JOHANNES GUTENBERG UNIVERSITÄT MAINZ

Mainzer Informatik-Berichte

Rainer Gimnich
Andreas Winter (Hrsg.)

9. Workshop

Software-Reengineering

(WSR 2007)

Mai 2007

Informatik-Bericht Nr. 1 / 2007

Institut für Informatik · Fachbereich 08

Johannes Gutenberg-Universität Mainz
Institut für Informatik – FB 8
D 55099 Mainz
ISSN 0931-9972

9. Workshop Software-Reengineering

der GI-Fachgruppe Software Reengineering (SRE)

02.-04. Mai 2007
Physikzentrum Bad Honnef

Die Workshops Software Reengineering (WSR) im Physikzentrum Bad Honnef wurden mit dem ersten WSR 1999 von Jürgen Ebert und Franz Lehner ins Leben gerufen, um neben den internationalen erfolgreichen Tagungen im Bereich Reengineering auch ein deutschsprachiges Diskussionsforum zu schaffen.

Ziel der Treffen ist es, einander kennen zu lernen und auf diesem Wege eine direkte Basis der Kooperation zu schaffen, so dass das Themengebiet eine weitere Konsolidierung und Weiterentwicklung erfährt.

Durch die aktive und gewachsene Beteiligung vieler Forscher und Praktiker hat sich der WSR als zentrale Reengineering-Konferenz im deutschsprachigen Raum etabliert. Dabei wird er weiterhin als Low-Cost-Workshop ohne eigenes Budget durchgeführt.

Auf Basis der erfolgreichen WSR-Treffen der ersten Jahre wurde 2004 die GI-Fachgruppe Software Reengineering gegründet, die unter http://www.uni-koblenz.de/sre präsent ist.

Die Fachgruppe organisiert inzwischen eine weitere Tagungsreihe RePro (Reengineering-Prozesse) mit dem Schwerpunkt Software-Migration. Außerdem organisiert sie – meist gemeinsam mit anderen GI-Fachgruppen – weitere Workshops mit den Schwerpunkten Reengineering und Services, Reengineering objektorientierter Systeme, Software-Architektur und -Migration und Software-Analyseverfahren.

Der WSR 2007 als zentrale Tagungsreihe bietet eine Vielzahl aktueller Reengineering-Themen, die gleichermaßen wissenschaftlichen wie praktischen Informationsbedarf abdecken sollen. Sie sind zusammengefasst in den Sitzungen:

- Reverse Engineering,
- Migrationsprojekte in der Praxis,
- Statische Analyse,
- Dynamische Analyse,
- Software-Evolution und -Transformation,
- Reengineering und Software-Wartung,
- Architektur-Analyse und -Migration,
- Qualitätssicherung.

Darüber hinaus enthält der WSR 2007 eine Demo-Sitzung im Plenum, in der ausgewählte Tools vorgeführt und diskutiert werden. Weitere Tools und Demo-Möglichkeiten stehen im Rahmen des Workshops zur Verfügung.

Der Fachgruppenleitung war es immer ein besonderes Anliegen, Hochschule und Industrie, d.h. Forschung und Praxis, zusammenzubringen und über den WSR die Möglichkeit des weiteren Austauschs bis hin zu gemeinsamen Projekten zu schaffen.

In dieser Hinsicht ist es erfreulich, dass in 2007 fast die Hälfte der Beiträge aus der Industrie kommt und dass die vorgetragenen Praxis-Themen vermutlich auch stärker für die Forschung interessant werden.

Die Organisatoren danken allen Beitragenden für ihr Engagement: den Vortragenden, Autorinnen und Autoren, „Tool-Demonstranten“. Unser Dank gilt auch den Mitarbeiterinnen und Mitarbeitern des Physikzentrums Bad Honnef, die es wie immer verstanden haben, ein angenehmes und problemloses Umfeld für den Workshop zu schaffen.

Andreas Winter, Universität Mainz
Volker Riediger, Universität Koblenz
Rainer Gimnich, IBM Frankfurt

9. Workshop Software-Reengineering

der GI-Fachgruppe Software Reengineering (SRE)
Bad Honnef
02.-04. Mai 2007

Mittwoch, 2. Mai 2007

10:45 – 11:00	**Begrüßung, Programmüberblick, Hinweise**	
11:00 – 12:30	**Reverse Engineering**	
	Jan Harder (Universität Bremen)	Rückgewinnung von Grammatik und Semantik von Visual Basic 6 zur Durchführung von statischen Programmanalysen
	Markus Knauß, Jochen Wertenauer (Universität Stuttgart)	Minimierung aufwändiger Berechnungen als Grundlage für konsistente und interaktive Programmvisualisierungen
	Roland Kapeller (Kapeller Unternehmensberatung, Köln)	Einsatz einer Basisontologie für das Reverse Software Engineering im Rahmen des Anforderungsmanagements für Produktlinien
12:30 – 14:00	**Mittagspause**	
14:00 – 15:30	**Migrationsprojekte**	
	Werner Teppe (Amadeus Germany)	Konfigurationsmanagement in einem großen industriellen Migrationsprojekt
	Rainer Gimnich (IBM, Frankfurt)	Komponentisierung in der SOA-Migration
	Harry M. Sneed (ANECON, Wien)	Migration in eine Service-orientierte Architektur
15:30 – 16:00	**Kaffeepause**	

16:00 – 18:00 **Code-Analyse und Tool-Demos**

Daniel Vinke, Meik Tessmer, Thorsten Spitta (Universität Bielefeld)	Quelltextanalyse eines grossen, neuen Produktivsystems
Daniel Speicher, Tobias Rho, Günter Kniesel (Universität Bonn)	JTransformer - Eine logikbasierte Infrastruktur zur Codeanalyse

Kurze Vorstellung der einzelnen Tools
Bauhaus (Universität Bremen/Universität Stuttgart/Axivion GmbH)
MigMan, Migration Manager
(pro et con Innovative Informatikanwendungen GmbH)
SRA - Software Reengineering Architektur
(pro et con Innovative Informatikanwendungen GmbH)

18:00 - 18:30 **Mitgliederversammlung Fachgruppe Software Reengineering**
- Genehmigung der Tagesordnung und des Protokolls der letzten Sitzung
- Bericht der Fachgruppenleitung
- Wahl des Leitungsgremiums
- Planung weiterer Veranstaltungen, u.a. WSR 2008 – 10 Jahre WSR

18:30 **Abendessen, anschließend traditioneller Spaziergang**

Donnerstag, 3. Mai 2007

09:00 - 10:30	**Dynamische Analyse**	
	Oliver L. Böhm (agentes)	Software-Archäologie mit AOP
	Martin Burger (Universität Saarbrücken)	JINSI: Isolation fehlerrelevanter Interaktion in Produktivsystemen
	Jochen Quante (Universität Bremen)	Online Construction of Dynamic Object Process Graphs
10:30 - 11:00	**Kaffeepause**	
11:00 - 12:30	**Software-Evolution und -Transformation**	
	Jens Krinke (FernUniversität Hagen)	Changes to Code Clones in Evolving Software
	Sven Wenzel (Universität Siegen)	How to trace model elements?
	Ralf Lämmel, Microsoft, USA	XML Schema Refactorings for X-to-O Mappings
12:30 - 14:00	**Mittagspause**	
14:00 - 15:00	**Reengineering und Software-Wartung**	
	Christof Mosler (RWTH Aachen)	Reengineering of State Machines in Telecommunication Systems
	Stefan Opferkuch (Universität Stuttgart)	Benötigen wir einen „Certified Maintainer“?
15:00 - 18:00	**Gemeinsame Unternehmung**	
18:30	**Conference Dinner**	
20:00	**Fortsetzung/Vertiefung der Tool-Demos** (nach Absprache)	

Freitag, 4. Mai 2007

09:00 - 10:30	**Architektur-Analyse und -Migration**	
	Jens Knodel (IESE, Kaiserslautern)	Three Static Architecture Compliance Checking Approaches – A Comparison
	Johannes Willkomm, Markus Voß (sd&m, Offenbach)	Referenzszenarien der Migration in Anwendungslandschaften
	Uwe Erdmenger, Denis Uhlig (pro et con, Chemnitz)	Konvertierung der Jobsteuerung am Beispiel einer BS2000-Migration
10:30 - 11:00	**Kaffeepause**	
11:00 - 12:30	**Qualitätssicherung**	
	Klaus Krogmann (Universität Karlsruhe)	Reengineering von Software-Komponenten zur Vorhersage von Dienstgüte-Eigenschaften
	Daniel Simon, Frank Simon (SQS, Köln)	Statische Code-Analyse als effiziente Qualitätssicherungsmaßnahme im Umfeld eingebetteter Systeme
	Cathrin Weiß, Rahul Premraj, Thomas Zimmermann, Andreas Zeller (Universität Saarbrücken)	Predicting Effort to Fix Software Bugs
12:30 - 14:00	**Mittagspause und Abreise**	

Der Workshop findet in Kooperation mit der Fachgruppe Software-Produktmanagement (WI-PrdM) der Gesellschaft für Informatik (GI) statt.

Rückgewinnung von Syntax und Semantik zur Analyse von Visual Basic 6 Programmen

Jan Harder
Arbeitsgruppe Softwaretechnik
Fachbereich 03, Universität Bremen
http://www.informatik.uni-bremen.de/st/
harder@informatik.uni-bremen.de

20. April 2007

1 Einführung

Visual Basic ist eine BASIC-Variante, die für die Programmierung von Windows-Applikationen erweitert wurde und in der kommerziellen Softwareentwicklung breite Verwendung findet. Im Rahmen der Anpassung an das .NET-Framework erfolgte eine Neuausrichtung der Sprache mit so tief greifenden Veränderungen an Syntax und Semantik, dass die Abwärtskompatibilität zu den früheren Versionen gebrochen wurde. Die Migration von pre-.NET-Programmen ist nur zum Teil automatisiert möglich und erfordert die manuelle Anpassung von Programmsequenzen, deren Semantik sich geändert hat. Zudem sind die ehemals verwendeten Systembibliotheken durch Aufrufe der .NET-Klassenbibliothek zu ersetzen, die über eine andere API verfügt und damit auch andere Nutzungsprotokolle voraussetzt.

Um die Migration leisten zu können, ist daher eine umfassende Kenntnis der Software und der darin bestehenden Abhängigkeiten notwendig. Bei Legacy-Systemen ist das Wissen über die Implementierung jedoch oft begrenzt, da die Software vor geraumer Zeit geschrieben wurde, Dokumentation fehlt oder nie erstellt wurde. Ebenso können die einstigen Entwickler das Unternehmen mitsamt ihres Wissens verlassen haben. Der entstehende Aufwand für das Programmverstehen, der schon bei den üblichen Aufgaben in der Softwarewartung etwa die Hälfte der Zeit beansprucht [1], ist hier also besonders hoch und unterstützende Analysen und Werkzeuge um so wertvoller.

Das Bauhaus-Projekt der Universitäten Bremen und Stuttgart [5] hat es sich zum Ziel gesetzt Softwareingenieure bei den Aufgaben des Programmverstehens zu unterstützen. Der *Resource Flow Graph* (RFG) ermöglicht es, unabhängig von der Programmiersprache die architektonisch relevanten Elemente von Programmquelltexten selektiv darzustellen und verschiedene Analysen durchzuführen, die Aufschluss über Struktur und Qualitätsaspekte der Software geben. Hierbei nimmt insbesondere die Rückgewinnung der Systemarchitektur aus den Quelltexten eine zentrale Rolle ein. Um die Migration zu den neuen .NET-Versionen zu unterstützen, wurde ein Analysewerkzeug realisiert, das den RFG für Visual-Basic-Programme der Version 6 – der letzten vor der .NET-Umstellung – erstellt.

2 Ausgangssituation

Zur Generierung des RFG müssen typische Aufgaben eines Compiler-Frontends durchgeführt werden, indem zunächst die Quelltexte geparst und dann die darin verwendeten Typen und Bezeichner aufgelöst werden. Zwar ist reichlich Dokumentation zu Visual Basic 6 verfügbar, allerdings fehlen Informationen, die für die Analyse unverzichtbar sind. So gibt es weder eine vollständige und korrekte Grammatik – die einzige Referenz ist der Compiler, dessen Quelltexte nicht einsehbar sind. Noch sind die Regeln, denen die Namensauflösung zugrunde liegt, im Detail beschrieben. Dieses undokumentierte Wissen über die Sprache musste schrittweise in experimentellen Verfahren wiedergewonnen werden, um die Analyse zu ermöglichen. Die dazu eingesetzten Vorgehensweisen werden im folgenden beschrieben.

3 Rückgewinnung der Grammatik

Ralf Lämmel und Chris Verhoef haben eine Vorgehensweise zum *Grammar-Recovery* für einen COBOL-Dialekt beschrieben und angewendet [2]. Hierbei wird aus den verfügbaren Informationen, wie etwa den Handbüchern, eine Ausgangsgrammatik erstellt. Diese wird schrittweise verfeinert, indem ein Parser für die Grammatik erzeugt wird, mit dem Programme der Sprache, deren Korrektheit durch den offiziellen Compiler sichergestellt werden kann, analysiert werden. Dabei auftretende Fehler werden behoben und der Test mit den Beispielprogrammen so lange wiederholt, bis diese fehlerfrei analysiert werden können.

Im Fall von Visual Basic bot die offizielle Referenz [3] einen Ausgangspunkt zum Erstellen der initialen Grammatik. Hier ist die Syntax einzelner Anweisungen beschrieben, jedoch nicht wie sich diese zu einem korrekten VB6-Programm zusammensetzen. Die-

ser fehlende Teil der Grammatik musste zunächst aufgrund von Annahmen und Erfahrungen mit anderen Sprachen gestaltet werden. Einige hilfreiche Informationen lieferten hierbei partielle Grammatiken anderer Autoren.

Als Testdaten diente eine große Menge industriellen Visual Basic 6 Codes mit mehr als 800.000 SLOC. Als besondere Erschwernis erwies sich bei der Verfeinerung der Grammatik die Unvollständigkeit und teils auch Fehlerhaftigkeit der offiziellen Dokumentation, die viele Relikte alter BASIC-Versionen, die noch immer unterstützt werden, verschweigt.

Vieles deutet zudem darauf hin, dass der VB6-Compiler nicht auf einer formalen Grammatikdefinition beruht sondern von Hand geschrieben wurde. Beispielsweise lässt sich kein allgemeines Regelwerk erkennen, nach dem sich entscheidet, ob ein Schlüsselwort reserviert ist oder nicht. Je nach Kontext, in dem ein Bezeichner verwendet wird, unterscheidet sich die Menge der reservierten Schlüsselwörter stark und ist offenbar willkürlich gewählt.

Dies spiegelt sich auch in einer Vielzahl von Mehrdeutigkeiten in der wenig restriktiven Syntax der Sprache wieder, die beim Parsen oft einen großen Lookahead erfordern. Daher wurde zur Implementierung des Parsers mit *ANTLR* [4] ein Generator verwendet, der es erlaubt, in Einzelfällen einen beliebigen Lookahead zu verwenden, der vom fest gewählten abweicht.

4 Nachbildung der Namensauflösung

Während für die Grammatik mit der Sprachreferenz ein hilfreicher Ausgangspunkt bestand, fehlte für die Rekonstruktion von Visual Basics Namensauflösung eine solche Quelle. Stattdessen fanden sich lediglich einige verstreute Hinweise in der Literatur, oft in Form von Anmerkungen, dass bestimmte Bezeichner eindeutig sein müssen. Bedeutende Hintergrundinformationen, etwa wie sich der globale Namensraum bei Namenskonflikten zusammensetzt, fehlten gänzlich.

Hier wurde zunächst ein initiales Modell auf der Grundlage von Beobachtungen, gezielten Tests und Erfahrungen mit anderen Sprachen erstellt. Die Identifikation von Namensräumen war durch die gezielte Generierung von Testfällen, die gleichnamige Bezeichner in unterschiedlichen Kombinationen gegenüberstellten, möglich. Testfälle, die vom VB6-Compiler abgewiesen wurden, beinhalteten Namensüberschneidungen, die die Sprache nicht erlaubt. Durch diese und weitere manuell erstellte Tests sowie dem Compiler als Validierungswerkzeug konnte so das initiale Modell konstruiert werden.

Dieses Modell wurde, analog zur Ausgangsgrammatik, anhand der Beispielprogramme getestet. Auftretende Fehler wurden schrittweise behoben. Hierbei ist generell zwischen zwei Arten von Fehlern zu unterscheiden: Namen, die nicht zugeordnet wurden, und solche, die dem falschen Symbol zugeordnet wurden. Während erstere Laufzeitfehler verursachen und daher leicht festzustellen sind, fallen falsche Zuordnungen nur dann auf, wenn sie zu Folgefehlern in der Namensauflösung führen oder bei manuellen Stichproben entdeckt werden.

5 Ergebnisse

Die Realisierung des Analysewerkzeugs fand in einem Zeitraum von etwa drei Monaten statt. Zwar sind die angewandten Methoden nicht exakt, ebenso hängen die Ergebnisse stark von der Güte der herangezogenen Beispielprogramme ab. Weitergehende Tests mit anderen VB6-Programmen zeigen jedoch, dass das Analysewerkzeug mit nur geringen Modifikationen auch hierfür erfolgreich eingesetzt werden kann.

Die erzeugten RFGs liefern Informationen, die aus den Quelltexten nicht direkt ersichtlich, für die Migration zu .NET jedoch bedeutsam sind. So lassen sich etwa potentielle, globale Auswirkungen lokaler Änderungen, wie sie bei der Anpassung der Semantik nötig sind, abschätzen, indem die Kontroll- und Datenabhängigkeiten der geänderten Funktionen untersucht werden. Die Architektur-Rückgewinnung ermöglicht es zudem geeignete architektonische Komponenten für eine inkrementelle Migration zu identifizieren. Dies ist insbesondere bei größeren Legacy-Systemen hilfreich, bei denen eine vollständige Migration in nur einem Schritt nicht möglich ist.

Zukünftig ist geplant den RFG auch für .NET-Programme zu generieren. Hiermit wird es möglich sein die vorhandenen Bauhaus-Werkzeuge zu nutzen, um die Einhaltung der Architektur während der Migration durch Refexionsanalysen zu überwachen.

Literatur

[1] R. K. Fjeldstadt and W.T. Hamlen. Application program maintenance study: Report to our respondents. In *Proceedings of GUIDE 48*, 1983.

[2] Ralf Lämmel and Chris Verhoef. Semi-automatic Grammar Recovery. *Software - Practice and Experience*, Vol. 31(15):1395–1438, Dezember 2001.

[3] Microsoft Developer Network Library - Visual Basic 6. Webseite, April 2007. `http://msdn2.microsoft.com/en-us/library/ms950408.aspx`.

[4] Terence Parr and R. W. Quong. Antlr: A Predicated-LL(k) Parser Generator. *Software - Practice and Experience*, Vol. 25(7), Juli 1995.

[5] Aoun Raza, Gunther Vogel, and Erhard Plödereder. Bauhaus – a tool suite for program analysis and reverse engeneering. In *Reliable Software Technologies – Ada-Europe 2006*, pages 71–82, Juni 2006.

Minimierung aufwändiger Berechnungen als Grundlage für konsistente und interaktive Programmvisualisierungen

Markus Knauß
Abteilung Software Engineering
Institut für Softwaretechnologie
Universität Stuttgart
www.informatik.uni-stuttgart.de/iste/se

Jochen Wertenauer
mail@jwertenauer.de

Einleitung

Um ein Programm zu bearbeiten, muss es zunächst verstanden werden. Das Verstehen wird unterstützt durch berechnete Zusatzinformation, wie zum Beispiel Programmstruktur oder Aufrufbeziehungen. Die Zusatzinformation kann zu dem Programm visualisiert werden. Die Visualisierung ist besonders nützlich, wenn sie konsistent mit dem bearbeiteten Programm und interaktiv bedienbar ist. Die Berechnung der Zusatzinformation ist aber aufwändig, was die Realisierung einer konsistenten und interaktiv bedienbaren Visualisierung schwer macht.

Eine Grundlage für eine konsistente und interaktiv bedienbare Visualisierung kann die Minimierung der Berechnung der Zusatzinformation sein. Die Minimierung wird erreicht, indem nicht mehr konsistente Zusatzinformation identifiziert und nur diese neu berechnet wird. Diese Idee wurde in einer Arbeit an der Universität Stuttgart untersucht (Wertenauer, 2007). Resultat ist das Werkzeug codation, mit dem nicht mehr konsistente Zusatzinformation identifiziert wird. Dieser Artikel berichtet über die Arbeit und die erreichten Ergebnisse.

Im nächsten Abschnitt beschreiben wir den Lösungsansatz. Anschließend stellen wir die technische Realisierung als Eclipse-Plugin vor. Danach berichten wir die Ergebnisse einer kleinen Fallstudie, in der das Werkzeug evaluiert wurde. Abschließend fassen wir zusammen und geben einen Ausblick auf mögliche weitere Arbeiten.

Lösungsansatz

Der Lösungsansatz beruht auf der Idee, Änderungen am Programmcode möglichst genau zu erkennen und nur die Zusatzinformation neu zu berechnen, die von den Änderungen betroffen ist. Um diese Idee zu realisieren, sind zwei Fragen zu beantworten:

1. Wie können Änderungen erkannt werden?
2. Wie kann die von einer Änderung betroffene Zusatzinformation erkannt werden?

Um die beiden Fragen zu beantworten und die Vorgaben einzuhalten, Änderungen möglichst genau zu erkennen und nur die Information neu zu berechnen, die tatsächlich von der Änderung betroffen ist, muss zunächst die Frage beantwortet werden:

3. Auf welche Teile des Programms bezieht sich die Information?

Im Rahmen der Arbeit an der Universität Stuttgart wurden die Fragen speziell für Java-Programme beantwortet. Darum beziehen sich die weiteren Ausführungen auf Java-Programme.

Die Berechnung der Zusatzinformation für Java-Programme beruht meist auf dem AST. Darum bezieht sich die Zusatzinformation für Java-Programme auf einzelne Knoten des ASTs.

Um Änderungen zu erkennen, wurde das Programm vor der Änderung mit dem nach der Änderung verglichen. Da die Bezüge zwischen Programm und Zusatzinformation über die AST-Knoten realisiert sind, müssen die beiden ASTs vor und nach der Änderung verglichen werden. Der Vergleich verwendet den Algorithmus für die Berechnung der Tree-Edit-Distance. Der Algorithmus berechnet die notwendigen Änderungen, die vom AST vor der Bearbeitung zum AST nach der Bearbeitung führen, in $O(n_1 n_2)$ (n_1, n_2: Knoten des AST vor und nach der Änderung) (Bille, 2005).

Da der Vergleich der vollständigen ASTs des Programms vor und nach der Änderung zu aufwändig ist, wurde der Algorithmus für Java-Programme angepasst. Der Algorithmus vergleicht nur noch ASTs von Methoden, lokalen Typen oder Variablen, die tatsächlich verändert wurden.

Ergebnis der Änderungsberechnung sind die geänderten AST-Knoten. Da sich die Zusatzinformation auf die AST-Knoten bezieht, kann schnell festgestellt werden, welche Zusatzinformation von Änderungen betroffen ist.

Technische Realisierung

Codation wurde als Eclipse-Plugin realisiert. Es ist in Eclipse als Project-Builder registriert. Das hat die folgenden Vorteile: codation ist unabhängig von der

Programmiersprache, wird bei jedem Speichern über die geänderten Dateien informiert und kann Ressourcen im Workspace ändern.

Codation implementiert ein Rahmenwerk für die Änderungserkennung. Das Rahmenwerk besteht aus den drei Schnittstellen: Annotation-Provider, Storage-Provider und File-Link-Provider. Mit dem Annotation-Provider werden Zusatzinformationen in codation eingebracht, der Storage-Provider kümmert sich um die Speicherung der Zusatzinformationen und der File-Link-Provider verbindet die Zusatzinformationen mit dem Programmcode. Der File-Link-Provider ist auch zuständig für die Identifikation der nicht mehr konsistenten Zusatzinformationen. Der Storage-Provider verwaltet die Zusatzinformationen und die Bezüge zwischen den Zusatzinformationen und dem Programmcode in eigenen Dateien. Hierdurch bleibt der Programmcode unverändert, was die Nutzung existierender Werkzeuge ermöglicht. Soll codation in einem eigenen Projekt genutzt werden, dann müssen diese drei Schnittstellen für die eigene Anwendung implementiert werden.

Der Ablauf der Änderungsberechnung in codation ist wie folgt:

1. Mit den Annotation-Providern werden die von der Änderung betroffenen Zusatzinformationen bestimmt.
2. Von der Änderung betroffene Zusatzinformation wird geprüft, ob eine Aktualisierung notwendig ist.
3. Annotation-Provider nicht mehr konsistenter Zusatzinformationen werden benachrichtigt.

Die Berechnung der Änderungsinformation wird im Hintergrund ausgeführt, um den Entwickler bei seiner Arbeit nicht zu behindern.

Fallstudie

Die Nützlichkeit von codation wurde in einer kleinen Fallstudie untersucht. Für die Fallstudie wurde eine Anwendung entwickelt, mit der Use-Cases mit dem Programmcode verknüpft werden können. So ist es möglich Use-Cases zu erkennen, deren Realisierung nach einer Änderung des Programms, zum Beispiel durch Testfälle, geprüft werden müssen. Es zeigte sich, dass codation gut geeignet ist für die Realisierung der Anwendung und die Änderungsberechnung sehr genau anzeigt, welche Use-Cases geprüft werden müssen. Allerdings ist festgestellt worden, dass die im Hintergrund laufende Änderungserkennung sehr aufwändig ist und die Arbeit behindern kann.

Um die Auswirkungen von Änderungen auf das Laufzeitverhalten von codation näher zu untersuchen, wurden Performanzmessungen durchgeführt. Es wurde festgestellt, dass die Laufzeit von der Größe der geänderten Methoden abhängt. Die Größe einer Methode spiegelt sich in einem größeren AST wieder, was zu einem höheren Aufwand in der Änderungsberechnung führt. Die Anzahl der Methoden in einer Klasse oder die Art der Änderung spielt hingegen keine Rolle.

Zusammenfassung und Ausblick

Dieser Artikel hat einen Ansatz vorgestellt, mit dem die aufwändige Berechnungen von Zusatzinformation zu einem Programm auf ein Minimum reduziert werden kann. Die Minimierung ist notwendig, um konsistente und interaktiv bedienbare Visualisierungen der Zusatzinformation zu ermöglichen. Durch die konsistente und interaktiv bedienbare Visualisierung wird das Verstehen und Bearbeiten des Programms, zwei wichtige Arbeiten in der Software-Wartung, unterstützt. Der Ansatz wurde im Werkzeug codation als Eclipse-Plugin realisiert. Für die Änderungserkennung in Java-Programmen auf Basis des AST wurde der Algorithmus für die Berechnung der Tree-Edit-Distance verwendet und angepasst. Die Nützlichkeit und Effizienz des Werkzeugs wurde in einer kleinen Fallstudie untersucht.

In zukünftigen Arbeiten kann die Berechnung der Änderungsinformation noch weiter optimiert werden, zum Beispiel indem Heuristiken eingesetzt werden, um die Anzahl der Knoten im untersuchten AST zu reduzieren. Ein anderer Ansatzpunkt für Optimierungen ist, zu berechnen, ab welcher AST-Größe es sinnvoller ist, eine Methode vollständig als geändert zu markieren, als detaillierte Änderungsinformationen zu berechnen.

codation kann von der Website: www.jwertenauer.de/ger/uni/da/index.shtml (3.4.2007) heruntergeladen werden. Das Werkzeug steht unter der Eclipse Public License (EPL) und kann in eigenen Projekten eingesetzt werden.

Literatur

Bille, Philip (2005): A Survey on Tree Edit Distance and Related Problems. Theoretical Computer Science, 337(1-3), 217-239.

Wertenauer, Jochen (2007): codation – Verbindung von Code mit Zusatzinformation. Diplomarbeit Nr. 2521, Universität Stuttgart.

Einsatz einer Basisontologie für das Reverse Software Engineering im Rahmen des Anforderungsmanagements für Produktlinien

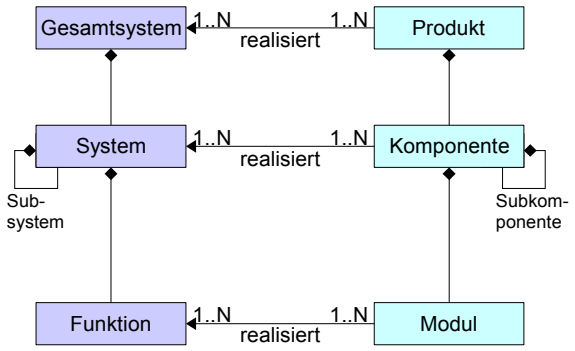
Roland Kapeller
Roland Kapeller Unternehmensberatung, Köln
r.kapeller@kapeller-ub.de

Zusammenfassung
Die steigende Komplexität elektronischer Steuergeräte in Kraftfahrzeugen erzwingt ein leistungsfähiges Anforderungsmanagement. Die Wiederverwendung von Software, die bereits Teil marktgängiger Produkte ist, kann durch nachträgliche Modellierung auf Grundlage einer Basisontologie effizient gestaltet werden.

Zielsetzung
Der Rückgriff auf bereits entwickelte Produkte läßt Designentscheidungen (einschließlich der Parametrisierung von Funktionen) als Anforderungen neben den Kundenanforderungen in die einzelnen Kundenprojekte miteinfließen. Diese „Systemarchäologie" sollte stets mit den Gestaltungsgrundsätzen der Systemspezifikation übereinstimmen. Gefordert wird daher ein *durchgängig* im System-Entwicklungsprozeß anwendbarer und *universeller* Ansatz der Klassifikation von Aussagen über (vorhandene) Produkte und (zu realisierende) Zielsysteme.

Ansatz
Es zeigt sich, daß eine **Basisontologie**, in deren Zentrum ein übersichtliches **Metamodell** steht, den Workflow bei mehreren wichtigen Aktivitäten unterstützen kann:
- Definition einer **Produktdefinition** als Basis der projekt- bzw. auftragsspezifischen Systemspezifikationen
- **Analyse des Lastenhefts** einschließlich des Vergleichs mit der Basisspezifikation
- **Reverse Engineering** vorhandener HW und SW zur Wiedergewinnung von Basisanforderungen (einschließlich der Anforderungen der Entwicklung)

Basisontologie
Bei Analyse und Design von Systemen wird zumeist (etwa in der MDA) zwischen *zwei* Beschreibungsebenen (bzw. Sichten) unterschieden. Im Requirements Management ist die Verwendung weiterer Sichten sinnvoll [1]. Das Minimum scheinen jedoch *drei* zu sein: Die **Verhaltensbeschreibung** beinhaltet *rein funktionale* Anforderungen. Alle Festlegungen hinsichtlich des Systemdesigns gehören hingegen zur **Strukturbeschreibung**. Wenn beide Sichten hierarchisch gegliedert sind, können die Beschreibungs-Elemente (*Klassen*) der verschiedenen Schichten einander zugeordnet werden (siehe folgende Abbildung).



Als dritte Sicht beinhaltet die **Schnittstellenbeschreibung** die in Verhalten und Struktur verwendeten Größen aller Art. Für die *Instanzen* der Elemente aller Sichten ist festgelegt, was zu ihrer Definition neben einem eindeutigen Namen angegeben werden muß. Die Kernelemente der Verhaltenssicht selbst sind dabei wie folgt definiert:
- Ein **System** besteht aus *Subsystemen* oder *Funktionen*. und entspricht (genau) einer Funktionalität.
- Ein **Subsystem** ist ein *System*, das einer Teilfunktionalität (ihres übergeordneten Systems) entspricht.
- Ein **Gesamtsystem** ist ein *System*, das kein *Subsystem* ist.
- Eine **Funktion** ist für das Erfülltsein von *Bedingungen* innerhalb einer Funktionalität verantwortlich. Dies bezieht sich auf die Systemgrenzen; Funktionen innerhalb eines Systems sollten daher nicht durch Größen *verkettet* werden.

Um diesen Kern gruppieren sich weitere Basiselemente:
- **Bedingung** (beinhaltet Größen)
- **Ereignis** (beinhaltet *Bedingung* mit Zeitpunktbezug)
- **Größe** (Informationen, Signale, Daten, mechanische oder überhaupt physikalische Wirkungen)
- **Zustand** (eines *Systems*, einer *Funktion*; Zustandsübergänge sind abhängig von *Bedingungen* und *Zuständen*)
- **Fehlerfall** (als benannter *Bedingung* innerhalb eines Systems)

Zu beschreiben sind damit (unter anderem):
- **Schnittstellen** (von *Systemen*, *Funktionen*, *Komponenten* und *Modulen*)
- **Systemsteuerung** (Zustandsmodell von *Systemen*, Vorbedingungen von *Funktionen*)

- **Wirkungsweise** (von *Funktionen*, insbesondere Betriebs- und Nachbedingungen)
- **Realisierungen** (von *Produkten*, *Komponenten* und *Modulen*)

Bei der vorgestellten Basisstruktur handelt es sich um ein *Beispiel*, das im folgenden noch *domänenspezifisch* verfeinert wird. Ihre praktische Anwendung kann auch die Klassifikation von Aussagen innerhalb der Spezifikation und die Formulierung von Systemanforderungen umfassen [2].

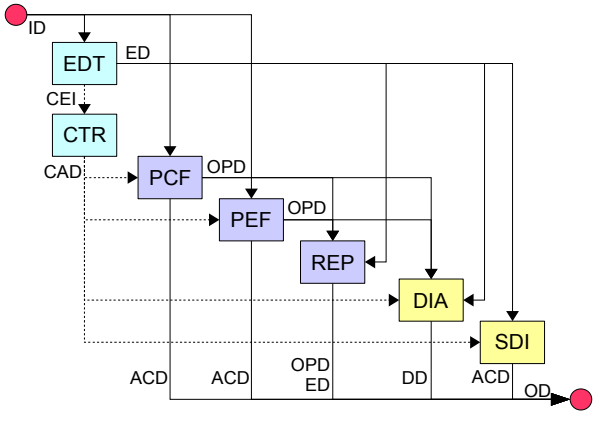
Die *Elemente der Struktursicht* sind über die in der Graphik angegebenen Beziehungen definiert. **Module** sind elektronische (HW) oder elektromechanische Module oder SW-Module (als **Pakete** aufzufassen).

System-Metamodell

Der funktionale Aufbau elektronischer Steuergeräte läßt sich verallgemeinern und führt zu einer „Zwiebelstruktur" aus Systemen und Funktionen (sowie deren Gruppen):

- **Sensorik** (setzt an der Systemgrenze beispielsweise physikalische in elektrische Größen um)
- **Steuerung**
  - *Eingangssignaltransformation* (setzt Signale in Daten um, etwa als A/D-Converter)
  - *Verarbeitung / Ablaufkontrolle*
  - *Ausgangssignaltransformation* (setzt Daten in Signale um, etwa als D/A-Converter)
- **Aktorik** (setzt bspw. elektrische in mechanische Größen an der Systemgrenze um)

Die **Verarbeitung / Ablaufkontrolle** besteht als Kern des eingebetteten Systems naturgemäß aus Datenverarbeitung (SW), die nach folgendem Modell folgt strukturiert werden kann (siehe folgende Abbildung).



Die dargestellten Elemente stellen *Funktionsgruppen* dar, deren Funktionen einen der folgenden Stereotypen besitzen:

- **CTR** = System Control
- **EDT** = Error Detection
- **PCF** = Processing Core Function
- **PEF** = Processing Error Function
- **REP** = Reporting (Ausgabe der Fehler und Diagnoseergebnisse)
- **DIA** = Diagnosis (von externem System initiiert)
- **SDI** = Self Diagnosis (steuert die Aktorik)

Außerhalb der *Verarbeitung / Ablaufkontrolle* bestehen weitere Funktionen mit eigenen Stereotypen, etwa für die Initialisierung und das Remote Control Interface.

Flüsse sind ebenfalls typisiert. Auf dieser Ebene:

- **ID** = Input Data
- **CEI** = Control Error Identification
- **CAD** = Control Activation & Deactivation
- **ED** = Error Data
- **ACD** = Actuator Control Data (Ansteuerung)
- **OPD** = Operation Data (Betriebsdaten)
- **DD** = Diagnosis Data
- **OD** = Output Data (::= ACD+OPD+ED+DD)

Anwendung im SW-Reengineering

Es wird empfohlen, *vor* Erstellung der *Strukturbeschreibung* die *Funktionalität* der SW zu untersuchen. Arbeitsschritte:

1. Synthese zu *Funktionen* (in der Beschreibungsform von Bedingungen) und Zuweisung des jeweils passenden Stereotypen, Bildung von *Funktionsgruppen*
2. Beschreibung der Schnittstellen ohne Parameter
3. Zusammenfassung der Funktionen zu Systemen, welche diese Funktionen steuern, Parametrisierung

Nützlich ist dabei ein CASE-Tool (heute eher SA/SD als UML, später SysML). Jedoch sollte dann ein Anforderungsmanagementtool die Anforderungen verwalten.

Fazit

Wie die Praxis bei einem namhaften Automobilzulieferer zeigt, bringt die Methode neben der **Beschleunigung der Angebotserstellung** und der **Erhöhung der Wiederverwendungsgrades** großen Nutzen durch die **Förderung einheitlicher Denk- und Kommunikationsmuster** bei den Mitarbeitern. Zudem werden vielfach **produktbezogene Verbesserungsmöglichkeiten** entdeckt.

Literaturhinweise

[1] Geisberger, Wußmann: Requirements Engineering eingebetteter Systeme (Softwaretechnik-Trends, 23/1, S. 6)

[2] Kapeller: Erstellung vollständiger Systemspezifikationen im Embedded Computing. (Softwaretechnik-Trends, 26/1, S. 15)

Komponentisierung in der SOA-Migration

Rainer Gimnich
IBM Software Group
SOA Advanced Technologies
Wilhelm-Fay-Str. 30-34, D-65936 Frankfurt
gimnich@de.ibm.com

Zusammenfassung

Service-orientierte Architektur (SOA) ist eine geschäftlich motivierte IT-Architektur, die die Integration von Geschäftsfunktionen als verbundene, wiederverwendbare Dienste (Services) unterstützt. SOA-Migration bezeichnet die Umsetzung existierender Architekturen – die sehr unterschiedlich geartet sein können – in eine SOA. Hierbei liegt der Schwerpunkt meist auf der Anwendungsarchitektur. Bei der SOA-Migration bieten Komponenten und Komposition auf unterschiedlichen Ebenen wichtige Hilfestellungen. Diese Leistungen werden hier untersucht und anhand von Projektbeispielen beschrieben.

1. SOA-Migration

Service-Orientierung bei Unternehmensarchitekturen ermöglicht eine Reihe von Vorteilen, u.a. höhere Flexibilität, kürzere Produktzyklen, stärkere Verzahnung von Fachbereich und IT, vorhandene Standards, leichtere Realisierung von Architekturprinzipien wie z.B. Wiederverwendung und lose Kopplung. Die Migration existierender Architekturen in SOA kann und sollte evolutionär erfolgen, d.h. in einzelnen Schritten mit geschäftlich motivierter Priorisierung [Gimnich/Winter 2005, Gimnich 2006b], z.B. beginnend mit der Partneranbindung, aufbauend auf Portallösungen. Migrationen erfolgen generell nach sorgfältiger Analyse der Gegebenheiten [Sneed et al. 2005] und einem realistischen Umstellungsplan. Eine 'SOA Roadmap' berücksichtigt dabei bereits geplante und laufende Projekte und stellt die frühzeitige Nutzung von SOA-Governance-Richtlinien/-Prozessen sicher [Gimnich 2006a].

2. Schichtenmodell und Vorgehen

Für die SOA-Realisierung hat sich ein Schichtenmodell bewährt, dass eine Zentrierung auf (Business) Services aus Nutzer- und Anbietersicht darstellt und dabei folgende Ebenen realisiert (s. auch Abb. 1):

- Nutzungsebene: Personen, Systeme (z.B. Portale)
- Geschäftsprozess-Ebene
- Service-Ebene (zentral)
- Service-Komponenten-Ebene
- Ebene der (existierenden) operationalen Systeme

Darüber hinaus gibt es Querschnittsfunktionen wie Governance, Metadaten-Management, unternehmensweite Integration (Enterprise Service Bus, ESB) u.a. Der Entwurf der SOA besteht primär in der Definition und Spezifikation der Elemente (Komponenten) auf diesen Ebenen und der Beziehungen der Elemente, sowohl innerhalb einer Ebene als auch übergreifend.

Für das Vorgehen bei SOA-Analyse und -Design hat sich ein kombiniertes Verfahren (Top-down / Bottom-up / Outside-in) bewährt, wie es z.B. in [Gimnich 2006b] an Fallstudien gezeigt wird. Hier geht es darum, genau die geschäftlich angemessenen Services und ihre Zusammenhänge zu ermitteln, zu spezifizieren und Realisierungsentscheidungen zu treffen. Eine 'Inflation' von Services ist zu vermeiden; für jeden Servicekandidaten ist seine Exponierung (zum Kunden, zu Partnern oder unternehmensintern) genau zu prüfen, zu entscheiden und einzuhalten (einschließlich Qualität/Leistung und Wartung). Die laut Analysten mächtigste Methode zur Service-Modellierung ist SOMA, eingeführt in [Arsanjani 2004], inzwischen publiziert und als Version 2.4 [SOMA 2006] allgemein verfügbar.

3. Komponentisierung

Komponentisierung ist Teil der SOA-Entwurfsmethode und hilft, die Komplexität der Erstellung einer angemessenen, geschäftsorientierten Zielarchitektur zu meistern. Die auf den einzelnen Ebenen verwendeten Komponentenkonzepte und -modelle unterscheiden sich dabei meist erheblich [Gimnich 2007].

Darüber hinaus dient die Komponentisierung dazu, auf jeder Ebene die Migration von ggf. existierenden und verwendbaren Elementen in die SO-Zielarchitektur zu steuern und durchzuführen.

3.1 Komponentisierung auf Geschäftsprozessebene

Hier werden – komplementär zu den Geschäftsprozessen – geschäftliche Komponenten (Business Components) gebildet. Diese stellen eigenständige, nicht-überlappende Gruppierungen von Geschäftsfunktionen dar, bei Banken z.B. Kontenverwaltung oder Auditing/Compliance. Bei der Beschreibung von Geschäftskomponenten lassen sich Business Services auf hoher Abstraktion ermitteln und später – in der Service-Modellierung – aus Ausgangspunkt nutzen. Ein komplexer Geschäftsprozess, wie z.B. Kreditantragsbearbeitung, wird hier als 'Kollaboration' zwischen den relevanten Geschäftskomponenten beschrieben und damit 'SOA-nah' dokumentiert.
Für die Migration lassen sich hierbei im sog. Business Operating Model (BOM) die existierenden IT-Assets (z.B. Anwendungen, Standardpakete, Unternehmensdatenbanken) sowie die eventuell etablierte IT-Governance auf hoher Ebene den Geschäftskomponenten zuordnen.

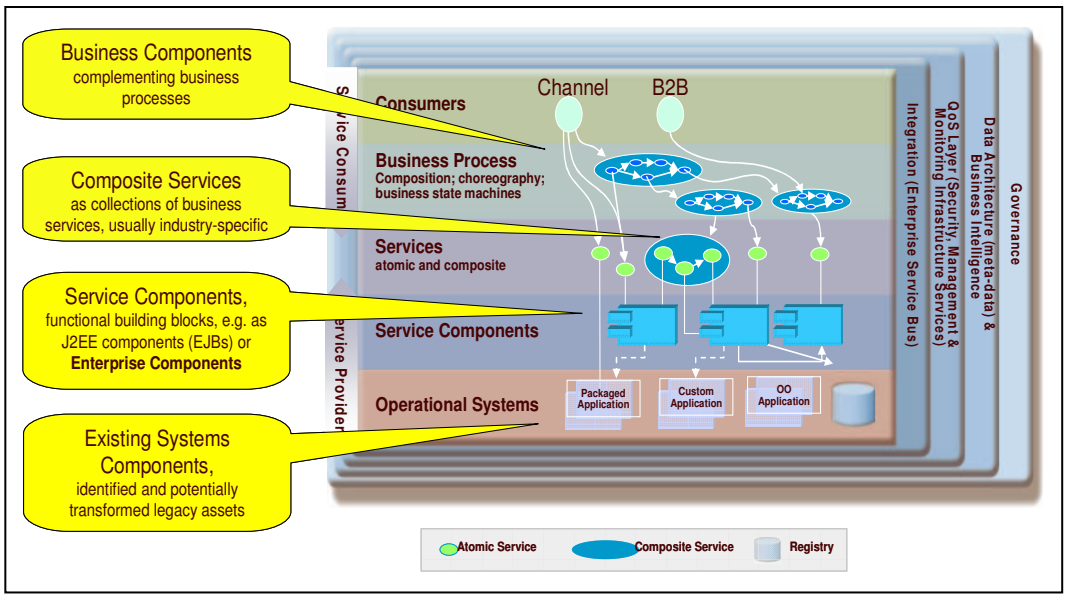


Abb. 1: Komponentenbildung pro Ebene

3.2 Komponentisierung auf Service-Ebene

Hier geht es um den Zusammenbau von Services zu komplexeren Strukturen (Composite Services), die eine größere Funktionalität realisieren und oft branchenspezifisch angelegt sind. Für die Migration sind existierende Services und ggf. Composite Services relevant, die in die SOA übernommen werden können.

3.3 Komponentisierung in der Service-Realisierung

Auf dieser Ebene liegen die 'klassischen' Komponentenmodelle wie die von J2EE, CORBA oder anderen Programmiermodellen vor. Für die SOA-Realisierung ist insbesondere eine Aggregation von Service-Komponenten interessant, wie sie z.B. durch das **Enterprise Component** Pattern erreicht werden kann [Arsanjani 2004]. Solche 'Unternehmenskomponenten' stellen dann meist die funktionale Grundlage für mehrere Services auf der Ebene darüber bereit.
Aus Migrationssicht sind alle Service-Komponenten wichtige Assets, die als Kandidaten für die Implementierung im Service-Modell registriert werden sollten.

3.4 Komponentisierung von operationalen Systemen (Legacy)

Auf dieser Ebene liegen in der Regel die meisten Assets vor, die für eine SOA-Migration in Betracht kommen. Oft handelt es sich um ‚klassische' Eigenentwicklung von Anwendungen, z.B. in COBOL oder C++, oder kommerzielle Anwendungspakete (‚Standardsoftware'). Auch wenn Komponenten und Beschreibungen auf den Ebenen darüber oft fehlen, Legacy-Sourcen oder für das Unternehmen parametrierte ‚Packages' liegen meist vor. Das Ziel der weiteren Verwendung in einer SOA ist nachvollziehbar, aus Qualitäts-, Stabilitäts-, Aufwands- und Kostengründen.

Methoden wie SOMA sehen die Einbindung existierender Systemen auf dieser Ebene in mehrfacher Hinsicht:

- zur Vollständigkeit des Service-Portfolios: bereits (konventionell) realisierte Business Services schließen oft Lücken in der Service-Kandidaten-Liste;
- zum Treffen von Entscheidungen über die Realisierung von Services: z.B. Integration und ggf. Transformation existierender Assets, gegenüber Einkauf oder Neuimplementierung der Funktionalität.

4. Bewertung und Ausblick

Der Nutzen der Komponentisierung wird anhand aktueller SOA-Projekte diskutiert. Abschließend wird die Bedeutung der gerade in Version 1 verabschiedeten Standardisierung einer Service Component Architecture [SCA 2007] eingeschätzt.

Literatur

[Arsanjani 2004] A. Arsanjani: Service-Oriented Modeling and Architecture. http://www.ibm.com/developerworks/webservices/library/ws-soa-design1/ , November 2004.

[Gimnich 2006a] R. Gimnich: Quality Management Aspects in SOA Building and Operating. SQM 2006 http://www.sqs-conferences.com/de/2006/programme_06.htm, May 2006.

[Gimnich 2006b] R. Gimnich: SOA Migration – Approaches and Experience. RePro 2006. GI Softwaretechnik-Trends Vol. 27 No. 1, 2006. http://pi.informatik.uni-siegen.de/stt/27_1/

[Gimnich 2007] R. Gimnich: Component Models in SOA Realization. SQM2007, http://www.sqs-conferences.com/, April 2007.

[Gimnich/Winter 2005] R. Gimnich, A. Winter: Workflows der Software-Migration. WSR 2005, Softwaretechnik-Trends 25:2, May 2005. Presentation: http://www.uni-koblenz.de/sre/Conferences/WSR/Wsr2005/Programm/gimnichwinter.pdf

[SCA 2007] Open SOA: Service Component Architecture Specifications. Final Version V1.0, March 21, 2007. http://www.osoa.org/display/Main/Service+Component+Architecture+Specifications

[Sneed et al. 2005] H. Sneed, M. Hasitschka, M.-T. Teichmann: Software-Produktmanagement. dpunkt, Heidelberg, 2005.

[SOMA 2006] IBM: RUP for Service-Oriented Modeling and Architecture V2.4 (Rational Method Composer plug-in) http://www.ibm.com/developerworks/rational/downloads/06/rmc_soma/, November 2006.

Migration in eine Service-orientierte Architektur

von
Harry M. Sneed
ANECON GmbH, Vienna
harry.sneed@anecon.com

1. Zur Bereitstellung von Web-Services

Eine Voraussetzung für die Einführung eines Service orientierten Architektur (SOA) ist die Bereitstellung der erforderlichen Web-Services. Denn im Gegensatz zu den Kindern werden sie nicht vom Storch gebracht. Der Anwender muss sie besorgen. Dies kann auf fünferlei Weise geschehen. Sie können entweder gekauft, gemietet, geliehen, gebaut oder aus vorhandenen Komponenten wieder gewonnen werden. Um diese alternative Ansätze Web-Services bereitzustellen geht es in diesem Beitrag.

In dem ersten Abschnitt werden die fünf Möglichkeiten für die Erlangung eines Webservices vorgestellt. Sie sind wie folgt:

- Einen Service zu kaufen
- Einen Service zu mieten
- Einen Service zu leihen
- Einen Service neu zu erstellen
- Einen Service aus vorhanden Komponenten zu erstellen

1.1. Web-Services einkaufen

Web-Services können von einem Software-Hersteller eingekauft werden. Die meisten klassischen ERP-Software-Hersteller bieten ihre Komponenten auch als Web-Services an, und jeder IT-Anwender kann sie kaufen. Dies gilt nicht nur für kommerzielle Software für den betrieblichen Einsatz, sondern auch für Konstruktions- und Wissenschaftssoftware, welche ebenso von der Stange gekauft werden kann. Die Vorteile solcher fertigen Stangenwaren-Web-Services ist:

sie sind wirklich verfügbar,
sie sind gut getestet und relativ verlässlich
sie werden mit Support angeboten [1]

Der zweite und der dritte Vorteil sollten nicht unterschätzt werden. Es bedarf eines signifikanten Aufwandes auch nur einen einfachen Web-Service in all seinen Möglichkeiten der Bedienung zu testen. Es ist auch bequem zu wissen, dass der Web-Service auf einer regelmäßigen Basis gewartet bzw. aktualisiert wird, und dass man sich als Kunde nicht darum kümmern muss.

Die Nachteile eines Web-Services von der Stange sind:

- Er ist in der Regel sehr teuer
- Er ist beschränkt in seinem Funktionsumfang
- Er ist für den Benutzer nicht veränderbar
- Er ist nicht selten zu groß und zu starr

Der größte Nachteil ist die fehlende Flexibilität. Die Verwendung von großen und starren Web-Services, wie in etwa einem kompletten Abrechnungssystem oder ein Kreditkartenvalidierungspacket ist wie das Bauen mit fertigen Betonwänden. Der Benutzer muss seine Geschäftsprozesse um die Web-Services herum bauen. Somit bestimmt der Web-Service den Prozess. Man kauft also nicht nur die Software, sondern auch den zugrunde liegenden Geschäftsprozess.

Für manche IT Anwender mag dies ein Segen sein. Die müssen keine Zeit und keine Aufwände investieren, um einen Geschäftsprozess zu definieren, aber die verlieren auch den Hauptvorteil eines Web-Services, die Flexibilität. Sie könnten auch den gesamten Prozess kaufen, wie sie es in der Vergangenheit taten. Es macht dann keinen Sinn, zu einer service-orientierten Architektur zu wechseln. Für andere Benutzer, die darauf aus sind, den Prozess ihren Bedürfnissen anzupassen ist dies eine nicht zu tolerierende

Beschränkung. Was wird aus dem Wettbewerb, wenn jeder Wettbewerber denselben Geschäftsprozess benutzt?

1.2. Web-Services mieten

Eine Alternative zum Kauf eines Web-Services ist es, ihn zu mieten. Viele Produzenten von Standardsoftware wie SAP und Oracle sind nun dabei, Pläne auszuarbeiten, um ihre Services auf einer Vermietungsbasis anzubieten. Anstelle die Software-Packete zu kaufen und sie innerhalb seiner Umgebung zu installieren, hat der IT-Anwender nun die Option, nur die Funktionalität die er benötigt zu verwenden und dies nur wann er sie benötigt, also auf Anfrage – Software on Demand. Er bezahlt dann nur für die tatsächliche Nutzung. Dieses Geschäftsmodell hat viele Vorteile gegenüber einem Kauf:

- Zunächst ist der Benutzer nicht gezwungen, die Software zu installieren und ständig zu aktualisieren
- Zweitens arbeitet er stets mit der aktuellen Version
- Drittens zahlt er nur für dass, was er wirklich nutzt

Software zu besitzen ist nicht immer vorteilhaft für den Anwender. Er muss die Kosten [Total Cost of Ownership] tragen. Nicht nur das er die Software in seiner Umgebung installieren und testen muss, er muss dies bei jedem neuen Release tun. Die Wartung der Software in seiner Umgebung ist eben teuer, aber der Vorteil ist, dass er sie an seine persönlichen Bedürfnisse anpassen kann. Er ist nicht gezwungen, seinen Prozess der Standardsoftware anzupassen, was der Fall wäre, wenn er sie mieten würde.

Dasselbe gilt für Web-Services. Wenn er sie kauft, kann er sie anpassen. Wenn er sie, in welcher Weise auch immer, mietet, muss er sie benutzen wie sie sind. Ist die Granularität der Software fein genug, kann dieses Problem beherrschbar bleiben. Er kann sie in seinen Prozess integrieren wie Ziegelsteine in eine Wand, aber wenn sie groß wie vorgefertigte Bauplatten sind, muss er seine Baupläne anpassen, um sie einbauen zu können. Andererseits ist er davon befreit, ständig neue Versionen zu installieren und zu testen. [2]

1.3 Web-Services ausleihen

Web-Services aus der Open-Source-Gemeinde zu entnehmen, ist, wie sich einen Web-Service zu leihen. Vertreter dieser Gemeinde lassen gerne andere ihre Arbeit verrichten, um sie dann mit oder ohne eigene Änderungen zu übernehmen. Auf der einen Seite wollen sie nicht für Software bezahlen, und auf der anderen Seite wollen sie sie nicht entwickeln. Open Source Web-Services werden als Allgemeingüter angesehen, die jeder nach belieben verwenden kann.

Es werden zwei Aspekte hier behandelt, der moralische und der rechtliche. Der moralische Punkt ist der, das Software, einschließlich Web-Services, intellektuelles Eigentum ist. Jemand hat seine wertvolle Zeit geopfert, um eine Lösung zu erstellen, die ein Problem löst. Wenn diese Lösung nun auch jemand anderem hilft, sollte diese Person bereit sein, dafür zu bezahlen. Anderen Falls verstößt er gegen die Prinzipien einer freien Marktwirtschaft. Aus diesem Grunde ist die Verwendung von Open Source Web-Services fraglich und nicht vereinbar mit der Gesellschaft, in der wir leben.

Die rechtliche Frage ist die Frage der Verantwortlichkeit. Wer ist Verantwortlich für das, was der entliehene Service macht? Natürlich sind es nicht die Autoren, da sie gar nicht wissen, wo und durch wen ihr intellektuelles Eigentum verwendet wird. Somit kann es nur der Anwender dieses Eigentums sein. In dem er einen Web-Service der Open-Source-Gemeinde verwendet, ist der Anwender frei, diesen so zu verändern, wie er ihn benötigt, aber dann muss er auch die Verantwortlichkeit für dessen Korrektheit und Zuverlässigkeit übernehmen., was wiederum bedeutet, dass er den Service in allen möglichen Fällen gründlich testen muss. Die meisten Anwender sind sich gar nicht bewusst, dass das Testen von Software bezüglich Zeit und Kosten so aufwändig ist, wie die Software zu entwickeln. In diesem Falle ist die Software den Testern noch fremd. Daher ist der Testaufwand größer als bei selbst entwickelter Software.Unzählige Studien haben bewiesen, dass der größte Zeitfaktor im Bereich der Software-Maintenance ist, die Software zu verstehen. [3] Bei Open-Source Code kommt dieser Faktor voll zum tragen.

Ein Verständnis für den Code zu entwickeln ist die größte Barriere, Open-Source zu verwenden. Man sollte es sich also zweimal überlegen, bevor man sich einen Web-Service von der Open-Source-Gemeinde entleiht. Es könnte sich herausstellen, dass man sich ein trojanisches Pferd ins Haus geholt hat.

1.4 Web-Services entwickeln

Web-Services können, wie jedes andere Softwarepaket, vom Anwender oder einem Vertragspartner entwickelt werden. Der Unterschied zu konventioneller Software ist, dass Web-Services, wenn sie korrekt definiert wurden, viel kleiner sind und leichter zu entwickeln sein sollten. Der andere Unterschied ist der, das Web-Services ein allgemeines Gut eines Unternehmens sein sollten, also für alle Einheiten des Unternehmens verfügbar sein sollten. Dies ist ein ernster Bruch in der Tradition der Unternehmen bezüglich der Frage, durch wen Software in dem Unternehmen bezahlt werden sollte.

In der Vergangenheit wurde sie IT-Abteilung als internes Software-Haus angesehen, mit dem Auftrag, für die verschiedenen Anwendungsbereiche als IT-Dienstleister zur Verfügung zu stehen. Benötigte die Marketing-Abteilung ein neues Customer-Relationship-Management-System, beauftragte sie die IT-Abteilung, eines zu entwickeln oder zu kaufen. Benötigte die Logistikabteilung ein neues Auftragsbearbeitungssystem, beauftragte sie die IT-Abteilung, eines für sie zu entwickeln. Es sind die Anwendungsbereiche, die über das Geld verfügen, und die werden es nur für etwas ausgeben wollen, das einen direkten Nutzen für sie hat.

Im Fall von Web-Services ist es nicht klar, wem sie gehören. Jeder im Netzwerk der Organisation kann auf sie zugreifen. Wer soll sie also bezahlen? Fakt ist, dass es ein großer Aufwand ist, gute Web-Services zu planen, zu entwickeln und zu testen. Sie sollten stabiler und verlässlicher sein als die Anwendungssysteme der Vergangenheit und sie sollten außerdem für die verschiedensten Zwecke und in verschiedenen Kontexten wieder verwendbar sein. Wie bei anderen Systeme kosten auch Web-Services drei mal so viel, als normale Single-User-Systeme. Anwendungsbereiche sind allerdings aber sehr abgeneigt gegenüber Projekten, die nicht genau ihren Anforderungen gewidmet sind und nur einen langfristigen Nutzen versprechen. Wenn sie ein Problem zu lösen haben, möchten sie es gleich und direkt gelöst bekommen, d.h. die Lösung soll auf ihre Anforderungen erfüllen und sonst nichts.

Einen Web-Service zu entwickeln ist eine langfristige Investition. [4] Es würde mindestens zwei Jahre dauern, bevor genug Services von ausreichender Qualität für die Anwender verfügbar sind, um ganze Prozesse daraus zu entwickeln. Es ist fraglich, ob die Anwender so lange warten wollen. Die Vorteile von selbst entwickelten Web-Services werden sich erst nach Jahren zeigen. In der Zwischenzeit muss die IT-Abteilung die laufenden Systeme warten. Dies bedeutet eine Doppelbelastung für die Organisation. Deswegen, und auch wegen der hohen Folgekosten, mag es keine attraktive Alternative sein, Web-Services selbst zu entwickeln. Die größte Barriere ist die benötigte Zeit, die zweite die Frage der Finanzierung.

1.5 Web-Services aus vorhandenen Systemen wiedergewinnen

Die fünfte und letzte Lösung ist, Web-Services aus bereits vorhanden Applikationen wieder zu gewinnen. Es mag wahr sein, das die existierenden Software alt, aus der Mode gekommen und schwierig zu warten ist, aber sie funktioniert. Und nicht nur dies, sie ist auch an die lokale Organisation angepasst. Sie passt zu den Daten und zur Umgebung der Organisation. Warum sollte man sie also nicht wieder verwenden? Das Ziel sollte nicht sein, die existierenden Anwendungen als Ganze zu nutzen, da sie vielleicht so nicht zu dem neuen Geschäftsprozess passen würden, sondern gewisse Teile zu extrahieren. Diese Teile können Methoden, Prozeduren, Module oder Komponenten sein. Das Wichtige ist nur, das sie unabhängig ausführbar sind. Dafür müssen sie gekapselt werden. Kapselungstechnologie ist der Schlüssel für die Wiederverwendbarkeit von Software. Es ist auch nicht so wichtig, in welcher Sprache die existierende Software geschrieben ist, so lange sie in der Server-Umgebung ausführbar ist. [5]

Da Anfragen an Web Services umgeleitet werden können ist es absolut legitim, verschiedene Server für verschiedene Sprachtypen zu haben. Es ist also durchaus möglich, einen COBOL und PL/I Service auf einem Server und einen C bzw. C++ Service auf einem anderen Server zu haben. Worauf es ankommt ist, dass die Komponenten mit einer standardisierten WSDL Schnittstelle ausgestattet sind, welche die Daten in der Anfrage in das lokal benötigte Format bringt und die die Ergebnisse wieder in das Datenformat der Anfrage konvertiert. Die Erstellung solche Schnittstellen kann automatisiert werden, so dass kein zusätzlicher Aufwand entsteht, als der, diese Schnittstellen zu testen [6]. Der Service selbst wurde durch den jahrelangen produktiven Einsatz getestet. Die geringen Kosten und die wenige Zeit, in der Web-Services auf diese Art erstellt werden können, sind die größten Vorteile dieses Ansatzes.

Die größten Nachteile sind:

- Die Software ist alt und nicht leicht zu verstehen
- Die Konvertierung von Daten aus einem externen in ein internes Format reduziert die Geschwindigkeit
- Es könnten irgendwann keine Programmierer mit Kenntnissen über Legacy-Systeme mehr verfügbar sein

Ein zusätzliches Problem ist es, den Datenstatus von einer Transaktion zur nächsten zu verwalten, wenn die Software nicht ursprünglich darauf ausgelegt war, ablaufsinvariant, bzw. reentrant, zu sein. Invarianz muss dann nachträglich in die Software implementiert werden. Dann stellt sich die Frage, wie man die verschiedenen Zustände der Benutzergruppen verwalten sollte. Dies ist kein web-service-spezifisches Problem, aber für Web-Services hat es eine besondere Bedeutung.

2. Ansätze zur Wiedergewinnung von Web-Services

Wer sich für das fünfte Alternativ entscheidet steht vor der Frage, wie man einen Web-Service aus existierender Software herausholen kann. Die Antwort auf diese Frage heißt Web Service Mining. Ein Großteil der benötigten Funktionalität einer Organisation wurde bereits auf die eine oder andere Weise implementiert. Die Funktionalität liegt begraben in ihrer Legacy Systemen. Web Service Mining dient dem Zweck sie ausfindig zu machen und wieder aufzubereiten. [7]

Um die bisherige Funktionalität für die Wiederverwendung mit Web-Services verfügbar zu machen, muss sie aus dem Kontext, in dem sie implementiert wurde, extrahiert und an die technischen Anforderungen einer service-orientierten Architektur angepasst werden. Dies beinhaltet vier Schritte:

- Entdecken
- Bewerten
- Extrahieren
- Anpassen

2.1 Entdecken potentieller Web-Services

Im Prinzip ist jede Funktion einer Legacy-Applikation ein potentieller Web-Service. Man sollte dabei merken, dass ein Großteil des Legacy-Codes aus technischen Funktionen besteht oder einer überholten Art der Datenhaltung oder der Kommunikation dient. Studien haben gezeigt, dass dieser Code fast zwei Drittel des gesamten Codes ausmacht. Dies lässt nur ein Drittel des Codes für die tatsächliche Erreichung der Applikationsziele. Dies ist der Code, der für das Mining von Interesse ist. Das Problem ist, das dieser Code in hohem Grad vermischt ist mit dem technischen Code. Innerhalb eines Code-Blocks, etwa einer Methode, einem Modul oder einer Prozedur, können sowohl Anweisungen für die Definition einer Datendarstellungsmaske sein als auch für die Berechnung der benötigten Werte. Beim Durchsuchen des Codes muss das Analysewerkzeug die Anweisungen mit einem betriebswirtschaftlichen Wert entdecken. [8]

Andererseits sind nicht alle betrieblichen Funktionen korrekt. Viele von ihnen werden über die Jahre veraltet sein. Das heißt, dass der applikationsrelevante Code identifiziert werden muss, er muss auch nach Gültigkeit geprüft werden. Dies wirft zwei Fargen auf:

- wie erkennt man, ob der Code anwendungsrelevant ist
- wie überprüft man, ob die Funktion noch einen Wert für den Anwender hat

Beide Themen werden eine Form der regelbasierten Entscheidungsfällung benötigen. Das wichtige dabei ist, dass der Benutzer in der Lage ist, die Regeln auf seine Bedürfnisse hin anzupassen, was bedeutet, dass das Analysewerkzeug in hohem Grad anpassungsfähig sein muss. Außerdem muss es schnell sein, denn es muss unter Umständen mehrere Millionen Zeilen Code analysieren um die davon für einen Web-Service relevanten zu identifizieren. Es wird also eine ausgefeilte Suchmaschine für Quellcode benötigt. Es könnte unmöglich werden, die Auswahl potentieller Webservices komplett ohne die Mitwirkung eines Mitarbeiters zu treffen. Es müsste also auch eine Dialogführung mit dem Benutzer in dem Werkzeug enthalten sein, um diesem die Möglichkeit zu geben, in die Suche einzugreifen und Entscheidungen zu treffen, die das Werkzeug nicht treffen kann.

Der Schlüssel zur Identifizierung potentieller Web-Services in existierendem Code wurde von diesem Autor bereits in einem früheren Paper bezüglich der Wiedergewinnung von Geschäftsregeln beschrieben [9]. Der Ansatz ist, die Namen der für die Applikation wichtigen Ergebnisdaten erst zu identifizieren und danach ihrer Entstehung zu verfolgen. Dies geschieht durch eine invertierte Datenflussanalyse. Der Datenfluss kann durch verschiedene Methoden oder Prozeduren in verschiedenen Klassen oder Modulen führen. Es ist wichtig, sie alle zu identifizieren. Ein Beispiel ist die Berechnung der Bonität im Kreditwesen. Das letztendliche Resultat ist die Bonität, aber es sind mehrere Klassen oder Module bei deren Berechnung involviert. Allerdings müssen sie alle bei der Erstellung eines Web-Services zur Bonitätsberechnung berücksichtigt und zusammengefasst werden. Dieses Problem ist verwandt mit dem der Impaktanalyse in der Software-Wartung.

2.2. Bewertung potentieller Web-Services

Ein weiteres Forschungsfeld ist die Bewertung der Code-Segmente, die für potentielle Web-Services in Frage kommen. Zunächst muss der Verwalter des Codes entscheiden, ob es sich überhaupt lohnt, Code-Fragmente aus dem System, in dem sie sich befinden zu extrahieren. Dies ist eine Frage der Wiederverwendbarkeit. Es müssen Metriken entwickelt werden, die Aussagen darüber treffen können, ob der Code wiederverwendbar ist oder nicht. Der Autor hat sich um dieses Thema bereits gekümmert und ein Paper dazu veröffentlicht [10]. Die Schlüsselmetrik ist die der Kapselungsfähigkeit. Ein Stück Code ist kapselbar, wenn es leicht aus seinem ihn umgebenden Code extrahiert werden kann. In diesem Fall hat es wenige externe Funktionsaufrufe, es erbt nichts und es teilt sich nur wenige Daten mit anderen Prozeduren. Dies bedeutet, dass man alle externen Funktionsaufrufe und alle Vererbungsbeziehungen zählen muss, genauso wie alle nicht lokale Daten, die außerhalb dieses Code-Blocks definiert werden. Diese Zählung muss dann in Relation zu der Größe des Code-Blocks, gemessen in Anweisungen, gebracht werden.

Dies wäre ein guter Ansatzpunkt, aber es reicht nicht aus. Es gibt noch die Fragen der Code-Qualität und des betrieblichen Wertes eines Code-Abschnitts zu klären. Es bleibt zu hinterfragen, ob der Code qualitativ gut sowie wertvoll genug ist, um in die neue Architektur übernommen werden zu können. Der betriebliche Wert muss gemessen werden, in dem gefragt wird, wie wertvoll die von dem Code-Abschnitt produzierten Ergebnisse sind. Es gab bisher nur wenig Forschung bezüglich dieser Frage. Schließlich muss berechnet werden, was es kostet, den Code zu extrahieren und dieses Ergebnis gegen den Wert der von ihm produzierten Ergebnisse gegenübergestellt werden. Dafür werden Metriken bezüglich Wartbarkeit, Testbarkeit, Interoperatibilität und Wiederverwendbarkeit des Codes wie auch bezüglich des betriebswirtschaftlichen Wertes der jeweiligen Funktionalität benötigt.

Die Evaluierung von potentiellen Web-Services ist kein triviales Forschungsthema und benötigt ausgereifte Metriken, um beherrscht zu werden. An dieser Stelle gilt der Aufruf an die Metrik-Forschungsgemeinschaft, ein System aus Maßen zu definieren, auf deren Basis es möglich ist, die Entscheidung zur Wiederverwendung zu fällen.

2.3 Extrahierung des Codes für den Web-Service

Wurde ein Code-Fragment als potentieller Web-Service identifiziert, ist der nächste Schritt, es aus dem System zu extrahieren, in dem es eingebettet ist. Dies kann eine hochkomplexe Aufgabe werden, ähnlich einer Organtransplantation besonders dann, wenn der Code sich nicht als separate Einheit kompilieren lässt. Prozedurale Module teilen sich globale Daten mit anderen Modulen im Hauptspeicher. Sie können auch andere Module aufrufen. Alle diese Abhängigkeiten müssen aufgelöst werden, um den Code aus seiner Umgebung extrahieren zu können. Objektorientierter Code ist generell leichter zu extrahieren, als prozeduraler, aber auch damit gibt es genug Probleme. Eine spezielle Klasse kann Elemente höherstufiger Klassen erben, die man nicht mit extrahieren möchte. Auch kann eine Klasse die Methoden einer fremden Klasse aufrufen, deren Ergebnisse wichtig für die weitere Verarbeitung sind. Diese Abhängigkeiten müssen entweder durch die Verflachung der Klassen – Class Flattening - oder durch Methodenauslagerung aufgelöst werden. Wie auch immer, keine dieser Lösungen ist einfach. Es bedarf noch der Forschung, die besten Methoden für die Extraktion von Methoden und Klassen zu bestimmen.

Ein besonders schwieriges Problem bei der Extraktion von Code aus Legacy-Systemen ist das der Isolierung von Funktionalität [11]. Besonders in objektorientierten Systemen sind Funktionen, sprich Anwendungsfälle, oft über viele Klassen diverser Komponenten verteilt. Eine Funktionen ist eine Kette verteilter Methoden, deren Aufruf durch ein Ereignis ausgelöst wird und ein vordefiniertes Ergebnis liefert. Dieses kann die Antwort auf eine Abfrage oder das Ergebnis einer Berechnung sein wie in etwas ein Preis oder eine Bonitätsbewertung. Um zu diesem Ergebnis zu gelangen müssen mehrere Methoden in verschiedenen Klassen in einer gegebenen Reihenfolge ausgeführt werden. Ein geplanter Web-Service wird in der Regel mit einer derartigen Funktion korrespondieren. Anwendungsbezogene Funktionen aus Komponenten zu extrahieren stellt die Reverse Engineeing Forschungsgemeinde vor eine schwierige Aufgabe. Es ist fraglich, ob es möglich ist, nur die Methoden zu extrahieren, die direkt von der gewollten Funktion benutzt werden, da diese Methoden Klassenattribute verwenden können, die einen Effekt auf andere Methoden haben. Andererseits wird die Extrahierung ganzer Klassen zu sehr großen Web-Services führen, bei denen ein Großteil des Codes nicht für die Erfüllung der vorgesehenen Aufgabe relevant ist. Dieses Problem zu lösen stellt, wenn es überhaupt lösbar ist, für die Forschungsgemeinde eine große Herausforderung dar.

2.4 Anpassung des Web-Service-Codes

Das letzte Forschungsthema ist die Anpassung der extrahierten Code-Einheiten an die Anforderungen eines Web-Services. Dies bedeutet, dass dieser mit einer WSDL-Schnittstelle ausgestattet werden muss. Die Eingabeparameter, welche diese zuvor von einer Parameterliste, einer Benutzerschnittstelle, einer Eingabedatei oder einer anderen Form der Dateneingabe erhalten haben, müssen nun als Argumente einer WSDL-Anfrage zugewiesen werden. Dies bedeutet, dass sie aus dem XML-Format der eingehenden SOAP-Nachricht konvertiert und in den internen Speicher des Web-Services geschrieben werden müssen. Die Ausgabewerte, die zuvor als Ausgabemasken, Rückgabewerte, Ausgabedateien, Berichte oder andere Formen der Datenausgabe ausgegeben wurden, müssen nun einer WSDL-Antwort zugewiesen werden. Dies impliziert, dass sie aus dem internen Speicher des Web-Services in eine ausgehende SOAP-Nachricht im XML-Zeichenformat geschrieben werden müssen. [12]

Alle diese Schritte können automatisiert werden. In der Tat ist die Anpassung des Codes der Teil, der sich am besten dafür eignet. Trotzdem ist die beste Methode, dies zu bewerkstelligen, noch zu finden. Für die Entwicklung von Anpassungswerkzeugen ist noch jede Menge Forschungsarbeit nötig.

3. Schlussfolgerung

Dieser Beitrag hat verschiedene Strategien für die Gewinnung von Web-Services für eine service-orientierte Architektur herausgestellt. Wie bereits darauf hingewiesen wurde, können sie gekauft, gemietet, geliehen, entwickelt oder aus vorhandenen Komponenten gewonnen werden. Es gibt Vor- und Nachteile für jede dieser Methoden. Die billigste und schnellste Lösung, abgesehen vom Übernehmen eines Web-Services aus der Open-Source-Gemeinde, ist die Gewinnung aus existierenden Applikationen im Besitzt des Anwenders.

Es existiert eine dringender Bedarf für Forschung im Bereich der Code-Wiedergewinnung. Zunächst sind Techniken für die Identifizierung von Code anhand der von ihm produzierten Ergebnisse nötig. Zweitens bedarf es Metriken für die Evaluierung der Wiederverwendbarkeit des identifizierten Codes. Drittens müssen Werkzeuge entwickelte werden, die in der Lage sind, die funktional zusammenhängenden Code-Abschnitte aus der Umgebung, in der sie eingebettet sind, zu extrahieren. Letztlich werden Werkzeuge benötigt um den Code zu kapseln und an die Anforderungen eines Web-Services anzupassen. Alle diese Schritte sind prinzipiell Themen für die Automatisierung, aber um dies zu erreichen, müssen die optimalen und verlässlichsten Wege, diese Schritte zu erledigen, noch gefunden und im Vergleich untereinander als solche bestätigt werden. In dieser Hinsicht kann die Forschungsgemeinde einen wertvollen Beitrag zur Migrationstechnik beisteuern.

Referenzen:

[01] Larson, G.: “Component-based Enterprise Frameworks”, Comm. Of ACM, Vol. 43, No. 10, Oct. 2000, p. 25
[02] Pak-Lok, P./Lau, A.: “The Present B2C Implementation Framework”, Comm. Of ACM, Vol. 49, No. 2, Feb, 2006, p. 96
[03] von Mayrhauser, A./Vans, A.: “Identification of Dynamic Comprehension Processes in large scale Maintenance”, IEEE Trans. on S.E., Vol. 22, No. 6, June, 1996, p. 424
[04] Bishop, J./Horspool, N.: “Cross-Platform Development – Software that lasts”, IEEE Computer, Oct. 2006, p. 26
[05] Aversano, L./Canfora, G./deLucia, A.: “Migrating Legacy System to the Web”, in Proc. of CSMR-2001, IEEE Computer Society Press, Lisabon, March 2001, p. 148
[06] Sneed, H.: “Wrapping Legacy COBOL Programs behind an XML Interface”, Proc. Of WCRE-2001, IEEE Computer Society Press, Stuttgart, Oct. 2001, p. 189
[07] Canfora, G./Fasolino, H./ Frattolillo, G.: “Migrating Interactive Legacy System to Web Services”, Proc. of CSMR-2006, IEEE Computer Society Press, Bari, March 2006, p. 23
[08] Sneed, H.: ”Integrating legacy Software into a Service oriented Architecture”, in Proc. of CSMR-2006, IEEE Computer Society Press, Bari, March 2006, p. 3
[09] Sneed, H./ Erdoes, K.: “Extracting Business Rules from Source Code”, Proc. of IWPC-96, IEEE Computer Society Press, Berlin, March, 1996, p. 240
[10] Sneed,H.: “Measuring Reusability of Legacy Software” in Software Process, Vol. 4, Issue 1, March, 1998, p. 43
[11] Greevy, O./Ducasse, S./Girba, T.: “Analyzing Software Evolution through Feature Views”, Journal of Software Maintenance & Evolution, Vol. 18, No. 6, Dec. 2006, p. 425
[12] Bodhuin, T./Guardabascio, E./Tortorella, M.: “Migrating COBOL Systems to the WEB”, WCRE-2002, IEEE Computer Society Press, Richmond, Nov. 2002, p. 329

Quelltextanalyse eines mittelgroßen, neuen Produktivsystems

Daniel Vinke, Meik Teßmer, Thorsten Spitta
Universität Bielefeld
Fakultät für Wirtschaftswissenschaften
post@danielvinke.de, {mtessmer|thspitta@wiwi.uni-bielefeld.de}

Abstract

Es wurde ein System von 200.000 LOC auf OO-Maße hin analysiert und die Abhängigkeiten der Maße statistisch analysiert. Das System ist seit 2001 im Einsatz und wird auch zur Zeit noch (2007) evolutionär weiter entwickelt. Einige Befunde aus der neueren Literatur wurden bestätigt, einige präzisiert. Wartungsintensive Klassen lassen sich zuverlässig erkennen.

1 Einführung

Quelltextanalysen sind nichts Neues (einen guten Überblick gibt Zuse [91]), auch vor der Verbreitung objektorientierter Software. Für objektorientierte Systeme braucht man spezifische Maße, von denen die „CK-Maße“ von Chidamber und Kemerer [CK94] als allgemein akzeptiert gelten [SK03]. Unklar definiert ist das Maß LCOM [SK03, 300]. Es erwies sich bei einer Validation der CK-Maße als einziges als nicht signifikant [BBM96].

An mittleren bis großen Systemen finden sich nur wenige Untersuchungen, noch weniger an produktiven [Vin07, 35]. Neben einer umfangreichen Laboruntersuchung [DKST05] wurde nur eine Analyse der Browserfamilie Mozilla gefunden [GFS05]. Beide Systeme sind in C++ geschrieben.

2 Untersuchungsobjekt

Das von uns analysierte System BIS (*Bielefelder Informations-System*) nennen wir *mittelgroß*: 190.000 LOC, 1430 Java-Klassen, 135 Packages. Von den Anfängen als Stundenplan-Verwaltungssystem für Studierende hat es sich inzwischen zum Raumplanungs-, Veranstaltungsverwaltungs- bis zum Prüfungsabwicklungssystem evolutionär entwickelt. Die Prüfungsabwicklung ist derzeit (Sommer 2007) noch recht rudimentär ausgeprägt. Große Teile des Systems sind als Auskunftsystem im Internet sichtbar: http://eKVV.uni-bielefeld.de/.

Da das System seit Ende 2003 unter Eclipse mit CVS weiterentwickelt und gepflegt wird, sind wir im Projekt SQUARE (*Software Quality and Refactoring Environment*) in der Lage, die Systemzustände seit Anfang 2004 in Form von Zeitreihen festzustellen, um z. B. Aussagen über Art und Verlauf der Evolution während der Entwicklung zu machen. Als erster Schritt war dazu eine Zeitpunkt-bezogene Analyse notwendig, von der wir hier berichten.

3 Untersuchung

Analysiert wurde der Stand des Systems vom Juni 2006. Wir wollten die CK-Maße erheben, ihren Nutzen mittels statistischer Analyse beurteilen und für das analysierte System kalibrieren.

Als Analysewerkzeug wurde *Understand* (for Java) verwendet. Die Alternative *Metrics* (Sourceforge) schied aus, da wir Quell- und keinen Bytecode analysieren wollten. Mit *Understand* hatten wir bereits Erfahrungen durch die Analyse eines deutlich größeren Systems (ca. 1 Mio LOC), das in C++ geschrieben war [Teß04]. Mittels *Understand* wurden alle CK-Maße ermittelt:

- **CBO** Coupling between Objects
- **LCOM** Lack of Cohesion in Methods
- **RFC** Response for a Class
- **DIT** Depth of Inheritance Tree
- **NOC** Number of Children
- **WMC** Weighted Methods per Class
- **NOM** Number of Methods

Es zeigte sich, dass die Daten für RFC nicht verwendbar waren, weil das Maß von *Understand* falsch berechnet wird, und dass LCOM nur bedingt verwendbar ist. Es ist bereits in [CK, 488] unpräzise definiert und wird von *Understand* als Prozentsatz ausgeben. Dies waren jedoch keine Hindernisse für die Analyse, weil wir ohnehin über Regressionsanalysen ermitteln wollten, welche Maße für das Auffinden wartungsintensiver Klassen benötigt werden. Danach ist RFC entbehrlich, weil es stark mit CBO und WMC korreliert ist.

Mit den ermittelten Maßen (ohne RFC, teilweise mit LCOM) wurde eine umfassende statistische Analyse mit dem System ***R*** durchgeführt, und zwar univariat (Häufigkeiten, Verteilungseigenschaften), bivariat (Korrelationsanalyse und Vergleich von jeweils

zwei Maßen mit Toleranzgrenzen) und multivariat (Hauptkomponentenanalysen). Bild 1 zeigt ein Beispielplot von max(VG)[1] gegen WMC. Man sieht z. B. eine Klasse mit fast 200 Methoden links oben, die aber nicht komplex ist. Interessant in diesen Plots sind immer die Einzelpunkte links oben, vor allem aber die rechts oben. Dies sind die „Ausreißer", die als wartungsintensiv interpretiert werden. Wir zählen in Bild 1 davon acht.

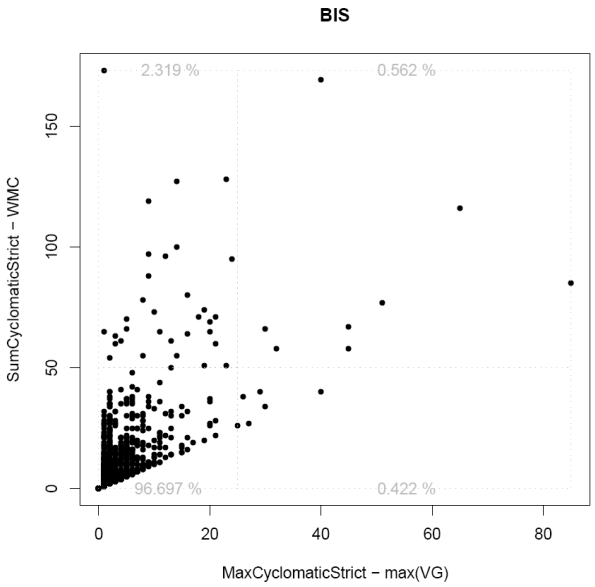


Bild 1: max(VG) korreliert mit WMC

Außerdem wurde Eclipse (mit 3.5 Mio LOC) analysiert, um die eigenen Kalibrierungen beurteilen zu können. Es enthält 12.000 Klassen, davon einige mit mehr als 10.000 LOC. Wir haben eine Klasse mit max(VG) > 1000 und drei mit VG > 500 gefunden. Strukturell sieht BIS in der Korrelationsanalyse zwischen WMC und CBO nicht schlechter aus als Eclipse.

4 Ergebnisse

Folgende Ergebnisse wurden ermittelt bzw. Angaben aus der Literatur bestätigt:

- LOC korreliert stark mit Komplexität
- Man findet wartungsintensive Klassen recht sicher mit mehreren Maßen. Diese sind miteinander korreliert.
- Diese Maße sind: LOC, max(VG), CBO und WMC (impliziert VG)
- Das Interaktionsmaß DIT*CBO bringt wichtige Zusatzinformationen
- Von den **OO-Maßen** braucht man **nur wenige**, es sind: WMC und DIT*CBO
- LCOM: Nach der Interpretation durch *Understand* haben recht viele Klassen einen **Kohäsionsmangel** von über 50%; oft **nahe 90%**.
- Im konkreten Fall des Systems BIS sind mindestens 70 (von rund 1400) Klassen wartungsintensiv. **5% halten wir für viel**.

5 Ausblick

Es besteht dringender Forschungsbedarf zur Kalibrierung der Maße, denn unsere Vorschläge beziehen sich nur auf einen Fall. Klar ist, dass in jedem Einzelfall kalibriert werden *muss*. Es sollte aber Toleranzgrenzen nach einer Klassifikation geben (etwa große / kleine oder einfache / komplexe Systeme).

Die erwähnten Längsschnittuntersuchungen sind in Arbeit.

Literatur

[BBM96] Basili, V. R.; Briand, L. C.; Melo, W. L.: A Validation of Object-Oriented Design Metrics as Quality Indicators. *IEEE Trans. Software Eng.* 22 (10): 751-761, 1994.

[CK94] Chidamber, S. R.; Kemerer, C. F.: A Metrics Suite for Object Oriented Design. *IEEE Trans. Software Eng.* 20 (6): 476–493, 1994.

[DKST05] Darcy, D. P.; Kemerer, C. F.; Slaughter, S. A.; Tomayko, J. E.: The Structural Complexity of Software: An Experimental Test. *IEEE Trans. Software Eng.* 31 (11): 982–995, 2005.

[GFS05] Gyimóthy, T.; Ferenc, R.; Siket, I.: Empirical Validation of Object-Oriented Metrics on Open Source Software for Fault Prediction. *IEEE Trans. Software Eng.* 31 (10): 897–910, 2005.

[Teß04] Teßmer, M.: Architekturermittlung aus Quellcode – Eine Fallstudie. Diplomarbeit, Universität Bielefeld, Technische Fak., Jan. 2004.

[Vin07] Vinke, D.: Quellanalyse eines mittelgroßen Produktivsystems – Eine Fallstudie zur Qualitätssicherung. Diplomarbeit, Universität Bielefeld, Fak.Wirtschaftswissenschaften, März 2007.

[Zus91] Zuse, H.: *Software Complexity*. Berlin et al., de Gruyter, 1991.

1 Die Komplexität nach McCabe. Die durchschnittliche Komplexität avg(VG) ist wenig aussagefähig.

JTransformer – Eine logikbasierte Infrastruktur zur Codeanalyse

Daniel Speicher, Tobias Rho, Günter Kniesel
Institut für Informatik III, Universität Bonn, Römerstrasse 164, 53117 Bonn
Email: {dsp, rho, kniesel}@iai.uni-bonn.de

Kurzfassung

Reengineering erfordert oft komplexe Analysen mit solider formaler Fundierung. Je weiter das Abstraktionsniveau des Formalismus dabei von dem der Implementierung entfernt ist, umso schwieriger ist die Implementierung und Wartung der Analyse. JTransfomer bietet daher eine logikbasierte Infrastruktur zur Codeanalyse, die eine angemessen direkte Umsetzung der formalen Grundlagen in ausführbaren Code erlaubt. Dies demonstrieren wir am Beispiel der Vorbedingungen für generalisierende Refactorings.

1. For Example Type Constraints

Als Fallstudie betrachten wir das auf Typbeschränkungen (‚type constraints') basierende Verfahren von Tip, Kiezun und Bäumer zur Entscheidung der Durchführbarkeit generalisierender Refactorings [3].

Zur Illustration der Grundgedanken untersuchen wir im Beispiel in Abb. 1. die Frage, ob in der Methode m wirklich der konkrete Typ B verwendet werden muss, oder auch der Obertyp A genügen würde? Der Aufruf der Methode aus dem Typ A auf der Variablen x steht dem nicht im Wege. Tatsächlich erzwingt jedoch die Methode n

```
1   package mini;
2
3   class A {
4       void p() {}
5   }                              Zuweisungen,
6   class B extends A {            Methodenaufrufe und
7       void q() {}                Rückgabeanweisungen
8   }                              stellen Bedingungen an
9                                  die Typen [E] der betei-
10  class C {                      ligten Ausdrücke E:
11      B m() {
12          B x = new B();         [new B()] ≤ [x]
13          x.p();                 [x] ≤ A
14          return x; }            [x] ≤ [C.m()]
15      void n() {
16          B y = m();             [C.m()] ≤ [y]
17          y.q(); }               [y] ≤ B
18  }
```

Abb. 1. Java Code mit Typbeschränkungen

indirekt die Verwendung des Typs B in Methode m, da dort auf dem Ergebnis von m eine Methode aus B aufgerufen wird. Der Ansatz von [3] erlaubt es diesen Zusammenhang formal abzuleiten.

Abb. 1 zeigt neben den unteren Codezeilen eine aus der jeweiligen Zeile abgeleitete Typbeschränkung. Z.B. folgt aus der Zuweisung des Rückgabewertes von m an y (Zeile 16), dass der Rückgabewert von m ein Subtyp des Typs von y sein muss. Aus den aufgeführten Ungleichungen ergibt sich die Ungleichungskette [x] ≤ [C.m()] ≤ [y] ≤ B und damit die behauptete Notwendigkeit. Die statischen Deklarationen des Typs von x oder y werden dabei nicht mit einbezogen, weil diese ja gerade zur Disposition stehen.

2. Logische Repräsentation und Analyse von Programmen

JTransformer [1] transformiert ein Java Programm in eine logische Faktenbasis, die die Grundlage für Programmanalysen darstellt. Wie in Abb. 2 illustriert, wird dabei jeder Knoten des abstrakten Syntaxbaumes eines Programms als ein logisches Faktum dargestellt [1]. Das Prädikatensymbol repräsentiert den Typ des Knotens. Der erste Parameter wird als eine eindeutige Identität dieses Knotens verwendet. Die anderen Parameter sind entweder primitive Werte (Zahlen, Namen) oder Identitäten anderer Knoten, die Referenzen auf diese Knoten darstellen[2].

```
classDef(clsC, pkgMini, 'C').
methodDef(mthM, clsC, 'm', clsB).
    localDef(varX, mthM, clsB, 'x', newB).
    newClass(newB, varX, ctrB, refB).
        ident(refB, newB, clsB).
    call(cllP, mthM, refX1, mthP).
        ident(refX1, cllP, varX).
    return(retX, mthM, refX2).
        ident(refX2, retX, varX).
methodDef(mthN, clsC, 'n', void).
    localDef(varY, mthN, clsB, 'y', cllM).
        call(cllM, varY, 'this', mthM).
    call(cllQ, mthN, refY, mthQ).
        ident(refY, cllQ, varY).
```

Abb. 2. Fakten für C

Prädikate können auch ganz oder teilweise über mitunter rekursive Ableitungsregeln definiert werden. Somit ist der Übergang von der Programmdarstellung zu Analysen dieser Darstellung fließend. Analysen sind lediglich weitere Prädikate.

Der Begriff des ‚declaration element' bezeichnet in [3] die Deklaration des statischen Typs von Methoden, Parametern, Feldern und lokalen Vari-

[1] Die Einrückung in Abb. 2 hat keine Bedeutung, sie verdeutlicht lediglich die Schachtelung der Elemente.

[2] So ist zum Beispiel das zweite Argument eines jeden Faktes eine Referenz auf den Elternknoten.

ablen, was sich in Prolog durch das folgende Prädikat decl_element ausdrücken lässt[3]:

```
decl_element(Elem, Type) :- methodDef(Elem, _, _, Type).
decl_element(Elem, Type) :- paramDef(Elem, _, Type, _, _).
decl_element(Elem, Type) :- fieldDef(Elem, _, Type, _, _).
decl_element(Elem, Type) :- localDef(Elem, _, Type, _, _).
```

Dies erlaubt erste Abfragen über unser Beispiel:

```
?- decl_element(mthM, clsB):  YES
?- decl_element(clIM, clsB):   NO
?- decl_element(Elem, clsB):  {ctrB, mthM, varX, und varY}
```

Die Abbildung eines Ausdrucks auf die Deklaration des davon referenzierten Elements wird durch das Prädikat referenced_decl umgesetzt:

```
referenced_decl(Ident, RefElem) :- ident(Ident, _, RefElem).
referenced_decl(Call, CalledM) :- call(Call, _,  _, CalledM).
referenced_decl(Call, CalledC) :- newClass(Call,_,CalledC,_).
```

4. Logikbasierte Umsetzung von ‚Type Constraints'

Im Beispiel müssen aufgrund der Verwendung der Methoden p und q die Variablen x und y mindestens den Typ A bzw. B haben. Die Zuweisungen und Ergebnisrückgaben implizieren dann Typbeschränkungen zwischen typdeklarierenden Elementen. Daraus ergibt sich eine Ungleichungskette, die zeigt, dass auch x den Typ B haben muss. Im Folgenden zitieren wir die entsprechenden formalen Definitionen und Implikationen aus [3] und geben dazu unsere Implementierung an.

Der Typ eines Ausdrucks auf dem die Methode m aufgerufen wird muss ein Subtyp sein von Decl(RootDef(M)), d.h. des allgemeinsten Typs der eine Deklaration von M enthält[4]:

(Call) call E.m() to a virtual method M
$\Rightarrow [E] \leq \text{Decl}(\text{RootDef}(M))$

Das folgende Prädikat constrained_by_type(E,T) setzt diese Folgerung direkt um. root_definition implementiert dabei die Funktion RootDef(M):

```
constrained_by_type(Elem, Type) :-
      call(_, _, CalledOn, CalledMethod),
      referenced_decl(CalledOn, Elem),
      root_definition(CalledMethod, RootMethod),
      methodDef(RootMethod, Type, _, _).
```

[3] Großschreibung in Regeln oder Fakten bezeichnet logische Variablen, Kleinschreibung Konstanten. Die ‚anonyme Variable' _ steht für beliebige Werte. Jede Regel ist als Implikation von rechts nach links zu lesen. So bedeutet z.B. die erste Regel „Wenn Elem eine Methodendefinition mit Rückgabetyp Type ist, dann ist Elem ein typdeklarierendes Element mit deklariertem Typ Type." Mehrere Regeln für das gleiche Prädikat sind disjunktiv verknüpft.

[4] Der Kürze der Darstellung halber ignorieren wir hier multiple Subtypbeziehungen.

Der Typ des Ausdrucks auf der rechten Seite einer Zuweisung muss ein Subtyp dessen auf der linken Seite sein:

(Assign) $E_1 = E_2 \Rightarrow [E_2] \leq [E_1]$

Das Prädikat constrained_by(L,U) stellt die Beschränkung des Typs des Elements L durch den Typ des Elements U dar:

```
constrained_by(Lower, Upper) :-
      assign(_, _, LHS, RHS),
      referenced_decl(LHS, Lower),
      referenced_decl(RHS, Upper).
constrained_by(Lower, Upper) :-
      localDef(Upper, _, _, _, InitExpression),
      referenced_decl(InitExpression, Lower).
```

Die Menge der *nicht generalisierbaren* Elemente, deren deklarierter Typ *C* sich nicht durch einen allgemeineren Typ *T* ersetzen lässt, umfasst alle Elemente die laut anwendbarer Typbeschränkungen Subtypen einer Klasse *C'* sein müssten, die *nicht* Obertyp von *T* ist. Hinzu kommen alle Elemente die Subtypen eines nicht generalisierbaren Elements sein müssten. Der einfach unterstrichenen Ausdruck entspricht constrained_by_type, der doppelt Unterstrichene constrained_by:

(Gen) $\text{Bad}(P,C,T) =$
$\{E \mid E \in \text{All}(P,C) \wedge \underline{[E] \leq C' \in \text{TC}_{\underline{\text{fixed}}}(P)}$
$\wedge \neg\, T \leq C' \} \cup$
$\{ E \mid E \in \text{All}(P,C) \wedge \underline{[E] \leq [E'] \in \text{TC}_{\underline{\text{fixed}}}(P)}$
$\wedge E' \in \text{Bad}(P,C,T) \}$

Das Prädikat not_generalizable(E,T) setzt dies um:

```
not_generalizable(Element, GeneralizedType) :-
      constrained_by_type(Element, Type, _),
      not( subtype(GeneralizedType, Type) ).
not_generalizable(Element, GeneralizedType) :-
      constrained_by(Element, Upper, _, _),
      not_generalizable(Upper, GeneralizedType).
```

Fazit

Dieser Beitrag gibt eine Idee davon, dass die in JTransfomer realisierte logikbasierte Umgebung eine deutliche Vereinfachung der Implementierung komplexer Programmalysen ermöglicht. Die Umsetzung formaler Systeme ist direkt rückverfolgbar zur theoretischen Grundlage.

Die Effizienz und Skalierbarkeit des Ansatzes wurde in [2] demonstriert.

Literatur

[1] JTransformer-Project, http://roots.iai.uni-bonn.de/research/jtransformer/.

[2] G. Kniesel, J. Hannemann, T. Rho: A Comparison of Logic-Based Infrastructures for Concern Detection and Extraction, LATE'07, March 12-16, 2007, Vancouver, BC

[3] Frank Tip, Adam Kiezun, Dirk Bäumer. Refactoring for Generalization Using Type Constraints. In *Proceedings of the 18th OOPSLA*, pages 13–26. ACM Press, 2003.

SW-Archäologie mit AOP (Praxisbericht)

Author: Oliver L. Böhm
agentes AG, Stuttgart
oliver.boehm@agentes.de

April 21, 2007

Abstract

Alte Software birgt oft viele Geheimnisse. Wenn man Glück hat, lässt sich zum Code, der untersucht und erweitert werden soll, noch die Dokumentation dazu auftreiben. Und wenn man sehr viel Glück hat, stimmt die Dokumentation sogar mit dem Code überein.

Leider sieht die Realität allzu oft anders aus: die Dokumentation ist hoffnungslos veraltet, die Entwickler von damals sind nicht mehr aufzutreiben und auch sonst gibt es niemanden, den man fragen könnte. Das einzige, worauf man sich verlassen kann, ist der Sourcecode. Aber leider ist dieser oft unverständlich...

Es gibt verschiedene Möglichkeiten, dem Code seine Geheimnisse zu entreißen. Ein vielversprechender Ansatz ist dabei der Einsatz von Aspekte, um unbekannte Codestellen zu erschließen und das Programmverhalten zu ergründen.

1 Die klassische Herangehensweise

Meist ist die Dokumentation oft die erste Anlaufstelle, um sich mit einer Alt-Anwendung vertraut zu machen. Auch wenn sie oftmals von der Realität abweicht, gibt sie doch einen wertvollen Einblick und wichtige Informationen, die den Einstieg in die Thematik erleichtern. Vor allem Übersichts-Dokumente über die Architektur, Infrastruktur und Randbedingen sind hier sehr hilfreich.

Um festzustellen, inwieweit der Code tatsächlich mit der Dokumentation übereinstimmt, sind Testfälle (sofern vorhanden) von eminenter Bedeutung. Sie bilden den Ausgangspunkt, um verschiedene Programmteile zu erkunden und deren Verhalten zu studieren. Die meisten Schwierigkeiten bestehen meist darin, die Testfälle zum Laufen zu bringen.

Hat man die Tests am Laufen, kann man sich an Refactoring-Maßnahmen wagen mit dem Ziel, die Businesslogik stärker zum Vorschein zu bringen und die Lesbarkeit und Wartbarkeit zu erhöhen. Vor allem J2EE-Anwendungen sind oftmals over-designed, was eine Einarbeitung meist erschwert.

Mit den neuen Erkenntnisse sollte nochmal die Dokumentation begutachtet werden: welche Dokumente müssen überarbeitet, welche Dokumente können ausgemistet oder/und in Form von neuen Testfällen ausgedrückt werden.

2 Unterstützung durch AOP

Der größte Manko bei Altanwendungen (und nicht nur dort) sind die Testfälle. Meistens fehlen sie oder sind genauso veraltet wie die Dokumentation. Ist die Anwendung noch in Betrieb, kann man versuchen, daraus Testfälle für die weitere Entwicklung abzuleiten. Und genau hier kann AOP helfen, fehlende Log-Informationen zu ergänzen oder die Kommunikation mit der Umgebung aufzuzeichen.

2.1 Schnittstellen beobachten

Die interessanten Stellen sind vor allem die Schnittstellen zur Außenwelt. Bei J2EE-Applikationen sind dies meist andere Systeme wie Legacy-Anwendungen oder Datenbanken, die über das Netzwerk angebunden werden. Hier reicht es oft aus, die Anwendung ohne Netzwerkverbindung zu starten und zu beobachten, wo überall Exceptions auftreten:

```
java.net.SocketException: java.net.ConnectException: Connection refused
    at com.mysql.jdbc.StandardSocketFactory.connect(StandardSocketFactory.java:156)
    at com.mysql.jdbc.MysqlIO.<init>(MysqlIO.java:283)
    at com.mysql.jdbc.Connection.createNewIO(Connection.java:2541)
    at com.mysql.jdbc.Connection.<init>(Connection.java:1474)
    at com.mysql.jdbc.NonRegisteringDriver.connect(NonRegisteringDriver.java:264)
    at java.sql.DriverManager.getConnection(DriverManager.java:525)
    at java.sql.DriverManager.getConnection(DriverManager.java:193)
    at bank.Archiv.init(Archiv.java:25)
    ...
```

Aus dieser Exception kann man erkennen, dass die Anwendung auf eine MySQL-Datenbank zugreift, aber kein Connection-Objekt vom DriverManager bekam. Mit diesem Wissen kann man sich alle Connection-Aufrufe genauer unter die Lupe nehmen:

```
after() returning(Object ret) :
        call(* java.sql.Connection.*(..)) {
    log.debug(getAsString(thisJoinPoint) + " = " + ret);
}
```

Mit dieser Anweisung wird am Ende jedes Connection-Aufrufs der Aufruf selbst mitsamt seinen Parametern (z. B. SELECT-Statements) und Rückgabewert ausgegeben (über die getAsString()-Methode, hier nicht abgebildet). Ähnlich kann man bei Netz- bzw. Socket-Verbindungen verfahren: man erweitert die Stellen (im AOP-Jargon "Joinpoints" genannt), über die Daten ausgetauscht werden.

Handelt es sich um eine interne Bibliothek oder Framework, das es zu betrachten gilt, sind vor allem

die öffentlichen Schnittstellen von Interesse. Hier kann man mit Aspektorientierten Sprachmitteln all die Methodenaufrufe ermitteln, die tatsächlich von außerhalb aufgerufen werden – denn oft sind nicht alle Methoden, die als "public" deklariert sind, für den Aufruf von außen vorgesehen.

```
public pointcut executePublic() :
    (execution(public * bank..*.*(..))
        || execution(public bank..*.new(..)))
    && !within(EnvironmentAspect);

public pointcut executeFramework() :
    execution(* bank..*.*(..)) || execution(bank..*.new(..));

public pointcut calledFromOutside() :
    executePublic() && !cflowbelow(executeFramework());

before() : calledFromOutside() {
    Signature sig = thisJoinPoint.getSignature();
    String caller =
        getCaller(Thread.currentThread().getStackTrace(), sig);
    log.info(caller + " calls " + sig);
}
```

Hier werden alle Methoden eines Bank-Frameworks (bank-Package) überwacht. Nur wenn der Aufruf nicht von innerhalb kam (!cflowbelow), wird vor der Ausführung der Methode oder Konstruktor der Aufrufer anhand des Stacktraces ermittelt (Methode getCaller(), hier nicht aufgelistet).

```
...
jsp.index_jsp._jspService(index_jsp.java:54) calls bank.Konto(int)
jsp.index_jsp._jspService(index_jsp.java:58) calls void bank.Konto.einzahlen(double)
jsp.index_jsp._jspService(index_jsp.java:60) calls double bank.Konto.abfragen()
...
```

Dies ist die Ausgabe, die den Aufruf aus einer JSP zeigt. Lässt man die Anwendung lang genug laufen, erhält man so alle Methoden, die von außerhalb aufgerufen werden.

2.2 Daten aufnehmen

Mit Java ist es relativ einfach möglich, Objekte abzuspeichern und wieder einzulesen. Damit lässt sich ein einfacher Objekt-Recorder bauen, um damit die Schnittstellen-Daten aufzunehmen:

```
after() returning(Object ret) : sqlCall() {
    objLogger.log(thisJoinPoint);
    objLogger.log(ret);
}
```

Hinter *thisJoinPoint* verbirgt sich der Context der Aufrufstelle, die der AspectJ-Compiler (eine AO-Sprache, die auf Java aufbaut, s. a. [Böh05]) als Objekt bereitstellt.

2.3 Aufnahmedaten einspielen

Wenn man die notwendigen Schnittstellen-Daten gesammelt hat, kann man anhand dieser Daten einen (einfachen) Simulator bauen. Man kennt die Punkte, über die diese Daten hereingekommen sind, man muss jetzt lediglich an diesen Punkte die aufgezeichneten Daten wieder einspielen:

```
Object around() : sqlCall() {
    Object logged = logAnalyzer.getReturnValue(thisJoinPoint);
    return logged;
}
```

Hat man so die Anwendung von der Aussenwelt isoliert und durch Daten aus früheren Läufen simuliert, kann man das dynamische Verhalten der Anwendung genauer untersuchen. So kann man weitere Aspekte hinzufügen, die automatisch Sequenz-Diagramme erzeugen oder wichtige Programmzweige visualisieren.

2.4 Code-Änderungen

Nach diesen Vorbereitungen kann mit den ersten Code-Änderungen (Refactorings[Fow01]) begonnen werden. Soll der Original-Code (noch) unverändert bleiben (was bei unbekannter Abdeckungen vorhandenen Testfällen sinnvoll sein kann), liefert die Aspektorientierung die Möglichkeit, den Code getrennt vom Original abzulegen. Man kann sogar verschiedene Varianten einer Komponente vorhalten und diese während der Laufzeit austauschen. Auf diese Weise kann man experimentell verschiedene Varianten und Code-Manipulationen ausprobieren, um deren Verhalten auf die Gesamt-Anwendung studieren zu können.

3 Fazit

Der Aufwand, sich in unbekanntem Code einzuarbeiten, wird häufig unterschätzt. Langfristig ist es meist wirtschaftlicher, vorhandenen Code komplett neu zu entwickeln, wenn die Dokumentation veraltet ist, der Code an vielen Stellen ausgewuchert ist und auch die Testfälle nicht vorhanden oder von zweifelhafter Qualität sind.

Oftmals hat man aber keine andere Wahl, als auf den bestehenden Code aufzusetzen, weil er die einzig gültige Quelle der Dokumentation darstellt. Und hier bietet die Aspektorientierung eine Vielzahl von Möglichkeiten, um

- zusätzliche Log-Möglichkeiten einzubauen,
- Schnittstellen zu überwachen,
- Objekt-Recorder zu implementieren und implantieren,
- die aufgenommenen Objekte wieder einzuspielen,
- u. v. m.

Daraus lassen sich weitere Testfälle schreiben und neue Erkenntnisse gewinnen, um Refactoring-Maßnahmen einzuleiten oder sich für eine Neuentwicklung zu entschließen. Allerdings entbindet auch AOP nicht von der Aufgabe, bestehenden und neuen Code so zu gestalten und umzuformen, dass künftige Generationen damit glücklich werden.

References

[Böh05] Oliver Böhm. *Aspekt-Orientierte Programmierung mit AspectJ 5.* dpunkt.verlag, 1 edition, 2005. ISBN-10 3-89864-330-15.

[Fow01] Martin Fowler. *Refactoring.* Addison-Wesley, 7 edition, 2001. ISBN 0-201-48567-2.

JINSI: Isolation fehlerrelevanter Interaktion in Produktivsystemen

Martin Burger, mburger@cs.uni-sb.de
Universität des Saarlandes, Fachrichtung Informatik
Saarbrücken, Deutschland

1 Einleitung

Die Fehlerbeseitigung in einem Softwaresystem beginnt zumeist mit der Reproduktion des Fehlers. Bei einem Produktivsystem ist dies besonders schwierig, da es von zahlreichen Diensten wie z.B. Datenbanken abhängt. Daher kann das System nicht ohne weiteres unter Laborbedingungen gestartet und beobachtet werden. Wir schlagen als Hilfsmittel für JAVA-Systeme JINSI vor: JINSI führt Teilkomponenten eines Produktivsystemes unter Laborbedingungen aus. Dazu zeichnet es zunächst die Interaktion dieser Komponente mit ihrer Umgebung im Produktivbetrieb auf und versorgt sie später im Laborbetrieb mit dieser Interaktion. Dort kann die Komponente nicht nur nach Belieben ausgeführt werden, sondern JINSI ermittelt auch den Bruchteil der Interaktion, der tatsächlich fehlerrelevant ist. Damit automatisiert JINSI zwei wesentliche Aufgaben der Fehlersuche: die Reproduktion und die Isolation eines Fehlers.

Bisherige Arbeiten zeichnen ganze Programmzustände auf, um einen Lauf zu wiederholen und zu untersuchen. Hierbei fallen allerdings sehr große Datenmengen an. Eine weitere Möglichkeit, um relevante Daten wie Parameter und Rückgabewerte zu speichern, besteht in der Serialisierung von Objekten. Diese ist allerdings bezüglich Speicher und Zeit sehr teuer. Die von JINSI aufgezeichnete Datenmenge ist im Vergleich winzig, da es die zu speichernden Daten auf ein Minimum beschränkt: (1) es werden nur die Objekte gespeichert, die Einfluss auf die Berechnung haben und (2) JINSI benötigt nur Objekt-Ids, Typen und skalare Werte für die spätere Wiedergabe. Dadurch werden benötigter Speicher und Zeit drastisch reduziert, was das Aufzeichnen in Produktivsystemen erlaubt.

JINSI arbeitet erfolgreich auf großen Systemen wie dem ASPECTJ-Compiler. Mit unseren Ansatz beträgt der Faktor für den zeitlichen Mehraufwand durch das Aufzeichnen nur 1,1 bis etwa 2. JINSI konnte in ersten Versuchen die Interaktion auf wenige fehlerrelvante Methodenaufrufe minimieren.

2 JINSI

Mittels Instrumentierung zieht JINSI[1] zunächst eine Grenze zwischen einer Komponente und ihrer Umgebung (Abb. 1); die Definition einer Komponente erfolgt durch Auswahl von Klassen oder Paketen. JINSI identifiziert die dortige Interaktion und instrumentiert den Bytecode so, dass ein- und ausgehende *Ereignisse*, wie Methodenaufrufe, Rückgabewerte und Feldzugriffe, abgefangen werden. Die Instrumentierung stellt auch sicher, dass nur Ereignisse aufgezeichnet werden, die die Grenze passieren. Die notwendige Instrumentierung kann vor Auslieferung oder auch zur Laufzeit im Produktivbetrieb geschehen und dauert auch bei großen Systemen nur wenige Sekunden. So kann die erneute Definition einer alternativen Grenze schnell erfolgen.

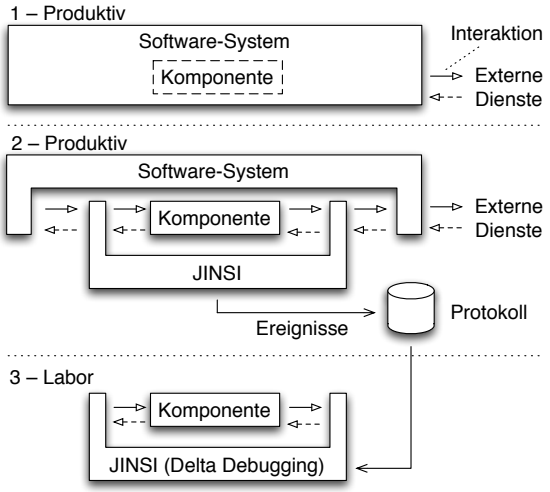


Abbildung 1: Eine Komponente in einem Softwaresystem (1) wird mittels Instrumentierung abgegrenzt. Danach kann JINSI die Interaktion im Produktiveinsatz abfangen und aufzeichnen (2). Im Labor kann der Fehler reproduziert und isoliert werden (3).

Daten, die während der Ausführung die Grenze passieren, können von skalaren Werten bis zu komplexen Objektgraphen reichen. Das Speichern von kompletten Objekten ist sehr aufwendig. So ist Serialisierung hinsichtlich Zeit und Speicher kostspielig. JINSI speichert stattdessen für Objekte, die für die Berechnung relevant sind, lediglich eine eindeutige Id und den Typ. Da auch alle ausgehenden Ereignisse aufgezeichnet werden, sind genügend Informationen für das spätere Abspielen vorhanden: Wird beispielsweise ein ausgehender Methodenaufruf abgefangen, so wird der Rückgabewert aufgezeichnet. Während der Wiedergabe ist der reale Wert nicht vonnöten; JINSI kann dann

[1] Für "JINSI Isolates Noteworthy Software Interactions"; "Jinsi" ist auch Suaheli und bedeutet "Methode".

den passenden Rückgabewert liefern: entweder einen skalaren Wert oder ein passendes Objekt[2]. Durch dieses *teilweise* Aufzeichnen fällt der Faktor des zeitlichen Overheads mit etwa 1,1 bis 2 klein aus, und die in XML gespeicherten Ereginisse sind auch bei langen Läufen komprimiert maximal einige Megabyte groß.

Vor dem Abspielen wird die Komponente erneut instrumentiert, so dass ausgehende Konstruktoraufrufe und Feldzugriffe behandelt werden können. Mittels der aufgezeichneten Interaktion kann dann der fehlerhafte Lauf auf der Komponente nach Belieben wiederholt werden. Diese Interaktion kann aber sehr viele irrelevant Ereignisse umfassen, z.B. wenn es sich um einen lange laufenden Serverdienst handelt. JINSI nutzt *Delta Debugging* (Zeller, 2005), um die Menge der Ereignisse durch systematisches Wiederholen des Laufes zu minimieren. Dabei werden alle nicht relevanten eingehenden Methodenaufrufe beseitigt. Da die Isolation ohne Umgebung, also z.B. ohne Anfragen an eine Datenbank, geschieht, kann der Fehler innerhalb weniger Sekunden bis Minuten vollautomatisch eingegrenzt werden.

Nach der Reduzierung auf die relevanten Ereignisse kann JINSI einen minimalen JUNIT Testfall erzeugen, der den Fehler in einer IDE wie ECLIPSE reproduziert. Dort können alle verfügbaren Werkzeuge genutzt werden, um den Defekt zu beseitigen.

3 Andere Arbeiten

JINSI ist der erste Ansatz, der das Aufzeichnen und Abspielen von Komponenteninteraktion mit der Isolierung relevanter Methodenaufrufe kombiniert. Frühere Arbeiten behandeln diese Ansätze getrennt. Lei and Andrews (2005) nutzen Delta Debugging um eine *zufällige* Sequenz von Methodenaufrufen zu minimieren, im Gegensatz zu zuvor unter realen Bedingungen aufgezeichneten Interaktionen. Delta Debugging beschreibt allgemein die Isolierung fehlerrelevanter Umstände (Zeller, 2005). Omniscient Debugging zeichnet *alle Zustände* eines Laufes auf (Lewis, 2003). JINSI reduziert den Lauf auf relevante Aufrufe, diese können mit ähnlichen Techniken untersucht werden. SCARPE (Orso and Kennedy, 2005) ist ein Werkzeug für das *selektive* Aufzeichnen und Abspielen von Interaktion in JAVA-Programmen; JINSI nutzt ähnliche Techniken (Orso et al., 2006).

4 Ergebnisse und Ausblick

JINSI implementiert eine kombinierte Lösung für zwei nicht triviale Aufgaben: Reproduzieren eines Fehlers und Beschränken auf die fehlerrelevante Komponenteninteraktion. Aus dieser Interaktion wird ein minimaler JUNIT-Testfall erzeugt, der den Fehler reproduziert. Unser Ansatz kann für komplexe Systeme im Produktiveinsatz angewandt werden und erlaubt die Analyse eines realen Fehlers im Labor. Neben ASPECTJ haben wir JINSI bereits erfolgreich auf dem E-Mail-Programm COLUMBA angewandt.

Durch die begrenzte Beobachtung einer Komponente und die Aufzeichnung weniger Objekt-Informationen wird die Menge der anfallenden Daten gering gehalten: Werden alle Interaktionen der zentralen ASPECTJ-Klasse `BcelWorld` beim Kompilieren des Timing Aspektes des mitgelieferten Beispieles "Telecom Simulation"[3] aufgezeichnet, so werden 5496 Ereignisse abgefangen. Diese sind als gzip-komprimierte XML-Datei lediglich 68 Kilobyte groß.

Die Instrumentierung der 3494 Klassen umfassenden Datei aspectjtools.jar dauert auf einem iMac mit Intel 2,0GHz Core Duo CPU nur 20,8s. Das Kompilieren des Aspektes benötigt mit der originalen Bibliothek 2,1s, mit der instrumentierten 4,5s. Hier beträgt der Faktor des zeitlichen Overheads 2,1. In länger laufenden Programmen ist mit einem Faktor von 1,1 bis 1,2 zu rechnen.

Mittels der aufgezeichneten Ereignisse kann ein Fehler reproduziert werden. JINSI hat in ersten Versuchen diese Interaktionen minimieren können. Der daraus resultierende JUNIT-Testfall umfasst nur einige wenige Methodenaufrufe auf der fehlerhaften Komponente.

JINSI ist noch ein Prototyp und wird weiterentwickelt. Eine Fallstudie mit dem Benchmark iBugs[4], welcher 369 echte Fehler in ASPECTJ enthält, soll den Nutzen belegen. Neben der Minimierung ausgehender Interaktion erscheint die von eingehender Interaktion interessant, um die Menge der relevanten Ereignisse weiter einzugrenzen. Eine Integration in ECLIPSE als Plug-In soll den Ansatz frei verfügbar machen.

Literatur

Y. Lei and J. H. Andrews. Minimization of randomized unit test cases. In *Proc. 16th IEEE Intl. Symp. on Software Reliability Engineering*, 2005.

B. Lewis. Debugging backwards in time. In *Proc. 5th Int. Workshop on Automated and Algorithmic Debugging*, 2003.

A. Orso and B. Kennedy. Selective Capture and Replay of Program Executions. In *Proc. of the 3rd Intl. ICSE Workshop on Dynamic Analysis*, 2005.

A. Orso, S. Joshi, M. Burger, and A. Zeller. Isolating relevant component interactions with jinsi. In *Proc. of the 4th Intl. ICSE Workshop on Dynamic Analysis*, 2006.

A. Zeller. *Why Programs Fail: A Guide to Systematic Debugging*. Morgan Kaufmann, 1st edition, 2005.

[2] JINSI verwendet hier sog. Mockobjekte. Diese implementieren die Schnittstelle des echten Objektes, ahmen das eigentliche Verhalten aber nur nach; das Verhalten kann von außen vorgegeben werden.

[3] `http://www.eclipse.org/aspectj/doc/released/progguide/examples-production.html`. Version 1.5.3.

[4] `http://www.st.cs.uni-sb.de/ibugs/`

Erzeugung dynamischer Objektprozessgraphen zur Laufzeit

Jochen Quante
Arbeitsgruppe Softwaretechnik
Fachbereich 3, Universität Bremen
http://www.informatik.uni-bremen.de/st/
quante@informatik.uni-bremen.de

1 Einführung

Objektprozessgraphen beschreiben den Kontrollfluss eines Programms aus Sicht eines einzelnen Objekts. Sie enthalten Informationen darüber, wie Objekte benutzt werden und wie diese Benutzungen in Beziehung zueinander stehen. Damit können sie für das Programmverstehen hilfreich sein (z.B. Protokollerkennung, Visualisierung). Allerdings ist die Extraktion solcher Graphen sehr aufwändig. Wir stellen eine neue dynamische Extraktionstechnik vor, die entsprechende Graphen schon während des Programmlaufs aufbaut und damit neue Anwendungen ermöglicht.

2 Extraktion dynamischer Objektprozessgraphen

Ein *Objektprozessgraph* ist ein Teilgraph des interprozeduralen Kontrollflussgraphen, der nur die Knoten und Kanten enthält, die für den Kontrollfluss aus Sicht eines bestimmten Objekts relevant sind [2]. Zur dynamischen Extraktion eines solchen Graphen wird zunächst das zu analysierende Programm so instrumentiert, dass alle Kontrollfluss- und Objekt-relevanten Informationen erfasst werden. Indem Instrumentierungspunkte auf Knoten und Ausführungsbeziehungen auf Kanten abgebildet werden, kann daraus ein entsprechender "Rohgraph" konstruiert werden, der durch Transformationen zum dynamischen Objektprozessgraphen (DOPG) wird. Das genaue Verfahren ist in [2] beschrieben. Abb. 1 zeigt ein Beispiel für die Konstruktion eines solchen Graphen während der Transformation.

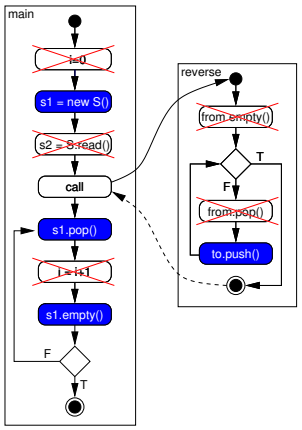
Im bisherigen Ansatz ("Offline") werden die Informationen aus der Instrumentierung in eine Datei geschrieben und erst nach Programmende analysiert. Der grobe Ablauf ist in Abb. 2(a) skizziert. Er beinhaltet einige aufwändige Schritte:

- Schreiben der Protokolldatei mit ca. 1-6 Mio. Ereignissen pro Sekunde. Dieser Schritt kann durch Kompression und Pufferung beschleunigt werden.
- Herausfiltern der relevanten Daten für einzelne Objekte. Die ursprüngliche Protokolldatei enthält die Daten für *alle* gewählten Objekte.
- Einlesen der gefilterten Daten zur Konstruktion des Graphen.

```
void main() {
  int i = 0;
  S s1 = new S();
  s2 = S.read();
  reverse(s2, s1);
  do {
    s1.pop();
    i = i + 1;
  } while (!s1.empty());
}

reverse(S from, S to) {
  while (!from.empty())
    to.push(from.pop());
}
```

(a) Quelltext



(b) DOPG für s1

Abbildung 1: Quelltext und DOPG für Anwendung eines Stack-Objekts. Die durchkreuzten Knoten werden im nächsten Schritt entfernt.

Um die Konstruktion zu beschleunigen, schlagen wir ein Verfahren vor, das auf das Schreiben der Protokolldatei verzichtet und die Graphen aufbaut, während das Programm noch läuft. Dies wird im folgenden genauer beschrieben.

3 Extraktionsalgorithmus

Die Konstruktion des Graphen zur Laufzeit ist in Abb. 2(b) skizziert und verläuft in folgenden Schritten (Details in [1]):

- Mitführung eines Graphen, der den aktuellen Aufrufpfad beschreibt. Dazu wird anhand der auftretenden Ereignisse ein Graph konstruiert (wie oben beschrieben). Wird eine Methode wieder verlassen, so wird der gesamte Methodenaufruf aus dem Graphen entfernt.
- Wenn eine zu beobachtende Klasse instanziiert wird, so wird dieser Graph kopiert. Die Kopie stellt den konkreten "Rohgraph" für dieses Objekt dar.
- Für alle weiteren Ereignisse werden sowohl zum Aufrufpfad-Graph als auch zu allen Objekt-Graphen entsprechende Kanten und Knoten hinzugefügt. Dabei werden Objekt-spezifische Ope-

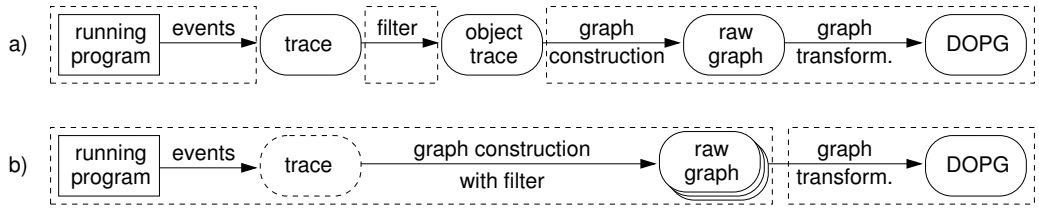


Abbildung 2: Offline- (a) und Online-Verfahren (b) im Vergleich.

rationen nur auf den zugehörigen Graphen angewendet. Bei den Kopien werden Methodenaufrufe nur dann entfernt, wenn der jeweilige Aufruf nicht relevant war. *Relevant* ist ein Aufruf genau dann, wenn er den Aufruf einer Methode oder den Zugriff auf ein Attribut des Objekts *oder* einen relevanten Aufruf enthält.

Damit wird zu jeder Instanz einer zu beobachtenden Klasse ein eigener Graph mitgeführt. Die Datenstrukturen werden so gewählt, dass ein effizientes Prüfen und Hinzufügen von Methodenaufrufen möglich ist. Da Java-Programme analysiert werden sollen, muss bei der Implementierung auch Multithreading berücksichtigt werden. Dies bedeutet, dass der Graph möglicherweise an mehreren Stellen gleichzeitig erweitert wird.

4 Fallstudie

Wir vergleichen nun den Laufzeitoverhead der Online- und Offline-Extraktionstechniken anhand mehrerer Beispiele. Dazu werden drei Java-Anwendungen unterschiedlicher Größe betrachtet: ArgoUML, J (ein Editor) und JHotDraw. Für jede der Anwendungen werden verschiedene Klassen herausgegriffen und für deren Instanzen DOPGs sowohl mit der Online- als auch mit der Offline-Variante erzeugt. Die benötigte CPU-Zeit wird gemessen. Da es sich bei allen Anwendungen um interaktive Systeme handelt, werden alle Szenarien mehrfach durchlaufen und Mittelwerte gebildet, um Variationen in der Bedienung auszugleichen.

Abb. 3 zeigt die Ergebnisse dieser Studie. Der durch die Instrumentierung verursachte Laufzeitoverhead ist sehr unterschiedlich: Für J liegt er zwischen Faktor 7 und 22, während er für JHotDraw bei 2 bis 3 liegt. In den meisten Fällen ist die Online-Variante aber mindestens genauso performant wie die Offline-Variante. In allen Fällen waren die Anwendungen ohne größere Verzögerungen weiterhin benutzbar.

Für das Szenario "Quad" brauchte die Online-Extraktion mehr Rechenzeit als für die Offline-Variante. Dies liegt darin begründet, dass in diesem Fall zehn Instanzen parallel verfolgt wurden, während in den anderen Fällen nur für jeweils eine Instanz ein Graph erstellt werden musste. Dies deutet darauf hin, dass die Online-Extraktion insbesondere in Fällen mit wenigen zu verfolgenden Objekten sinnvoll einsetzbar ist.

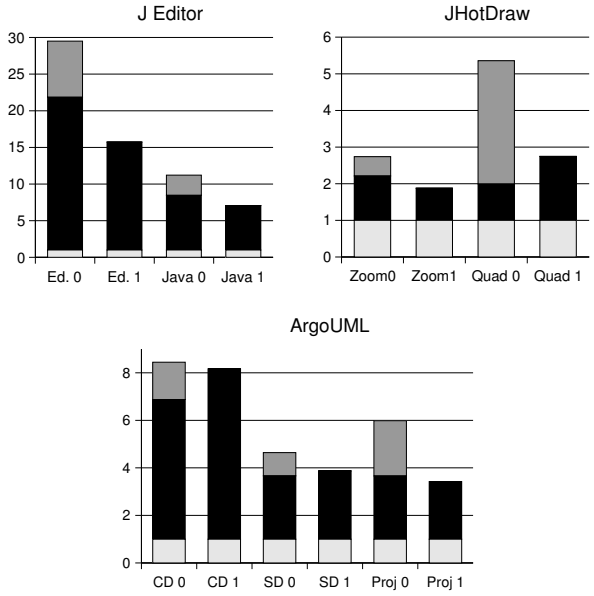


Abbildung 3: Ergebnisse der Fallstudie. Auf der Y-Achse ist der Laufzeitoverhead (schwarz) als Faktor im Vergleich zur normalen Laufzeit angegeben. Der linke Balken (0) steht jeweils für die Offline-Extraktion, der rechte Balken (1) für die Online-Variante. Für die Offline-Extraktion ist zusätzlich die Zeit für das Filtern angegeben (grau).

5 Fazit

Die Fallstudie zeigt, dass die Konstruktion von dynamischen Objektprozessgraphen in der Praxis effizient möglich ist. In vielen Fällen, insbesondere bei einer geringen Anzahl betrachteter Objekte, bremst die Online-Extraktion das beobachtete Programm weniger als das reine Schreiben der Protokolldatei. Damit ergeben sich ganz neue Anwendungsmöglichkeiten: Die Graphen können zur Laufzeit visualisiert werden und so einen Einblick geben, wie sich bestimmte Benutzeraktionen in Bezug auf ein Objekt auswirken.

Literatur

[1] Jochen Quante. Online construction of dynamic object process graphs. In *Proc. of 11th CSMR*, pages 113–122, 2007.

[2] Jochen Quante and Rainer Koschke. Dynamic object process graphs. In *Proc. of 10th CSMR*, pages 81–90, 2006.

Changes to Code Clones in Evolving Software

Jens Krinke
FernUniversität in Hagen, Germany
krinke@acm.org

1 Introduction

Although cut-copy-paste (-and-adapt) techniques are considered bad practice, every programmer is using them. Because such practices not only involve duplication but also modification, they are called *code cloning* and the duplicated code is called *code clone*. A *clone group* consists of the code clones that are clones of each other. Code cloning is easy and cheap during software development, but it makes software maintenance more complicated, because errors may have been duplicated together with the cloned code and modifications of the original code often must also be applied to the cloned code.

There has been little empirical work that checks whether the above mentioned problems are relevant in practice. Kim et al. [2] investigated the evolution of code clones and provided a classification for evolving code clones. Their work already showed that during the evolution of the code clones, common changes to the code clones of a group are fewer than anticipated. Geiger et al. [1] studied the relation of code clone groups and change couplings (files which are committed at the same time, by the same author, and with the same modification description), but could not find a (strong) relation. Therefore, this work will present a study that (in)validates the following hypothesis:

> *During the evolution of a system, code clones of a clone group are changed commonly.*

Of course, a system may contain bugs like that a change has been applied to some code clones, but has been forgotten for other code clones of the clone group. For stable systems it can be assumed that such bugs will be resolved at a later time. This results in a second hypothesis:

> *During the evolution of a system, if code clones of a clone group are not changed commonly, the missing changes will appear in a later version.*

This work will validate the two hypotheses by studying the changes that are applied to code clones during 200 weeks of evolution of two open source software systems.

2 Experiment Setup

For the study the version histories of two open source systems have been retrieved: The first system is `jhotdraw`. It is a drawing application and framework written in Java. Its goal is the demonstration of good use of design patterns. Its version archive is available via CVS and for the study, the `jhotdraw6` CVS-module has been used. The second system is a subsystem of `eclipse`. Its version archive is also available via CVS and for the study, the `org.eclipse.jdt.core` CVS-module has been used.

For both systems, the sources have been retrieved based on their status on 200 different dates, such that each version is exactly one week later or earlier than the next or previous version. For both systems, only the Java source files have been analyzed. Also, the source files have been transformed to eliminate spurious changes between versions: Comments have been removed from the sources and afterward, the source files have been reformatted with the pretty printer *Artistic Style*. The transformed sources are saved to a repository.

The changes between the versions of the system have been identified by the standard *diff* tool. For each version v of the analyzed system, the changes between version v and $v + 1$ (the version of the next week) have been identifed. Also, the changes between a version in week v and and in five weeks later ($v + 5$) have been identified. For each of the 200 versions, the clone groups have been identified by the use of the clone detection tool *Simian* from RedHill Consulting Pty. Ltd.

The analysis has been done on 195 version because the versions of the last five weeks were used only for the generation of the changes between the versions. For each week w, $0 \leq w < 195$, the tool has generated the list of clone groups with common and non common changes, based on the clone groups of the analyzed system in week w and the changes from week w to week $w + 1$ (and to week $w + 5$).

3 Results

This section will present the results of the study as described in the previous section. It will first describe the results for `eclipse` and `jhotdraw` independently and will then present common results together with the effects of comment removal. Also, the part of the study that validates the second hypothesis is presented.

3.1 Results for `eclipse`

Most of the times when changes occur to clone groups, only one or two clone groups have non common changes. It is also rarely the case that during one week more than three clone groups are affected by any kind of change. Table 1 shows the accumulated numbers for all 198 weeks.

	eclipse		jhotdraw	
	1 week	5 weeks	1 week	5 weeks
clone groups with non common changes	105	188	30	41
clone groups with common changes	79	135	19	20

Table 1: Comparison for non common changes during one and five weeks

This reveals that in the `org.eclipse.jdt.core` system, clone groups have more often non common changes than common changes. This suggests that if a code clone is changed, it is more often adapted in the environment it is used in, than it is modified in the same way like the other code clones in its group.

3.2 Results for `jhotdraw`

The development of `jhotdraw` was much less active and happened in bursts. For only 21 versions occur changes to the clone groups, half of them due to irrelevant changes. The three bursts occur in week 76, 141, and 182. The burst in week 76 shows an interesting behavior: Although 16 clone groups are affected by changes, only two are affected by non common changes. A manual inspection of the changes revealed that during that week, a massive code cleanup has been applied to the source code with a lot of small refactorings. Again, non common changes to more than two clone groups are rare.

The numbers in Table 1 are similar to the numbers from `eclipse`. Again, clone groups have more often non common changes than common changes. This supports the assumption that if a code clone is changed, it is more often adapted in the environment it is used in, than it is modified in the same way like the other code clones in its group.

3.3 Evolution of Changed Clone Groups

Up to now, the presented results only gave evidence for the validation of the first hypothesis. The second hypothesis "*During the evolution of a system, if code clones of a clone group are not changed commonly, the missing changes will appear in a later version*" has also been validated within this study. If this hypothesis is true, changes to a clone group that are not applied to all code clones of a group will appear in a later version. The study only considered the changes to a system that appear within one week. If the hypothesis is true, there have to be more common changes if a longer time period is considered. To validate this, the study has been repeated with a time distance of five weeks instead of one as it is assumed that missed changes to a clone will be detected and fixed within four weeks.

The study generated the changes that appear within the next five weeks for the state of the software system at 195 weeks (0 to 194). Table 1 shows the numbers of changed clone groups within one week and the number of changed clone groups within five weeks. However, for the five weeks comparison, only the weeks were non common changes occurred have been used, i.e. the weeks where no (relevant) changes or common changes occurred within one week were ignored for the five week comparison.

The numbers show that during the five week period much more changes occur than within only the first week. Moreover, the non common changes increase much more than the common changes. This indicates that the second hypothesis is not valid. If it would be valid, the common changes would increase much more. However, there are exceptions to this general observation as manual inspection revealed: For example, in week 125 in `eclipse` six clone groups are changed in a non common way within the following week. Within the next four weeks, four of the of the six clone groups receive additional changes such that they are now changed in a common way during the five week period. In weeks 19, 59, 118, and 144 the changes are similar: two (for week 144, three) clone groups that have non common changes within one week have only common changes within five weeks. For `jhotdraw`, manual inspection did not reveal such behavior.

These observations suggest that the second hypothesis occurs in practice, but not very often. Moreover, during a longer period of observed time, many more non common changes to clone groups appear, such that occurrences of changes that change non common changes to common changes are hard to identify.

4 Conclusions

The study showed that the hypotheses are not valid generally. The study showed that clone groups more often have non common changes than common changes, invalidating the first hypothesis. The study also showed that the second hypothesis is only valid partially, because the non common changes appear much more often. However, a small amount of such changes that turn non common changes to a clone group to common changes in a later version has been observed.

References

[1] Reto Geiger, Beat Fluri, Harald C. Gall, and Martin Pinzger. Relation of code clones and change couplings. In *Proceedings of the 9th International Conference of Funtamental Approaches to Software Engineering (FASE)*, pages 411–425, March 2006.

[2] M. Kim, V. Sazawal, and D. Notkin. An empirical study of code clone genealogies. In *Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering (ESEC/FSE)*, pages 187–196, 2005.

How to Trace Model Elements?

Sven Wenzel
Software Engineering Group, University of Siegen
wenzel@informatik.uni-siegen.de

Motivation In model-driven engineering models enjoy the same status as customary source code. During the development process, many different versions or variants of models are created. Thereby, developers need to trace model elements between different versions to answer questions like *Since when does element X exist in this model?* or *When did element Y disappear?*. Regarding variants developers are often interested if certain elements or groups of elements (e.g. containing a bug) also exist in other variants. Questions from the analytical point of view, such as logical coupling, require traceability of model elements, too.

Traceability of elements becomes a challenge, since in daily practice, models are often stored as textual files, e.g. XMI, in versioning systems such as CVS or SVN. However, these systems are not aware of models, i.e. syntax and semantics inside the textual files. Subsequently, we present an approach for fine-grained element tracing based on difference computation. We furthermore offer a model-independent visualization of our trace information.

Differencing Models The main task of tracing is to locate the correspondence of an element in another model. The comparison of two succeeding documents and the location of correspondences respectively is known as difference computation and a daily task in software configuration management. Modern difference tools are able to compare models on an appropriate level of abstraction, which is basically a graph with tree-like structure; models are composed of elements which in turn have sub elements. They are not exactly trees due to cross references.

For our tracing approach, we use a generic similarity-based algorithm, called SiDiff, which was part of our prior research [1]. Instead of relying on persistent identifiers it computes similarities between model elements in a bottom-up/top-down algorithm, according to the tree-like structure of the models. If two elements reveal a unique similarity and they are not similar to other elements as well, they are matched. A threshold ensures a minimum similarity to avoid unsuitable matches. Elements of cycles are compared repeatedly as long as new matches can be found. Thereby, SiDiff can compare even less tree-like structures, e.g. Petri nets, whose elements are not qualified by their compositional structure but by their neighborhood to other elements. In a pre-phase a hash value is calculated for each element and elements with identical hash values are matched. Thereby, runtime of pairwise comparison is reduced and precision is improved since identical elements provide trustworthy fix points for comparison of neighbor elements.

Computation of Trace Information The comparison of two models provides information about correspondences, i.e. the same element occuring in both models. In order to compute trace information about an element in a model version, this model version is compared with each direct successor, either in the same branch or in parallel branches. The successors are compared with all their successors, and so on.

Starting from the basic model, for each element that occurs also in the succeeding models a track is created, e.g. for element *A* in Fig. 1. A track is a chain of model elements representing the same element occuring in different model revisions. Due to model variants the track may split into branches forming the shape of a tree.

The elements can either occur in the succeeding models without any changes or they have been changed. Unchanged elements are located immediately by the hashing functionality of the difference algorithm. Elements that have been changed from one revision to another are located by the similarity computation facilities. Thereby the threshold defined for each element type prevents from correspondences between elements with significant changes.

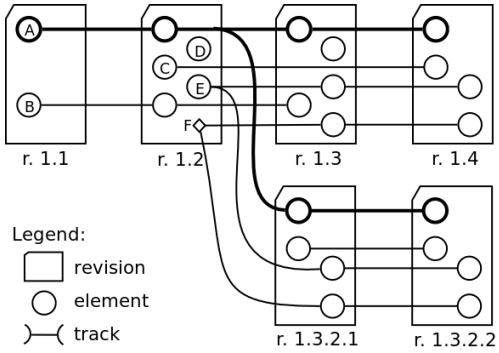


Figure 1. Examples of tracks

If an element cannot be located in any subsequent revision of a model the track of that element ends; e.g. track *B* ends in revision 1.3. Elements that do not have correspondences in an adjacent revision are compared to the elements of the next following revision. Thereby the traceability of elements is enhanced without including elements into a track, that do not reliably correspond. In this case a track may contain gaps, e.g. track *C*. Gaps may also cover branching points of a model, e.g. the one marked with *F* concatenating the tracks of two branches. Elements without any corresponding partner, e.g. element *D*, do not become part of any track.

Tracing of Elements In order to trace elements at daily work, the tracks computed above must be visualized in a meaningful way for developers. Therefore, we have implemented our approach as Eclipse plug-in using the GEF framework. A screenshot is shown in Fig. 2.

Tracks are visualized in an abstract representation independent of any model type. Rectangles represent different revisions of the given model, inside a rectangle

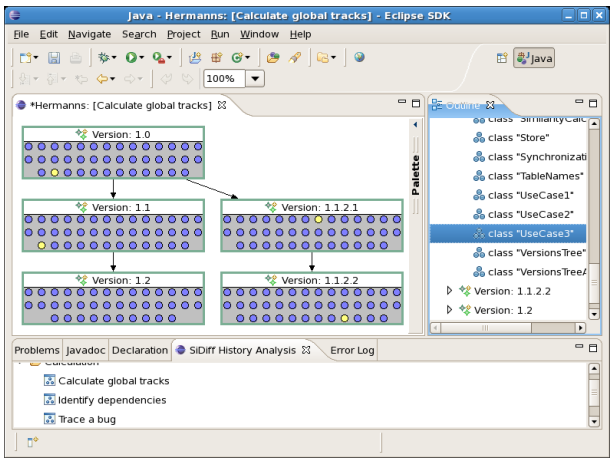


Figure 2. Screenshot of the analysis tool

each model element is represented by a small colored circle. The color provides information depending on the current analysis task. An outline view shows a list of all revisions and their elements inside. Both the graphical representation and the outline view allow developers to select model elements. Tool tips show further information about the elements. Filters can reduce the set of displayed elements. The panel on the lower side provides different analysis tasks to choose from. The current implementation supports four different analysis tasks:

Global Track Analysis. The analysis of global tracks is the simpliest task. It only uses the track information computed as described above. If a user selects a single model element in any revision of the model history, the occurences in all other revisions are highlighted despite the fact that the elements might have been slightly modified. One can immediately see (a) since when a given element exists in the history, (b) where an element of a revision disappeared or (c) where it occurs in other revisions or variants. Given the examplary screenshot above, a class named *UseCase3* is selected; it occurs and is marked yellow respectively in each revision except revision 1.2.

Tracing a Bug. A bug usually consists of more than just one model element. We assume that a bug is only present in another version (and should be fixed there) if the set of elements involved in the bug occurs as a whole with only small changes in the other version. Therefore, similarities of the element set in different versions as a whole are also computed and must be above an additional threshold for the other fragment to be considered as a repetition of the bug. In bug tracing analysis the selected bug elements are colored blue, while the bug candidates in the other revisions are colored regarding their probability to be a bug. Unconcerned elements are greyed out.

Finding dependencies. Dependency analysis, formerly known as logical coupling, provides information about software elements bearing a relation to each other although that relation is not obvious. These relations are based on the changes applied to the elements, e.g. whenever element *A* has been changed, element *B* was changed, too. Due to our fine-grained tracing we are able to provide this dependency information for each element within the model history. Once an element has been selected we are able to follow its track through the whole history. For each revision where the traced element has been changed, we check other elements that are also affected by modifications. Those elements are colored according to their degree of dependency.

Identifying Day Flies. Day flies are elements that occur only in one revision of the model history. They are usually a sign for models that are changed without taking a long view. Technically they can be identified very easily as they are not part of any track. In our visualization those elements are shown by brown circles. This provides a quick overview about model revisions containing day flies. The outline view depicts the elements themselves, if the user want to examine them in detail.

Evaluation

We evaluated our approach and its visualization methodologies in an empirical case study involving 30 developers. The probands had to perform different analysis problems on given model histories; first manually with a modeling tool of their choice and afterwards with help of our approach. Both times they had to fill in a questionaire asking for time exposure, experiences, etc.

First analysis covered a history of models with 25 to 30 classes unknown to the probands, which is typical for reverse engineer's work. Although the models are rather small for daily practice, the tool-assisted analysis provided significant enhancements. Summarized over the four analysis scenarios, time exposure was reduced by more than 75% in 75% of the cases. Beside time reduction, our approach computed all results correctly, while the probands produced erroneous results during manual analysis; we estimate an error rate of 30%. Furthermore, day flies were nearly impossible to be detected manually.

Half of the probands also analyzed models designed by themselves during a software development course. Despite the fairly good knowledge of their history 86% of the probands preferred the tool-assisted analysis; already models of 20 classes are too large to keep an overview. The latter group of probands allowed verification of the traces computed by our approach; all information has been judged to be correct as expected, since the correspondence analysis used for the trace computation has been tested intensively in the past.

Summarized over all scenarios the probands preferred essentially the tool solution; mainly explained by performance, trustworthy of results and simplicity. The visualization itself got good ratings for illustration and graphical controls; only the handling was not clearly rated to be intuitive. In general the benefit was considered very high; especially for bug tracing and dependency analysis which are hard to solve manually. The identification of day flies, however, was rated to be of limited usefullness.

References

1. U. Kelter, J. Wehren, and J. Niere. A generic difference algorithm for UML models. In *Proc. of SE 2005*, Essen, Germany, March 2005.

XML schema refactorings for X-to-O mappings — Extended abstract —*

Ralf Lämmel, Microsoft Corp., Redmond, USA

Extended abstract

Executive summary We deal with the overall problem of mapping XML schemas to object models, where the resulting object models are meant for programmatic, schema-aware access to XML data. The provision of such 'X-to-O mappings' involves various challenges; one of them is *variation in style of schema organization*. Without appropriate countermeasures, such variation affects the resulting object models in undue manner. We tackle this challenge by devising *transformations for normalizing styles of schema organization*, thereby improving the quality of X-to-O mapping results.

An illustrative scenario Consider the following XML tree of an order with two items; someone is ordering two copies of the XSD 1.1 standard as well as three copies of the Cobol 2008 standard:

```
<Order Id="4711">
  <Item Id="xsd-1.1-standard">
    <Price>42.88</Price>
    <Quantity>2</Quantity>
  </Item>
  <Item Id="cobol-2008-standard">
    <Price>37.99</Price>
    <Quantity>3</Quantity>
  </Item>
</Order>
```

A reasonable XML schema for orders may designate a single element declaration for orders with a nested element declaration for items; an item comprises a price and a quantity; cf. Fig. 1. *What sort of object model do we want for the given schema?* Let us assume that we readily committed to a straightforward mapping as follows: we map simple XML schema types to similar built-in types of the OO language at hand; we map repeating elements such as Item to list collections; we use 'classes with plain fields' for an OO representation of XML trees. *Then there is still the question of whether or not to preserve nesting.* Both mapping options are explored in Fig. 2. Both options have clearly their pros and cons, and hence it is useful to provide normalizations that can be requested for the automated derivation of object models.

Contributions of our work In addition to nesting of content models, there are a few further style issues

```
<!-- A global element declaration -->
<xs:element name="Order">
 <xs:complexType>
  <xs:sequence>
   <!-- A local element declaration -->
   <xs:element maxOccurs="unbounded" name="Item">
    <xs:complexType>
     <xs:sequence>
      <xs:element name="Price" type="xs:double"/>
      <xs:element name="Quantity" type="xs:int"/>
     </xs:sequence>
     <xs:attribute name="Id" type="xs:string" use="required"/>
    </xs:complexType>
   </xs:element>
  </xs:sequence>
  <xs:attribute name="Id" type="xs:int" use="required"/>
 </xs:complexType>
</xs:element>
```

Figure 1: The orders schema (in 'Russian Doll' style)

```
// Flat classes for orders and their items
public class Order
{
    public int            Id;
    public List<Item>     Item;
}
public class Item
{
    public string         Id;
    public double         Price;
    public int            Quantity;
}
```

```
// Nested classes for orders and their items
public class Order {
    public int                Id;
    public List<ItemType>     Item;
    public class ItemType     {
        public string         Id;
        public double         Price;
        public int            Quantity;
    }
}
```

Figure 2: Different object models for orders

that challenge X-to-O mappings in a similar manner. As a remedy, we devise a systematic software-transformation approach for style normalization. It turns out that some forms of style normalization (or conversion) cannot be accomplished by transformations at the level of XML schemas alone; instead, some amount of work must be delegated to the back-end of the X-to-O mapping so that the outgoing object models are transformed. Our transformational setup leverages functional OO programming, and abstract data types for the 'subjects under transformation'. In particular, we put to work C# 3.0 and LINQ. This form of a transformational setup should be of interest well beyond the theme of X-to-O mappings.

*A full paper, "Style normalization for canonical X-to-O mappings", appeared in the proceedings of ACM SIGPLAN 2007 Workshop on Partial Evaluation and Program Manipulation (PEPM '07).

Reengineering of State Machines in Telecommunication Systems

Christof Mosler
Department of Computer Science 3, RWTH Aachen University
Ahornstr. 55, D-52074 Aachen
christof.mosler@rwth-aachen.de

Abstract

In our reengineering project, we regard architecture modeling and reengineering techniques of telecommunication systems. Our work concerns analysis and restructuring of such systems, including the re-design and re-implementation. As telecommunication systems are often planned and modeled using state machines, it seems reasonable to analyze the systems also on the more abstract representation level of state machines. This paper presents a reengineering approach, in which we extract the state machines from PLEX source code, verify their consistency with the original specification documents, and use the information for further reengineering steps.

1 Introduction

In the E-CARES[1] research project, we study concepts, languages, methods, and tools for understanding and restructuring of complex legacy systems from the telecommunications domain. The project is a cooperation between Ericsson Eurolab Deutschland GmbH (EED) and Department of Computer Science 3, RWTH Aachen University. The current system under study is Ericsson's AXE10, a mobile-service switching center (MSC) comprising more than ten million lines of code written in PLEX (**P**rogramming **L**anguage for **EX**changes) [2]. This language was developed in about 1970 at Ericsson and is still in wide use within the company. PLEX systems are embedded real-time systems using the signaling paradigm, thus they pose additional requirements regarding fault tolerance, reliability, availability, and response time.

According to the model introduced by Byrne [1], a reengineering process comprises three phases: reverse engineering, modification, and forward engineering. All of them are supported by our reengineering tool environment (for details see [3]). In the reverse engineering phase, we use different sources of information. The most important and reliable one is the source code of the PLEX system. For representation of the legacy systems, we follow a graph-based approach, i.e. all underlying structures are graphs and editing operations are specified by graph transformation rules. We parse the source code and instantiate a graph describing the system structure, comprising its communication, control flow, and data flow. This *structure graph* consists of a main system node, subsystem nodes, block nodes, and the nodes for different program parts of each block (e.g. subroutines, signal entry points, data structures). The nodes are connected to each other by various kinds of edges (e.g. contains, goto, calls, from source, to target). On this graph, the user can perform various types of analyses by using different visualization and query techniques.

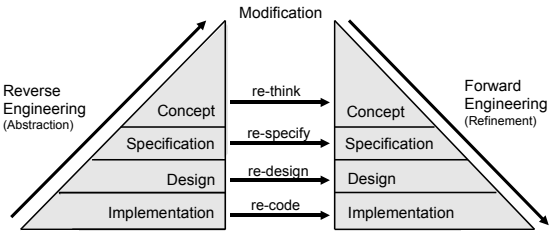


Figure 1: Reengineering Model by Byrne [1]

The tool supports also the modification phase by offering some complex algorithms for suggesting how to improve the legacy software. The user can interactively adapt the suggested transformations taking the semantics of the software into account or edit the graph manually. All modifications of the structure graph can later be propagated from the design level to the implementation level by using a TXL-based tool to generate the new source code.

2 Extraction and Modification of State Machines

When developing telecommunication systems, engineers often think in terms of state machines and protocols. Indeed, many parts of the AXE10 system were originally specified with state machines. We argue that reengineering can only be successful if this representation level is also considered. Up to now, all modification approaches in our project concerned the structure graph, corresponding to the design level of the model in figure 1. The extension presented in this paper focuses on the specification level, where parts

[1]The acronym E-CARES stands for **E**ricsson **C**ommunication **AR**chitecture for **E**mbedded **S**ystems.

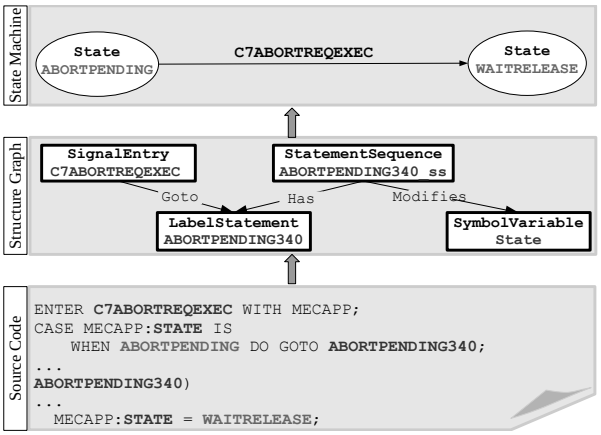


Figure 2: Extraction of State Machines

of the PLEX system are additionally represented as state machines. According to the model, we lift the abstraction level from design to specification level.

Generally, there is no explicit support for the implementation of state machines in PLEX. They are simulated by global enumeration variables whose possible values represent the possible states of the PLEX program during execution. Following the graph-based approach, our reengineering tool instantiates the state machines as independent graphs, allowing to apply different analyses and graph transformations. These *state machine graphs* are derived from the structure graph when certain patterns appear. Usually, these patterns comprise an assignment to the state variable, which indicates a transition in the state machine.

The example in figure 2 shows parts of the PLEX source code, the corresponding subgraph of the structure graph, and the extracted part of the state machine graph. The source code starts with an ENTER statement waiting for an incoming signal named C7ABORTREQEXEC. In the next line, a CASE statement on the STATE value of the received record MECAPP decides whether to jump to the statement sequence labeled ABORTPENDING340. Given the current block state ABORTPENDING, the new assignment of the STATE variable indicates the new state WAITRELEASE. The structure graph created for this code comprises all the information required for the state machine extraction. It is one of the major patterns used in our extraction algorithm. The extraction of the state machine starts at the signal entry points and looks for paths to statement sequences that change the state variable. For each assignment to the state variable there is a new transition created that is labeled with the signal name. The name of the source state is stored in the GOTO edge and the name of the target state is contained in the analyzed MODIFIES edge. The resulting state machine graph is shown in the upper part of the figure.

Before continuing developing concepts and tool extensions for state machine analysis, we wanted to guarantee the utility of the extracted information. We extracted the state machines of several PLEX blocks and by using a simple parser compared them automatically with the specification files we got from Ericsson. Our final results showed that the extracted state machines contained all states and all relevant transitions described in the specification documents (e.g. for one of the major system blocks comprising 17 states and 53 transitions). All occasionally emerging problems during the analysis originated in outdated versions of our PLEX files.

After the extraction, the additional information visible in the state machine graphs can be taken into account when re-designing the structure graph. Currently, our most important area of application for state machine analysis is merging and splitting of PLEX blocks. Merging of small blocks into one bigger block can reduce the number of exchanged signals and thus improve the system performance. After the merging, the new block usually implements several state machines which could also be merged in order to reduce the state space. Splitting a block into several independent blocks is usually done when the block reaches a size which makes maintenance too painful. If a block implements several state machines, the splitting of the structure graph should result in two blocks each containing one of the original state machines. If the block contains only one state machine, a preceding splitting of this state machine and the adaption of the structure graph (e.g. by providing a new state variable and dividing the state space) are required before splitting the structure graph.

3 Outlook

The introduced extraction algorithm has been successfully implemented. Also some basic re-design algorithms for the structure graph considering the state machines have been developed. Future work concerns the study of simple operations directly for the state machine graphs and the corresponding changes to be propagated to the structure graph. The goal is to prove generally that automatic source code generation after restructuring of state machine graphs is possible.

References

[1] Byrne, Eric J.: *A Conceptual Foundation of Software Re-engineering*. In *ICSM '92*, pages 226–235. IEEE Computer Society Press, 1992.

[2] Marburger, André: *Reverse Engineering of Complex Legacy Telecommunication Systems*. Shaker Verlag, Aachen, Germany, 2004. ISBN 3-8322-4154-X.

[3] Mosler, Christof: *E-CARES Project: Reengineering of Telecommunication Systems*. In Lämmel, R., J. Saraiva and J. Visser (editors): *GTTSE'05*, number 4143 in *LNCS*, pages 437–448, Braga, Portugal, 2006. Springer.

Benötigen wir einen „Certified Maintainer“?

Stefan Opferkuch
Abteilung Software Engineering, Institut für Softwaretechnologie
Universität Stuttgart

Zusammenfassung

Qualifizierungsnachweise („Zertifikate“) haben in der Software-Branche eine lange Tradition. Durch sie sollen anerkannte und standardisierte Qualifizierungen in verschiedenen Bereichen des Software-Engineerings geschaffen werden. Dieser Artikel untersucht die Vorteile einer solchen Qualifizierung für die Software-Wartung und umreißt, welche Inhalte eine Qualifizierung umfassen muss.

1 Einführung

In der Software-Branche existieren schon seit vielen Jahren Qualifizierungsnachweise in Form von Zertifikaten verschiedener Hard- und Software-Hersteller. So bieten SAP, Microsoft, IBM, Sun, Cisco, Red Hat und viele andere Unternehmen die Möglichkeit an, sich durch eine Schulung in speziellen Technologien der jeweiligen Unternehmen weiterzubilden und diese Weiterbildung durch ein Zertifikat zu belegen.

In den letzten Jahren werden auch in Technologie-neutralen Bereichen des Software-Engineerings zunehmend Zertifizierungen angeboten. Die wohl bekannteste ist der Certified Tester [1]. Weitere ähnliche Zertifizierungen existieren oder entstehen für die Gebiete Software-Architektur, Projektmanagement und Requirements Engineering.

Allen Zertifizierungsprogrammen ist dabei gleich, dass fundierte Grundlagen zu einem Gebiet des Software-Engineerings kompakt gelehrt werden sollen. Dieses Prinzip ist auch für weitere Gebiete denkbar. Es stellt sich also die Frage: Benötigen wir für die Software-Wartung einen „Certified Maintainer“, also eine nachweisbare Qualifizierung für die Wartung? Welche Vorteile würden sich daraus ergeben, und welche Inhalte sollten in einer solche Qualifizierung berücksichtigt werden?

2 Vorteile einer Qualifizierung für die Wartung

Um die Vorteile einer Qualifizierung in der Software-Wartung zu untersuchen, ist ein Modell der optimalen Qualifikationen eines Software-Warters hilfreich. Ein solches Modell ist in Abbildung 1 dargestellt. Die Basis der Qualifikationen bilden die Grundlagen des Software-Engineerings. Darauf aufbauend kennt der Software-Warter die Grundlagen der Wartung und verfügt auf oberster Ebene über Kenntnis des von ihm zu wartenden Objekts.

Wenn man dieses Modell daraufhin untersucht, wie die einzelnen Qualifikationen typisch erworben werden, so zeigt sich, dass die Grundlagen des Software-Engineerings durch eine Software-Engineering-Ausbildung erlernt werden. Die Kenntnis über die konkret zu wartende Software erlangt der Software-Warter im Unternehmen, entweder durch Unternehmens-interne Schulungen oder durch „learning by doing“. Die Grundlagen der Software-Wartung kommen dabei häufig zu kurz. In der Software-Engineering-Ausbildung wird die Software-Wartung als Themengebiet stark vernachlässigt und allenfalls oberflächlich besprochen. Beim Erlernen der Kenntnis über die zu wartende Software steht die konkrete Wartungssituation im Vordergrund. Der Fokus liegt also auf der speziellen Wartung, nicht auf den Grundlagen.

Eine separate, systematische Qualifizierung in den Grundlagen der Software-Wartung würde jedoch eine Reihe von Vorteilen mit sich bringen:

- In der Wartung qualifizierte Mitarbeiter können sich schneller die Kenntnisse und Handlungsweisen für die Wartung einer konkreten Software aneignen.
- Das Grundlagenwissen der Wartung ist von der Arbeit an einer konkreten Software unabhängig und lässt sich von einer Software auf eine andere übertragen.
- Es wird ein Nachweis über die Qualifikation für die Wartung geschaffen, die von der Wartung einer konkreten Software unabhängig ist.
- Durch die einheitlichen Vorgaben für die Durchführung der Wartung, wie sie in der Qualifizierung erlernt werden, wird die Qualität der Wartung erhöht.
- Die Organisation der Ausbildung wird durch die Möglichkeit von Wartungstrainings, in denen die Grundlagen der Software-Wartung mehreren Personen gelehrt werden können, vereinfacht.

Die Einführung einer nachweisbaren Qualifizierung für die Software-Wartung bildet also eine überaus sinnvolle Ergänzung zu den Grundlagen des Software-Engineerings und der Kenntnis der zu wartenden Software.

Im Folgenden werden die Inhalte einer solchen Qualifizierung untersucht.

3 Inhalte der Qualifizierung

Um die Inhalte der Qualifizierung für die Software-Wartung festzulegen, ist zu klären, welche einzelnen Qualifikationen für die Wartung benötigt werden.

Dazu stellen wir uns vor, dass ein Unternehmen einen Mitarbeiter für die Wartung einer Software sucht. Über welche Qualifikationen soll dieser verfügen?

- Um den Ablauf der Wartung zu verstehen, sollte der Mitarbeiter den *Wartungsprozess* kennen.

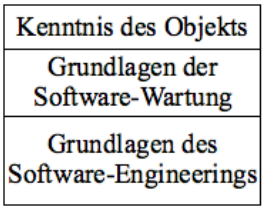


Abbildung 1: Qualifikationen eines Software-Warters

Er sollte nachvollziehen können, welche Phasen die Software-Wartung umfasst und welche Abhängigkeiten zwischen den Phasen bestehen.

- Da Software-Wartung immer die Arbeit an einer bestehenden Software ist, sollte der Mitarbeiter die Verwaltung dieser Software beherrschen, also den Umgang mit dem *Configuration Management*.
- Bei der Durchführung der Wartung ist zunächst das Problemverstehen und die Formulierung der Benutzerwünsche von zentraler Bedeutung. Der Mitarbeiter sollte sich also im Umgang mit *Wartungsanfragen* und im *Anforderungsmanagement* auskennen.
- Nach dem Problemverstehen folgt das Programmverstehen. Der Mitarbeiter sollte also in *Programmanalyse* und *Software-Architektur* geschult sein.
- Um Änderungen durchführen zu können, benötigt der Mitarbeiter Qualifikationen im *Software-Entwurf*, in der *Implementierung* und der *Dokumentation*.
- Zur Prüfung der Änderungen muss der Mitarbeiter über Fähigkeiten im *Software-Test*, speziell im Regressionstest und im funktionalen Test, hierbei insbesondere im Glass-Box-Test, und in der Durchführung von *Software-Inspektionen* verfügen.
- Für die Auslieferung der Software sollte der Mitarbeiter Kenntnisse im *Release-Management* besitzen.
- Schließlich sollte sich der Mitarbeiter mit *Werkzeugen* auskennen, die die verschiedenen Tätigkeiten der Software-Wartung unterstützen.

Aus dieser Auflistung lassen sich nun im Umkehrschluss Themengebiete einer Qualifizierung für die Software-Wartung ableiten:

- Wartungsprozesse
- Configuration Management
- Anforderungs- und Änderungsmanagement
- Software-Architektur und -Entwurf
- Software-Test und -Inspektion
- Release Management
- Wartungswerkzeuge

4 Schlussfolgerung und Ausblick

Durch den Certified Maintainer könnte eine standardisierte und transparente Aus- und Weiterbildung für die Software-Wartung entstehen. Es könnte zwischen Grundlagenwissen für die Wartung allgemein und Spezialwissen bezüglich der Wartung einer bestimmten Software unterschieden werden. Somit wäre z. B. denkbar, dass alle neuen Mitarbeiter, die in der Software-Wartung tätig sein sollen, eine Qualifizierung für die Wartung erwerben, um sich anschließend in eine spezielle Wartungsumgebung einzuarbeiten.

Erfahrenen Mitarbeitern böte sich durch den Qualifizierungsnachweis eine Möglichkeit, ihre Qualifikation bezüglich der Software-Wartung gegenüber ihrem aktuellen oder einem neuen Arbeitgeber zu belegen.

Bei der Planung der Qualifizierung für die Wartung gibt es einige Punkte zu beachten. Vor allem ist bei der Planung der Lehrinhalte auf die praktische Relevanz und die Umsetzbarkeit des Lehrstoffs zu achten. Ein durchgängiges Beispiel aus der Praxis als zentrales Element der Schulung wäre eine Möglichkeit der Gestaltung, um diesen Anforderungen gerecht zu werden.

Des Weiteren müssen die einzelnen inhaltlichen Themen für die Qualifizierung weiter ausgearbeitet und verfeinert werden. Dabei stellt sich die Frage, welche gesicherten Erkenntnisse über diese Themen vorliegen und wie die Erkenntnisse so aufbereitet werden können, dass sie in der Qualifizierung verwendet werden können.

Abschließend stellt sich die Frage nach dem didaktischen Gesamtkonzept, so dass die Inhalte optimal vermittelt werden können.

Literatur

[1] SPILLNER, ANDREAS und TILO LINZ: *Basiswissen Softwaretest: Aus- und Weiterbildung zum Certified Tester; Foundation Level nach ASQF- und ISTQB-Standard*. dpunkt-Verlag, 2. überarb. Auflage, 2004.

Three Static Architecture Compliance Checking Approaches – A Comparison[1]

Jens Knodel

Fraunhofer Institute for Experimental Software Engineering (IESE),
Fraunhofer-Platz 1, 67663 Kaiserslautern, Germany
knodel@iese.fraunhofer.de

Abstract

The software architecture is one of the most important artifacts created in the lifecycle of a software system. One instrument to determine how well an implementation conforms to the planned specification is architecture compliance checking. This paper compares three static architecture compliance checking approaches, namely Reflexion models, relation conformance rules, and component access rules.

Keywords: component access rules, architecture compliance checking, relation conformance rules, Reflexion model, SAVE, software architecture.

1. Introduction

The software architecture is one of the most crucial artifacts within the lifecycle of a software system. Decisions made at the architectural level have a direct impact on the achievement of business goals, functional and quality requirements. However, our experiences with industrial partners show that architectural descriptions are seldom maintained. On the contrary, architecture descriptions (if available at all) are often insufficient and/or incomplete. Late changes during implementation, conflicting quality requirements only resolved while implementing the system and technical development constraints often results in the fact that the implemented architecture is not compliant to the intended architecture, even for the first release of the software system. During the evolution of a system such effects intensify and the planned and the implemented architecture diverge even more. Thus, there is a high demand for assurance of architectural compliance.

2. Static Compliance Checking Approaches

The compliance of the architecture can be checked statically (i.e., without executing the code) and dynamically (at run-time). In this paper, we compare the three most popular approaches for static architecture compliance checking of a system:

- **Reflexion models**: Reflexion models [3] compare two models of a software system against each other, typically, an architectural model (the planned or intended architecture on high-level of abstraction) and a source code model (the actual or implemented architecture on a low-level of abstraction). The comparison requires a mapping between the two models, which is a human-based task.
- **Relation Conformance Rules**: Relation conformance rules (see [4], [1]) enable specifying allowed or forbidden relations between two components. The rules are specified by regular expression for the names of origin and the target of the components of the system. Relation conformance rules can detect similar defects as reflexion models, but the mapping is done automatically for each conformance check.
- **Component Access Rules**: Component access rules enable specifying simple ports for components, which other components are allowed to call. These rules help to increase the information hiding of components on a level, which might not be possible to achieve by the implementation language itself. Once a routine is public, it is accessible by the whole system, not only selected subset of it. Component Access Rules were inspired by ports in ADLs and exported packages in OSGi.

Static architecture evaluation of either one of the three approaches results in the assignment of a compliance types to each relation between two components. The compliance type includes c**onvergences** (relations allowed or implemented as intended), **divergences** (relations not allowed or not implemented as intended), and **absences** (relations intended but not implemented). We implemented all three static architecture evaluation approaches as features of the Fraunhofer SAVE tool (Software Architecture Visualization and Evaluation) and they will be compared in the next section.

[1] This work was performed in the project ArQuE (**Ar**chitecture-Centric **Qu**ality **E**ngineering) which is partially funded by the German ministry of Education and Research (BMBF) under grant number 01IS F14.

3. Comparison

Table 1 shows the comparison results for the three architecture compliance checking approaches. It assumes the general application case of the compliance checking approaches, however there might be exceptions. The table was derived based on our practical experiences gained in several industrial and academic case studies.

4. Conclusion

Architecture compliance checking is a sound instrument to detect architectural violations (i.e., deviations between the intended architecture and the implemented architecture). This paper presents a short comparison of three static compliance checking approaches (Reflexion models, relation conformance rules, and component access rules, please refer to [2] for a detailed comparison). The comparison supports the architects in their decision making on which compliance checking approach to apply based on their goals and context.

References

[1] Holt, R.C., (2005), "Permission Theory for Software Architecture Plus Phantom Architectures", School of Computer Science, University of Waterloo, September 2005.

[2] Knodel, J.; Popescu, D. "A Comparison of Architecture Compliance Checking Approaches". 6th IEEE/IFIP Working Conference on Software Architecture, Mumbai, India, 2007.

[3] Murphy, G.C.; Notkin, D. & Sullivan, K.J. (2001), "Software Reflexion Models: Bridging the Gap between Design and Implementation", IEEE Trans. Software Eng. 27(4), 364-380.

[4] van Ommering, R., Krikhaar, R., Feijs, L. (2001), "Languages for formalizing, visualizing and verifying software architectures", Computer Languages, Volume 27, (1/3): 3-18

Table 1 – Comparison of Static Architecture Compliance Checking Approaches

Dimension	Reflexion Model	Relation Conformance Rules	Component Access Rules
Input	source code, architecture	source code, (architecture)	source code, (component spec.)
Stakeholders	architect, (developer)	architect, (developer)	component engineer, (developer)
Manual Steps	• creating of the architectural model • mapping of architectural model elements to code • reviewing results	• creating and formalizing of relation conformance rules • reviewing results	• creating and formalizing of component access rules • reviewing results
Probability of False Positives	• low	• high	• low
Defect Types	• unintended dependencies • misusage of interfaces • (misuse of patterns) • violation of architectural styles	• unintended dependencies • misusage of interfaces • (misuse of patterns) • violation of architectural styles	• unintended dependencies • broken information hiding • misusage of interfaces
Transferability	• version: rework mapping • variant: partial refinement of architectural model and mapping • major restructuring: rework • different system: not possible	• version: no consequences • variant: no consequences • major restructuring: review rules • different system: yes, if naming conventions are applied	• version: no consequences • variant: no consequences • major restructuring: review rules • different system: yes, if component is reused
Ease of Application	• intuitiveness: high • iterative refinements: yes • learning curve: high	• intuitiveness: high • iterative refinements: limited • learning curve: medium	• intuitiveness: medium • iterative refinements: limited • learning curve: medium
Multiple View Support	• yes, each model mapping pair can capture a different view. • it is possible to have mapping that crosscut the decomposition hierarchy • commonalities, overlaps, and variation can be achieved by comparing the mappings	• yes, each rule set can capture a different view • no hierarchy crosscutting possible since the rules depend on the hierarchy	• no, multiple views are not supported • no hierarchy crosscutting possible since the rules depend on the hierarchy
Restructuring Scenarios	• supported by modifications to model and mapping pairs • monitoring and tracking possible	• no support for what-if analyses • monitoring and tracking possible	• no support for what-if analyses • no support for monitoring and tracking of restructurings
Architectural Decision Support	• limited	• limited	• limited

Referenzszenarien der Migration in Anwendungslandschaften

Johannes Willkomm, Dr. Markus Voß

sd&m Research
sd&m AG software design & management
Berliner Str. 76, 63065 Offenbach
johannes.willkomm@sdm.de, markus.voss@sdm.de

1. Einleitung

Die sd&m AG hat ihre Erfahrungen aus Dutzenden von Migrations-Projekten in ein Vorgehen zur Planung solcher Vorhaben gefasst. In [1] haben wir darüber bereits berichtet. Kern dieses Vorgehens ist eine systematische und auf klaren Konzepten beruhende Elimination von Freiheitsgraden zur schrittweisen Ableitung von Ziel-Architektur und Ziel-Migrationsszenario. Der Gesamtprozess ist in Abbildung 1 illustriert.

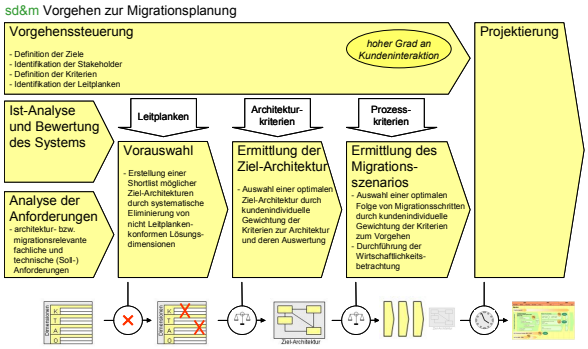


Abbildung 1: Migrationsplanung

In [2] haben wir über einige praxiserprobte Lösungsmuster aus den Bereichen Vorgehenssteuerung, Ist-Analyse und Ermittlung der Ziel-Architektur berichtet. In diesem Beitrag liegt der Schwerpunkt auf der Ermittlung des Migrations-Szenarios (Roadmap) unter Nutzung so genannter *Referenzszenarien*.

2. Langfristige Evolution der Anwendungslandschaft

Bei der Untersuchung von Migrationsszenarien ist es wesentlich, diese in den Gesamtkontext der langfristigen Evolution einer Anwendungslandschaft einzuordnen. Dabei hat es sich als praktikabel erwiesen, zwischen drei Ausprägungen der Anwendungslandschaft zu unterscheiden:

- Ist-Anwendungslandschaft: Im Rahmen der Ist-Analyse wird die Ist-Anwendungslandschaft erhoben und beschrieben. Sie ist über Jahre und Jahrzehnte gewachsen, meist ohne eine globale Planung. Daher finden sich in jeder Ist-Anwendungslandschaft Kompromisse und architektonische Irrtümer.
- Ideal-Anwendungslandschaft: Die Ideal-Anwendungslandschaft wird im Rahmen einer IT-Strategie konzipiert. Sie stellt eine auf Agilität, Effektivität und Effizienz optimal ausgerichtete Strukturierung der Komponenten einer Anwendungslandschaft dar und dient so als „Leuchtturm“ für alle zukünftigen Umbaumaßnahmen. Sie wird im Folgenden als gegeben vorausgesetzt.
- Soll-Anwendungslandschaft: Im Rahmen der Ermittlung des Migrationsszenarios wird die Soll-Anwendungslandschaft erstellt. Sie wird durch konkret geplante Projekte angestrebt.

Die folgende Abbildung illustriert diesen Zusammenhang.

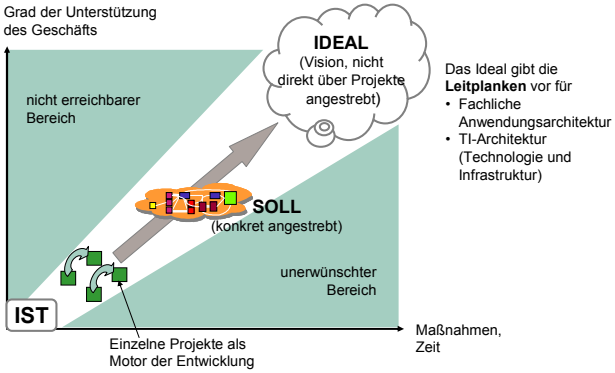


Abbildung 2: Anwendungslandschafts-Vorhaben

Im Folgenden werden Hintergründe zur Ermittlung von Soll-Anwendungslandschaft und Roadmap beschrieben.

3. Ermittlung von Soll-Anwendungslandschaft und Roadmap

Die Ermittlung von Soll-Anwendungslandschaft und Roadmap erfolgt in zwei Stufen. In der ersten Stufe, der Analyse, werden die wesentlichen Einflussfaktoren analysiert und gegeneinander abgewogen. In der zweiten Stufe, der Synthese, werden die notwendigen Schritte definiert sowie sinnvolle Stufen zur Umgestaltung identifiziert.

Im Rahmen der Analyse wird eine Delta-Betrachtung zwischen Ist und Ideal durchgeführt. Ziel der Analyse ist es, einen Teilbereich der Anwendungslandschaft zu identifizieren, deren Umgestaltung zuerst anzugehen ist. Teilbereiche können dabei beispielsweise eine oder mehrere Domänen der Anwendungslandschaft sein. Für die konkrete Auswahl dieser Bereiche steht eine Kosten-Nutzen-Betrachtung unter Berücksichtigung der Architekturkriterien im Vordergrund. Ergebnis der Analyse ist die Soll-Anwendungslandschaft, die man als den Bebauungsplan der Anwendungslandschaft mit einer Soll-Architektur der in den nächsten 3-5 Jahren umzugestaltenden Flächen verstehen kann.

Ziel der Synthese ist es, den Prozess zur Umgestaltung der Ist-Anwendungslandschaft durch Schritte und Stufen hin zum Soll zu detaillieren. Das Ergebnis der Synthese

ist eine Roadmap. Hierbei werden zunächst auf der Basis von Referenzszenarien die notwendigen Schritte, d.h. die durchzuführenden Teilaufgaben identifiziert. Die Schritte werden dabei unter Berücksichtigung der Effizienz in eine Reihenfolge gebracht. Danach gilt es sinnvolle Stufen zu finden. Stufe bedeutet hierbei die Produktivsetzung einer vorgenommenen Änderung der Anwendungslandschaft. Stufen fassen hierbei mehrere Schritte aus Kostengründen zusammen oder können einen Schritt aus Risikogründen in mehrere Stufen unterteilen. Die Folgende Grafik stellt diesen Sachverhalt dar.

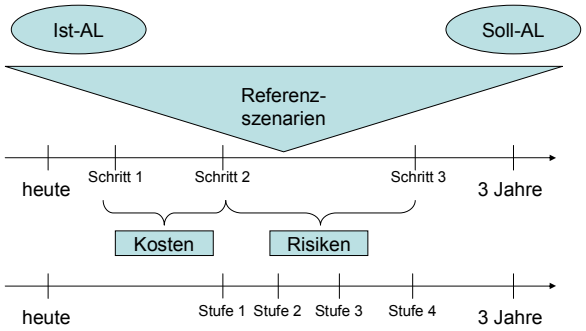


Abbildung 3: Zweistufiger Prozesse – Schritte und Stufen

Die Referenzszenarien kondensieren die Erfahrungen von sd&m aus Anwendungslandschafts-Migrationsprojekten. Sie manifestieren sich in Form einer Reihe von aufeinander aufbauenden Fragen, die in der sukzessiven Beantwortung zur Roadmap führen. Erster Schritt dabei ist die Frage nach der Notwendigkeit zur frühen Migration von Bestandskomponenten (vgl. auch [3]):

- **Bestandskomponenten zuerst migrieren?** Die Erstellung der im Soll vorgesehenen Bestandskomponenten (vgl. [4]) ist in vielen Fällen in einem ersten Schritt notwendig. Damit wird eine nachhaltige Basis gelegt, um die im Soll vorgesehenen Funktions- und Prozesskomponenten solide zu implementieren. Ob dieser erste Schritt am Anfang durchzuführen ist, hängt davon ab, ob Erweiterungen und Neustrukturierungen an der Datenstruktur der Bestände notwendig sind. Das ist in der Regel der Fall.
  Beispiel: In einem Projekt für einen Baustoffhändler war der Prozess zur Vergabe von Warenkrediten zu optimieren. Die Partner-Struktur der betroffenen Systeme gestattete keine einheitliche Sicht auf den Geschäftspartner. Aus diesem Grund wurde in einem ersten Schritt eine zentrale Partnerverwaltung geschaffen. Die Pflege von Partnerdaten fand nun ausschließlich über diese Bestandskomponente statt. Diverse Altsysteme wurden mit diesen Partnerdaten versorgt. Erst dann wurden nächste Schritte in der Optimierung der Kreditvergabe angegangen.

Nach dem Schritt „Bestandskomponenten zuerst?“ erfolgt eine Identifizierung der Schritte auf Basis von fachlichen Gesichtspunkten. Bewährt haben sich hierbei folgende Alternativen:

- **Randfunktionen vor Kernfunktionen migrieren?** In historisch gewachsenen Anwendungslandschaften sind Randfunktionen meist in größeren Anwendungssystemen versteckt oder durch individuelle Komponenten abgedeckt. Bewährt hat sich, die Randfunktionen zunächst durch standardisierte Lösungen herauszulösen, bevor die Kernfunktionen angegangen werden.
  Beispiel: Bei einem Pharmagroßhändler wurde nach der Einführung zentraler Bestandskomponenten zunächst Randfunktionen wie Lageroptimierung und Reportingfunktionen durch Standardsoftware abgelöst, bevor der eigentliche Kernprozess angegangen wurde.
- **Entlang der Wertschöpfungskette migrieren?** Manchmal ist es sinnvoll, die Funktionen entlang der Wertschöpfungskette sukzessive herauszulösen. Hierbei kann die Wertschöpfungskette von vorne oder von hinten „aufgerollt“ werden.
  Beispiel: In einem Bankenprojekt wurde nach Herauslösung der Komponenten zur Verwaltung von Bankleitzahlen (Bestandskomponenten zuerst) zunächst die Clearing-Annahme, dann die Clearing-Verarbeitung und letztendlich die Versendung von Clearing-Informationen umgestaltet.
- **Anhand der Prioritäten aus dem Geschäft migrieren?** Bei nur geringer Verzahnung der abzulösenden Funktionalitäten kann auch eine sukzessive Herauslösung anhand der geschäftlichen Prioritäten erfolgen.
  Beispiel: In einem Projekt für eine Behörde wurde nach Prüfung der Wohlstrukturiertheit der Bestandskomponenten Such- und Pflegefunktionen zu Personendaten abhängig von der Dringlichkeit ihrer Ablösung sukzessive aus dem Altsystem herausgelöst.

Wurden auf diese Weise die notwendigen Schritte und deren Reihenfolge identifiziert, gilt es dann die Stufen zu finden. Die Zusammenfassung von mehreren Schritten zu einer Stufe ist trivial. Bei der Unterteilung eines Schrittes in mehrere Stufen spielt vor allem eine Strategie eine Rolle, die immer wieder auftritt:

- **Mit fachlicher Partitionierung migrieren?** Hierbei wird ein Schnitt durch die Datenbereiche gemacht und eine Stufe beispielsweise mandantenweise umgesetzt.
  Beispiel: Ein Reiseanbieter führte sein neues Regulierungssystem zur Risikominimierung gestuft nach Zielgebieten ein (Mallorca zuerst).

Literatur

[1] Voß, M.: *Synthese eines Vorgehens zur Migrationsplanung*. Workshop Software-Reengineering (WSR 2006), GI Softwaretechnik-Trends, Band 26, Heft 2, 2006

[2] Boos, J., Voß, M., Willkomm, J., Zamperoni, A.: *Lösungsmuster in der Planung industrieller Migrationsprojekte*. Workshop Reengineering-Prozesse (RePro 2006), GI Softwaretechnik-Trends, Band 27, Heft 1, 2007

[3] Schaumann, P.: *Altsysteme von außen nach innen ablösen*. IT Fokus 7/8, 2004

[4] Humm, B., Hess, A., Voß, M.: *Regeln für serviceorientierte Architekturen hoher Qualität*. Informatik-Spektrum 6/2006, Springer Verlag, 2006

Konvertierung der Jobsteuerung am Beispiel einer BS2000-Migration

Uwe Erdmenger, Denis Uhlig
pro et con Innovative Informatikanwendungen GmbH, Annaberger Straße 240, 09125 Chemnitz
Uwe.Erdmenger@proetcon.de, Denis.Uhlig@proetcon.de

Software-Migration beschränkt sich nicht nur auf die Konvertierung von Programmen, Bildschirmmasken und Dateien/Datenbanken. Zu einem produktiven System gehören auch Skripten bzw. Jobs zur Ablauforganisation und Protokollierung.

Der vorliegende Beitrag beschreibt eine toolgestützte Konvertierung der unter BS2000 eingesetzten Jobsteuersprache **SDF (System-Dialog-Facility)** nach **Perl**. Es werden daraus Schlußfolgerungen zur toolgestützten Konvertierung von Jobsteuersprachen allgemein gezogen.

1 Eigenschaften der Quellsprachen

Sprachen zur Jobsteuerung unterscheiden sich in einigen wesentlichen Eigenschaften von Programmiersprachen. Die wichtigsten Unterschiede sollen in den folgenden Punkten erläutert werden.

Syntaktische Komplexität: Jobs werden meist interpretativ abgearbeitet. Dabei wird typischerweise eine befehlsweise Abarbeitung durchgeführt. Dadurch ist die syntaktische Komplexität geringer als bei üblichen Programmiersprachen.

Abhängigkeit vom Betriebssystem: In Jobs wird ungekapselt auf Funktionen des Betriebssystems zugegriffen (keine Bibliotheken, wie bei Programmiersprachen üblich). Ein Beispiel ist bei SDF die direkte Nutzung von Jobvariablen zur Inter-Prozess-Kommunikation.

Einbettung externer Programme: Da Jobs häufig zur Automatisierung von Abläufen eingesetzt werden, besteht die Notwendigkeit, externe Programme in die Jobs einzubetten, um sie zu steuern und um Ergebnisse zu kontrollieren. Eingebettet sein können z.B. Systemprogramme bzw. Utilities wie SORT oder EDT, oder Aufrufe und Parameter von Programmen.

2 Wahl der Zielsprache

Die genannten Eigenschaften beeinflussen die Auswahl der Zielsprache. Dafür sind folgende Kriterien wesentlich:

Sprachumfang: Die Zielsprache soll möglichst viele der im ersten Abschnitt genannten Eigenschaften unterstützen. Alle nicht unterstützten Funktionalitäten müssen in der Zielsprache nachgebildet werden können.

Erweiterbarkeit: Es muss möglich sein, die Zielsprache durch Bibliotheken zu erweitern. Auf diese Weise ist es einfach und effizient möglich, für die konvertierten Jobs Funktionen, welche die Zielsprache nicht bietet, verfügbar zu machen. Des Weiteren sollen Module für die zukünftige Weiterentwicklung zur Verfügung stehen (z. B. Datenbankanbindung).

Portabilität: Die Zielsprache soll auf mehreren Hardwareplattformen zur Verfügung stehen, um zukünftige Migrationen zu erleichtern.

Für die Konvertierung von SDF fiel die Wahl auf Perl als Zielsprache. Perl ist ebenso eine interpretative Sprache, welche die wesentlichen Eigenschaften von Jobsteuersprachen bietet (Variablensubstitution, Einbettung externer Programme, ...). Perl ist auf den verschiedensten Plattformen verfügbar. Damit ist auch das Kriterium der Portabilität erfüllt. Auch die Erweiterbarkeit ist gegeben. Perl kann über spezielle Schnittstellen auf Bibliotheken und eine Vielzahl von Modulen im Comprehensive Perl Archive Network (CPAN) zurückgreifen. Ebenso erfüllt der Sprachumfang die geforderten Kriterien.

3 Werkzeug

Für die Konvertierung entwickelte pro et con ein Werkzeug **SDF-to-Perl-Translator (S2P)**. S2P besteht aus drei Komponenten. Die erste Komponente *Parse* führt eine Syntaxanalyse durch, deren Ergebnis ein Syntaxbaum der SDF-Prozedur ist. Dieser ist mittels verschachtelter Hashes und Arrays implementiert. Dieser Baum wird anschließend in ein Repository abgelegt, um den Aufwand für zukünftige Schritte zu minimieren.

Die zweite Komponente *Analyse* erlaubt Analysen über die SDF-Prozeduren. Dabei kann die Analysebasis eine einzelne Prozedur (z.B. Analyse von Befehlen oder Parameterlisten) oder auch das gesamte Repository sein. Diese Batch-Analysen erstellen z. B. die Aufruf-Hierarchie oder führen eine Suche nach Clones, also ähnlichen Prozeduren, durch. Die Ergebnisse der Analysen werden als .csv-Dateien oder als HTML-Dokument angeboten.

Die dritte Komponente *Convert* realisiert die eigentliche Übersetzung der SDF-Prozedur. Bei der Konvertierung der sequentiell abgearbeiteten Befehle werden zwei Typen unterschieden: Befehle, die ein Äquivalent in Perl besitzen, werden durch diese abgebildet. So wird aus dem Sprungbefehl `SKIP-COMMANDS` ein `goto` in der generierten Perl-Funktion. Alle anderen Befehle ohne Perl-Äquivalent erfordern eine Nachbehandlung. Bei der Konvertierung wird der Name in Kleinbuchstaben gewandelt und die Parameterübergabe wird in eine für Perl typische Notation

überführt. Die Implementierung dieser Befehle erfolgt in einem Laufzeitsystem. Das folgende Beispiel zeigt eine solche Konvertierung.
Original:

```
/  ADD-FILE-LINK LINK-NAME=SORTOUT -
/                FILE-NAME=I.TST.DAT
```

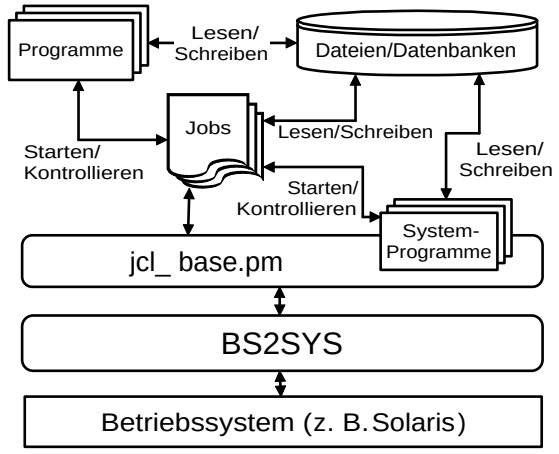
generiertes Perl:

```
add_file_link (LINK_NAME => 'SORT_OUT',
               FILE_NAME => 'I.TST.DAT');
```

Das Ergebnis der Komponente *Convert* ist eine zur SDF-Prozedur semantisch äquivalente Perl-Funktion, die alle konvertierbaren SDF-Befehle beinhaltet. Trotzdem bleibt immer ein gewisser Teil Handarbeit. Nicht konvertierbare SDF-Kommandos werden als Perl-Kommentar in den Zielcode übernommen, um diese manuelle Arbeit zu erleichtern.

4 Laufzeitsystem

Zur Ausführung der konvertierten Skripten wird zusätzlich ein Laufzeitsystem benötigt, welches die von Perl nicht unterstützen Befehle und Funktionalitäten nachbildet. Beim Laufzeitsystem der aus SDF generierten Perl-Prozeduren wurde ein zweischichtiger Aufbau gewählt (Vgl. Abbildung). Die systemnahen Funktionalitäten sind in der Komponente BS2SYS konzentriert. BS2SYS ist eine in C implementierte Bibliothek, die als Shared Object bereitgestellt wird. Eine Funktion aus dieser Bibliothek ist z.B. die Konvertierung von Jobvariablen und Dateien.



Die übrige Funktionaliät des Laufzeitsystems wird im Modul jcl_base.pm in Perl realisiert.

Der Quellcode des Laufzeitsystems wurde in maschinell auswertbarer Form kommentiert. Das Werkzeug **PerlDoc2** von pro et con erstellt daraus eine Nutzerdokumentation in HTML.

5 Konvertierungsbeispiel

Das folgende Beispiel demonstriert einen Fall, bei dem eine SDF-Funktionalität in Perl nachgebildet wurde.

SDF verfügt über eine spezielle Technik, um auf Fehler zu reagieren. Schlägt ein Kommando fehl, wird ein so genannter Spin-Off ausgelöst. Dieser kann als eine Ausnahme (Exception) betrachtet werden. Das bedeutet, es werden nach einem Fehler alle Folgebefehle übersprungen. Gestoppt wird der Spin-Off erst, wenn ein Befehl erreicht wird, der als Aufsetzpunkt dienen kann (z. B. `SET-JOB-STEP`). Das nachfolgende Beispiel zeigt einen solchen Fall:

```
/ REMARK ******************************************
/ REMARK TEST AUF EXISTENZ DER DATEI I.TST.DAT
/ REMARK ******************************************
/ .FILETEST SHOW-FILE-ATTRIBUTES FILE-NAME=I.TST.DAT
/      WRITE-TEXT 'FILE I.TST.DAT IST VORHANDEN'
/      SKIP-COMMANDS TO-LABEL=WEITER
/      SET-JOB-STEP
/      WRITE-TEXT 'FILE I.TST.DAT IST NICHT VORHANDEN'
/ .WEITER
```

Der Befehl `SHOW-FILE-ATTRIBUTES` gibt die Eigenschaften der im Parameter `FILE-NAME` angegebenen Datei aus. Ist die Datei vorhanden und der Befehl damit erfolgreich, wird die nachfolgende Textausgabe ausgeführt und mittels `SKIP-COMMANDS` die Fehlerbehandlung übersprungen. Ist die Datei allerdings nicht vorhanden, werden alle Befehle bis zu `SET-JOB-STEP` übersprungen. Im Beispiel wird einfach eine Meldung ausgegeben und die Verarbeitung wird fortgesetzt.

Eine solche Technik existiert in Perl nicht und wurde deshalb wie folgt emuliert: Im ersten Schritt werden alle Aufsetzpunkte im Job ermittelt und mit einer Sprungmarke versehen, sofern sie noch keine besitzen. Zusätzlich muss nun an jeder Sprungmarke ein Befehl eingefügt werden, der im Laufzeitsystem den nächsten Aufsetzpunkt für den Spin-Off setzt. Wird nun durch einen Befehl eine Ausnahme ausgelöst, ermittelt das Laufzeitsystem den nächsten Aufsetzpunkt und springt zur entsprechenden Sprungmarke. Der resultierende Perl-Text sieht dann wie folgt aus:

```
REMARK("******************************************");
REMARK("TEST AUF EXISTENZ DER DATEI I.TST.DAT     ");
REMARK("******************************************");
FILETEST:
   NEXT_STEP('SPIN_OFF_LAB_1');
   show_file_attributes(FILE_NAME=>'I.TST.DAT');
   write_text('FILE I.TST.DAT IST VORHANDEN');
   goto WEITER;
SPIN_OFF_LAB_1:
   NEXT_STEP('SPIN_OFF_LAB_2');
   write_text('FILE I.TST.DAT IST NICHT VORHANDEN');
WEITER:
   NEXT_STEP('SPIN_OFF_LAB_2');
```

Wie beschrieben, wird mittels `NEXT_STEP` der nächste Aufsetzpunkt bekannt gemacht. Tritt in der Perl-Variante in dem im Laufzeitsystem nachgebildeten Befehl `show_file_attributes` ein Fehler auf, wird das zuletzt registrierte Spin-Off-Label ermittelt (in diesem Fall `SPIN_OFF_LAB_1`) und angesprungen. Ist dieses erreicht, wird der nächste Aufsetztpunkt gesetzt und die Fehlerbehandlung durchgeführt.

6 Zusammenfassung

S2P wurde erfolgreich in einem BS2000-Migrationsprojekt eingesetzt. Unsere praktischen Erfahrungen besagen, dass es möglich und sinnvoll ist, nicht nur Programme und Daten, sondern auch die Jobs (teil-)automatisch mit Toolunterstützung zu migrieren.

Reengineering von Software-Komponenten zur Vorhersage von Dienstgüte-Eigenschaften

Klaus Krogmann
Lehrstuhl für Software-Entwurf und -Qualität
Institut für Programmstrukturen und Datenorganisation
Universität Karlsruhe (TH)
krogmann@ipd.uka.de

1 Einleitung

Die Verwendung von Komponenten ist ein anerkanntes Prinzip in der Software-Entwicklung. Dabei werden Software-Komponenten zumeist als *Black-Boxes* aufgefasst [1], deren Interna vor einem Komponenten-Verwender verborgen sind. Zahlreiche Architektur-Analyse-Verfahren, insbesondere solche zur Vorhersage von nicht-funktionalen Eigenschaften, benötigen jedoch Informationen über Interna (bspw. die Anzahl abgearbeiteter Schleifen oder Aufrufe externer Dienste), die von den vielen Komponentenmodellen nicht angeboten werden.

Für Forscher, die aktuell mit der Analyse nicht-funktionaler Eigenschaften von komponentenbasierten Software-Architekturen beschäftigt sind, stellt sich die Frage, wie sie an dieses Wissen über Komponenten-Interna gelangen. Dabei müssen existierende Software-Komponenten analysiert werden, um die benötigten Informationen über das Innere der Komponenten derart zu rekonstruieren, dass sie für anschließende Analyse-Verfahren nicht-funktionaler Eigenschaften genutzt werden können. Bestehende Verfahren konzentrieren sich auf die Erkennung von Komponenten oder bspw. das Reengineering von Sequenzdiagrammen gegebener Komponenten, fokussieren aber nicht auf die Informationen, die von Vorhersageverfahren für nicht-funktionale Eigenschaften benötigt werden (vgl. Abschnitt 2).

Der Beitrag dieses Papiers ist eine genaue Betrachtung der Informationen, die das Reengineering von Komponenten-Interna liefern muss, um für die Vorhersage der nicht-funktionalen Eigenschaft Performanz (im Sinne von Antwortzeit) nutzbringend zu sein. Dazu wird das Palladio Komponentenmodell [2] vorgestellt, das genau für diese Informationen vorbereitet ist. Schließlich wird ein Reengineering-Ansatz vorgestellt, der dazu geeignet ist, die benötigten Informationen zu gewinnen.

2 Grundlagen

Um sinnvolle Analysen der nicht-funktionalen Eigenschaften einer Komponente zu ermöglichen, werden Informationen über die Interna einer Komponente benötigt. In Abbildung 1 werden die Interna als Abhängigkeiten zwischen einer angebotenen und benötigten Schnittstelle einer Komponente dargestellt. So kann der Aufruf von angebotenen Diensten

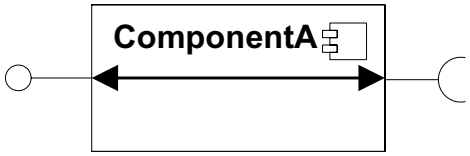


Abbildung 1: Abhängigkeiten zwischen angebotener und benötigter Schnittstelle

der Komponente (links) dazu führen, dass auch Dienste auf der benötigten Schnittstelle der Komponente (rechts) aufgerufen werden. Abhängig davon, wie der benötigte Dienst von einer externen Komponente bereitgestellt wird (beispielsweise niedrige oder hohe Ausführungszeit), verändert sich auch die vom Benutzer wahrgenommene Antwortzeit der betrachteten Komponente [3].

Um diese grob skizzierten Abhängigkeiten in ein Modell abzubilden, das die Grundlage für die Vorhersage nicht-funktionaler Eigenschaften einer Komponenten-Architektur ist, wurde das Palladio Komponentenmodell (PCM) geschaffen. Das PCM ist eine domänenspezifische Sprache, für die Analyse- und Simulations-Methoden [4, 2] existieren, die zur Vorhersage von Antwortzeiten, Ressourcenauslastung, usw. dienen. Zur Definition von Abhängigkeiten zwischen angebotenen und benötigten Diensten sind dabei die sogenannten Dienst Effekt Spezifikationen (SEFF) essentiell. SEFFs beschreiben das innere Verhalten einer Komponente in Form von Kontroll- und Datenfluss. Das Ziel eines SEFFs ist dabei, eine *Abstraktion* des inneren Verhaltens zu bieten.

In SEFFs wird grundsätzlich zwischen komponenteninternem Verhalten und externen Dienstaufrufen unterschieden. Darüber hinaus existieren Kontrollflusskonstrukte für Schleifen, Verzweigungen, Ressourcen-Aquisition und -Freigabe (insgesamt als „Aktionen“ bezeichnet). *Parametrisierung und parametrische Abhängigkeiten* erlauben es Abhängigkeiten zwischen Eingabeparametern eines Dienstes und Schleifenausführungen, Verzweigungswahrscheinlichkeiten, Parametern von externen Dienstaufrufen, usw. zu definieren. Die Erfassung von *Ressourcennutzung* (bspw. CPU-Einheiten, Festplattenzugriffe) ermöglicht eine Abbildung auf Hardware. Verzweigungswahrscheinlichkeiten und Schleifenausführungszahlen können

darüber hinaus auch *stochastisch* beschrieben werden – bspw. wenn die Angabe einer parametrischen Abhängigkeit hochkomplex und berechnungsintensiv würde.

Bislang existieren keine automatisierten Reengineering Ansätze, die alle benötigten Daten von SEFFs aus bestehenden Komponenten gewinnen – manuelle Ansätze scheiden aus Aufwandsgründen aus.

3 Reengineering Ansatz

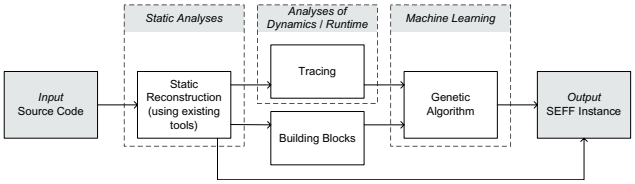


Abbildung 2: Angestrebter Reengineering Prozess

Wie im vorangehenden Kapitel beschrieben wurde, sind zahlreiche Daten für ein Modell der Komponenteninterna zur Vorhersage nicht-funktionaler Eigenschaften zu erfassen. Das Ziel eines solchen Modells ist es nicht, existierenden Quellcode möglichst genau abzubilden, sondern eine Abstraktion zu schaffen, die nur noch Parameter und parametrischen Abhängigkeiten usw. enthält, die für die Vorhersageverfahren signifikant sind.

Zu diesem Zwecke wollen wir statische Analyse, wie sie von existierenden Werkzeugen unterstützt wird, mit der Laufzeitanalyse in Form von Tracing kombinieren. Die aus beiden Schritten resultierenden Daten werden einem genetischen Algorithmus [5] zugeführt. Die Aufgabe des genetischen Algorithmus' ist das Auffinden einer Abstraktion parametrischer Abhängigkeiten. Dabei werden insbesondere dienstgüte-invariante Parameter herausgefiltert und funktionale Abhängigkeiten vereinfacht. In Abbildung 2 wird der vorgeschlagene Reengineering Prozess skizziert.

Der Reengineering-Vorgang beginnt mit der Eingabe von Quellcode, das Ziel der Vorgangs ist eine Modellinstanz eines SEFFs. Die statische Analyse liefert den Kontrollfluß der SEFFs. Zugleich wird eine für das Tracing benötigte Instrumentierung auf Basis der statischen Struktur vorgenommen. Ziel ist es hierbei, *gezielt* Laufzeitdaten zu erheben, um die anfallenden Datenmengen zu reduzieren. Daneben liefert die statische Analyse Bausteine – den initialen Genom – für den genetischen Algorithmus. Dies kann beispielsweise eine lineare Funktion für die Ausführungsdauer einer `for`-Schleife sein.

Die im Tracing-Schritt erhobenen und bereinigten Daten werden als Zielfunktion des genetischen Algorithmus' verwendet, um den Genom zu verbessern. Dabei muss der *Trade-Off* zwischen einfacher Berechenbarkeit der funktionalen Abhängigkeiten und die bereinstimmung mit den Tracing-Daten gelöst werden.

4 Fazit

Der angestrebte Mischung aus statischem und dynamischem Reengineering versucht die Vorteile beider Varianten zu vereinen. Die statische Analyse liefert wichtige Informationen für die Kontrollflussstrukturen des SEFFs, wohingegen die dynamische Analyse (Tracing) typische Probleme, wie die Bestimmung komplexer Schleifenabbruchbedingungen, vereinfacht.

Das Verhalten von Komponenten wird für die dynamische Analyse zur Laufzeit beobachtet. Daher ist es notwendig, dass Testdaten in die beobachteten Komponenten eingegeben werden. Dabei werden die gleichen Probleme wie beim Testen von Software auftreten: es kann niemals der gesamte Eingaberaum in endlicher Zeit abgedeckt werden. Entsprechend sind aussagekräftige Testdaten eine Grundvoraussetzung um sinnvolle Ergebnisse zu erlangen.

Zukünftige Arbeiten werden sich mit der Implementierung und Evaluation der vorgestellten Verfahrens-Idee beschäftigen.

Literatur

[1] Szyperski, C., Gruntz, D., Murer, S.: Component Software: Beyond Object-Oriented Programming. 2 edn. ACM Press and Addison-Wesley, New York, NY (2002)

[2] Becker, S., Koziolek, H., Reussner, R.: Model-based performance prediction with the palladio component model. In: Workshop on Software and Performance (WOSP2007), ACM Sigsoft (2007)

[3] Reussner, R.H., Schmidt, H.W.: Using Parameterised Contracts to Predict Properties of Component Based Software Architectures. In Crnkovic, I., Larsson, S., Stafford, J., eds.: Workshop On Component-Based Software Engineering (in association with 9th IEEE Conference and Workshops on Engineering of Computer-Based Systems), Lund, Sweden, 2002. (2002)

[4] Koziolek, H., Firus, V.: Parametric Performance Contracts: Non-Markovian Loop Modelling and an Experimental Evaluation. In: Proceedings of FESCA2006. Electronical Notes in Computer Science (ENTCS) (2006)

[5] Koza, J.R.: Genetic Programming – On the Programming of Computers by Means of Natural Selection. third edn. The MIT Press, Cambridge, Massachusetts (1993)

Statische Code-Analyse als effiziente Qualitätssicherungsmaßnahme im Umfeld eingebetteter Systeme

Dr. Daniel Simon (SQS AG), daniel.simon@sqs.de
Dr. Frank Simon (SQS AG), frank.simon@sqs.de

1 Einleitung

In diesem Papier werden Erfahrungen aus gemeinsamen Projekten des Kunden und der SQS bei der Durchführung proaktiver Qualitätssicherungsmaßnahmen im Umfeld von eingebetteten Telekommunikationssystemen erläutert. Dabei werden insbesondere die Vorteile von frühzeitig ergreifbaren Qualitätssicherungsmaßnahmen mit Hilfe statischer Code-Analysewerkzeuge sowie deren ungefähre Quantifizierung erkennbar.

1.1 Projektkontext

Der Kunde entwickelt Kommunikations- und Informationssysteme für sicherheitskritische Anwendungen. Mit einem hohen Marktanteil im Bereich der Sprachvermittlungssysteme für die Flugsicherung hat sich das Unternehmen zum Weltmarktführer in diesem Segment entwickelt. Darüber hinaus findet die Technik des Kunden bei Command & Control Systemen (Rettung, Polizei, Feuerwehr und Schifffahrt), im TETRA-Mobilfunk sowie bei Eisenbahnen und im öffentlichen Verkehr Anwendung. Hohe Ausfallsicherheit, ein rascher Verbindungsaufbau sowie Flexibilität und Vernetzbarkeit beim Einsatz kennzeichnen die Produkte des Kunden.

Die SQS Gruppe (SQS) ist der größte unabhängige Anbieter für Software-Test- und Qualitätsmanagement-Dienstleistungen in Europa. Für sehr dezidierte Services existieren sogenannte Kompetenzzentren, die ihre Dienste von zentraler Stelle in die Projekte tragen. Der im vorliegenden Fall betroffene Aspekt „Statische Analyse“ wird hierbei durch das 16 Personen starke Kompetenzzentrum Code-Quality-Management betreut.

2 Aufgabenstellung

An die Kommunikations- und Informationssysteme, die der Kunde entwickelt, werden besondere Ansprüche in Hinsicht auf Zuverlässigkeit und Verfügbarkeit bei zugleich limitierten Hardware-Ressourcen gestellt. Die Systeme bestehen zum Teil aus speziell für die jeweilige Anwendung entwickelten Hardwarekomponenten und entsprechender Software, die eigens zur Kontrolle und Steuerung dieser Hardware maßgeschneidert implementiert und gewartet werden muss. Dabei reicht die Palette der Softwarearten von einfachen Hardwaretreibern für Standardbetriebssysteme bis hin zu komplexen Benutzerschnittstellen für Endanwender.

Das Gesamtsystem wird vor der Abnahme durch den Kunden intensiv getestet und auf seine Funktionsfähigkeit hin untersucht. Die Tests der Gesamtsysteme sind oftmals sehr aufwändig, da für entwickelte Endgeräte das Einsatzumfeld beim Kunden aus Kostengründen oder aus technologischen Gründen nur schwer oder gar nicht vor Ort aufgebaut werden kann.

Durch den Einsatz von zusätzlichen statischen Analysetechniken soll daher bereits im Vorfeld der durchzuführenden dynamischen Tests die Fehlerrate verringert werden, um damit den Ablauf der Tests durch geringere Fehlerraten zu beschleunigen. Die vor der Durchführung der Qualitätssicherungsmaßnahmen aufgestellte Hypothese lautet, dass auch in der Praxis durch Qualitätsaudits mittels statischer Code-Analysen Laufzeitfehler vorab identifiziert und beseitigt werden können, die andernfalls nur durch (möglicherweise intensives) Testen gefunden werden. Damit sollten die Projekte insgesamt von einem reduzierten Aufwand bei der Testdurchführung profitieren und eine höhere Qualität der Gesamtsysteme für den Kunden erzielen.

3 Statische Analysen im Vergleich zu dynamischen Tests

Dynamisches Testen im Umfeld des Kunden ist in frühen Phasen der Software-Entwicklung sehr aufwendig, da beispielsweise Software von Kommunikationssystemen im Verbund mit der noch in der Entwicklung befindlichen Hardware arbeiten muss. In späteren Phasen muss die Software im Zusammenspiel mit vielen Umsystemen getestet werden, die am Standort der Entwicklung nicht oder noch nicht zur Verfügung stehen. Eine zusätzliche Herausforderung entsteht durch die geographische Verteilung von Entwicklungsgruppen für Hard- und Software.

Die statischen Code-Analysen bieten im Vergleich zu dynamischen Tests unter anderem die folgenden Vorteile:

- Abdeckung von zusätzlichen Qualitätsaspekten wie Wartbarkeit und Analysierbarkeit.
- Verwendbarkeit von technisch abhängigen und fachlich unabhängigen Prüfregeln zur Kommunikation und Homogenisierung in Entwicklergruppen (z.B. gewünschte Architekturen oder Designs).
- Weitgehende Automatisierbarkeit der Prüfungen, ohne signifikante Initialaufwände (wie z.B. bei Capture&Replay-Werkzeugen) zu benötigen.
- Durchführbarkeit für einzelne Software-Fragmente lange vor der Integration zu einem lauffähigen Gesamtsystem.

- Durchführungszeitpunkt unabhängig vom Stand des Projekts (für die Prüfung von Regeln idealerweise schon zu Beginn, aus Werkzeugsicht später aber genauso möglich).
- Durchführung der statischen Analyse mit geringen Kenntnissen der Fachlichkeit möglich und daher minimal-invasiv für den Projektverlauf.
- Vollständigkeit der Analyse aufgrund der statischen Gesamtbetrachtung des Systems (der Abdeckungsgrad dynamischen Tests schwankt in Abhängigkeit der Testabdeckung, die in aller Regel aber 80% einer C0-Abdeckung nicht übersteigt).
- Einfache Vergleichbarkeit der Messergebnisse mit anderen Systemen/ Versionen durch Benchmarking, da die statische Sicht weniger abhängige Variable besitzt und daher reproduzierbarere und damit einfacher vergleichbare Messergebnisse liefert.

Als Verfeinerung der allgemeinen Zielsetzung der Qualitätssicherung erfolgte beim Kunden eine Fokussierung auf Indikatoren mit großem Einfluss auf die ISO9126-Qualitätseigenschaften [1] Effizienz, Stabilität und Zuverlässigkeit. Auf den Aspekt der Wartbarkeit der Software wird darüber hinaus nur sekundär Wert gelegt, obwohl der Kunde zum Teil erhebliche Wartungs- und Supportanforderungen seiner Kunden erfüllen muss: Die erwartete Lebensdauer der Systeme liegt in der Regel über 25 Jahre. Diese nicht zwangsläufig der Total-Cost-of-Ownerships geschuldete Sichtweise ist typisch für Embedded Systems.

3.1 Abgrenzung

Die aufgeführten Vorteile der statischen Analyse motivieren deren zusätzlichen Einsatz, können die dynamischen Tests aber keinesfalls vollständig ersetzen, da

- keinerlei Aussagen über die Funktionalität der Software abgeleitet werden können (Validierung),
- Performance und Ressourcenverbrauch (Speicher, CPU) nur schwer vollständig erfasst werden,
- bei einigen Indikatoren entweder notwendige konservative Annahmen der statischen Analysen viele falsch-positive Kandidaten liefern oder nur einen Teil der möglichen Fehlerstellen identifizieren.

Der erwartete wirtschaftliche Vorteil der zusätzlichen statischen Analyse liegt neben der erreichbaren Ganzheitlichkeit (z.B. Wartbarkeit) darin, dass die Probanden der dynamischen Analyse eine deutlich bessere Eingangsqualität haben und die dynamischen Tests nur noch solche Abweichungen finden, für deren Identifikation dynamische Tests tatsächlich notwendig sind. Letztlich dient die statische Code-Analyse damit der Effizienzsteigerung dynamischer Tests.

4 Projektablauf

Initiiert wurde die Qualitätsoffensive seitens der Projektleitung, die als gesamtverantwortlich für die jeweils untersuchten Systeme zeichnet. Insgesamt wurde die Software dreier verschiedener Anwendungen in unterschiedlichen Technologien untersucht. Die Software ist in mehreren verschiedenen Teilprojekten entwickelt worden und hat C-, C++- und C#-Anteile.

Das erklärte Ziel der Assessments ist die Identifikation von Risiken bzgl. der ISO9126 (vgl. oben). Da für die einzelnen Indikatoren eine Null-Fehlergrenze als zu restriktiv und in der Praxis als nicht sinnvoll eingeschätzt wird, wurde zur Bewertung der Indikatoren das allgemeine Risiko mit Hilfe von Benchmarking gegen das SQS-Benchmark-Repository ermittelt, das initiiert durch das Forschungsprojekt QBench mittlerweile Risikokennzahlen von knapp 200 Großprojekten aus der Industrie besitzt [6].

Der Überblick über das Vorgehen ist in Abbildung 1 dargestellt und wird im Folgenden weiter detailliert.

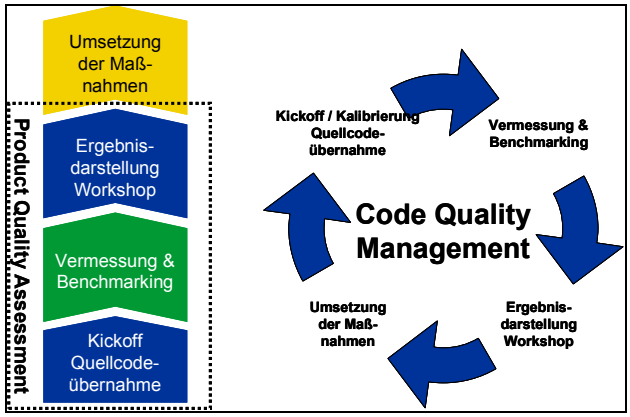


Abbildung 1: Einmaliges Vorgehen (PQA) und kontinuierliches Qualitätsmanagement (CQM)

4.1 Quellcode-Übernahme

Der Projektstart der Assessments bestand jeweils in einem Kickoff mit den Entwicklern. Vor Ort wurde die Codeübernahme und Erfassung der allgemeinen Projektparameter durchgeführt. Dazu gehören unter anderem die verwendeten Programmiersprachen, die eingesetzten Frameworks und ganz wesentlich die Inventarisierung der Fremdanteile (3rd party, open-source). Zum Zeitpunkt des Kickoffs ist die Identifikation von generiertem Code von besonderer Bedeutung, da zumindest einige der untersuchten Indikatoren für generierten Code anders gewichtet werden. Die Beteiligung der Kunden-Entwickler ist für diesen Schritt unbedingt erforderlich, um die nachfolgend durchgeführte Analyse des Codes zielgerichtet

durchführen zu können. Die Verständigung mit der Entwicklung auf die gemeinsame Codebasis ist auch wegen der Normierung der Messergebnisse ein *sine-qua-non* für die anschließend ohne Beteiligung der Entwickler durchgeführte Analyse.

4.2 Vermessung

Die Code-Analyse und die vorläufige Risikoidentifikation mit Hilfe des SQS-Benchmarking-Repository wurde im Anschluss ohne Mitwirkung des Kunden durch die SQS durchgeführt. Aus zwei wesentlichen Gründen kommt bei der Analyse ein ganzer Werkzeug-Zoo zum Einsatz, dessen Resultate im Anschluss dem Kunde in kondensierter und konsolidierter Form übergeben werden. Im konkreten Fall sind dies CAST [2], Bauhaus [3], Software Tomograph [4], Splint [5] sowie eine Reihe von Perl-, Shell-, und sonstigen Skripten.

- Einerseits sind die Analysatoren primär technologiegetrieben und bieten etwa für bestimmte Sprachen besonders gute, für andere dagegen keine Unterstützung. Darüber hinaus besitzen die Werkzeuge für unterschiedliche Techniken unterschiedliche Detailtreue und Abstraktionsfähigkeit: Während manche Werkzeuge in einer Technik den vollständigen Kontrollfluss analysieren, abstrahieren andere davon und stellen lediglich grobere, auf Modulebene verfügbare Informationen dar.
- Zweitens sind die Ergebnisse solcher Werkzeuge immer noch nur durch Experten effektiv und effizient anwendbar und zwischen den Werkzeugen untereinander vollkommen inkompatibel. Bereits an recht einfach anmutenden Metriken wie „Lines of Code“, die in allen Werkzeugen auch als solche bezeichnet werden, scheitert ohne entsprechende Kenntnisse die Vision, konsistente und objektive Kennzahlen zu erheben. Erst die Erfahrung um die jeweiligen Schwächen der Werkzeuge erlaubt das punktuelle Zusammenstellen von Ergebnissen unterschiedlicher Werkzeuge zu einer Gesamtsicht.

4.3 Ergebnisanpassung und -anreicherung

Die Ergebnispräsentation vor Entwicklern und die Anreicherung in Form einer Risikobewertung der vermessenen Indikatoren erfolgen nach der Konsolidierung der Analyseergebnisse. Um zu einer Gesamtbewertung der Projekte zu gelangen, wird auf der Grundlage bidirektionaler Qualitätsmodelle [6] eine Aggregation der Bewertung der Einzelindikatoren durchgeführt. Dabei kommt eine durch SQS vorgegebene Standardgewichtung der Indikatoren in Bezug auf die ISO-Eigenschaften zum Einsatz, die gegebenenfalls projektspezifisch angepasst werden kann. Damit sind quantitative Aussagen bzgl. der Ausprägung eines Systems entlang der ISO9126 möglich.

Die Ableitung von konkreten Maßnahmen in Bezug auf die konkreten Befunde im Quellcode der Software bedarf darüber hinaus kundenspezifischer Diskussion: So kann eine Liste hochpriorisierter Risiken z.B. dadurch relativiert werden, dass ein Großteil der Symptome in Systemteilen liegen, die kurzfristig durch andere Module ersetzt werden. Neben dem allgemeinen auf Basis des Benchmarkings ermittelten Risikos ist daher die Integration der Kundenstrategie wichtig. Eine weitere Dimension, die ebenfalls der Anreicherung durch den Kundenkontext bedarf, betrifft den Aufwand, der mit der Behebung gefundener Risiken verbunden ist. Hier existieren zwar Abschätzungen auf Basis langjähriger Erfahrungen; diese hängen aber in jedem Fall vom konkreten Kontext (z.B. welche Werkzeugunterstützung kann vorausgesetzt werden) ab. Werden diese drei Dimensionen Benchmark-Risiko, Kunden-Risiko und Aufwand gleichzeitig betrachtet ergibt sich in Folge ein Entscheidungsraum, der Grundlage einer abgestimmten Priorisierung ist. Ziel ist die Identifikation der risikobehaftetsten und aufwandsminimalsten Maßnahmen. Eine vereinfachte Darstellung der Parameter ist in **Fehler! Verweisquelle konnte nicht gefunden werden.** zu sehen.

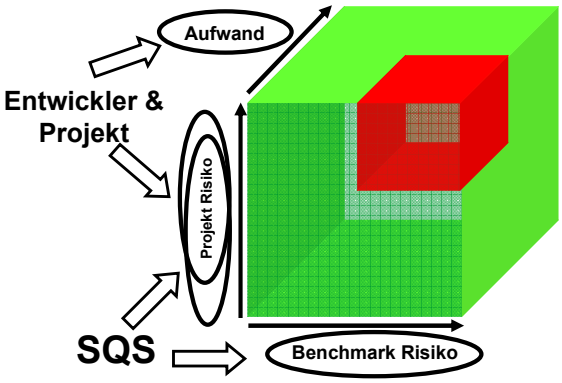


Abbildung 2: Priorisierung der Maßnahmen.

Nicht zuletzt wurden die Ergebnisse des Assessments auch dem Kunden-Management präsentiert. Die Herausforderung besteht hierbei in der kompakten Aufbereitung von komplexen Ergebnisberichten (die als Arbeitsgrundlage an die Entwickler gegeben werden) auf wenige, nachvollziehbare und aussagekräftige Fakten.

5 Erfahrungen und Ergebnisse

Die Methoden und Konzepte der statischen Codeanalyse lassen sich in der Praxis mit Aufwänden zu einem beliebigen Zeitpunkt in die Projekte einbringen, die im Vergleich zu den Aufwänden des Blackboxtestens sehr gering sind (im vorliegenden Projekt knapp 10%). Allerdings gilt auch hier: je früher desto besser, da bereits durch die Einführung derartiger Qualitätssicherungsmaßnahmen das Bewusstsein der einzelnen Entwickler ge-

schärft wird und die Entstehung von problematischem Code von vorneherein unterbleibt.

Die statische Codeanalyse steht damit frühzeitig bereit, nach Schwächen und Risiken im Quellcode der Applikationen zu suchen - lange bevor der erste Blackboxtest durchgeführt werden kann. Sie lässt sich gezielt zur Suche von Codestellen einsetzen, die zur Laufzeit erhebliche Auswirkungen auf den Programmablauf haben. Dieses Vorgehen ist aufgrund der sehr geringen notwendigen Beistelleistungen (z.B. keine lauffähige Umgebung, keine repräsentativen Testdaten, keine systematischen Testfälle) sehr viel effizienter durchzuführen.

Da die Fehler sehr techniknah identifiziert werden, ist das Lösungsvorgehen meist direkt ableitbar und zügig umsetzbar. Im konkreten Fall konnten durch die statische Analyse der Projekte aus dem Hause des Kunden Codestellen identifiziert werden, die im Falle einer Ausführung des Programms mit Sicherheit zu Fehlverhalten des System und schließlich zum Absturz der Software geführt hätten. Da diese vor aufwendigen dynamischen Tests identifiziert und behoben werden konnten, waren die dynamischen Tests insgesamt weniger aufwendig.

Im vorliegenden Projekt dominierten von den Vorteilen der statischen Analyse (vgl. oben) vor allen Dingen folgende Punkte:

- Vollständigkeit: Etwa 20% der identifizierten Fehler wären vermutlich kaum systematisch durch dynamische Tests identifiziert worden. Hierzu zählen insbesondere die Bereiche Speicherverwaltung und Initialisierungsanomalien.
- Erhöhung der Eingangsqualität des dynamischen Tests für deren Aufwandsreduktion: Auch wenn in der Praxis kein sauberer empirisch tauglicher Versuchsaufbau existierte, so konnte im vorliegenden Projekt zumindest im Vergleich zu ähnlichen Systemen des gleichen Unternehmens eine Aufwandsreduktion des dynamischen Tests bis zum Erreichen von Test-Ende-Kriterien um ca. 10-15% erreicht werden.
- Ganzheitlichkeit: Auch wenn die Eigenschaft Wartbarkeit nicht im direkten Fokus der Analyse stand (vgl. oben) so wurde sie dennoch optimiert, da auch diese Abweichungen Teil des Maßnahmenpaketes waren. Quantifizierungen dieser Wartbarkeitseinsparung aus anderen Projekten belegen hier Werte zwischen 10% und 30% (vgl. z.B. [7], [8], [9]).

Darüber hinaus hat der frühe Zeitpunkt einer statischen Analyse den Vorteil, das Thema Qualität frühzeitig bis ins Management hinein zu kommunizieren. Damit schafft sie durchgehend und kontinuierlich (Automatisierbarkeit!) Transparenz im Hinblick auf die Qualität der Software. Auch die Möglichkeit, damit Lieferantensteuerung zu einer aktiven und aufgrund der Objektivität partnerschaftlichen Tätigkeit werden zu lassen, wird beim Kunden als sehr positiv angesehen. Somit wird ein nachhaltiger Beitrag zur Verbesserung der Produkte sowohl entwicklungsintern als auch extern für Kunden geleistet.

6 Referenzen

[1] ISO/IEC 9126-1:2001

[2] CAST APM, http://www.castsoftware.com/

[3] Bauhaus Suite, Axivion GmbH, http://www.axivion.de/

[4] Software Tomograph, Software Tomographie GmbH, http://www.software-tomography.de/

[5] Splint, http://www.splint.org/

[6] Simon, F., Seng, O., Mohaupt, Th., *Code Quality Management* dpunkt-Verlag, Mai 2006

[7] Studemund, M., Rioth, C., Simon, F. „ROI-Modell für das Reengineering zur Optimierung technischer SW-Qualität“, Proceedings der WSR2005

[8] Richter, U., Simon, F.: „Mit Code-Metriken Wartungskosten senken: Controlling technischer Qualität“, Proceedings der Metrikon 2005

[9] Conte, A., Simon, F.: „Partnerschaftliche Lieferantensteuerung mittels technischer Anforderungen“, Proceedings der SQM2007

Predicting Effort to Fix Software Bugs

Cathrin Weiß
Saarland University
weiss@st.cs.uni-sb.de

Rahul Premraj
Saarland University
premraj@cs.uni-sb.de

Thomas Zimmermann
Saarland University
tz@acm.org

Andreas Zeller
Saarland University
zeller@acm.org

Abstract

Predicting the time and effort for a software problem has long been a difficult task. We present an approach that predicts the fixing effort *for an issue. Our technique leverages existing issue tracking systems: given a new issue report, we search for similar, earlier reports and use their average time as a prediction. Our approach thus allows for early effort estimation, helping in assigning issues and scheduling stable releases. We evaluated our approach on the JBoss project data and found that we can estimate within* ± 7 *hours of the actual effort.*

1. Introduction

In this paper, we address the problem of estimating the time it takes to fix an *issue* (an *issue* is either a bug, feature request, or task) from a novel perspective. Our approach is based on leveraging the experience from earlier issues—or, more prosaic, to extract issue reports from bug databases and to use their features to estimate *fixing effort* (in person-hours) for new, similar problems. These estimates are central to project managers, because they allow to plan the cost and time of future releases.

Our approach is illustrated in Figure 1. As a new issue report r is entered into the bug database (1), we search for the existing issue reports which have a description that is most similar to r (2). We then combine their reported effort as as estimate for our issue report r (3).

2. Data Set

Most development teams organize their work around a *bug database*. Essentially, a bug database acts as a big list of issues—keeping track of all the bugs, feature requests, and tasks that have to be addressed during the project. Bug databases scale up to a large number of developers, users—and issues. An issue report provides fields for the *description* (what causes the issue, and how can one reproduce it), a *title* or *summary* (a one-line abstract of the description), as well as a *severity* (how strongly is the user affected by the issue?).

For our experiments, we use the JBoss project data that uses the Jira issue tracking system to organize issue reports. Jira is one of the few issue tracking systems that supports effort data. To collect issues from JBoss, we developed a tool [3] that crawls through the web interface of a Jira database and stores them locally. As inputs for our experiment, we only use the title and the description of issues since they are the only two available fields at the time the issue is reported. In Table 1, we list the prerequisites for issues to qualify for our study. In total, 567 issues met these conditions and finally became the inputs to our statistical models.

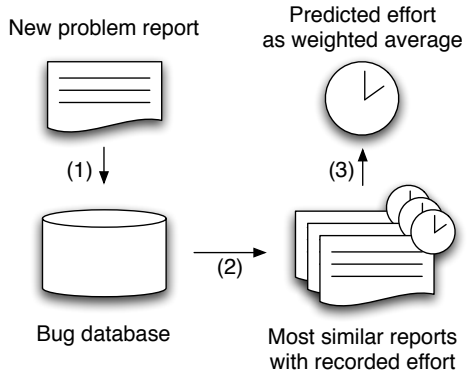


Figure 1. Predicting effort for an issue report

Table 1. Prerequisites for issues.

	Count
Issues reported until 2006-05-05	11,185
Issues with	
– effort data (*timespent_sec* is available)	786
– valid effort data (*timespent_sec*$\leq$*lifetime_sec*)	676
– *type* in ('Bug', 'Feature Request', 'Task', 'Sub-task')	666
– *status* in ('Closed, 'Resolved')	601
– *resolution* is 'Done'	575
– *priority* is not 'Trivial'	574
Issues indexable by Lucene	**567**

3. Predicting Effort for Issue Reports

To estimate the effort for a new issue report, we use the nearest neighbor approach [2] to query the database of re-

solved issues for textually similar reports. Analogous to Figure 1, a *target* issue (i.e., the one to be estimated) is compared to *previously* solved issues to measure similarity. Only the *title* (a one-line summary) and the *description*, both of them are known *a priori*, are used to compute similarity. Then, the k most similar issues (*candidates*) are selected to derive a estimate (by averaging *effort*) for the target. Since the input features are in the form of free text, we used Lucene [1] (an open-source text similarity measuring tool) to measure similarity between issues.

We also use another variant of kNN (α-kNN) to improve confidence in delivered estimates. Here, only predictions for those issues are made for which similar issues exist within a threshold level of similarity. Prosaically, all candidates lying within this level of similarity are used for estimation.

To evaluate our results, we used two measures of accuracy. First, Average Absolute Residuals (*AAR*), where residuals are the differences between actual and predicted values. Smaller *AAR* values indicate higher prediction quality and vice versa. Second, we used *Pred(x)*, which measures the percentage of predictions that lie within $\pm x\%$ of the actual value, x taking values 25 and 50 (higher *Pred(x)* values indicate higher prediction quality).

4. Results

Figure 2 shows the *AAR*, *Pred(25)* and *Pred(50)* values for when varying the k parameter from 1 to 10. The *AAR* values improve with higher k values, i.e., the average error decreases. Since, the *Pred(25)* and *Pred(50)* values worsen (i.e., decrease), there is no optimal k in our case. Overall the accuracy for kNN is poor. On average, the predictions are off by 20 hours; only 30% of predictions lie within a $\pm$50% range of the actual effort, which we speculate to be an artefact of diversity in issue reports.

The α-kNN approach does not suffer from diversity as much as kNN. In Figure 3, we show the accuracy values for α-kNN when incrementing the α parameter from 0 to 1 by 0.1. We used $k = \infty$ for this experiment to eliminate any effects from the restriction to k neighbors. Note that *Feedback* indicates the percentage of issues that can be predicted using α-kNN. The combination of $k = \infty$ and $\alpha = 0$ uses *all previous* issues to predict effort for a new issue (naïve approach without text similarity). It comes as no surprise that accuracy is at its lowest, being off by nearly 35 hours on average.

However, for higher α values, the accuracy improves: for $\alpha = 0.9$, the average prediction is off by only 7 hours and almost every second prediction lies within $\pm$50% of the actual effort value. Keep in mind that higher α values increase the accuracy at the cost of applicability; for $\alpha = 0.9$, our approach makes only predictions for 13% of all issues. Our future work will focus on increasing the *Feedback* values by using additional data, such as discussions on issues.

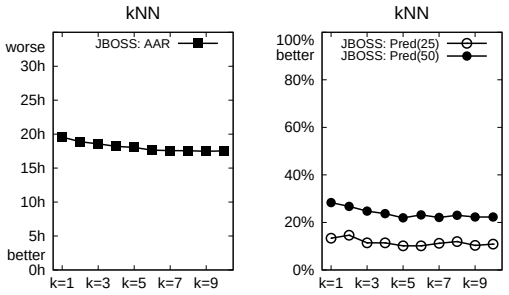


Figure 2. Accuracy values for kNN.

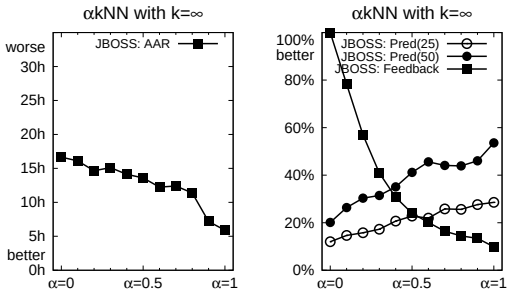


Figure 3. Accuracy for α-kNN with k=∞.

5. Conclusions and Consequences

Given a sufficient number of earlier issue reports, our automatic model makes predictions that are very close for issues. As a consequence, it is possible to estimate effort at the very moment a new bug is reported. This should relieve managers who have a long queue of bug reports waiting to be estimated, and generally allow for better allocation of resources, as well for scheduling future stable releases. The performance of our automated model is more surprising if one considers that our effort predictor relies only on two data points: the title, and the description. However, some fundamental questions remain to be investigated such as what is it that makes software tedious to fix? To learn more about our work in mining software archives, please visit

`http://www.st.cs.uni-sb.de/softevo/`

References

[1] E. Hatcher and O. Gospodnetic. *Lucene in Action*. Manning Publications, December 2004.

[2] M. Shepperd and C. Schofield. Estimating software project effort using analogies. *IEEE Transactions on Software Engineering*, 23(12):736–743, November 1997.

[3] T. Zimmermann and P. Weigerber. Preprocessing CVS data for fine-grained analysis. In *Proceedings of the First International Workshop on Mining Software Repositories*, pages 2–6, May 2004.

Tool-Demos

Bauhaus

Anbieter/Hersteller: Universität Bremen/Universität Stuttgart/Axivion GmbH
Referent: Jochen Quante, Jan Harder
Kurzbeschreibung:
Bauhaus bietet einen Werkzeugkasten zur Unterstützung von Software-Evolution mit Hilfe statischer und dynamischer Analysen. Sowohl quellcodenahe als auch architekturrelevante Informationen werden extrahiert, abstrahiert und visualisiert.
weitere Informationen:
`http://www.informatik.uni-bremen.de/st/`,
`http://www.bauhaus-stuttgart.de/`,
`http://www.axivion.com/`

BS2 MigMan - Migration Manager für BS2000 Architekturen

Anbieter/Hersteller: pro et con Innovative Informatikanwendungen GmbH
Referent: Denis Uhlig
Kurzbeschreibung:
BS2 MigMan ist eine IDE, welche eine automatische Migration von BS2000 Applikationen in UNIX-Umgebungen realisiert. Das Tool ist eine Eclipse-Anwendung mit folgenden Funktionalitäten:

- Anwendungsprogramme im BS2000 eigenen COBOL-Dialekt werden in allgemeingültige COBOL-Dialekte für UNIX konvertiert.
- Ein in das Tool integrierter Sprachkonvertierer übersetzt die BS2000-eigene Systemprogrammiersprache SPL in C++.
- SDF-Prozeduren werden nach Perl konvertiert.

Die komfortable graphische Oberfläche garantiert die Kontrolle über alle MigMan-Funktionen. Die integrierte Versionsverwaltung basiert auf dem Open Source Produkt Subversion und realisiert das Speichern verschiedener Projektzustände.
weitere Informationen:
`http://www.proetcon.de`

SRA - Software Reengineering Architektur

Anbieter/Hersteller: pro et con Innovative Informatikanwendungen GmbH
Referent: Uwe Erdmenger
Kurzbeschreibung:
SRA ist ein Werkzeugkasten zur Entwicklung von Migrations-Tools. Alle Migrations-Werkzeuge der Firma pro et con entstanden unter Nutzung von SRA. SRA selbst ist ebenfalls eine Eigenentwicklung. Daraus werden präsentiert:

Parsergenerator BTRACC:
Generiert aus einer formalen Beschreibung den Quellcode eines Parsers. Im Vergleich zu anderen Parsergeneratoren (z.B. yacc) entfallen durch das zugrundeliegende Backtracking-Verfahren Notationsbeschränkungen der Eingabegrammatik und die Attributierung ist wesentlich komfortabler. Die Demonstration erfolgt anhand eines Praxisbeispiels (Generierung eines SPL-Parsers).

Tree Handler zur formalen Notation von Syntaxbäumen:
Eingabe für das Werkzeug ist eine formale Notation eines Syntaxbaumes. Tree Handler generiert daraus den Code für die Klassen des Syntaxbaumes. Tree Handler wird eingesetzt bei der Entwicklung von Sprachkonvertierern (Translatoren).

C-Gen zur Generierung von C/C++-Quellcode:
Der letzte Schritt bei einer toolgestützen Sprachkonvertierung besteht in der Zielcode-Generierung. C-Gen generiert automatisch aus der internen Repräsentation eines Programmes (Syntaxbaum) den strukturierten Zielcode. Mittels eines Config-Files wird auf das Zielcode-Layout nutzerspezifisch Einfluß genommen (z.B. Blockeinrückungen, Zeilenumbrüche, Formatierungen,...).

weitere Informationen:
`http://www.proetcon.de`