*
A database adheres k-anonymity if for each row, there are at least $k-1$ other rows with the same attribute values of the quasi-identifiers. Rows with the same attribute values of their quasi-identifiers are called a equivalence class.

In order to achieve k-anonymity, attribute values can be generalized and rows can be suppressed.

Different weaknesses of and attacks on k-anonymity have been identified [MKGV07]. So-called *Homogeneity Attacks* as well as *Background Knowledge Attack* can be used to break k-anonymity. If an attacker knows that a particular individual is in the disclosed database and all sensitive information cells of the k corresponding rows have the same value (alternatively: the attacker can rule out particular values by background knowledge), she can infer information that intuitively should be protected by the property of k-anonymity. Consider for example the table in Figure 3. This database adheres 2-anonymity. If an attacker knows an individual that is 30 years old and lives in area code 3524 and knows that he is in the database, she can infer that the individual has the flu.

Because these attacks the property of l-diversity [MKGV07] has been defined. l-diversity constrains the attribute values of the sensitive information column in an equivalence class:

Definition 2 *l-diversity*
A database adheres l-diversity if in each equivalence class, there are at least l "well-represented" values in the sensitive information column.

There are different interpretations of "well-represented" [MKGV07]. Although l-diversity has been defined in order to eliminate homogeneity and background knowledge attacks, they are still possible. For example if the used interpretation of "well-represented" does not consider semantic similarities as for example "gastritis" and "stomach pain" (c.f. Figure 3).

quasi-identifier		*sensitive information*
age	area code	disease
15-34	352*	flu
15-34	352*	flu
35-40	345*	gastritis
35-40	345*	stomach pain

Figure 3: A medical database adhering 2-anonymity the attributes age and area code have been generalized in order to achieve 2-anonymity.

Because of the weaknesses of l-diversity, t-closeness has been defined [LL07]:

Definition 3 *t-closeness*
An equivalence class is said to have t-closeness if the distance between the distribution of a sensitive attribute in this class and the distribution of the attribute in the whole table is no more than a threshold t. A table is said to have t-closeness if all equivalence classes have t-closeness.

t-closeness shifts the problem of defining a resilient anonymity property to the problem of finding a resilient distance measurement for distributions. Moreover, if t is small, the table contains not more information than the distributions of attribute values, which is in contrast to the initial intention of disclosing whole databases instead of just summaries.

4.2 General Problem of k-anonymity, l-diversity, and t-closeness

Besides the weaknesses and attacks described in Section 4.1, k-anonymity, l-diversity, and t-closeness have another fundamental weakness.They describe the result of an anonymization process, not the process itself. An attacker can, knowing the anonymization process, learn information from the disclosed database that should intuitively not be learned from a database adhering k-anonymity, l-diversity, or t-closeness. In the remainder of this section, we will provide two examples. Consider for example the database depicted in Figure 4. In general, a database has more than one versions adhering k-anonymity. Figure 5, for

name	age	area code	disease
Alice	31	3524	flu
Bob	31	3456	flu
Carol	35	3524	gastritis
X	35	3456	stomach pain

Figure 4: A medical database

example, depics two versions of the database in Figure 4 adhering 2-anonymity.

age	area code	disease
3*	3524	flu
3*	3456	flu
3*	3524	gastritis
3*	3456	stomach pain

(a)

age	area code	disease
31	3*	flu
31	3*	flu
35	3*	gastritis
35	3*	stomach pain

(b)

Figure 5: Databases adhering 2-anonymity derived from the table in Figure 4

Consider an anonymization process as follows. If "X" = "Eve", disclose Table 5(a), otherwise disclose Table 5(b). Now, although the results adhere 2-anonymity, an attacker learns if Eve is in the database just by knowing the anonymization process and looking at which attribute was generalized.

Consider as another example a database about a city comprised of three city parts A, B, and C with 7, 9 and 20 inhabitants, respectively. The database consists of a quasi-identifier the city part, and a binary sensitive information attribute p. Let the anonymization process merge city parts as long as the database does not adhere 2-diversity and try to minimize the number of merges. If all city parts are merged in the disclosed database, an attacker can infer information based on the number of people with p according to the table in Figure 6. Consider for example 8 people with p living in the city. If anybody in city part A

# people with property p	information attacker can infer
1	(*nothing to infer*)
2 - 7	all inhabitants with p live in the same city part
8,9	nobody living in part A has p
10 - 15	all inhabitants with p live in part C
16 - 20	all inhabitants with p live in part C or all inhabitants of A and B have p
21	(*impossible*)

Figure 6: Information an attacker can infer depending on the number of people with attribute p in the disclosed database adhering 2-diversity.

had p and without loss of generality nobody in C had p the anonymization process would have stopped after merging A and C. In the other case (people in B and C have p) the anonymization process would have stopped immediately.

These examples show, that, knowing an anonymization process, an attacker can infer information that should be protected by l-diversity.

4.3 differential privacy

In contrast to k-anonymity, l-diversity, and t-closeness that describe the result of the anonymization process, *differential privacy* [Dwo06] describes the anonymization process itself. This fixes the problem described in the previous section. The intention is to minimize the risk of an individual joining a database. Formally, differential privacy is defined as follows [Dwo08]:

Definition 4 *differential privacy*
A randomized function $\mathcal{K}$ gives ε-differential privacy if for all data sets D_1 and D_2 differing on at most one element and all $S \subseteq Range(\mathcal{K})$:

$$Pr[\mathcal{K}(D_1) \in S] \leq exp(\varepsilon) \times Pr[\mathcal{K}(D_2) \in S]$$

Informal, this means that if a function adheres diffential privacy, the probability of getting different results from data sets differing on at most one row should be less than a certain threshold.

In practice this notion is achieved by adding noise to results. Therefore a trusted instance called "curator" is placed between clients and the database (cf. Figure 7). The curator forward the clients' queries and adds noise according to assumptions about the universe of all possible databases.

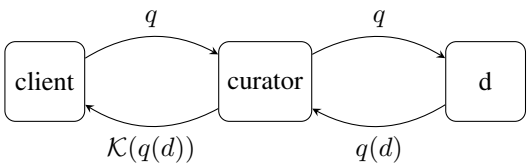


Figure 7: An architecture for differential privacy. A trusted curator is placed between the clients and the database d and transforms query results in order to achieve differential privacy

This is also the biggest drawback of this notion. In order to achieve differential privacy by adding noise, you have to know every entity that possibly can be added to your database.

4.4 k-Ind-IPC

Another notion, that describes the anonymization process instead of the result is k-Ind-IPC [HHK+10]. Intuitively, k-Ind-IPC hides relations within an equivalence class:

Definition 5 *k-Ind-IPC*
Let d and d' be databases, where d' can be gain by partitioning d into parts of at least k-rows, permuting the attribute values indepentdently within each column and each part and

uniting the resulting. A anonymization function f provides k-Ind-IPC if for all databases d and d'. $f(d)$ is indistinguishable from $f(d')$

For a more formal definition please refer to [HHK+10].

Since this notion is motivated from secure database outsourcing, there are methods for outsourcing databases that provide k-Ind-IPC [Hub10]. These methods combine the approaches presented in Section 3.3 and involve separating the database service, deploying the parts on different servers and deploying an adapter on the client. Only a limited set of queries, however, can be executed efficiently and, due to the separations of duties approach, different servers are needed.

5 Conclusion

In this paper, we presented methods that potentially can solve the privacy problems inherent to service outsourcing. We evaluated these methods with respect to performance and provable security. Then, we presented approaches for secure database outsourcing, that apply these methods. We have seen that thees approaches imply special architectures. Most these concrete approaches do not provide a practical security guarantee. Therefore we examined potential candidates for formalizations of security notions for outsourced databases. Differential privacy and k-Ind-ICP seem viable candidates. Since differential privacy, however, is originated from a different scenario, it is unclear how this notion can be applied in a secure database outsourcing scenario.

For future work we want to examine how to build databases that provide k-Ind-ICP or differential privacy, and what architectures these notions imply. We also want to examine if and how differential privacy and k-Ind-ICP relate. This can lead to a better understanding of these notions and also to new methods that support them.

References

[ABG+] Gagan Aggarwal, Mayank Bawa, Prasanna Ganesan, Hector Garcia-Molina, Krishnaram Kenthapadi, Rajeev Motwani, Utkarsh Srivastava, Dilys Thomas, and Ying Xu. Two Can Keep a Secret: A Distributed Architecture for Secure Database Services. *CIDR 2005*.

[BBO07] Mihir Bellare, Alexandra Boldyreva, and Adam O'Neill. Deterministic and Efficiently Searchable Encryption. In *CRYPTO*, pages 535–552, 2007.

[BJN00] Dan Boneh, Antoine Joux, and Phong Nguyen. Why Textbook ElGamal and RSA Encryption are Insecure (Extended Abstract), 2000.

[Bla93] Matt Blaze. A cryptographic file system for UNIX. In *CCS '93: Proceedings of the 1st ACM conference on Computer and communications security*, pages 9–16, New York, NY, USA, 1993. ACM.

[BS03] Adrian Baldwin and Simon Shiu. Hardware Encapsulation of Security Services. In Einar Snekkenes and Dieter Gollmann, editors, *Computer Security â ESORICS 2003*, volume 2808 of *Lecture Notes in Computer Science*, pages 201–216. Springer Berlin / Heidelberg, 2003.

[CCD88] David Chaum, Claude Crépeau, and Ivan Damgard. Multiparty unconditionally secure protocols. In *STOC '88: Proceedings of the twentieth annual ACM symposium on Theory of computing*, pages 11–19, New York, NY, USA, 1988. ACM.

[CDV+05] Alberto Ceselli, Ernesto Damiani, Sabrina De Capitani Di Vimercati, Sushil Jajodia, Stefano Paraboschi, and Pierangela Samarati. Modeling and assessing inference exposure in encrypted databases, 2005.

[CdVF+] Valentina Ciriani, Sabrina De Capitani di Vimercati, Sara Foresti, SUSHIL JAJODIA, Stefano Paraboschi, and PIERANGELA SAMARATI. Combining Fragmentation and Encryption to Protect Privacy in Data Storage.

[DL07] Jeffrey S. Dwoskin and Ruby B. Lee. Hardware-rooted trust for secure key management and transient trust. In *CCS '07: Proceedings of the 14th ACM conference on Computer and communications security*, pages 389–400, New York, NY, USA, 2007. ACM.

[DVJ+03] Ernesto Damiani, S. De Capitani Vimercati, Sushil Jajodia, Stefano Paraboschi, and Pierangela Samarati. Balancing confidentiality and efficiency in untrusted relational DBMSs, 2003.

[Dwo06] Cynthia Dwork. Differential Privacy. *Automata, Languages and Programming*, pages 1–12, 2006.

[Dwo08] Cynthia Dwork. Differential Privacy: A Survey of Results. *Theory and Applications of Models of Computation*, pages 1–19, 2008.

[Gas04] William Gasarch. A Survey on Private Information Retrieval. *Bulletin of the EATCS*, 82:72–107, 2004.

[Gen09] Craig Gentry. Fully homomorphic encryption using ideal lattices. In *STOC '09: Proceedings of the 41st annual ACM symposium on Theory of computing*, pages 169–178, New York, NY, USA, 2009. ACM.

[GMW87] Oded Goldreich, Silvio Micali, and Avi Wigderson. How to play ANY mental game. In *STOC '87: Proceedings of the nineteenth annual ACM symposium on Theory of computing*, pages 218–229, New York, NY, USA, 1987. ACM.

[HHK+09] Christian Henrich, Matthias Huber, Carmen Kempka, Jörn Müller-Quade, and Mario Strefler. Towards Secure Cloud Computing. In *Proceedings of the 11th International Symposium on Stabilisation, Safety, and Security of Distributed Systems (SSS 2009)*, 2009.

[HHK+10] Christian Henrich, Matthias Huber, Carmen Kempka, Jeorn Mueller-Quade, and Ralf Reussner. Technical Report: Secure Cloud Computing through a Separation of Duties. `https://sdqweb.ipd.kit.edu/huber/reports/sod/technical_report_sod.pdf`, 2010.

[HILM02] Hakan Hacigümüs, Bala Iyer, Chen Li, and Sharad Mehrotra. Executing SQL over encrypted data in the database-service-provider model. In *Proceedings of the 2002 ACM SIGMOD international conference on Management of data*, pages 216–227. ACM, 2002.

[HMT04] Bijit Hore, Sharad Mehrotra, and Gene Tsudik. A privacy-preserving index for range queries. In *VLDB '04: Proceedings of the Thirtieth international conference on Very large data bases*, pages 720–731. VLDB Endowment, 2004.

[Hub10] Matthias Huber. Towards Secure Services in an Untrusted Environment. In Barbora Bühnová, Ralf H. Reussner, Clemens Szyperski, and Wolfgang Weck, editors, *Proceedings of the Fifteenth International Workshop on Component-Oriented Programming (WCOP) 2010*, volume 2010-14 of *Interne Berichte*, pages 39–46, Karlsruhe, Germany, June 2010. Karlsruhe Institue of Technology, Faculty of Informatics.

[KC04] Murat Kantarcioglu and Chris Clifton. Security Issues in Querying Encrypted Data. Technical report, 2004.

[LL07] Ninghui Li and Tiancheng Li. t-Closeness: Privacy Beyond k-Anonymity and l-Diversity. 2007.

[MKGV07] Ashwin Machanavajjhala, Daniel Kifer, Johannes Gehrke, and Muthuramakrishnan Venkitasubramaniam. l-Diversity: Privacy beyound k-Anonymity. *Cornell University*, page 52, March 2007.

[NIS09] NIST. NIST - Cloud Computing. `http://csrc.nist.gov/groups/SNS/cloud-computing/`, 2009.

[Pea09] Sinai Pearson. Taking Account of Privacy when Designing Cloud Computing Services. *HP Laboratories*, 2009.

[RSA78] R.L. Rivest, A. Shamir, and L. Adleman. A Method for Obtaining Digital Signatures and Public-Key Cryptosystems. *Communications of the ACM*, 21:120–126, 1978.

[Swe02] Latanya Sweeney. k-Anonymity: A Model for Protecting Privacy. *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems*, 10(5):557–570, 2002.

[TCG] TCG. Trusted Compting Group. `http://www.trustedcomputinggroup.org/`.

Software Requirements Specification in Global Software Development – What's the Difference?

Frank Salger

Capgemini
Carl-Wery-Strasse 42
81739 Munich
frank.salger@capgemini.com

Abstract: Global software development (GSD) is becoming the standard approach in the software industry. In software projects, the quality of the software requirements specification (SRS) is a crucial success factor. However, little is known about how to write high quality SRS within GSD projects. The question is: "What notation of quality shall we employ for SRS that are used in GSD projects?" In this paper, we use the IEEE Standard 830 as a basis to answers this question. This standard defines several quality attributes of 'good SRS'. For each of the quality attributes, we explain its specific role in GSD and point out means to achieve compliance with it. We argue that the negative impact in not complying with the quality attributes is exacerbated in GSD. Combining these results with our previous work on inspections, we derive our main conclusion: The quality attribute which has to be treated most carefully in GSD is 'external completeness'. The paper includes a concise checklist to assess SRS in GSD projects. The results thus provide tangible benefits to practitioners. We also point out research questions with high relevance for the industry.

1 Introduction

It is well known that requirements engineering (RE) has a strong influence on project success or failure [SGI03, DO05]. This influence becomes even stronger in global software development (GSD) where teams in geographically different sites work together as one team in order to create complex software systems. In fact it is often concluded that requirements engineering is one of the main challenges in GSD (see e.g. [PAE03]). This conclusion seems to be reasonable as the software requirements specification (SRS) usually constitutes the pivotal work product by means of which much of the communication between the different sites which are involved in GSD takes place [HPB05, HM03]. Taking into account the increased importance of offshoring and outsourcing in modern software development [WFV06, He07a] it is surprising that the question: "What notation of quality shall we employ for SRS that are used in GSD projects?" has not been explicitly addressed yet.

Now, why is RE so difficult in GSD projects? This question can be answered along the main differences of GSD to co-located projects. These specific GSD-challenges are:

Geographical distribution: Geographical distribution impedes 'whiteboarding', that is, standing next to each other, drawing models and the like at the white board to clarify specification issues or search for flaws in requirements. It also hinders to use 'rich' communication, in particular mimics and gestures. Probably the biggest impact of geographical distribution might come from missing ad hoc meetings at the water cooler, at the coffee machine or at lunch. It is there, where lots of information is exchanged, even without the team members noticing it [He07b]: By now, it is well known that we cannot write down all information that is necessary to build a system in a SRS and that large portions of the missing information is transferred by means of such unplanned "ad hoc meetings" (see, e.g., [Da05]). *Different language:* Using different languages involves repeated translations. This 'language hopping' opens a wide space to introduce all kinds of ambiguities. *Different cultures:* The effect of different cultures is often explained along Hofstede's cultural dimensions [Ho10]. For example, Indians are considered to be less assertive than Germans. This might lead to situations, where an Indian programmer does not point out flaws of a SRS, in order not to let the German requirements engineer 'loose his face'. *Time zone difference:* Although time zone difference can be advantageous for several disciplines like testing (see, e.g. [HH07]), other disciplines are negatively effected by reducing the overlap of common working hours. With respect to RE, there simply might be less time available for discussions between the onshore requirements engineer and the offshore developer. *Different engineering backgrounds:* Different engineering backgrounds might entail usage of different processes, methodologies and tools. Apart from slowing down the lead time to productive work, different terminologies can seriously affect a distributed team by opening the door for ambiguities and misunderstanding [ED01], [B et al 09].

Regarding the immense challenges that are newly introduced in GSD settings, surprisingly little research has been conducted on the topic of requirements engineering in GSD. While there exist some valuable research results on process related aspects of writing software requirements specifications in GSD projects (see, e.g., [G et al 08]), the author is not aware about discussions, what "quality" of a SRS would mean in GSD projects. To tackle this research gap, we here leverage the IEEE Standard 830 - Recommended Practice for Software Requirements Specifications [IEEE98], a well known and largely accepted definition of 'quality' of a SRS. Among other information, the standard provides a characterization of good SRS. There, a SRS is considered to be of high quality if it satisfies a number of quality attributes, i.e., if it is:

- *Correct:* "An SRS is correct if, and only if, every requirement stated therein is one that the software shall meet." [IEEE98]

- *Unambiguous:* "An SRS is unambiguous if, and only if, every requirement stated therein has only one interpretation." [IEEE98]

- *Complete:* In [IEEE98], three aspects of 'completeness' are distinguished: A SRS is considered complete, if and only if a) All significant requirements are captured (called 'external completeness' hereafter). b) The definition of the responses of the software to all realizable classes of input data in all realizable classes of situations is given (called 'internal completeness' hereafter). c) Full labels and references to all figures, tables, and diagrams in the SRS and definition of all terms and units of measure are given (also subsumed under 'internal completeness' hereafter).

- *Consistent:* "Consistency refers to internal consistency. If an SRS does not agree with some higher-level document, such as a system requirements specification, then it is not correct." [IEEE98]

- *Ranked for importance/stability:* "An SRS is ranked for importance and/or stability if each requirement in it has an identifier to indicate either the importance or stability of that particular requirement." [IEEE98]

- *Verifiable:* "An SRS is verifiable if, and only if, every requirement stated therein is verifiable. A requirement is verifiable if, and only if, there exists some finite cost-effective process with which a person or machine can check that the software product meets the requirement." [IEEE98]

- *Modifiable:* "An SRS is modifiable if, and only if, its structure and style are such that any changes to the requirements can be made easily, completely, and consistently while retaining the structure and style." [IEEE98]

- *Traceable:* "An SRS is traceable if the origin of each of its requirements is clear and if it facilitates the referencing of each requirement in future development or enhancement documentation." [IEEE98]

In the remainder of this paper we investigate the question, what role the above described quality attributes play in GSD projects. We provide a qualitative valuation which quality attributes are even more crucial to be obeyed than in co-located projects. This valuation is derived from various project dig-outs, numerous interviews with senior managers and architects, as well as literature research and empirical research conducted at Capgemini [F et al 07]. Single aspects of our results can already be backed by existing research literature. The main contribution of this work thus is a consolidated overall view on SRS quality in global software development projects based on the IEEE-standard 830. We believe that the article is beneficial for practitioners to get a concise overview on the specifics of writing good quality SRS in global software development projects. It also contributes to fill an existing knowledge gap identified in [DE06]: "[…] more information has to be gathered to establish a connection between quality assurance techniques and certain quality attributes."

The next section describes the context of this work: The company, the system type considered and one of the GSD projects which where the basis of this research. In section 3 we scrutinize the quality attributes that are explicitly stated in the IEEE-standard with respect to its role in GSD.

We also discuss quality assurance techniques that help to comply with these attributes. Further, we reframe the discussion into a concise checklist which we hope will serve the practitioner to set-up or assess his own GSD projects. In Section 4, we give a short summary, draw our conclusions, indicate our future work and point out some research questions with relevance to the industry.

2 Context

Capgemini employs about 95000 people in more than 30 locations around the world. With a global network of more than 40.000 experts, Capgemini's Technology Services (TS) is dedicated to helping clients design, develop and implement technical projects of all sizes through the creation of project architecture, software package implementation, application development, consulting in IT technologies and innovative solutions. For more information: http://www.capgemini.com/services-and-solutions/technology/. The business information systems developed at Capgemini Germany support core business processes of our customers (and as such are critical to the business of our customers) but are usually not critical to life. This paper focuses on custom solution development projects of large scale business information systems. Business information systems can consist of several hundred use cases, user interfaces and domain entities. Complex concepts for print output, batch processing, archiving, authorization and multi-tenancy are almost always necessary to meet the requirements of the customer. We now shortly describe one such custom solution development project. This case study will subsequently be used to explain how specific techniques have been to achieve a high quality SRS.

Project Alpha: In this project Capgemini Germany was engaged to build the retirement management application of a large German insurance company and the migration of the data of the legacy application landscape into this new system. The project started in 2003 and the last customer release went into production mode in 2007. In the average, about 100 developers worked in this project. The project leveraged offshoring of the design, the implementation and testing. The project was distributed over three sites: a) Munich, Germany, where requirements engineering, analysis & design, implementation, testing and deployment took place. b) Zurich, Switzerland, where analysis & design, implementation and testing took place. c) Wrozlaw, Poland where design, implementation and testing took place. The final SRS consisted of several hundred use cases and user interface specifications, as well as complex cross-cutting concepts, e.g. for multi-tenancy or the time dependent handling of business domain objects. A major challenge was the very strict performance requirements on data-intensive batch processes. Moreover, business rules where very complex since numerous legal exceptions had to be considered. Instead of using a commercial requirements management tool, the SRS was maintained in a custom-build tool. This allowed us to extensively generate documentation and configuration code from the SRS and helped to keep the SRS and the design in sync.

3 Quality of SRS in Global Projects

In this section we investigate how the above discussed quality attributes are affected by GSD. We will argue that meeting all quality attributes becomes even more important in GSD than in co-located project setting. For each single quality attribute, we discuss why it becomes more important in GSD. We then succinctly propose techniques how to achieve the quality attribute in GSD, based on our experiences. Some of these techniques are well known and should also be applied in co-located projects. However, we also highlight a couple of techniques that are less known and especially helpful in GSD. We describe the techniques as "checks". The checks are independent of each other and can be used separately. At this moment, we have no empirical data about interrelation of the techniques. E.g., we cannot yet say, whether one technique is more effective than another in order to satisfy a specific quality attribute. Thus, we suggest to use all proposed techniques in order to increase the likelihood of achieving the respective quality attribute. We cannot claim yet, that the list of proposed techniques is complete.

Quality Attribute: ***Correct, Unambiguous, Complete, Consistent***

Why more important: We discuss these four quality attributes together because they are uniformly affected by the GSD-challenges: First, *geographical distribution* impedes the intensive communication necessary to resolve these kinds of defects in SRS. Even working interactively in front of a whiteboard, it can be daunting o resolve flaws in SRS - let alone by using only electronic communication tools. Writing SRS in a *different language* than ones native language is very challenging and often leads to ambiguities in SRS. These issues often lead to delays and quality problems with the resulting system. *Time zone differences* leave less time for the onshore requirements engineer and the offshore developer to discuss issues with a SRS. It also entails asynchronous working which makes it more difficult to quickly reveal ambiguities.

Check:

Constructive quality assurance applied? Correctness and completeness largely escape classical appraisal and failure-based quality assurance techniques as suggested by the results from other researchers [Da05] as well as our own previous work [SEH09]. It is thus especially important in GSD to embrace systematic constructive quality assurance methods in order to arrive at correct and complete SRS baselines from which to start offshore development. Project Alpha used Joint Application Development sessions to gain a common understanding of the system to be built.

Incremental development used? Incremental development and continuous integration are mentioned as a technique to tackle the problem with correctness and completeness of SRS: Increments allow to 'fail early' and gather early feedback based on tested increments. Missed core requirements often lead to substantial rework of architectures, design and large portions of low-level code. Our experience is that such massive changes are much more difficult to be executed in GSD as compared to co-located projects. Early increments often point out missed requirements and can even substitute detailed specifications to some extend.

In project alpha was developed in four major increments and several minor increments. This approach especially helped to evolve the cross-cutting concepts of the SRS, and avoided that teams of later increments made the same mistakes as their team mates in previous increments.

Test cases derived, checked and supplied to the offshore team? Deriving test cases for SRS often point out flaws, omissions and ambiguities: It is simply not possible to derive test cases for underspecified requirements. In project Alpha, we heavily used this technique to improve the quality of the SRS. The most valuable improvement was gained by having the end users specify (acceptance) test cases, where the requirements engineer conducted walkthroughs on the SRS based on these test cases. This approach surfaced a couple of serious flaws in the SRS. This technique alongside with handing over the test cases to the offshore team seems to be especially promising in GSD be acceptance test driven development [SE10, SSC04].

Quality Attribute: ***Ranked for importance/stability***

Why more important? These quality attributes are indirectly impacted by *geographical distribution*. Put it simple, this kind of distribution makes it more difficult to manage a project (see, e.g., [Cu08]). This in turn can make it necessary to implement management techniques like earned value, which in turn is based on a concept of 'value', or 'importance'. Further, some authors also advise not to offshore the development of business critical functionality (see, e.g., [BK05]), which should also be an aspect of the definition of importance. A same argument is often used with respect to stability: Due to hindered communication in GSD, it seems to be unreasonable to offshore the development of highly unstable functionality, as instability necessarily leads to frequent communication. Communicating changes and the necessary re-planning are more difficult to handle due to distance [S et al 06].

Check:

Business domain itself stable? Which parts of the SRS will be offshored? Are these parts related to highly volatile business subject matter? If so, the project should be able to answer how they will manage frequent changes to SRS and how they will cascade these changes to the offshore development team. Although the business domain of project Alpha was highly complex due to numerous legal exceptions, it was quite stable. Thus, stability of the business domain did not pose any extra challenges.

Change request management process appropriate? Highly related to the first check, defined processes for change request (CR) management are always especially important in GSD. In particular, it is necessary to implement a process which can handle the many issues that come up during fine design and programming [S et al 06]. In project Alpha, a multi-level CR management process was implemented. Minor changes (triggered from then development team or the user group) could be authorized by the project managers (client side and project side) themselves, but all significant changes of course had to pass the executive change management board.

SRS baselined and signed-off? It is always important to define clear SRS baselines to start development from, and always difficult to judge when to draw the baseline [Wi03]. However, in GSD a good baseline becomes even more important as every change or update to the requirements package must be propagated to a globally distributed site. Using modern requirements engineering tools with full access of all sites, this problem is to some extend mitigated. But still, it remains challenging to communicate changes of requirements or packages thereof remotely. In project alpha, parts of the SRS (so called 'work packages') where first validated by the client. After that, the responsible requirements engineer together and the offshore development team conducted a so called 'work package handover checkpoint' to ensure that the work package was complete from a developers point of view.

Quality Attribute: *Verifiable*

Why more important? In the end, it is the client who will verify that the system satisfies the SRS as stated. To do so, he of course must understand the SRS. However, a dilemma arises, as "representations that improve the requirements specification for the developer may be counterproductive in that they diminish understanding to the user and vice versa." ([3], see page 5). Now, some authors argue that SRS must be more detailed in GSD settings in order to support the work of the offshore developer {attrition, less business context} (see, e.g., [S et al 06]). Basically, this is an indirect consequence of *geographical distribution* as they cannot participate in the requirements workshops and miss the information shared in ad hoc meetings. Thus, the dilemma is exacerbated in GSD, as adding even more 'developer oriented' information further diminishes the probability that the client understands the SRS. Hence, in principle, the treatment of the quality attribute 'verifiable' must be rethought in GSD. Luckily, most often, 'in-verifiability' is a consequence of under-specification. This can be reduced by adding detail which is required by client and developer alike.

Check:

Test cases derivable? Testing is the prime technique to verify requirements. Testing relys on test cases. If these are not derivable, a requirement is not verifiable. In project Alpha, the requirements engineer started to derive test cases already during the software requirements specification phase. Moreover, the business subject matter experts reviewed these test cases which also helped to improve the specification [SE10].

Weak phrases avoided? Verifiability is closely related to absent ambiguity. And a lot of ambiguity can be avoided if so called weak phrases are eliminated. In project Alpha, a rather mathematical and semi-formal approach was taken to specify business logic. This helped to avoid weak phrases.

Precise quantification used? Especially for non-functional requirements, precise quantification is necessary. One possibility to achieve precise quantification is to use highly structured (yet practical) languages, like, e.g. Planguage [Gi05]. In project Alpha, the technical chief architect reviewed all non-functional requirements and ensured that they were precise enough to derive a valid software architecture.

Quality Attribute: *Modifiable*

Why more important? Of all quality attributes discussed in this article, this is the one which is least impacted by GSD. Nevertheless, *different engineering backgrounds* triggered discussions in some of our projects. Sometimes, the offshore team wanted all information related to a use case be bundled together with the use case. This eased the task of allocating work to the developers. However, this would have resulted in redundancy (e.g., copying business rules to incorporate them into different 'use case bundles'). We refrained from this approach as this clearly would have made the SRS less modifiable, accepting a potential reduction in development efficiency. If we consider the ability to propagate modifications to the according stakeholders than *geographical distribution* also impact this quality attributes, as distribution reduces the developer awareness of requirement modifications [Da07].

Check:

Sufficient tool support employed? In most nontrivial software projects there are too many requirements to be left unmanaged. Specifically in GSD, tools should support the notification of requirements changes to the involved people. Vendors of requirements management tools acknowledge the trend towards GSD and take provision to update their tools to these settings. Project Alpha used a requirements management tool. But, this tool did not support direct notification mechanism to flag requirements changes to developers. However, the role of a 'topic responsible' was introduced, who was responsible to propagate the changes of requirements of his topic to the team. In this project, this approach proved to be effective as well.

Quality Attribute: *Traceable*

Why more important? Traceability plays an important role in GSD [L et al]. Traces between artefacts on the same level of abstraction are important from a developer point of view. For example, a clear mapping of the domain objects referenced by the use cases onto the logical data model might be helpful for the offshore developers. But traces between higher level requirements and, e.g., use cases are also especially helpful for offshore developers as they usually did not participate in the requirements workshop and thus miss the overall view on the business subject matter. As 'higher level' requirements (like features or business processes) often provide more overview that the 'lower-level' uses cases, a developer can use the traces to gain additional information on the business subject supported by the use case he implements.

Check:

Traces between high level requirements and SRS exist? Check that the constituent parts of the SRS can be traced back to the high level requirements, and vice versa. For example, it is usually necessary to trace high level features down to the use cases that 'implement' them. In project Alpha, a lightweight excel-based tool was used to implement traces between high level requirements (business processes) and system-level use cases. This proved to be sufficient, as the mapping between business processes and system-level use cases was usually a one-to-many relation.

Traces between SRS primitives exist? The appropriateness of this kind of tracing directly impacts developer efficiency: As he implements the functionality he will have to navigate through the SRS in order to get all necessary information. Especially important are the traces between use cases and user interfaces, as well as between use cases and business rules. In project Alpha, a specific naming scheme was used to map SRS primitives to each other (like, e.g., use interfaces to use cases). Again, this proved to be sufficient due to the rather simple mapping between the different primitives.

Traces between SRS and downstream artefacts can be set up? Already when a SRS template is made up one should think about traceability to downstream artefacts that will be spawned by the SRS (like design artefacts, test cases, etc.). The critical decision is, to what level of detail unique identifiers (IDs) will be used. For example: Will use cases be uniquely identified? Or will steps of use cases be uniquely identified? In GSD, IDs play an important role, as most communication happens over telephone and email. There, it is a tremendous advantage if one can clearly point about what he is talking using IDs. In project Alpha, IDs where used down to the level of use case steps and business rules. This allowed to map fine-grained functional parts of the specification onto design primitives, which proved to be valuable especially in the long maintenance phase.

Although project Alpha applied various of the discussed techniques, it still faced considerable requirements engineering related challenges. Probably the most critical challenge was the issue of knowledge transfer: Although a rather detailed and granular SRS has been created, the offshore developers did sometimes miss information necessary to implement the system. Sometimes the developers missed even more details for specific complex requirements. On the other hand side, they sometimes missed the overall context. We believe that answering the question of right granularity of SRS is especially critical in GSD projects. In Table 1, we present a checklist condensed from the discussion above. We believe that this checklist can be valuable to assess GSD projects – also by practitioners of other companies. In the table, "!" flags a challenge which is exacerbated by global distribution, and "*" flags aspects which might help to address the respective challenge. These aspects should be carefully assessed in GSD projects to assure project success.

Quality Attribute	Why more important?	Check!
Correct Unambiguous Complete Consistent	! Distance impedes communication which makes resolving non-compliance to all these quality attributes more difficult ! Non-native language can introduce ambiguities ! Time zone differences leave less time so resolve potential issues	* Constructive quality assurance applied (like, e.g., joint application development, prototyping)? * Incremental development used? * Test cases derived and checked?
Ranked for Importance/ Stability	! Distribution makes project management more difficult and demands for advanced techniques like, e.g., earned value analysis	* Business domain stable? * Change request management appropriate? * SRS baselined and signed-off by

	! Instable business domain triggers frequent changes to SRS, whose propagation is more difficult in distributed teams	customer before development starts? * Allocation of all 'work packages' for offshore development rationalized?
Verifiable	! If a requirement is not verifiable it most often is underspecified. This will lead to problems during development, which makes lots of communication necessary. But communication is impeded in GSD.	* Test cases derivable? * Weak phrases avoided? * Precise quantification used (e.g., Planguage)?
Modifiable	! Distribution limits awareness of offshore developers for modifications ! Distribution makes propagation of modifications more difficult	* Sufficient tool support employed? * Offshore developer have direct access to requirements management tool?
Traceable	! Offshore developers have less business background ! Considerable attrition rates in offshore nations	* Traces between high level requirements and SRS exist? * Traces between SRS primitives exist? * Traces between SRS and downstream artefacts can be set up?

Table 1 SRS Checklist for Global Software Development

4 Summary, Conclusion and Future Work

We now briefly summarize the results of this paper before we draw our conclusions. By indicating our future work, we will close our discussion. *Summary:* This paper offers preliminary answers to the question "What notion of quality shall we employ for SRS that are used in global software development (GSD) projects?" We based our discussion on several quality attributes that characterize high quality of a SRS as suggested by the IEEE-standard 830 – Recommended Practice for Software Requirements Specifications. For each of those quality attributes, we explained how the specific GSD challenges exacerbate the relevance of complying with the quality attribute. Further, we discussed various techniques which help to achieve compliance with the quality attributes. We also explained how these techniques where applied within a large GSD project executed at Capgemini CSD GSA. Finally, we derived a concise checklist which can be leveraged by practitioners to set-up or assess their GSD-projects. After having discussed the effect of GSD on the quality attributes of the IEEE-standard, we can now put these results in the context of our previous work and the results of other researches and draw our major conclusions. *Conclusion:* In our previous work, we argued that not all quality attributes are equally effectively checked by inspections [SEH09]. Basically we found that the quality attributes "internal completeness", "consistency", "understandability", "verifiability", "modifiability", "traceability" and "ranked for importance/stability" can be very effectively checked by inspections. Less effectively but sufficiently addressed is the quality attribute "correctness". However, only insufficiently addressed is the quality attribute "external completeness".

The main reason is that omissions of requirements are very difficult to spot for any reviewer: A peer reviewer often has the problem of being 'routine-blinded'. In turn, a project external reviewer is not as intimately familiar with the business domain as to judge completeness [Da05]. As we here consider the SRSs of custom business information systems, the project team members are sometimes the only experts in the company for the according business subject matter. Apart from that, customer needs constantly change [Da05]. Not only appraisal-based quality assurance techniques fail to assess 'completeness'. To worsen the situation, other quality assurance techniques like testing do often not catch requirements omissions, as (acceptance) tests run only against those requirements which have been gathered and which are actually written down within the SRS.

But requirements omissions are not only hard to spot. They are also the type of requirements defects, which are most expensive to correct. We thus conclude that 'external completeness' is probably the most important quality attribute to strive for in GSD. From our project experience we did not experience that completely new quality attributes for SRS show up in the context of GSD. On the other hand, we have argued that compliance to the SRS quality attributes becomes even more important in GSD, and that failure to do so exacerbates the risk of project failure – even more than in co-located projects. Thus, requirements engineering in GSD seems not to be different from what a software engineer has to do. The difference is that high quality requirements engineering is even more critical in GSD than in co-located projects!

Future Work: In our future work we are going to investigate the question, how SRS should look like in agile GSD projects, taking into account challenges like long maintenance phases of business information systems. For our near term work we plan to provide an empirical basis for the (qualitative) results offered in this paper. Subsequently, we also want to extend this study on the quality characteristics which are only implicitly mentioned by the IEEE-standard 830. Finally, we want to formulate a research question of which we believe that the answers would be highly beneficial to the industrial practice: Writing SRS is always a balancing act of keeping the SRS readable for both, the customer and the developer. It is sometimes argued that SRS in GSD must be more detailed than in co-located projects. An interesting research question would thus be: "Does GSD intensify the problem of keeping a SRS readable for all involved stakeholders?" And – if yes – how to tackle this issue?

References

[B et al 09] Bartelt, C., Broy, M., Herrmann, C., Knauss, E., Kuhrmann, M., Rausch, A., Rumpe, B., and Schneider, K.: Orchestration of Global Software Engineering Projects - Position Paper. In Proceedings of the 2009 Fourth IEEE international Conference on Global Software Engineering - Volume 00 (July 13 - 16, 2009). IEEE Computer Society, Washington, DC, 2009. 332-337.

[BK05] BITKOM. Leitfaden Offshoring. Bundesverband Informationswirtschaft, Telekommunikation und neue Medien e.V. 2005.

[Cu08] Cusumano, M. A.: Managing Software Development in Globally Distributed Teams. Communications of the ACM, Vol. 51, No. 2, ACM. 2008. 15—17.

[Da07] Damian, D.: Collaboration Patterns and the Impact of Distance on Awareness in Requirements-Centred Social Networks. 15th IEEE International Requirements Engineering Conference, RE 2007, October 15-19th, 2007, New Delhi, India 2007.
[Da05] Davis, A. M.: Just Enough Requirements Engineering. Dorset House Publishing. 2005.
[DE06] Denger, C. and Elberzhager, F.: A Comprehensive Framework for Customizing Quality Assurance Techniques, IESE-Reprt No 118.06/E, Fraunhofer IESE, Germany. 2006.
[DO05] Degner, C. and Olsson T.: Quality Assurance in Requirements Engineering. Engineering and Managing Software Requirements. Springer. Berlin-Heidelberg. 2005.
[ED01] Ebert, C. and De Neve, P.: Surviving Global Software Development. IEEE Softw. 18, 2 (Mar. 2001), 2001. 62-69.
[F et al 07] Fabriek, M., van den Brand, M., Brinkkemper, S., Harmsen, F. and Helms, R.: Improving offshore communication by choosing the right coordination strategy. Technical Report UU-CS-2007-021. Department of Information and Computing Sciences. Utrecht University. 2007.
[Gi05] Gilb, T.: Competitive Engineering. Elsevier Butterworth Heinemann, Netherlands, 2005.
[G et al 08] Gorschek, T., Fricker, S., Felt, R., Wohlin, C., and Mattsson, M.: 1st International Global Requirements Engineering Workshop, GREW'07. SIGSOFT SE. Notes 33, 2008.
[He07a] Herbsleb, J. D.: Global Software Engineering: The Future of Socio-technical Coordination. In 2007 Future of Software Engineering (May 23 - 25, 2007). International Conference on Software Engineering. IEEE Computer Society, Washington, DC, 2007. 188-198.
[He07b] Herbsleb, J. D.: Global Software Engineering: The Future of Socio-technical Coordination. In 2007 Future of Software Engineering (May 23 - 25, 2007). International Conference on Software Engineering. IEEE Computer Society, Washington, DC. 2007. 188-198.
[HH07] Hussey, J., M. and Hall, S., E.: Managing Global Development Risks. Auerbach Publications. 2007.
[HM03] Herbsleb, J. D. and Mockus, A.: An Empirical Study of Speed and Communication in Globally Distributed Software Development. IEEE Trans. Softw. Eng. 29, 2003. 481-494.
[Ho10] Hofstede™ Cultural Dimensions. http://www.geert-hofstede.com/ (accessed 08/19/2010)
[HPB05] Herbsleb, J. D., Paulish, D. J., and Bass, M.: Global software development at siemens: experience from nine projects. In Proceedings of the 27th international Conference on Software Engineering, ICSE '05. ACM, New York, NY, 2005. 524-533.
[IEEE98] IEEE Std. 830-1998 – Recommended Practice for Software Requirements Specifications, The Institute of Electrical and Electronics Engineers, Inc. New York, 1998.
[L et al 04] Lormans, M., Dijk, H. v., Deursen, A. v., Nocker, E., and Zeeuw, A. d.: Managing Evolving Requirements in an Outsourcing Context: An Industrial Experience Report. In Proceedings of the Principles of Software Evolution, 7th international Workshop (September 06 - 07, 2004). IWPSE. IEEE Computer Society, Washington, DC, 2004. 149-158.
[PAE03] Prikladnicki, R. and Audy, J. L. N. and Evaristo, R.: Global Software Development in Practice: Lessons Learned. Software Process Improvement and Practice; 8. 2003. 267-281.
[SSC04] Sengupta, V. Sinha, and S. Chandra, "Test-Driven Global Software Development," 7th International Workshop on Global Software Development, 2004.
[SEH09] Salger, F., Engels, G. and Hofmann, A.: Inspection Effectiveness for Different Quality Attributes of Software Requirement Specifications: An Industrial Case Study, In Proceedings of the 7th ICSE Workshop on Software Quality (WoSQ 09), IEEE Press, 2009. 15-21.
[SE10] Salger, F. and Engels, G.: Knowledge transfer in global software development: leveraging acceptance test case specifications. In Proceedings of the 32nd ACM/IEEE international Conference on Software Engineering – Vol. 2. ICSE '10. ACM, New York, NY, 2010. 211-214.
[SGI03] The Standish Group International Inc. CHAOS Chronicles v3.0. West Yarmouth. 2003.
[S et al 06] Sangwan, R., Bass, M., Mullik, N., Paulish, D., J., Kazmeier, J.: Global Software Develop-ment Handbook. Auerbach Publications. 2006.
[WFV06] William A., Frank M. and Vardi, M., Y (Eds.).: Globalization and Offshoring of Software. ACM task force. ACM. 2006.
[Wie03] Wiegers, K. E.: Software Requirements. Microsoft Press. Redmond, Washington. 2003

Anwendungsentwicklung mit Plug-in-Architekturen: Erfahrungen aus der Praxis

Jörg Rathlev
Fachbereich Informatik, Universität Hamburg
rathlev@informatik.uni-hamburg.de

Abstract: In der Anwendungsentwicklung kommen zunehmend Plug-in-basierte Ansätze zum Einsatz. Die Verwendung Plug-in-basierter Techniken hat Auswirkungen auf die Entwicklung der Software, in der die spezifischen Eigenschaften von Plug-ins berücksichtigt werden müssen. Dieser Beitrag identifiziert basierend auf Erfahrungen aus der Praxis Entwurfsfragen, die sich beim Einsatz von Plug-ins stellen, und diskutiert Entwurfsalternativen und deren Auswirkungen.

1 Einleitung

Die Anforderungen an ein Softwaresystem unterscheiden sich je nach Anwendungskontext und können sich im Laufe der Zeit verändern. Um kostspielige Neuentwicklungen zu vermeiden, sollte Software daher anpassbar und erweiterbar gestaltet sein. Plug-in-basierte Entwicklung ist ein Ansatz, der verspricht, die Implementierung von erweiterbarer und anpassbarer Software zu unterstützen und zu vereinfachen. Unter anderem durch die Verbreitung Plug-in-basierter Plattformen wie Eclipse gewinnt dieser Ansatz an Bedeutung.

Die Entscheidung, ein Softwaresystem Plug-in-basiert zu entwickeln, hat Auswirkungen auf dessen Architektur und die Organisation des Entwicklungsprojektes. Offensichtlich muss das System geeignet in Plug-ins aufgeteilt werden. Daneben gibt es aber noch weitere, auf den ersten Blick weniger offensichtliche Auswirkungen, die aus den spezifischen Eigenschaften von Plug-ins folgen. In der praktischen Anwendung sehen Entwickler sich daher mit einer Reihe neuer Entwurfsfragen konfrontiert, wenn sie Plug-in-basierte Techniken einsetzen.

Dieser Beitrag identifiziert basierend auf Erfahrungen aus der Praxis die Gründe, aus denen Plug-in-Architekturen eingesetzt werden, sowie Entwurfsfragen und organisatorische Herausforderungen, die sich beim Einsatz von Plug-ins stellen.

Die in diesem Beitrag präsentierten Ergebnisse basieren auf Erfahrungen aus vier verschiedenen Softwareprojekten. Bei dem ersten Projekt handelt es sich um ein seit fünf Jahren laufendes Industrieprojekt, in dem eine integrierte Werkzeugumgebung für die Steuerung und Überwachung technischer Anlagen entwickelt wird. Der Autor dieses Beitrags war im Rahmen einer Forschungskooperation für drei Jahre als Softwareentwickler in diesem Projekt tätig.

Das zweite Projekt ist ein Forschungsprojekt, das vom Autor in beobachtender Rolle be-

gleitet wird. In diesem Projekt soll ein komponentenbasiertes Rahmenwerk für die Entwicklung von Leitstand-Systemen entwickelt werden. Sowohl das erste als auch das zweite Projekt basieren in der Implementierung auf der Eclipse Rich Client Platform.

Um die Erkenntnisse abzusichern, wurden Interviews mit Entwicklern aus zwei weiteren Projekten durchgeführt. Bei dem dritten Projekt handelt es sich um die Entwicklung eines Plug-in-Rahmenwerks für die .NET-Plattform, das von einem Hamburger Unternehmen für Anwendungen im Bereich betrieblicher Umweltinformationssysteme eingesetzt wird. Das vierte Projekt ist ein Forschungsprojekt an der Universität Hamburg, in dem dieses Rahmenwerk eingesetzt wird, um ein System zu entwickeln, das den Handel mit Emmissionsrechten unterstützt.

Außerdem wurden die Organisation und Architektur des Eclipse-Projektes untersucht. Als Informationsquellen dienten hier die eigenen Erfahrungen des Autors mit der Eclipse-Plattform sowie die verfügbare Literatur und Dokumentation.

Dieser Beitrag gibt nachfolgend zunächst eine kurze Einführung in die Plug-in-basierte Entwicklung. Anschließend wird beschrieben, mit welcher Motivation in den beobachteten Projekten Plug-in-basierte Techniken eingesetzt wurden. Danach werden die beobachteten organisatorischen und technischen Herausforderungen und Fragestellungen diskutiert.

2 Plug-ins und Plug-in-Architekturen

Der Begriff *Plug-in* wird mit verschiedenen Bedeutungen verwendet, sowohl in der Literatur als auch in der Praxis.

Im Allgemeinen versteht man unter einem Plug-in eine Erweiterung einer vorhandenen Anwendung, wobei die Erweiterung erst zur Laufzeit von der Anwendung dynamisch geladen wird [Mar06]. Dies ermöglicht es, die Erweiterung unabhängig von der Anwendung auszuliefern und nachträglich zu installieren. Die Integration der Erweiterung in die Anwendung erfolgt über eine Plug-in-Schnittstelle. Ist die Spezifikation dieser Schnittstelle offengelegt, so können Plug-ins nicht nur vom Hersteller der Anwendung selbst, sondern auch von Dritten entwickelt werden. Plug-ins können aber in der Regel nur die Basisanwendung erweitern, die Erweiterung anderer Plug-ins ist nicht möglich.

Eine solche Plug-in-Schnittstelle wird heute von vielen am Markt etablierten Softwareprodukten bereitgestellt. Zu den bekanntesten Beispielen gehören Bildbearbeitungswerkzeuge und Web-Browser.

So genannte *reine Plug-in-Architekturen* verbinden das Plug-in-Konzept mit Konzepten der komponentenbasierten Entwicklung [Rat08]. Bei einer solchen Architektur werden Plug-ins als das grundlegende Zerlegungskonzept für das gesamte Softwaresystem verwendet. Die Software wird nicht in eine Anwendung und Plug-ins getrennt, sondern sie wird vollständig aus Plug-ins gebaut. Technisch besteht kein Unterschied zwischen den Plug-ins, die die Basisanwendung implementieren, und den Plug-ins, die diese erweitern. Jedes Plug-in kann Schnittstellen zur Erweiterung durch andere Plug-ins bereitstellen [Bir05].

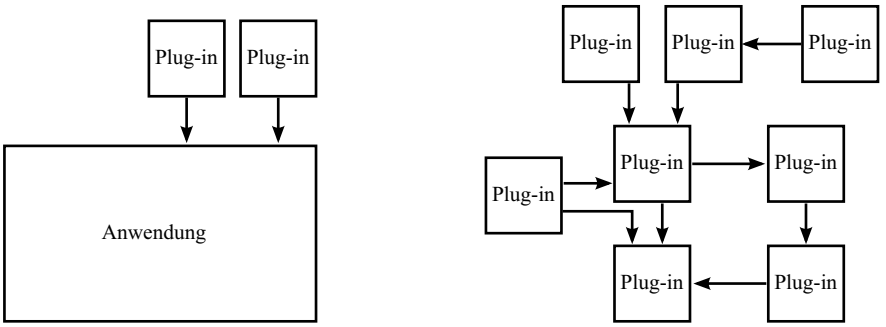


Abbildung 1: Erweiterbare Anwendungen und reine Plug-in-Architekturen

Anders als bei einer erweiterbaren Anwendung kann bei einer Anwendung mit einer Plug-in-Architektur nicht nur das bestehende System erweitert werden, sondern Anwender oder Drittentwickler können auch Teile der Basisanwendung austauschen oder entfernen und die Anwendung so an spezielle Anwendungskontexte anpassen.

In Abbildung 1 werden der klassische Plug-in-Begriff für erweiterbare Anwendungen und das Konzept der reinen Plug-in-Architektur schematisch gegenübergestellt.

Plug-ins sind Softwarekomponenten im Sinne von [SGM02], die Code zur Benutzung durch andere Plug-ins exportieren bzw. von anderen Plug-ins importieren können. Neben dieser Benutzungsbeziehung verfügen Plug-in-basierte Systeme aber vor allem über einen *Erweiterungsmechanismus*. Über diesen kann ein Plug-in zur Laufzeit dynamisch ermitteln, welche Erweiterungen von anderen Plug-ins angeboten werden, und auf diese zugreifen [Rat08]. Der Erweiterungsmechanismus kann technisch auf unterschiedliche Weise realisiert werden, die konkrete Realisierung spielt für die Betrachtungen in diesem Beitrag jedoch keine Rolle.

Im weiteren Verlauf dieses Beitrags wird der Begriff „Plug-in“ im Sinne von Plug-ins in einer reinen Plug-in-Architektur verwendet, wenn nicht explizit etwas anderes gesagt wird.

3 Gründe für die Wahl einer Plug-in-Architektur

Für die Entscheidung, eine Anwendung Plug-in-basiert zu entwickeln, gibt es verschiedene Gründe. Naheliegend ist die Wahl einer Plug-in-Architektur, um die Erweiterbarkeit einer Anwendung zu verbessern.

Durch die Erweiterbarkeit ermöglichen es Plug-ins, die Entwicklungszyklen verschiedener Bestandteile einer Anwendung voneinander zu entkoppeln, sodass schnell ein Kernsystem fertiggestellt werden kann, das die Mindestanforderungen eines Anwendungsbereichs erfüllt. Zusätzlich gewünschte Funktionen können dann zu einem späteren Zeitpunkt in Form von Plug-ins erstellt werden, die dieses Minimalsystem erweitern [Mar06].

Während Erweiterbarkeit auch durch eine klassische Plug-in-Schnittstelle in einer erweiterbaren Anwendung erreicht werden kann, verspricht die Wahl einer Plug-in-Architektur zusätzliche Verbesserungen bei der Anpassbarkeit. In einer Plug-in-Architektur werden für den Austausch von Teilen eines Systems dieselben technischen Mittel verwendet wie für die Erweiterung des Systems. Teile eines Softwaresystems, die als Plug-ins implementiert sind, können einfach gegen eine alternative Implementierung ausgetauscht werden oder auch aus dem System entfernt werden. Für unterschiedliche Anwendungskontexte können so unterschiedliche Konfigurationen des Systems erstellt werden, die jeweils genau die für den Kontext benötigten Plug-ins umfassen. Voraussetzung hierfür ist natürlich, dass die Software geeignet in Plug-ins aufgeteilt wurde.

Eine weitere Motivation für die Wahl einer Plug-in-Architektur, die in allen für diesen Beitrag untersuchten Projekten genannt wurde, war der Wunsch, bei der Entwicklung der Software mit anderen Unternehmen kooperieren zu können oder eine Plattform zu etablieren, auf deren Basis auch Dritte individuelle Lösungen entwickeln können. Eine solche Kombination von kooperativer Entwicklung über Unternehmensgrenzen hinweg einerseits und individueller Anpassung andererseits wird auch als ein Software-Ökosystem bezeichnet [Bos09]. Für Unternehmen kann die Entwicklung einer Plattform einen Wettbewerbsvorteil darstellen. Cusumano fasst in [Cus10] verschiedene Arbeiten zum Thema Plattform-Strategien zusammen.

Schließlich wird die Verwendung einer Plug-in-Architektur teilweise damit begründet, dadurch die Funktionalität eines vorhandenen, Plug-in-basierten Rahmenwerks verwenden zu können. Zum Beispiel wurde in einigen Projekten die Entscheidung für die Eclipse Rich Client Platform unter anderem damit begründet, dass diese umfangreiche, vorgefertigte Funktionen für die Gestaltung der Benutzeroberfläche mitliefert.

4 Organisatorische Aspekte

Wie im vorangegangenen Abschnitt geschildert, war in allen untersuchten Projekten eines der Ziele, eine Plattform für die Entwicklung individuell angepasster Softwarelösungen zu etablieren. Dieses Ziel steht im Konflikt mit dem Ziel, für einen konkreten Kontext eine funktionsfähige, einsatzfertige Anwendung zu entwickeln. Zwischen beiden Zielen muss eine geeignete Balance gefunden werden.

In allen untersuchten Softwareprojekten lag eine Mischung aus Anwendungs- und Plattformentwicklung vor:

- In dem ersten Projekt war das Ziel, eine Anwendung zur Steuerung der eigenen technischen Anlagen zu entwickeln. Gleichzeitig wurde als Basis für diese Anwendung eine eigene, offene Plattform entwickelt mit dem Ziel, dass andere Einrichtungen mit ähnlichen Anforderungen diese Plattform nutzen können und sich nach Möglichkeit an ihrer Weiterentwicklung beteiligen.
- In dem zweiten Projekt soll eine Plattform für die Entwicklung von individuellen Leitstand-Implementierungen entwickelt werden. Dazu werden zunächst für ver-

schiedene Projektpartner individuelle Anwendungen implementiert. In diesen sollen dann wiederverwendbare, gemeinsame Bestandteile identifiziert und zu einer Plattform zusammengeführt werden.

- Das dritte Projekt war ein Projekt zur Plattformentwicklung, in dem ein .NET-Rahmenwerk entwickelt wurde. Dieses wird jetzt eingesetzt, um kundenspezifische Anwendungen zu entwickeln.
- Im vierten Projekt soll eine Anwendung entwickelt werden. Diese soll erweiterbar sein um individuelle Anpassungen, beispielsweise zur Anbindung an vorhandene Informationssysteme.

Bei der Anwendungsentwicklung ist das primäre Ziel, für konkrete Anwendungsfälle eine Lösung zu entwickeln, mit der die Anforderungen der Endanwender erfüllt werden, ohne dass weitere Anpassungen nötig sind. Die Anwendungsentwicklung findet meist im Rahmen von einzelnen Projekten statt. Der Projekterfolg ist dabei definiert durch die erfolgreiche Fertigstellung einer einsetzbaren Anwendung.

Bei der Plattformentwicklung dagegen sind die Zielgruppe Softwareentwickler, die die Plattform einsetzen. Eine Plattform muss ausreichend allgemein gehalten werden, um für verschiedene Anforderungen nutzbar zu sein. Die Plattformentwicklung ist meist eher langfristig ausgerichtet und bildet die Basis für zukünftige Projekte, führt jedoch als solche nicht zum Projekterfolg in Anwendungsprojekten.

Die Plattformentwicklung ist mit einem höheren initialen Aufwand verbunden als die Anwendungsentwicklung. Dies liegt zum einen daran, dass bei der Anforderungsermittlung die Anforderungen mehrerer Einsatzkontexte erfasst und berücksichtigt werden müssen. Zum anderen steigt auch der Testaufwand, weil zusätzliche Integrationstests notwendig werden.

Die Plattformentwicklung kann auch die Flexibilität einschränken, weil Schnittstellen einer gemeinsam genutzten Plattform nicht einseitig geändert werden können, ohne dass dadurch die Kompatibilität eingeschränkt würde. Bei der Anwendungsentwicklung dagegen ist es häufig möglich, Schnittstellen anzupassen, weil sich alle Nutzer der Schnittstelle unter der Kontrolle der Entwickler befinden. Bei einer Plattform, die auch von Dritten genutzt wird, ist dies nicht der Fall. Entwickler müssen dies berücksichtigen, wenn sie Schnittstellen entwerfen und entscheiden, welche Schnittstellen als Teil der Plattform freigegeben werden.

Für den Umgang mit dem Konflikt zwischen Plattform- und Anwendungsentwicklung haben wir in den untersuchten Projekten verschiedene Strategien beobachtet, die nachfolgend dargestellt werden.

Das zweite und vierte untersuchte Projekt befinden sich noch am Anfang ihrer Implementierungsphasen, sodass hier noch keine Aussagen getroffen werden können.

Im ersten Projekt gab es keine expliziten Richtlinien oder Prozesse für Änderungen an der Plattform. In der Praxis wurden geplante Änderungen an der Plattform über eine Mailingliste abgesprochen, die von Entwicklern aus den an der Entwicklung beteiligten Organisationen abonniert wurde. Kleinere Änderungen, beispielsweise das Hinzufügen einer

kleineren Funktion oder Bugfixes, wurden teilweise auch ohne Absprache vorgenommen.

Ein Risiko dieser Strategie ist, dass es unbeabsichtigt zu einer parallelen Entwicklung ähnlicher Funktionalität in verschiedenen Anwendungs-Plug-ins kommen kann. Dieses Risiko steigt insbesondere dann, wenn die Anwendungsentwicklung unter Termindruck stattfindet, weil Entwicklern dann die Zeit fehlt, eine gemeinsame Implementierung von Basisfunktionen zu entwerfen und in die Plattform einfließen zu lassen. In dem beobachteten Projekt entstanden mehrfach solche parallel entwickelten Funktionen, die umfangreiche Umbauten erforderten, um sie in die Plattform einfließen zu lassen.

Von einem externen Entwickler wurde außerdem die Einschätzung geäußert, dass der Aufwand, Änderungen an der Plattform zu koordinieren, zu hoch sei. Er hat daher von eigenen Anwendungs-Plug-ins gemeinsam benötigte Funktionen teilweise nicht in die „offizielle“ Plattform einfließen lassen, sondern zusätzlich organsationseigene Plattform-Plug-ins entwickelt.

Im dritten Projekt wird die Plattform nur noch im Rahmen von Projekten zur Anwendungsentwicklung weiterentwickelt. Neue Funktionen entstehen zunächst in der Anwendungsentwicklung und werden in die Plattform eingebracht, wenn es als sinnvoll erachtet wird, sie als Teil der Plattform zur Verfügung zu stellen. Diese Strategie ähnelt der Strategie, die bei der Weiterentwicklung der Eclipse-Plattform angewendet wird.

Bei der Weiterentwicklung von Eclipse werden neue Funktionen bewusst erst dann in die Plattform aufgenommen, wenn es auch einen konkreten Nutzer für die jeweilige Schnittstelle gibt. Dabei wird zwischen öffentlichen und veröffentlichten Schnittstellen unterschieden (vgl. dazu [Fow02]), d. h. es werden technisch auch solche Schnittstellen für die Nutzung durch andere Plug-ins exportiert, die noch kein Teil der offiziellen Plattform sind. Durch diese Vorgehensweise soll sichergestellt werden, dass eine Schnittstelle ausreichend ausgereift und im praktischen Einsatz getestet ist, bevor sie Teil der Eclipse-Plattform wird. Die Unterscheidung zwischen öffentlichen und veröffentlichten Schnittstellen erfolgt dabei mit Hilfe von Namenskonventionen [dR01, GB03].

5 Technische Aspekte

Neben den organisatorischen Fragen hat der Einsatz einer Plug-in-Architektur natürlich auch Auswirkungen auf den softwaretechnischen Entwurf eines Softwaresystems. Nachfolgend werden Entwurfsfragen diskutiert, die sich in diesem Kontext stellen.

5.1 Austausch und Wiederverwendung von Plug-ins

Der Einsatz von Plug-ins ermöglicht es, Teile einer Anwendung zur Ladezeit oder sogar erst zur Laufzeit auszutauschen oder zu entfernen. Dabei können zwei Arten von Austauschbarkeit unterschieden werden: erstens der Austausch eines Plug-ins gegen ein anderes Plug-in (Austausch) und zweitens der Einsatz eines Plug-ins als Erweiterung in einer

anderen Anwendung (Wiederverwendung).

5.1.1 Austausch

Durch den Erweiterungsmechanismus einer Plug-in-Architektur wird vorrangig der Austausch von Plug-ins unterstützt. Über diesen Mechanismus kann ein Plug-in zur Laufzeit dynamisch die installierten Erweiterungen auffinden und auf diese zugreifen. Ein Plug-in, das Erweiterungen akzeptiert, bildet somit eine Art Rahmenwerk für seine Erweiterungen.

Die Kontrolle darüber, wann welche Erweiterungen geladen werden, liegt bei den Plug-ins, die diese Erweiterungen verwenden. Dadurch werden nicht zwingend alle angebotenen Erweiterungen auch tatsächlich zur Laufzeit in das System eingebunden. Es stellt technisch keinen Fehler dar, wenn ein Plug-in eine Erweiterung anbietet, für die sich im System zur Laufzeit keine Nutzer finden. Somit können Erweiterungen auch für optionale Systemteile angeboten werden, von denen der Entwickler einer Erweiterung nicht wissen kann, ob sie in einem konkreten System installiert sind.

Diese Eigenschaft von Plug-in-Systemen ist wünschenswert, weil sie zu einer losen Kopplung von Plug-ins führt und die Voraussetzung dafür schafft, verschiedene Systemteile und Erweiterungen miteinander zu integrieren, ohne Abhängigkeiten zwischen diesen zu erzwingen.

Dennoch kann dieses Verhalten in der Entwicklungspraxis auch zu Schwierigkeiten führen. Vor allem die Fehlersuche wird erschwert, wenn eine Erweiterung entwickelt wurde, die vom System nicht wie erwartet eingebunden wird. Weil hier technisch kein Fehler vorliegt, fehlt Entwicklern in dieser Situation zunächst ein Anhaltspunkt, wo die Ursache liegen könnte. Eine triviale, in der Praxis aber häufig beobachtete Ursache für solche Probleme sind zum Beispiel Tippfehler bei der Angabe von textuellen IDs. Mit zunehmender Praxiserfahrung fällt es Entwicklern leichter, solche Fehler aufzuspüren.

5.1.2 Wiederverwendung

Die Wiederverwendung von Plug-ins in anderen Kontexten wird durch Erweiterungsmechanismen nur eingeschränkt verbessert. Theoretisch kann ein Erweiterungsmechanismus zwar verwendet werden, um Wiederverwendung zu ermöglichen. Insbesondere die oben genannte Möglichkeit, Erweiterungen für unterschiedliche Kontexte bereitzustellen, von denen dann jeweils nur die im konkreten Kontext nutzbaren aufgefunden und eingebunden werden, erlaubt sehr flexible Szenarien.

In der Praxis haben wir jedoch beobachtet, dass viele Plug-ins implizite Annahmen über ihren Ausführungskontext machen, durch die ihre Einsetzbarkeit in anderen Kontexten eingeschränkt wird. Dazu gehören beispielsweise Annahmen darüber, welche der angebotenen Erweiterungen in jedem Fall in das System eingebunden werden, oder Annahmen darüber, wie die Benutzeroberfläche des erweiterten Systems aufgebaut ist und wo Erweiterungen in diese eingebunden werden können.

Folgendes Beispiel aus dem ersten untersuchten Projekt hilft, dieses Problem zu illustrieren: Ein Plug-in, das urpsrünglich für die in dem Projekt entstandene Werkzeugum-

gebung entwickelt wurde, sollte nun auch für Softwareentwickler zur Verfügung gestellt werden, die es in ihre integrierte Entwicklungsumgebung einbinden möchten. Aus technischer Sicht schien dies möglich, da sowohl die Werkzeugumgebung aus dem Projekt als auch die Entwicklungsumgebung (die Eclipse IDE) auf der Eclipse-Plattform basierten. In der Praxis stellte sich jedoch heraus, dass die Nutzung des Plug-ins in der Eclipse IDE nur mit Einschränkungen möglich war, weil durch die abweichenden Menüstrukturen in der Eclipse IDE gegenüber der Werkzeugumgebung in der IDE nicht alle Menüpunkte eingebunden wurden.

Solche impliziten Annahmen können (zumindest mit den in den untersuchten Projekten eingesetzten Techniken) nicht vollständig explizit gemacht werden, ohne dadurch gleichzeitig die Wiederverwendbarkeit noch stärker einzuschränken. Würde ein Plug-in eine Abhängigkeit zu den anderen Plug-ins deklarieren, die einen den Annahmen entsprechenden Kontext bilden, so wären zwar die Abhängigkeiten explizit, ein Einsatz in einem anderen Kontext jedoch überhaupt nicht mehr möglich.

Entwickler müssen daher beim Entwurf von Plug-ins darauf achten, implizite Abhängigkeiten soweit wie möglich zu vermeiden, ohne dadurch die Verwendbarkeit des Plug-ins unnötig einzuschränken.

5.2 Verwendung unterschiedlicher Erweiterungsmechanismen

Aktuelle Plug-in-Rahmenwerke bieten häufig verschiedene Erweiterungsmechanismen nebeneinander an. In der Eclipse Rich Client Platform beispielsweise können sowohl Extension Points [GB03] als auch OSGi Services [WHKL08] als Erweiterungsmechanismus verwendet werden.

Für Entwickler wirft diese Situation einerseits die Frage auf, welchen der verfügbaren Mechanismen sie verwenden sollen, wenn sie eine Erweiterungsschnittstelle bereitstellen möchten (mehr zu dieser Frage im nachfolgenden Abschnitt), und andererseits die Frage, wie über verschiedene Mechanismen angebotene Erweiterungen miteinander kombiniert werden können.

Zu Problemen führt die Kombination verschiedener Mechanismen dann, wenn aus einer Erweiterung heraus, die über einen Mechanismus instanziert wird, auf Erweiterungen zugegriffen werden soll, die über einen anderen Mechanismus angeboten werden. Im ersten Projekt konnten wir dieses Problem mehrfach beobachten, weil dort der Bedarf bestand, aus über den Extension-Point-Mechanismus eingebundenen Erweiterungen heraus auf OSGi Services zuzugreifen. Ein direkter Zugriff ist hier nicht möglich, weil die über den Extension Point eingebundene Erweiterung von der Eclipse-Plattform instanziert wird und dabei keine Referenz auf die OSGi-Umgebung erhält.

Von den Entwicklern wurde dieses Problem dadurch umgangen, dass entsprechende Referenzen auf die OSGi-Umgebung oder die benötigten Services in statischen Variablen gespeichert wurden. Dadurch kam es jedoch mehrfach zu Fehlern durch unzutreffende Annahmen über die Lebenszyklen der Objekte, zum Beispiel weil versucht wurde, auf noch nicht verfügbare Services zuzugreifen.

Dasselbe Problem wurde auch im zweiten Projekt beobachtet. Hier wurde von Entwicklern der Vorschlag geäußert, eine bestimmte Konfiguration der Startreihenfolge der Plug-ins vorzuschreiben, um eine größere Kontrolle über die Lebenszyklen zu erlangen.

In der neuen Eclipse-Version 4.0, die im Juni 2010 veröffentlicht wurde, wird für Erweiterungen Dependency Injection unterstützt [Art09]. Das macht es möglich, für Erweiterungen deklarativ bestimmte Abhängigkeiten anzugeben, diese werden dann bei der Erzeugung der Erweiterung automatisch von der Eclipse-Plattform in die Erweiterung injiziert. Es muss sich noch zeigen, ob dieser Mechanismus die genannten Probleme beheben kann.

Einen experimentellen Dependency-Injection-Mechanismus für die Integration von OSGi Services hat Bartlett entwickelt [Bar].

Ein vergleichbares Problem, das ebenfalls auf inkompatible Lebenszyklen der von Plug-ins instanzierten Objekte zurückzuführen ist, kann sich bei der Portierung von Plug-in-basierten Anwendungen ins Web ergeben. Im Eclipse-Umfeld existiert mit der Rich Ajax Platform [RAP] ein Rahmenwerk, das diese Portierung ermöglichen soll. Im ersten Projekt haben wir dieses Rahmenwerk untersucht und dabei festgestellt, dass unterschiedliche Lebenszyklen das größte Hindernis bei der Portierung bilden. Für eine Web-Anwendung wäre ein zusätzlicher Session-Lebenszyklus notwendig, der in einer Desktop-Anwendung jedoch in der Regel nicht vorgesehen ist [CHRM09].

5.3 Erweiterbarkeit

Wird in einem Plug-in Funktionalität implementiert, die anderen Plug-ins zur Verfügung gestellt werden soll, so stellt sich die Frage, ob dies über einen Erweiterungsmechanismus realisiert werden soll oder ob das Plug-in eine entsprechende Programmierschnittstelle (API) exportieren soll. Wird ein Erweiterungsmechanismus verwendet, so übernimmt das Plug-in gegenüber Klienten die Rolle eines Rahmenwerks. Das bedeutet, es kontrolliert, wann welche Erweiterungen verwendet werden. Exportierte APIs hingegen können von anderen Plug-ins ähnlich wie eine gewöhnliche Bibliothek importiert und aufgerufen werden.

In keinem der untersuchten Projekte existierten Richtlinien für die Entwickler dazu, für welche Aufgaben ein Erweiterungsmechanismus verwendet werden sollte.

Wird ein Erweiterungsmechanismus verwendet und stehen in der verwendeten Plattform mehrere alternative Mechanismen zur Verfügung, so stellt sich zusätzlich die Frage, welcher von diesen verwendet werden sollte. Auch hierzu gab es in keinem Projekt Richtlinien. Für die Eclipse-Plattform finden sich im Internet Vergleiche der verschiedenen Mechanismen, beispielsweise [Bar07].

6 Auswirkungen auf die Benutzbarkeit

Eine Plug-in-Architektur zu verwenden, kann sich auch auf die Benutzbarkeit einer Anwendung auswirken. Bereits zuvor angesprochen wurde das Problem, dass Plug-ins implizite Annahmen über den Aufbau der Benutzeroberfläche machen können, in die sie eingebunden werden.

Eine andere Eigenschaft Plug-in-basierter Systeme, die zu Problemen führen kann, ist die verzögerte Aktivierung (Lazy Activation) von Plug-ins. Im Eclipse-Rahmenwerk wird die Implementierung eines Plug-ins erst direkt vor der Ausführung des Codes geladen. Zuvor verwendet die Plattform deklarative Informationen, um die von dem Plug-in beigetragenen Erweiterungen bereits an der Oberfläche darstellen zu können. Dadurch soll die Ladezeit von Eclipse-basierten Anwendungen verringert werden [GB03].

Für den Benutzer ist nicht erkennbar, zu welchen an der Oberfläche dargestellten Elementen die zugehörige Implementierung bereits geladen wurde. Dies kann zu nicht erwartungskonformen Verhalten der Anwendung führen, wenn der Anwender ein Element als „aktiv, aber im Hintergrund“ wahrnimmt, es tatsächlich jedoch noch nicht aktiviert wurde.

Im ersten Projekt trat dieses Problem im Zusammenhang mit Ansichten (Views) auf, die Nachrichten über das Netzwerk empfangen und auflisten. Befand sich eine solche Ansicht beim Start der Anwendung im Hintergrund, so wurde sie von Eclipse bereits dargestellt, der zugehörige Code jedoch noch nicht geladen. Der Benutzer nahm diese Ansicht jedoch als „im Hintergrund geöffnet“ wahr und erwartete, dass auch bereits im Hintergrund Nachrichten empfangen werden. Tatsächlich wurde die Netzwerkverbindung jedoch erst aufgebaut, sobald der Benutzer die Anwendung zum ersten Mal in den Vordergrund holte. Dieses Verhalten war für die Benutzer unerwartet. Es führte außerdem dazu, dass eine im Hintergrund geöffnete Ansicht sich in zwei unterschiedlichen Zuständen befinden konnte, die an der Oberfläche nicht unterscheidbar waren.

Um solche Probleme zu vermeiden, sollten Entwickler bei der Programmierung Anwendungen mit einer Plug-in-Architektur darauf achten, dass durch die Erweiterungsbeziehungen zwischen Plug-ins keine Inkonsistenzen zwischen dem tatsächlichen und dem vom Benutzer wahrnehmbaren Zustand der Anwendung entstehen können.

7 Zusammenfassung

Basierend auf Erfahrungen aus Softwareprojekten in der Praxis wurden in diesem Beitrag Besonderheiten bei der Entwicklung Plug-in-basierter Software dargestellt. Plug-ins ermöglichen es, Software zu entwickeln, die auch nach der Auslieferung beim Anwender verändert und aktualisiert werden kann. Dies schafft die Möglichkeit, Software über Organisationsgrenzen hinweg kooperativ zu entwickeln. Technisch ermöglicht wird diese Flexibilität durch einen komponentenbasierten Ansatz in Verbindung mit einem Erweiterungsmechanismus, der es ermöglicht, Plug-ins mit Erweiterungen dynamisch in ein System einzufügen und aus diesem zu entfernen.

Organisatorisch müssen Unternehmen, die Plug-ins gewinnbringend einsetzen möchten, eine geeignete Entwicklungsstrategie wählen. Der Konflikt zwischen der Plattform- und der Anwendungsentwicklung bildet hier ein Spannungsfeld, in dem geeignete Kompromisse gefunden werden müssen. Dieses Spannungsfeld hat auch Auswirkungen auf die Gestaltung der Architektur eines Softwaresystems, weil es den Entwurf und die Freigabe der Schnittstellen zwischen seinen Komponenten beeinflusst.

In der Praxis haben wir unterschiedliche Strategien beobachtet, mit denen Unternehmen diesen Konflikt zu adressieren versuchen. Bisher können wir keine eindeutige Antwort darauf liefern, welche Strategie in welcher Situation am besten geeignet ist.

Auf der technischen Seite stellt Entwickler vor allem die hohe Dynamik in Plug-in-basierten Systemen vor Herausforderungen. Entwickler müssen ermitteln, an welchen Stellen Erweiterbarkeit erforderlich ist, um fachlich motivierte Änderungsszenarien zu unterstützen, und an welchen Stellen eine klassische, programmatische Schnittstelle bevorzugt werden sollte, um den Entwicklungsaufwand zu reduzieren.

Als problematisch herausgestellt haben sich in der Praxis vor allem die Wiederverwendung von Plug-ins in Kontexten, für die diese ursprünglich nicht entworfen wurden, und die gleichzeitige Nutzung verschiedener Erweiterungstechniken. Bei letzterem müssen Entwickler damit umgehen, dass Erweiterungen von unterschiedlichen Containern instanziert werden. Aktuelle Neuentwicklungen wie Eclipse 4.0 versprechen hier Abhilfe durch die bessere Unterstützung von Techniken wie Dependency Injection, werden jedoch in der Praxis bisher kaum eingesetzt.

Literatur

[Art09] John Arthorne. *White Paper: e4 Technical Overview*, 2009. `http://www.eclipse.org/e4/resources/e4-whitepaper.php`, letzer Abruf 8.10.2010.

[Bar] Neil Bartlett. *Extensions2Services*, Projekt-Website. `http://github.com/njbartlett/Extensions2Services`, letzter Abruf 8.10.2010.

[Bar07] Neil Bartlett. *A Comparison of Eclipse Extensions and OSGi Services*, 2007. `http://www.eclipsezone.com/articles/extensions-vs-services/`, letzter Abruf 6.10.2010.

[Bir05] Dorian Birsan. On plug-ins and extensible architectures. *Queue*, 3(2):40–46, 2005.

[Bos09] Jan Bosch. From software product lines to software ecosystems. In Dirk Muthig und John D. McGregor, Hrsg., *SPLC*, Jgg. 446 of *ACM International Conference Proceeding Series*, Seiten 111–119. ACM, 2009.

[CHRM09] M. Clausen, J. Hatje, J. Rathlev und K. Meyer. Eclipse RCP on the Way to the Web. In *ICALEPCS 2009 – Proceedings*, Seiten 884–886, 2009.

[Cus10] Michael Cusumano. Technology strategy and management: The evolution of platform thinking. *Commun. ACM*, 53(1):32–34, 2010.

[dR01] Jim des Rivières. How to Use the Eclipse API, 2001. `http://www.eclipse.org/articles/article.php?file=Article-API-Use/index.html`, letzter Abruf 8.10.2010.

[Fow02] Martin Fowler. Public versus Published Interfaces. *IEEE Software*, 19(2):18–19, 2002.

[GB03] Erich Gamma und Kent Beck. *Contributing to Eclipse: Principles, Patterns, and Plug-Ins*. Addision-Wesley, Boston, 2003.

[Mar06] Klaus Marquardt. Patterns for Plug-ins. In *Pattern Languages of Program Design 5*, Seiten 310–335, Addison-Wesley, Upper Saddle River, NJ, 2006.

[RAP] o.V. *Rich Ajax Platform (RAP)*, Projekt-Website. `http://www.eclipse.org/rap/`, letzter Abruf 8.10.2010.

[Rat08] Jörg Rathlev. Plug-ins: an Architectural Style for Component Software. In Ralf Reussner, Clemens Szyperski und Wolfgang Weck, Hrsg., *Proceedings of the thirteenth International Workshop on Component-Oriented Programming (WCOP 2008)*, Seiten 5–9, 2008.

[SGM02] Clemens Szyperski, Dominik Gruntz und Stephan Murer. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley, London, 2. Auflage, 2002.

[WHKL08] Gerd Wütherich, Nils Hartmann, Bernd Kolb und Matthias Lübken. *Die OSGi Service Platform*. dpunkt, Heidelberg, 2008.

Workshops

Dritter Workshop zu „Design For Future – Langlebige Softwaresysteme“

Stefan Sauer[1], Christof Momm[2], Mircea Trifu[3]

[1] Universität Paderborn
sauer@s-lab.upb.de
[2] SAP AG
christof.momm@sap.com
[3] Forschungszentrum Informatik Karlsruhe
mtrifu@fzi.de

Software altert relativ zu ihrer Umgebung. Denn die Umgebung, in der Software eingesetzt wird, verändert sich kontinuierlich. Sowohl die Anforderungen an die Software als auch die verwendete Hardware und Infrastruktur sind einem stetigen Wandel ausgesetzt. Wird die Software nicht entsprechend weiterentwickelt, so altert sie rasch relativ zu ihrer Umgebung. Dieses Problem ist vor allem im Bereich der großen betrieblichen Informationssysteme unter dem Oberbegriff Legacy bekannt. Durch die zunehmende Bedeutung langlebiger, komplexer eingebetteter Systeme und die steigende Vernetzung von Systemen wird sich die Situation in Zukunft noch weiter verschärfen. Wenn Alterungsprozesse von Teilen das Gesamtsystem gefährden, sprechen wir von der Erosion von Softwaresystemen. Diese Probleme haben enorme ökonomische Bedeutung. Aktuelle Ansätze in der Softwaretechnik, insbesondere in den Bereichen modellbasierte Entwicklungsmethoden, Lifecycle-Management, Softwarearchitektur, Requirements Engineering und Re-Engineering, können dazu beitragen, die Situation zu verbessern, wenn sie geeignet weiterentwickelt und angewandt werden. Derartige Ansätze werden im Workshop Design for Future 2011 diskutiert.

Ziel des Workshops ist es, Wissenschaftler und Praktiker zusammenzubringen, die an der Entwicklung oder der Anwendung aktueller Ansätze interessiert sind, um die verschiedenen Facetten und Herausforderungen der Software-Alterung zu beherrschen. Hierzu streben wir die Entwicklung neuer Konzepte, Methoden, Techniken und Werkzeuge der Softwaretechnik an, um die erheblichen Investitionen, die in große Softwaresysteme getätigt werden, zu schützen und massive Probleme durch zunehmende Software-Alterung zu verhindern. Im Workshop sollen sowohl wissenschaftliche und in der Praxis entstandene Lösungen als auch praktische Erfahrungen berichtet und diskutiert werden, um die Entstehung neuer Legacy-Probleme und die Erosion von Software zu verhindern.

Vierter Workshop zur Software-Qualitätsmodellierung und -bewertung (SQMB 2011)

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
{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. Viele aktuelle Projekte und Initiativen, wie ISO 25000, CISQ, Squale oder Quamoco, zeigen den noch immer vorhandenen Forschungsbedarf. Eine umfassende Behandlung von Qualität wird in diesen Initiativen durch Qualitätsmodelle und darauf aufbauenden Bewertungen erwartet. Der Workshop SQMB 2011 hat zum Ziel die in den früheren Ausgaben erarbeiteten Erfahrungen und Problemstellungen zu vertiefen und Fortschritte darin zu diskutieren.

Dieser Workshop hat das Ziel, Erfahrungen mit Qualitätsmodellierung und -bewertung zu sammeln und gemeinsam neue Forschungsrichtungen zu entwickeln. Dieses mal wird ein spezieller Fokus auf eingebettete Softwaresysteme gerichtet.

Forschung für die zivile Sicherheit: Interdisziplinäre Anforderungsanalyse zwischen Praxis und Softwarelösung (FZS)

Birgitta König-Ries[1], Rainer Koch[2], Stefan Strohschneider[1]

[1] Friedrich-Schiller-Universität Jena
{birgitta.koenig-ries, strohschneider}@uni-jena.de
[2] Universität Paderborn
r.koch@cik.uni-paderborn.de

Die „Forschung für die zivile Sicherheit“ stellt ein Beispiel für einen Forschungsbereich dar, der durch sehr heterogene Fragegestellungen und Stakeholder-Gruppen geprägt ist. Die darauf begründete Interdisziplinarität von Forschungsaktivitäten bedeutet Chance und Herausforderung zugleich. Dies betrifft insbesondere die Erforschung der Anwendung von IT-Systemen, die den Kern vieler nationaler und internationaler Aktivitäten bilden. Fragestellungen reichen dabei von der IT-Unterstützung für den vorbeugenden Brandschutz über Ansätze zur besseren Patientenverfolgung in Großschadenslagen und Entscheidungsunterstützungswerkzeugen bei Evakuierungen bis hin zu organisationsübergreifenden Kommunikationsplattformen und Lernumgebungen.

Ein wichtiges Element fast aller Projekte in diesem Bereich ist die explizit als solche benannte nicht-informatorische „Begleitforschung“. Die Kooperation der klassischen technischen Disziplinen und Partnern aus der Wirtschaft wird dabei auf Partner verschiedener Fachgebiete zum interdisziplinären Austausch erweitert. Neben der Psychologie spielen auch andere Fachgebiete wie z.B. die Kommunikations-, Rechts- und Kulturwissenschaft, Human-Factors-Forschung oder auch Pädagogik sowie verschiedene weitere Gebiete der Geistes-, Sozial- und Verhaltenswissenschaften eine Rolle. Allgemeines Ziel ist es, eine bessere Annäherung an den Sachverhalt bzw. eine höhere Annäherung an den Realitätsbereich zu erreichen.

Mit Blick auf Produkt- und insbesondere Softwareentwicklungsprozesse ergeben sich inhaltlich und methodisch Potentiale in der Schnittstelle, die besonders für das Requirements Engineering und die Anforderungsanalyse Verbesserungen versprechen. Oftmals treffen hier jedoch unterschiedliche wissenschaftliche Konzepte, Arbeitsmethoden und Forschungsschwerpunkte aufeinander; der Transfer oder die Frage nach der Transferierbarkeit der Ergebnisse der nicht-technischen Partner in die Entwicklung der neuen technischen Systeme und Lösungen gestaltet sich als Herausforderung bei der Projektarbeit.

Schwerpunktthema des Workshops sollen daher Fragen sein, die Schnittstelle betreffen:

- Wo liegen die Potentiale der interdisziplinären Anforderungsanalyse, welche Risiken müssen berücksichtigt werden?

- Wie kann potentiell skeptische Einstellung gegenüber zusätzlicher IT konstruktiv durch interdisziplinäre Forschung aufgegriffen werden?
- Wie können Methoden aus verschiedenen Disziplinen aufeinander abgestimmt werden?
- Welche Werkzeuge sind zur anforderungsbezogenen Kommunikation geeignet?
- Wie groß sind die inhaltlichen und methodischen Überschneidungen und Entfernungen?
- Wie können fundierte Lessons Learned und Best Practices erarbeitet werden?

Oder provokant formuliert: Fehlt der Bedarf oder fehlt die Lösung? Wie kommt die Expertise aus den Köpfen der „Begleitforscher“, in die Köpfe der „Techniker“ und zurück?

Für das Schwerpunktthema sind Beiträge sowohl von Informatikern als auch aus anderen Forschungsdisziplinen erwünscht. Erfahrungsberichte aus Projekten sind dabei ebenso willkommen wie Arbeiten, die sich etwa mit den unterschiedlichen Wissenschaftstraditionen in den beteiligten Disziplinen oder Strategien des Wissenstransfers beschäftigen.

Die zivile Sicherheitsforschung wird hier als Beispiel betrachtet. Neben Beiträgen zu diesem Schwerpunktthema richtet sich dieser Workshop auch an allgemeine Arbeiten und Ergebnisse in anderen Bereichen anwendungsbezogener Forschung. Im Rahmen des Workshops soll auch diskutiert werden, in welchem Rahmen die Diskussion zum Schwerpunktthema verstetigt werden kann.

Zweiter Workshop zur Zukunft der Entwicklung softwareintensiver, eingebetteter Systeme (ENVISION 2020)

Kim Lauenroth[1], Klaus Pohl[1], Wolfgang Böhm[2], Manfred Broy[2]

[1] Universität Duisburg-Essen
{kim.lauenroth@paluno, klaus.pohl@sse}.uni-due.de
[2] Technische Universität München
{boehmw, broy}@in.tum.de

Softwareintensive, eingebettete Systeme unterstützen den Menschen schon heute in vielen Bereichen des Lebens – sichtbar und unsichtbar. Beispielsweise verbessern sie im Automobil die Sicherheit, regulieren das Klima in Gebäuden oder steuern medizinische Geräte bis hin zu ganzen Industrieanlagen. Experten prognostizieren für die Zukunft eine rasante Zunahme softwareintensiver, eingebetteter Systeme.

Die Ausweitung des Funktionsumfangs und die zunehmende Vernetzung eingebetteter Systeme führen gleichzeitig zu einer rasanten Zunahme der Komplexität dieser Systeme, die auch im Entwicklungsprozess Berücksichtigung finden muss. Existierende Vorgehensweisen und Methoden stoßen bereits unter den heutigen Rahmenbedingungen (z.B. Zeit- und Kostendruck) an ihre Grenzen. Existierende Ansätze und Methoden müssen aufgrund der wachsenden Herausforderungen in Frage gestellt und in Teilen neu konzipiert werden.

Der Workshop ENVISION 2020 verfolgt das Ziel, die Entwicklung und Diskussion zukünftiger Ansätze, Vorgehensweisen und Methoden zur Entwicklung softwareintensiver, eingebetteter Systeme zu fördern. Wir laden zu diesem Workshop Beiträge von Forschern und Praktikern ein, die diese Diskussion stimulieren und die Konzeption neuer, verbesserter Entwicklungsansätze mit gestalten wollen. Ein besonderes Augenmerk gilt dabei modellbasierten Entwicklungsansätzen.

Innovative Systeme zur Unterstützung der zivilen Sicherheit: Architekturen und Gestaltungskonzepte

Anna Maria Japs[1], Benedikt Birkhäuser[2]

[1] Universität Paderborn
japs@cik.uni-paderborn.de
[2] Bergische Universität Wuppertal
birkhaeuser@uni-wuppertal.de

In den letzten Jahren sind verstärkt Forschungsanstrengungen entstanden, um IT-Systeme zur Unterstützung aller beteiligten Akteure in Szenarien der zivilen Sicherheit (Bevölkerungsschutz, Katastrophenschutz, etc.) zu entwickeln. Entwicklung und Nutzung solcher Systeme unterliegen besonderen Rahmenbedingungen durch ihre Öffentlichkeitswirksamkeit und Notwendigkeiten der Verlässlichkeit, Anpassbarkeit/Flexibilität, Sicherheitsrelevanz, Nachvollziehbarkeit und Interoperabilität solcher IT-Systeme. Ziel des Workshops ist es, Designer und Systementwickler aus diesem Bereich zusammenzubringen, um Erfahrungen und Ansätze für Architekturen, Anwendungen und Vorgehensweisen zu diskutieren. Die Innovativität der diskutierten Ansätze kann dabei sowohl auf der besonders gelungenen Umsetzung der Rahmenbedingungen als auch auf der Erschließung und Einbindung neuer technologischer Konzepte kommen. Neben den Forschenden in diesem Bereich möchten wir auch explizit forschungsinteressierte Praktiker zur Darstellung ihrer Konzepte und Erfahrungen auf diesem Workshop einladen.

Ziele:

- Darstellung von Ansätzen aktueller Forschungsprojekten aus der Domäne.
- Entwicklung von Best-Practice Lösungen.
- Austausch und Diskussion von Erfahrungen aus bereits umgesetzten Ansätzen.
- Identifizierung aktueller Herausforderungen.
- Ausblick auf zukünftige Möglichkeiten im Bereich der Systementwicklung für Public Safety.

Produktlinien im Kontext: Technologie, Prozesse, Business und Organisation (PIK 2011)

Andreas Birk[1], Klaus Schmid[2], Markus Völter[3]

[1] Software.Process.Management
andreas.birk@swpm.de
[2] Universität Hildesheim
schmid@sse.uni-hildesheim.de
[2] voelter - ingenieurbüro für softwaretechnologie, itemis AG
voelter@acm.org

Produktlinien sind heute in vielen Bereichen der Software-Industrie vertreten, von eingebetteten Systemen bis zu betrieblichen Informationssystemen. Sie ermöglichen höhere Produktivität, steigern die Qualität und verbessern die strategischen Positionen der Unternehmen. Dennoch bergen Produktlinien für viele Unternehmen noch bedeutende Herausforderungen und Risiken. Die Gründe liegen teilweise im technischen Bereich. So sind viele Produktlinien-Technologien für den breiten Einsatz in der Praxis noch nicht genügend ausgereift und miteinander integriert. Die wohl größten Herausforderungen stellen sich in den Wechselwirkungen zwischen den technischen Verfahren mit den Prozessen sowie dem organisatorischen und geschäftlichen Kontext der Produktlinienentwicklung.– Wie müssen die technologischen Ansätze auf diese Wechselwirkungen ausgerichtet sein? Welche Potenziale bieten neue technologische Entwicklungen in unterschiedlichen Einsatzfeldern?

Der Workshop bietet ein Forum für die deutschsprachige Community zu Software- Produktlinien (SPL) und fördert den Erfahrungsaustausch zu SPL. Er verfolgt insbesondere die Ziele:

- Den Dialog zwischen Praxis und anwendungsorientierter Forschung fördern
- SPL-Erfahrungen und neue SPL-Technologien vorstellen
- Eine Standortbestimmung der SPL-Technologie in Forschung und Praxis vornehmen

Workshop und Fachgruppentreffen der FG OOSE – „Evolutionäre Software- und Systementwicklung – Methoden und Erfahrungen“ (ESoSyM 2011)

Gregor Engels[1], Bernhard Schätz[2], Matthias Riebisch[3], Christian Zeidler[4]

[1] Universität Paderborn
engels@upb.de
[2] fortiss GmbH
schaetz@fortiss.org
[3] Technische Universität Ilmenau
matthias.riebisch@tu-ilmenau.de
[4] ABB Forschungszentrum Ladenburg
christian.zeidler@de.abb.com

Bei der Entwicklung von Softwaresystemen sind immer komplexere Anforderungen zu erfüllen. Gleichzeitig steigen die Forderungen nach langfristiger Nutzbarkeit bei einfacher Änderbarkeit der Systeme. Evolutionäre Entwicklung und modellbasierte Methoden helfen, diese Ziele zu erreichen und Risiken der Entwicklung besser zu beherrschen. Allerdings bestehen Differenzen zwischen der Entwicklung von Ansätzen in der Forschung und der Anwendung in industrieller Softwareentwicklung. Der Workshop soll breiten Raum für die Erörterung problemangepasster Ansätze und Ergebnisse der Forschung sowie der Problemstellungen und Erfahrungen der Industrie bieten. Darüber hinaus soll er auch eine Zusammenfassung und Darstellung der Arbeiten der Arbeitskreise der Fachgruppe OOSE ermöglichen.

Der Workshop setzt die folgenden Schwerpunkte:

- Modellbasierte Methoden für evolutionäre Entwicklung
- Methoden und Ansätze zur Unterstützung langlebiger Systeme Methoden mit Unterstützung für spezifische Eigenschaften eingebetteter Systeme
- Produktlinien- und Komponenten-Methoden
- Effizienz der Entwicklungstätigkeiten im Software-Lebenszyklus Verbindung zwischen Werkzeugen
- Unterstützung bei Entwurfsentscheidungen
- Transfer und Anpassung von Forschungsergebnissen an betriebliche Erfordernisse
- Erfahrungsberichte aus Einführung und Anwendung von neuen Ansätzen

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: Biometrics 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)

P-145 Knut Hinkelmann, Holger Wache (Eds.)
WM2009: 5th Conference on Professional Knowledge Management

P-146 Markus Bick, Martin Breunig, Hagen Höpfner (Hrsg.)
Mobile und Ubiquitäre Informationssysteme – Entwicklung, Implementierung und Anwendung
4. Konferenz Mobile und Ubiquitäre Informationssysteme (MMS 2009)

P-147 Witold Abramowicz, Leszek Maciaszek, Ryszard Kowalczyk, Andreas Speck (Eds.)
Business Process, Services Computing and Intelligent Service Management
BPSC 2009 · ISM 2009 · YRW-MBP 2009

P-148 Christian Erfurth, Gerald Eichler, Volkmar Schau (Eds.)
9th International Conference on Innovative Internet Community Systems
I^2CS 2009

P-149 Paul Müller, Bernhard Neumair, Gabi Dreo Rodosek (Hrsg.)
2. DFN-Forum Kommunikationstechnologien
Beiträge der Fachtagung

P-150 Jürgen Münch, Peter Liggesmeyer (Hrsg.)
Software Engineering
2009 - Workshopband

P-151 Armin Heinzl, Peter Dadam, Stefan Kirn, Peter Lockemann (Eds.)
PRIMIUM
Process Innovation for Enterprise Software

P-152 Jan Mendling, Stefanie Rinderle-Ma, Werner Esswein (Eds.)
Enterprise Modelling and Information Systems Architectures
Proceedings of the 3rd Int'l Workshop EMISA 2009

P-153 Andreas Schwill, Nicolas Apostolopoulos (Hrsg.)
Lernen im Digitalen Zeitalter
DeLFI 2009 – Die 7. E-Learning Fachtagung Informatik

P-154 Stefan Fischer, Erik Maehle Rüdiger Reischuk (Hrsg.)
INFORMATIK 2009
Im Focus das Leben

P-155 Arslan Brömme, Christoph Busch, Detlef Hühnlein (Eds.)
BIOSIG 2009:
Biometrics and Electronic Signatures
Proceedings of the Special Interest Group on Biometrics and Electronic Signatures

P-156 Bernhard Koerber (Hrsg.)
Zukunft braucht Herkunft
25 Jahre »INFOS – Informatik und Schule«

P-157 Ivo Grosse, Steffen Neumann, Stefan Posch, Falk Schreiber, Peter Stadler (Eds.)
German Conference on Bioinformatics 2009

P-158 W. Claupein, L. Theuvsen, A. Kämpf, M. Morgenstern (Hrsg.)
Precision Agriculture Reloaded – Informationsgestützte Landwirtschaft

P-159 Gregor Engels, Markus Luckey, Wilhelm Schäfer (Hrsg.)
Software Engineering 2010

P-160 Gregor Engels, Markus Luckey, Alexander Pretschner, Ralf Reussner (Hrsg.)
Software Engineering 2010 – Workshopband
(inkl. Doktorandensymposium)

P-161 Gregor Engels, Dimitris Karagiannis Heinrich C. Mayr (Hrsg.)
Modellierung 2010

P-162 Maria A. Wimmer, Uwe Brinkhoff, Siegfried Kaiser, Dagmar Lück-Schneider, Erich Schweighofer, Andreas Wiebe (Hrsg.)
Vernetzte IT für einen effektiven Staat
Gemeinsame Fachtagung Verwaltungsinformatik (FTVI) und Fachtagung Rechtsinformatik (FTRI) 2010

P-163 Markus Bick, Stefan Eulgem, Elgar Fleisch, J. Felix Hampe, Birgitta König-Ries, Franz Lehner, Key Pousttchi, Kai Rannenberg (Hrsg.)
Mobile und Ubiquitäre Informationssysteme
Technologien, Anwendungen und Dienste zur Unterstützung von mobiler Kollaboration

P-164 Arslan Brömme, Christoph Busch (Eds.)
BIOSIG 2010: Biometrics and Electronic Signatures Proceedings of the Special Interest Group on Biometrics and Electronic Signatures

P-165 Gerald Eichler, Peter Kropf, Ulrike Lechner, Phayung Meesad, Herwig Unger (Eds.) 10th International Conference on Innovative Internet Community Systems (I^2CS) – Jubilee Edition 2010 –

P-166 Paul Müller, Bernhard Neumair, Gabi Dreo Rodosek (Hrsg.) 3. DFN-Forum Kommunikationstechnologien Beiträge der Fachtagung

P-167 Robert Krimmer, Rüdiger Grimm (Eds.) 4th International Conference on Electronic Voting 2010 co-organized by the Council of Europe, Gesellschaft für Informatik and E-Voting.CC

P-168 Ira Diethelm, Christina Dörge, Claudia Hildebrandt, Carsten Schulte (Hrsg.) Didaktik der Informatik Möglichkeiten empirischer Forschungsmethoden und Perspektiven der Fachdidaktik

P-169 Michael Kerres, Nadine Ojstersek Ulrik Schroeder, Ulrich Hoppe (Hrsg.) DeLFI 2010 - 8. Tagung der Fachgruppe E-Learning der Gesellschaft für Informatik e.V.

P-170 Felix C. Freiling (Hrsg.) Sicherheit 2010 Sicherheit, Schutz und Zuverlässigkeit

P-171 Werner Esswein, Klaus Turowski, Martin Juhrisch (Hrsg.) Modellierung betrieblicher Informationssysteme (MobIS 2010) Modellgestütztes Management

P-172 Stefan Klink, Agnes Koschmider Marco Mevius, Andreas Oberweis (Hrsg.) EMISA 2010 Einflussfaktoren auf die Entwicklung flexibler, integrierter Informationssysteme Beiträge des Workshops der GI-Fachgruppe EMISA (Entwicklungsmethoden für Informationssysteme und deren Anwendung)

P-173 Dietmar Schomburg, Andreas Grote (Eds.) German Conference on Bioinformatics 2010

P-174 Arslan Brömme, Torsten Eymann, Detlef Hühnlein, Heiko Roßnagel, Paul Schmücker (Hrsg.) perspeGKtive 2010 Workshop „Innovative und sichere Informationstechnologie für das Gesundheitswesen von morgen“

P-175 Klaus-Peter Fähnrich, Bogdan Franczyk (Hrsg.) INFORMATIK 2010 Service Science – Neue Perspektiven für die Informatik Band 1

P-176 Klaus-Peter Fähnrich, Bogdan Franczyk (Hrsg.) INFORMATIK 2010 Service Science – Neue Perspektiven für die Informatik Band 2

P-177 Witold Abramowicz, Rainer Alt, Klaus-Peter Fähnrich, Bogdan Franczyk, Leszek A. Maciaszek (Eds.) INFORMATIK 2010 Business Process and Service Science – Proceedings of ISSS and BPSC

P-178 Wolfram Pietsch, Benedikt Krams (Hrsg.) Vom Projekt zum Produkt Fachtagung des GI-Fachausschusses Management der Anwendungsentwicklung und -wartung im Fachbereich Wirtschafts-informatik (WI-MAW), Aachen, 2010

P-179 Stefan Gruner, Bernhard Rumpe (Eds.) FM+AM`2010 Second International Workshop on Formal Methods and Agile Methods

P-180 Theo Härder, Wolfgang Lehner, Bernhard Mitschang, Harald Schöning, Holger Schwarz (Hrsg.) Datenbanksysteme für Business, Technologie und Web (BTW) 14. Fachtagung des GI-Fachbereichs „Datenbanken und Informationssysteme“ (DBIS)

P-181 Michael Clasen, Otto Schätzel, Brigitte Theuvsen (Hrsg.) Qualität und Effizienz durch informationsgestützte Landwirtschaft, Fokus: Moderne Weinwirtschaft

P-183 Ralf Reussner, Matthias Grund, Andreas Oberweis, Walter Tichy (Hrsg.) Software Engineering 2011 Fachtagung des GI-Fachbereichs Softwaretechnik

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