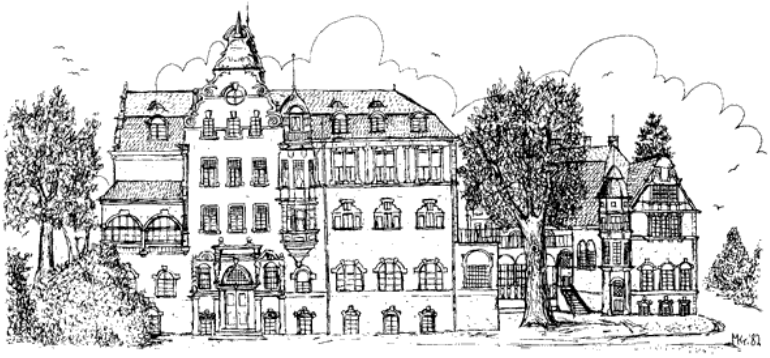
17. Workshop Software-Reengineering und -Evolution

der GI-Fachgruppe Software-Reengineering (SRE)

Bad Honnef

4.-6. Mai 2015



17. Workshop Software-Reengineering und -Evolution

der GI-Fachgruppe Software Reengineering (SRE)

4.-6. Mai 2015
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 (wie etwa SANER und ICSME) auch ein deutschsprachiges Diskussionsforum zu schaffen. Seit dem letzten Jahr haben wir explizit das Thema Software-Evolution in den Titel mit aufgenommen, um eine breitere Zielgruppe anzusprechen und auf den Workshop aufmerksam zu machen. Damit ist das neue Kürzel entsprechend WSRE.

Ziel der Treffen ist es nach wie vor, 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 WSRE als zentrale Reengineering-Konferenz im deutschsprachigen Raum etabliert. Dabei wird er weiterhin als Low-Cost-Workshop ohne eigenes Budget durchgeführt. Bitte tragen auch Sie dazu bei, den WSRE weiterhin erfolgreich zu machen, indem Sie interessierte Kollegen und Bekannte darauf hinweisen.

Auf Basis der erfolgreichen WSR-Treffen der ersten Jahre wurde 2004 die GI-Fachgruppe Software Reengineering gegründet, die unter http://www.fg-sre.gi-ev.de/ präsent ist. Durch die Fachgruppe wurden seitdem neben dem WSRE auch verwandte Tagungen zu Spezialthemen organisiert. Seit 2010 ist der Arbeitskreis „Langlebige Softwaresysteme“ (L2S2) mit seinen „Design For Future“-Workshops (DFF) aufgrund der inhaltlichen Nähe ebenfalls bei der Fachgruppe Reengineering aufgehängt. Alle zwei Jahre findet seitdem ein gemeinsamer Workshop von WSRE und DFF statt.

Der WSRE ist weiterhin die zentrale Tagungsreihe der Fachgruppe Software-Reengineering. Er bietet eine Vielzahl aktueller Reengineering-Themen, die gleichermaßen wissenschaftlichen wie praktischen Informationsbedarf abdecken. In diesem Jahr gibt es wieder Vorträge zu einem weiten Spektrum von Reengineering-Themen.

Wir sind sicher, dass der diesjährige WSRE – auch dank Ihrer Teilnahme – wieder zu einem lohnenden Ereignis wird und viele spannende Diskussionen und neue Kontakte hervorbringen.

Die Organisatoren danken allen Beitragenden für ihr Engagement – insbesondere den Vortragenden, Autorinnen und Autoren. Unser Dank gilt auch den Mitarbeiterinnen und Mitarbeitern des Physikzentrums Bad Honnef, die es sicherlich wie immer verstehen werden, ein angenehmes und problemloses Umfeld für den Workshop zu schaffen.

Volker Riediger, Universität Koblenz-Landau
Jochen Quante, Robert Bosch GmbH, Stuttgart
Jens Borchers, Sopra Steria Consulting, Hamburg
Jan Jelschen, Universität Oldenburg

17. Workshop Software-Reengineering & Evolution
4.-6. Mai 2015
Tagungsprogramm

Montag, 4. Mai			Slot #
bis 11:00	Anreise		
	Text Mining		
11:00		Welcome	1
11:15	Jochen Quante	Software Reengineering Bibliometrics - People, Topics, and Locations	2
11:45	Marcel Heinz and Ralf Lämmel	Verbesserung einer aus Wikipedia gewonnenen Ontologie	3
12:15	Mittagspause		
	Analysis		
14:00	Arne Wichmann and Sibylle Schupp	Visual Analysis of Control Coupling for Executables	4
14:30	Martin Wittiger and Timm Felden	Recognition of Real-World State-Based Synchronization	5
15:00	Torsten Görg	Performance Tuning of PDG-based Code Clone Detection	6
15:30	Kaffeepause		
	Model Based Development		
16:00	Klaus Müller and Bernhard Rumpe	A Methodology for Impact Analysis Based on Model Differencing	7
16:30	Dilshodbek Kuryazov and Andreas Winter	Towards Model History Analysis Using Modeling Deltas	8
17:00	Domenik Pavletic and Syed Aoun Raza	Multi-Level Debugging for Extensible Languages	9
17:30	Fachgruppensitzung SRE		10
18:00	Abendessen und traditioneller Spaziergang		
Dienstag, 5. Mai			
	Migration		
09:00	Johannes Meier, Dilshodbek Kuryazov, Jan Jelschen and Andreas Winter	A Quality Control Center for Software Migration	11
09:30	Tilmann Stehle and Matthias Riebisch	Establishing Common Architectures in a Process for Porting Mobile Applications to new Platforms	12
10:00	Harry Sneed	Namensänderung in einem Reverse Engineering Projekt	13
10:30	Werner Teppe	Data reengineering and migration to prepare a legacy application platform migration	14
11:00	Kaffeepause		
	Quality		
11:15	Nils Göde	Quality Control in Action	15
11:45	Jens Borchers	Software-Qualitätsmanagement im Rahmen von Application Management Services	16
12:15	Martin Brandtner, Philipp Leitner and Harald Gall	Profile-based View Composition in Development Dashboards	17
12:45	Mittagspause		
	Tool Demos		
14:00	Nils Göde	Teamscale	18
14:20	Arne Wichmann	Kuestennebel: Visual Analysis of Control Coupling for Executables	19
14:40	Dilshodbek Kuryazov	Q-MIG	20
15:00	Social Event: Besuch/Führung im Arp-Museum Rolandseck		
	abends Conference Diner (ab ca. 18:30)		
Mittwoch, 6. Mai			
	Architecture		
09:00	Michael Langhammer and Klaus Krogmann	A Co-evolution Approach for Source Code and Component-based Architecture Models	21
09:30	Robert Heinrich, Kiana Rostami, Johannes Stammel, Thomas Knapp and Ralf Reussner	Architecture-based Analysis of Changes in Information System Evolution	22
10:00	Jens Knodel, Matthias Naab and Balthasar Weitzel	Modularity – Often Desired, Too Often Failed	23
10:30	Kaffeepause		
	Project Support		
11:00	Jan Jelschen, Johannes Meier and Andreas Winter	SENSEI Applied: An Auto-Generated Toolchain for Q-MIG	24
11:30	Marvin Grieger and Masud Fazal-Baqaie	Towards a Framework for the Modular Construction of Situation-Specific Software Transformation Methods	25
12:00	Hakan Aksu and Ralf Lämmel	API-related Developer Profiling	26
12:30	abschließendes Mittagessen		

Text Mining

Jochen Quante – Software Reengineering Bibliometrics - People, Topics, and Locations

Marcel Heinz und Ralf Lämmel – Verbesserung einer aus Wikipedia gewonnenen Ontologie

Software Reengineering Bibliometrics – People, Topics, and Locations

Jochen Quante

Robert Bosch GmbH, Corporate Research
Stuttgart, Germany

The statistical analysis of publication data can provide insightful information about the state and evolution of a research field. We report about application of such an analysis on software reengineering publications. The analysis of authors, titles and abstracts results in an overview about relevant people, topics, and locations.

1 Introduction

Researchers often face the challenge of getting an overview over a research field. A good starting point is literature research. However, this easily results in thousands of documents, and it is very hard to get an overall picture. This is where statistical analyses can help. By using data mining techniques, this data can be leveraged to a level where it really gives an overview of the most relevant persons, topics, and locations. We applied such an approach to the analysis of scientific publications on software reengineering. It is based on an idea by Hassan et al.[1] who did a similar analysis ten years ago.

2 Data Collection

We start by identifying the most relevant conferences. The selection of conferences guarantees a certain degree of quality (peer reviews). It also has the advantage that people meet there, so the probability to find collaborations between people is higher than in journals. Next, a bibliographic database is queried for publications from these conferences. We use Scopus[1], which provides not only authors and title, but also a lot of additional information: Abstracts, addresses of authors, citation information, etc. This data is then exported and analyzed in a proprietary mining tool.

3 Mining Approach

The following information can be mined from this data:

- Ranked authors lists: Authors with the highest number of publications, or with the highest number of direct or indirect collaborations. This requires to match identical authors, which is a challenge of its own (e. g., typos, variants in writing the name, multiple people with same name).

[1] http://www.scopus.com/

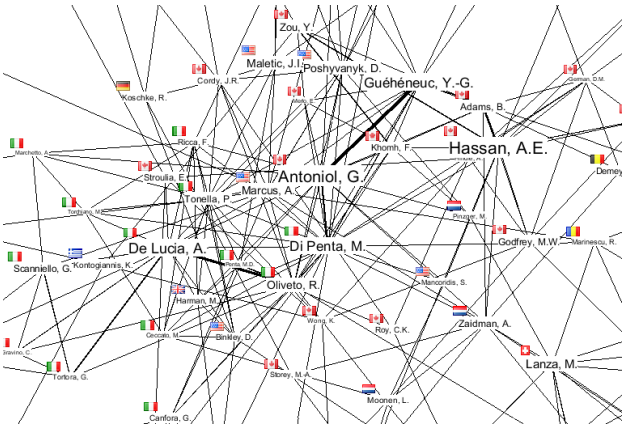


Figure 1: Collaborations between authors with at least 10 publications.

- Collaboration graph: Who has written papers together?

- Geographical authors map: Where do people come from? What are the central locations of research in this field? This information can be derived by using Scopus' affiliation information. Unfortunately, the affiliation is a free text field, which means that text mining techniques are needed to derive the location (city and country) out of this. For example, sometimes only the name of the institute is given.

- Topic map: What are the main topics of these publications? Which topics are related? Who is active in which of these topics? There are several approaches to do topic mining. We integrated word group counting and an advanced topic mining approach [2].

- Trend analysis: Which topics are becoming more frequent, which ones are fading out? When topics have been identified, the number of papers on this topic over the years can be analyzed: Is it increasing or decreasing? However, only really strong trends can be identified this way. "Weak signals" cannot be found using such an approach.

4 Results

We took as a basis all publications from CSMR, ICPC, ICSM and WCRE from 2004 until 2013, plus ICSE

Author	Country	#Pub.	#CoAu.
Hassan	Canada	58	61
Antoniol	Canada	56	70
De Lucia	Italy	46	(45)
Di Penta	Italy	45	62
Guéhéneuc	Canada	44	62
Koschke	Uni Bremen	22	34
Nierstrasz	Uni Bern	21	32
Sneed		21	4
Pinzger	Uni Klagenfurt	15	28
Knodel	IESE Kaiserslt.	15	27

Table 1: Authors with most publications and most distinct co-authors (overall and German-speaking).

City	Country	#Auth.	#Pub.
Montreal	Canada	83	201
Delft	Netherlands	27	90
Kingston	Canada	36	76
Lugano	Switzerland	24	61
Salerno	Italy	22	58
Bern	Switzerland	17	49
Vienna	Austria	35	41
Bremen	Germany	21	35
Kaiserslautern	Germany	24	29
Stuttgart	Germany	14	24

Table 2: Where most publications and authors come from (overall and Germany only).

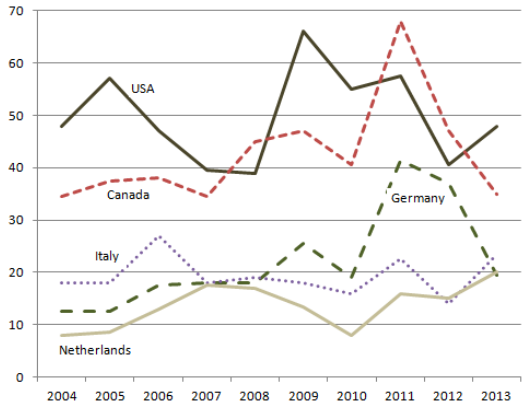


Figure 2: Where most publications came from (year vs. number of publications).

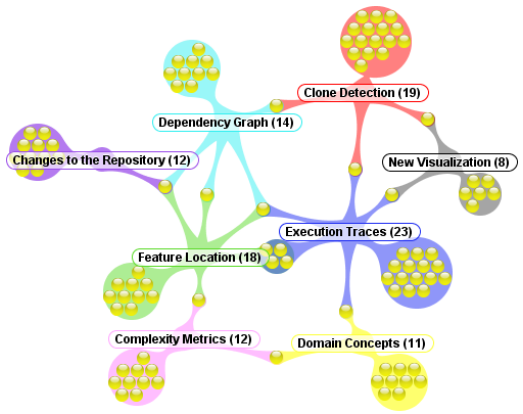


Figure 3: Topic map for ICPC publications (excerpt).

publications with keyword "maintenance". This corpus contains 2,547 publications by 3,433 different authors from 63 different countries.

Looking at authors shows that the most active ones come from Canada. The ones with the highest number of publications also have the highest number of co-authors (see Table 1). Table 1 also shows the "Top 5" German-speaking authors. Collaborations can best be shown in a graph. Figure 1 shows collaborations between authors who have published more than 10 papers. The strongest collaborations can naturally be found between Professors and their (former) Ph. D. students. For other collaborations, it can be noted that they are more common between people who reside close to each other.

Another question is where the center of research in this area is. Figure 2 shows where most papers came from during the last 10 years. USA and Canada are on top, but Germany has also been quite active for some time. Such a visualization can be useful to see if other players (e. g., from China) are coming up.

Another aspect of publications concerns content: You want to know what the main topics of research are. When looking at word group frequencies in titles, terms like "reverse engineering", "source code" or "software maintenance" are identified. Looking at the whole abstracts results in more specific terms like "web applications", "execution traces" or "aspect oriented programming". A more advanced technique is *topic mining* [2]. Applying this technique to all abstracts from ICPC results in topics as shown in Figure 3. Such a *topic map* also shows how topics are related: Each small circle is a publication, and the topic areas cover all papers that deal with this topic. This technique can be used to create a "landscape" of a research topic.

5 Summary

We have shown how bibliometrics and text mining can give insights into a given research topic. Such techniques can be helpful for getting a first overview for further analysis of a research field. It can also be used to continuously monitor a field in order not to miss relevant trends and changes.

References

[1] A. E. Hassan and R. C. Holt. The small world of software reverse engineering. In *Proc. of 11th Working Conference on Reverse Engineering (WCRE 2004)*, pages 278–283, 2004.

[2] S. Osinski and D. Weiss. A concept-driven algorithm for clustering search results. *IEEE Intelligent Systems*, 20(3):48–54, 2005.

Verbesserung einer aus Wikipedia gewonnenen Ontologie
(Extended Abstract)

Marcel Heinz Ralf Lämmel
Arbeitsgruppe Softwaresprachen, Universität Koblenz-Landau

Zusammenfassung

Wikipedia bietet die zur Zeit größte online verfügbare Enzyklopädie. Verschiedene Arbeiten zielen darauf ab, das Wissen aus Wikipedia in eine Ontologie zu fassen. Nachdem eine Ontologie erstellt wurde, kann ihre Qualität mit verschiedenen Verfahren bewertet werden. Zur Analyse einer aus Wikipedia gewonnenen Ontologie schlagen wir verschiedene Bad Smells vor. Wir schlagen auch verschiedene Transformationen zur Verbesserung der Qualität vor. Wir wenden unseren Ansatz in einer Fallstudie an, welche die Domäne der Computersprachen betrifft. Diese Vorgehensweise ist inspiriert durch Ansätze des Software Reverse und Reengineering und der Evaluation von Ontologien.

1 Einleitung

Ontologien entsprechen analysierbaren Wissensbasen. Die enthaltenen Informationen werden meist aus verschiedenen Quellen extrahiert. Zur Zeit bietet Wikipedia die größte online verfügbare Enzyklopädie und ist daher das Ziel verschiedener Ansätze zur Extraktion von vorhandenen Informationen.

Als Fallstudie dient in dieser Arbeit Wikipedia's Kategorie zu Computersprachen, deren Informationen das Verstehen von Software-Produkten unterstützen können. Besonders technologische Abhängigkeiten, wie die zu Betriebssystemen, sind von Bedeutung.

Unser Ansatz leitet aus einer gegebenen Domäne eine Ontologie ab, in welcher primär taxonomische Relationen erfasst werden. Danach übertragen wir das Konzept von Bad Smells aus der Softwaretechnik auf die Qualitätsanalyse einer aus Wikipedia gewonnenen Ontologie. Dabei werden strukturelle und semantische Aspekte berücksichtigt. Die Mechanismen zur Erkennung von Bad Smells basieren entweder auf Metriken oder auf Muster. Unsere Vorschläge wurden durch vorangegangene Arbeiten wie die von Fowler et al. [4] oder Rosenfeld et al. [5] inspiriert. Schließlich identifizieren wir geeignete Transformationen zur Verbesserung der Qualität.

Im einfachsten Fall benutzen wir Refactorings, welche den 'Informationsgehalt' einer Ontologie nicht ändern. In andern Fällen müssen wir den 'Informationsgehalt' aber ändern, etwa im Sinne von Prunings zum Entfernen entdeckter, falscher Information. Wir berufen uns hier auch auf Conesa und Olivé, die beschreiben, wie der Informationsgehalt der Cyc Ontologie systematisch sinnvoll verändert werden kann [2].

Die Leitfrage an dieser Stelle ist daher, ob sich bekannte Konzepte aus der Softwaretechnik (Software Reverse und Re-engineering) und Evaluation von allgemeinen Ontologien auf eine speziell aus Wikipedia extrahierte Ontologie zur Qualitätsverbesserung übertragen lassen.

2 Extraktion einer Ontologie

Unser Ansatz konzentriert sich auf die im Folgenden beschriebenen, strukturierten Informationen von Wikipedia, welche in einer Ontologie erfasst werden.

Das erste Ziel einer Extraktion betrifft den Kategorien-Graph. Angefangen bei einer Wurzelkategorie, etwa der Kategorie zu Computersprachen, werden alle direkten und transitiven Unterkategorien in Hat-Unterkategorie-Beziehungen erfasst.

Die Artikel in den Kategorien bilden das zweite Ziel einer Extraktion. Jeder Artikel beschreibt eine Entität. Jede Entität wird der enthaltenden Kategorie durch eine Hat-Entität-Beziehung zugeordnet. Bei einem Artikel werden nur der Name des Artikels und Informationen aus einer eventuell vorhandenen Infobox extrahiert. Die Informationen einer Infobox stellen dabei prägnante Attribute zu der im Artikel beschriebenen Entität dar. Eine Infobox wird auf ein 'AttributeSet' abgebildet, wobei der Name des verwendeten Infobox Templates festgehalten wird. Die extrahierten Attribute werden dem 'AttributeSet' zugeordnet.

Das dritte Ziel besteht aus den Überkategorien von Artikeln und Kategorien. Während initial nur die Kategorien erfasst werden, welche von der gewählten Hauptkategorie erreichbar sind, werden hier auch die Namen der Kategorien erfasst, welche nicht von der Hauptkategorie erreichbar sind. Daraus ergeben sich weitere Hat-Unterkategorie- und Hat-Entität-Beziehungen.

3 Analyse einer Ontologie

Verschiedene Qualitätsmängel in der extrahierten Ontologie sind nicht immer eindeutig. Wenn eine Kategorie beispielsweise viele Artikel beinhaltet, dann ist dies kein eindeutiges Zeichen dafür, dass Artikel in Unterkategorien verschoben werden müssen. Allerdings ist eine solche Tatsache ein Anzeichen dafür, dass eine

Verbesserung durchgeführt werden könnte und entspricht damit einem Bad Smell.

Wir haben eine Reihe von Bad Smells identifiziert. Zu jedem Bad Smell gehört dessen Name, seine Beschreibung, sein Erkennungsmechanismus sowie die Nennung eines positiven und eines negativen Treffers.

Ein beispielhafter Bad Smell ist *Bloated Category*. Mit ihm werden Kategorien erfasst, die sehr viele Artikel beinhalten. Er basiert auf den Bad Smells *Large Class* von Fowler et al. und *Large Category* von Rosenfeld et al. [5]. Ein positiver Match für diesen Bad Smell ist die Kategorie zu 'XML-based standards', welche 260 Artikel enthält. Die entsprechende Seite auf Wikipedia ist bereits mit einem Hinweis markiert, dass Artikel in Unterkategorien verschoben werden sollen. Ein negativer Match ist die Kategorie 'Free software programmed in C' mit 474 Artikeln. Es ist nicht offensichtlich, wie diese Kategorie in sinnvolle Unterkategorien aufzuteilen wäre.

Die Namen weiterer Bad Smells sind in Tabelle 1 aufgelistet. Die gelisteten Bad Smells mit semantischem Bezug basieren auf den Ideen von Baumeister et al. [1] und Fahad und Qadir [3], welche sich auf Anomalien allgemein in Ontologien konzentrieren.

Overcategorization	Redundant Relation
Lazy Category	Speculative Generality
Cycle	Chain of Inheritance
Missing Category	Partition Error
Twin Categories	Multi Topic
Topic Inconsistency	Semantical Distance

Tabelle 1: Liste der vorgeschlagenen Bad Smells

4 Verbesserung einer Ontologie

Bei der Verbesserung der Qualität von Ontologien lassen sich verschiedene Begriffe unterscheiden. Refactorings entsprechen Transformationen, bei denen die Semantik der Ontologie erhalten bleibt. Prunings dagegen verändern das vorhandene Wissen von Ontologien. Wir schlagen einen entsprechenden Katalog mit Name, Beschreibung und Kontext zu jedem Refactoring und Pruning vor.

Ein beispielhaftes Refactoring ist 'Remove redundant has-entity'. Der Artikel 'SPARQL' ist sowohl in der Kategorie 'Data modeling languages' als auch in der Kategorie 'Computer languages' vorhanden. 'Data modeling languages' ist eine direkte Unterkategorie von 'Computer languages'. Daher ist die Beziehung zwischen 'Computer languages' und 'SPARQL' redundant und wird durch dieses Refactoring entfernt. Dank der Transitivität der Beziehung bleibt das Wissen der Ontologie erhalten.

Ein beispielhaftes Pruning ist 'Abandon entity'. Wenn ein Artikel zu wenig Bezug zu seiner Kategorie oder zur Domäne hat, kann die entsprechende Ressource aus der Ontologie entfernt werden. In der Kategorie 'Computer languages' sind Computerspiele ebenfalls erreichbar. Der Artikel zu 'Doom (1993 video game)' wird beispielsweise erfasst. Auf ihn trifft jedoch der Bad Smell 'Semantical Distance' zu. Von insgesamt 38 Kategorien, die diesen Artikel enthalten, ist nur eine Kategorie von 'Computer languages' aus erreichbar. Daher kann der Artikel aus der Ontologie entfernt werden, in dem alle Relationen, die ihn betreffen, gelöscht werden.

Die Namen weiterer Refactorings und Prunings werden in Tabelle 2 aufgeführt.

Abandon Category	Rename Element
Abandon Entity	Change Topic
Remove Has-SubCategory	Add Missing Category
Remove Has-Entity	Unite Attributesets
Collapse Hierarchy	Extract Entity
Remove unreachable Element	Extract SubCategory
Lift Cycle	Move Entity
	Move Category

Tabelle 2: Refactorings und Prunings

5 Zusammenfassung

Wir wenden bekannte Konzepte aus der Softwaretechnik ebenfalls auf Ontologien an. Eine von Wikipedia gewonnene Ontologie kann auf Bad Smells untersucht werden. Wann es sich wirklich um einen Mangel handelt, muss vom jeweiligen Betrachter entschieden werden. Bereits Fowler et al. erwähnen, dass die menschliche Intuition in diesem Belang wichtig ist.

Zur Reparatur übertragen wir Refactorings aus der Softwaretechnik und Prunings aus dem Bereich der Qualitätsverbesserung von allgemeinen Ontologien auf die extrahierte Ontologie. Diese Art von Verfahren kann nicht nur zur Analyse und Verbesserung einer extrahierten Ontologie verwendet werden, sondern mit gewissen Anpassungen auch zur Qualitätssicherung von Domänen in Wikipedia selbst eingesetzt werden.

Literatur

[1] Joachim Baumeister and Dietmar Seipel. Verification and refactoring of ontologies with rules. In *Managing Knowledge in a World of Networks*, pages 82–95. Springer, 2006.

[2] Jordi Conesa and Antoni Olivé. Pruning ontologies in the development of conceptual schemas of information systems. In *Conceptual Modeling–ER 2004*, pages 122–135. Springer, 2004.

[3] Muhammad Fahad and Muhammad Abdul Qadir. A framework for ontology evaluation. *ICCS Supplement*, 354:149–158, 2008.

[4] Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts. *Refactoring: improving the design of existing code*. Addison-Wesley, 1999.

[5] Martin Rosenfeld, Alejandro Fernández, and Alicia Díaz. Semantic wiki refactoring. a strategy to assist semantic wiki evolution. In *Fifth Workshop on Semantic Wikis Linking Data and People 7th Extended Semantic Web Conference Hersonissos, Crete, Greece, June 2010*, page 132. Citeseer, 2010.

Analysis

Arne Wichmann and Sibylle Schupp –Visual Analysis of Control Coupling for Executables

Martin Wittiger and Timm Felden – Recognition of Real-World State-Based Synchronization

Torsten Görg – Performance Tuning of PDG-based Code Clone Detection

Visual Analysis of Control Coupling for Executables

Arne Wichmann, Sibylle Schupp
Technische Universität Hamburg-Harburg, Hamburg, Germany
{arne.wichmann, schupp}@tuhh.de

Program comprehension of stripped executables is hard because neither modules and function names, nor any other structural information are available. We introduce an algorithm that, using morphological operations, highlights fan-in, fan-out, and module coupling in the adjacency matrix of the control flow graph and thus allows initial orientation at function level.

This paper introduces the structures of interest and our algorithm, and analyzes the `yaboot` bootloader.

1 Control Coupling Analysis of Stripped Executables

A general problem of analyzing stripped executables is that very little information is available. Specifically, a disassembler can only help to regroup the code to functions, but on its own cannot generate useful function names, or recreate modules of the code. A first orientation in the code needs a method that is independent of debug information, execution traces, or known libraries and operating systems.

In this paper we present an algorithm that reconstructs information about functions of interest, modules, and their control coupling and thereby gives an overview of the executable. An exploitable property for such an algorithm is that linkers keep the functions and object files in the sequence they were given during compilation and thereby encode such information into the program's addresses and its control flow graph (compare [2]). Our algorithm mines the control flow graph for fan-in, fan-out, and reference clusters between functions using morphological operations on the graph's adjacency matrix and presents them in a comprehensive plot, so that the analyst can get a rough overview of the executable.

While alternative visual coupling analyses exist, they are usually applied to structures extracted from high-level code [1] or use trace information [7].

Table 1: Interpretations of Visual Structures

Structure	Interpretation
Diagonal	Local Control Flow
Box/Triangle on Diagonal	Module
Box/Triangle not on Diag.	Module Coupling
Global Vertical	Fan-In/Library
Vertical near Diag.	Helper Function
Global Horizontal	Fan-Out/Dispatch
Horizontal near Diag.	Dispatch Function

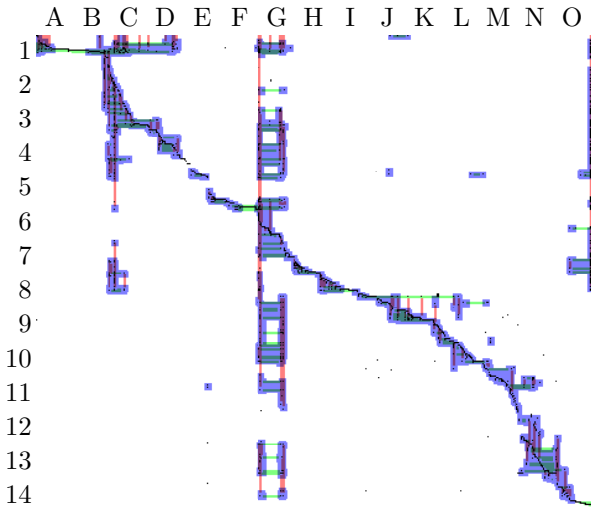


Figure 1: Control Flow Plot for `yaboot`/fc15

2 Visual Analysis of `yaboot`

Figure 1 shows an example of the `yaboot` bootloader for PowerPC from the Fedora Core 15 Linux distribution with rough row (A..O) and column (1..14) notations for orientation. Each horizontal row in the plot represents one of the about 350 functions, each column tracks control accesses in 128 byte steps. We describe the structures of interest and their interpretation:

A1..D1 Big green horizontal line: A setup/dispatch module with large fan-out.

B2..C8 Several red vertical lines: A library module with high fan-in.

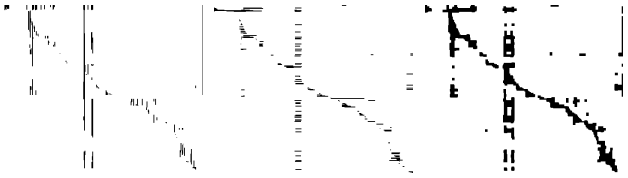
G1..G14 Long red vertical lines: Basic library functions with high fan-in.

J9..N14 Different shapes near diagonal, little coupling in J1..N8 and A9..I14: External library with good separation.

Manual inspection using information from debug symbols confirms the interpretation: A1 is a module that contains an interactive shell, B2 is a wrapper module to the openfirmware (BIOS like), G1 is the C library and J9 is the ext2fs library.

3 Control Flow Plots and Visual Structures

The goal is to create an algorithm that produces a control flow plot, like the one shown in the example above. A control flow plot is based on a binary adja-



(a) H (red) (b) V (green) (c) B (blue)

Figure 2: Separated Results for `yaboot/fc15`

cency matrix of control accesses (similar to [6]), where both axes represent numerically ordered program addresses. To represent function sizes, outgoing accesses are grouped per function (horizontal) and incoming accesses (vertical) use bins in the size of an average basic block.

In such a plot different key visual structures (see Table 1) can be mined and highlighted that provide semantics for an analyst. The diagonal line is caused by intraprocedural control flow, like jumps and next instructions, and its steepness represents the functions sizes. Modules can be identified by their intra-module coupling, which shows up as boxes or triangles next to the diagonal line. Coupling between modules shows up as boxes or triangles disconnected from the diagonal. Setup functions with high fan-out will produce a scattered horizontal line, helper and library functions with high fan-in will leave a similar but vertical line.

The scope of functions and modules can be estimated by their access lines' length as well as their relative closeness to the diagonal. Longer lines correlate to broader access across modules and shorter lines near the diagonal identify intramodular functions.

4 Morphological Highlighting of Structures

We provide a simple algorithm to highlight the structures in the image using basic image processing operations like erosion, dilation, opening (erode + dilate), and closing (dilate + erode) from mathematical morphology. An erosion operation leaves only pixels in the image, where a structuring element fully fitted in the original image. A dilation adds pixels wherever a structuring element aligned with a pixel from the original image has a pixel.

A custom script in IDA Pro [3] extracts the control flow graphs as plots, which are then processed using the octave image package [5].
Listing 1 shows the code used to highlight the structures in the plot. The separated results (see Figures 2a to 2c) are composed in an additional step to form the result image (see Figure 1). The line detection in both horizontal and vertical direction first uses *closing* on the image with a long line as structuring element to create initial lines, then *erodes* with a shorter element to keep only long lines. The line is finally broadened by a *dilation* with an orthogonal structure.

```
H = imdilate(imerode(imclose(J,ones(hlen,1)),
        ones(hlen/3,1)),ones(1,vlen/10));
V = imdilate(imerode(imclose(J,ones(1,vlen)),
        ones(1,vlen/3)),ones(hlen/10,1));
B = H + V + J;
s = ones(blen/3,blen/3);
B = imclose(imdilate(imclose(B,s)-B,s),s);
```

Listing 1: Line and Block Emphasis Algorithm

To create the boxes and triangles in the image, the original image is combined with the detected lines and then *closed* using a block structuring element to form a blurred version. Single dots and lines are removed by subtracting the combined image, and the resulting gaps are removed in one *dilation* step. The remaining structures in the image are again *closed* to produce the final area information.

5 Summary and Future Work

The algorithm works independently of the architecture and therefore supports all the processor modules in IDA Pro. The prototype implementation was run on a testsuite [4] of several embedded executables as well as executables from the CPU2006 benchmark. Manual checks of the results are consistent with debug as well as reverse-engineered information. The next steps are to integrate the implementation in a reverse engineering workflow and to expand the evaluation in both quality and quantity.

References

[1] Johannes Bohnet and Jürgen Döllner. "Visual Exploration of Function Call Graphs for Feature Location in Complex Software Systems". In: *Proceedings of the 2006 ACM Symposium on Software Visualization*. SoftVis '06. Brighton, United Kingdom: ACM, 2006, pp. 95–104.

[2] Peter J. Denning. "The Locality Principle". In: *Communications of the ACM* 48.7 (July 2005), pp. 19–24.

[3] *IDA Pro: Interactive Disassembler*. URL: `http://www.hex-rays.com/products/ida/index.shtml`.

[4] *Küstennebel*. URL: `http://www.tuhh.de/sts/research/projects/kuestennebel.html`.

[5] *Octave-forge: Image*. URL: `http://octave.sourceforge.net/image/`.

[6] Reese T. Prosser. "Applications of Boolean Matrices to the Analysis of Flow Diagrams". In: *Papers Presented at the December 1-3, 1959, Eastern Joint IRE-AIEE-ACM Computer Conference*. IRE-AIEE-ACM '59 (Eastern). Boston, Massachusetts: ACM, 1959, pp. 133–138.

[7] Daniel Quist and Lorie M. Liebrock. "Reversing Compiled Executables for Malware Analysis via Visualization". In: *Information Visualization* 10.2 (2011), pp. 117–126.

Recognition of Real-World State-Based Synchronization

Martin Wittiger and Timm Felden
University of Stuttgart, Institute of Software Technology

Abstract

In the real world, safety-critical embedded systems use state-based synchronization to avoid data races. Using constraint solving to tackle state, we have improved upon existing static data race analysis.

1 Introduction

Data races form a class of programming errors. They are notoriously difficult to find by software testing, yet cause severe problems. Safety-critical embedded systems rely on functional correctness that can only be achieved in the absence of data races. This domain, therefore, particularly requires tools to mitigate the risks in this area. Therefore, the development of static analysis tools, which can either detect data races or prove their absence, is a well-established research problem.

Whenever multiple tasks of a concurrent system access shared resources such as communication variables, software developers use synchronization patterns. Desktop software mostly relies on mutexes and monitors. Embedded systems avoid these two patterns. In our experience, almost all embedded systems employ interrupt enable/disable patterns to synchronize tasks. Static analysis tools handle these patterns easily and efficiently. In addition, state-based synchronization is prevalent. It is often hand-crafted to fit a specific system and typically relies heavily on scheduling properties such as task priorities.

State-based synchronization is generally hard to deal with using static analysis. Keul [1] performs an analysis step called simple path exclusion that recognizes state machines used for synchronization. Schwarz et al. [2] use the term flag-based synchronization to denote essentially the same thing. Both approaches only recognize simple patterns. They both pose strict, virtually syntactically verifiable requirements on variables forming state machines.

2 State Analysis

Our goal is to classify data races using constraint solving. Consider the following two small examples. Both examples involve a state variable `s` that we assume to be of an atomic type and thus free of data races and a shared variable `d` that might be subject to a data race. Greek letters are used to denote tasks.

In the first example, we we show a variation of the ordinary state-based locking. We want new states to be added easily, thus we use an *untaken* state 0 that is not owned by any thread. For the sake of simplicity, we restrict ourselves to two tasks with one access each. The resulting state machine is depicted in figure 1. A pseudo-C implementation is shown in figure 2.

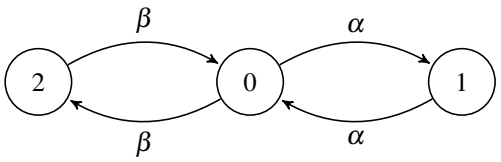


Figure 1: State transitions in our first state machine.

Task α	Task β
`if(s == 0) {`	`if(s == 0) {`
`  s = 1;`	`  s = 2;`
`  d = 4;`	`  d = 7;`
`  s = 0; }`	`  s = 0; }`

Figure 2: A state machine with two tasks and three states.

The second example (figure 3) features three tasks and a slightly degenerated state machine.

We will discuss whether the examples contain data races after taking a closer look at our approach.

2.1 Tool Composition

The analyses described in this paper build upon several layers of preexisting analyses. Our toolchain processes safety-critical C-code. In a first step, a pointer analysis, refined by escape analysis, is performed on the AST. We then perform a lockset analysis and establish a flow-, thread- and partly context-sensitive call graph using a project-specific concurrency configuration. An in-depth description is given by Keul [1]. All analyses are designed to be conservative, i. e. there will be a *warning* for each data race. Warnings are pairs of accesses to memory locations. In practice, many of them are false positives that we want to be able to safely exclude from the list.

Task α	Task β	Task γ
`if(s == 1) {`	`if(s == 2) {`	`s++;`
`  d *= 17;`	`  d += 37;`	
`  s = d; }`	`  s = d; }`	

Figure 3: A state machine with three tasks and multiple states.

This work was in part funded within the project ARAMiS by the German Federal Ministry for Education and Research with the funding ID 01IS11035. The responsibility for the content remains with the authors.

Afterwards, we perform dead code elimination and run a constant folding and propagation analysis. This is very important since constants used in synchronization mechanisms in real code are rarely just integer literals.

In the next stage, state variables have to be selected. This step is discussed in the next section as it has to be performed manually at the moment. Now, the call graph is projected on the effects on the selected state variables. This means that each statement in every basic block in every procedure in each thread is replaced by a conservative approximation of its effect. For instance, if a statement provably cannot change the value of any state variable, it is replaced by a `nop`.

When assigning to a dereference, for instance `*p = 7`, one cannot in general know, which memory location is affected. Such an assigment may thus be transformed into several weak updates on any state variable `p` may point to. However, if `p` points to a specific memory location, a single (strong) update is produced whenever this is a state variable or otherwise a `nop` is emitted. We treat function pointers in a similar fashion. So, the straightforward usage of pointer analysis leads to a natural translation that conservatively approximates pointers. As a whole this takes a form of abstract interpretation.

The projection also retains and marks conflicting accesses from warnings. The projection result including those data race marks is then transferred to a constraint language. The solver then discards all warnings whose accesses can be shown not to be concurrently reachable.

In our implementation we use CSP_M (machine-readable CSP, see [3]) as a constraint language and the FDR2 [4] refinement checker. Certain peculiarities of FDR2 require us to preprocess the CSP_M output of our tool before passing it on. When processing small examples like the ones shown in this paper, FDR2 answers instantly. Solver runtime presumably becomes an issue in more complex settings.

2.2 Choice of State Variables

We do not consider all variables eligible to represent state. To remain conservative, we have identified five criteria:

- Accesses to state variables must be atomic. This can be achieved in several ways: By declaring them atomic, by disabling certain compiler optimizations and using appropriate types, or by suitably enabling and disabling interrupts. This atomicity implies the absence of superfluous assignments and data races.
- State variables must be declared `volatile`. The C memory models do not provide strong enough guarantees about the visibility of updates to non-volatile variables to base synchronization on.
- Any state variable must be compared to a constant in a path predicate at least once. If they are not, they cannot possibly provide a means of synchronization.
- A variable that is never assigned a constant should not be used as it probably is of no use.
- State variables should be communication variables, i. e. they should be accesible from more than one task. Any variable local to a single task is likely to contribute little or nothing to synchronization.

Our tool rejects proposed state variables if they appear to be involved in data races and warns the user when non-volatile variables are used.

2.3 Examples Unveiled

When we use our prototype to examine the example code from figure 2, it refuses to eliminate the warning on the variable `d` seemingly protected by state-based synchronization. This is simply because the synchronization is broken—indeed, figure 1 is intentionally misleading.

State machines are safe when each state is owned by at most one task. States that have multiple owners will typically yield data races, while states with no owner result in deadlocks, as they cannot be left. Here, `s == 0` allows both tasks to access the shared resource and introducing priorities would not change anything.

The second example is tricky: It seems that Task γ allows for uncoordinated state changes breaking the state pattern. If, however, the priority of γ is lower than that of both α and β, the mechanism works and there is no data race. If γ has the highest priority, there is a data race on `d`. Our tool reports both cases correctly and is not misled by the curious assignments of `d` to `s` nor by the wrap-around semantics of `s`.

The real-life embedded C code we have encountered inspires this example. The ability to recognize this pattern leads to the elimination of false positives, which in turn saves QA staff time.

3 Conclusion and Future Work

Judging whether a given implementation of state-based synchronization works as intended is difficult and error-prone. We are confident that in future our approach will scale to industry-sized programs and dispense with manual identification of variables. Our work provides a means to verify the correctness of a given implementation. We improve upon existing solutions by reducing the requirements on state variables. We have demonstrated how our constraint-solving approach works on small programs and enables programmers to verify that synchronization patterns work as intended.

References

[1] S. Keul, "Tuning Static Data Race Analysis for Automotive Control Software," in *11th IEEE International Working Conference on Source Code Analysis and Manipulation*, 2011, pp. 45–54.

[2] M. D. Schwarz, H. Seidl, V. Vojdani, and K. Apinis, "Precise Analysis of Value-Dependent Synchronization in Priority Scheduled Programs," in *LNCS*, vol. 8318, 2014, pp. 21–38.

[3] C. A. R. Hoare, *Communicating Sequential Processes*. Prentice Hall International, 2004.

[4] Formal Systems (Europe) and Oxford University, "Failures-Divergence Refinement: FDR2 User Manual," 2010.

Performance Tuning of PDG-based Code Clone Detection

Torsten Görg
University of Stuttgart
Universitätsstr. 38, 70569 Stuttgart, Germany
torsten.goerg@informatik.uni-stuttgart.de

Abstract: *This paper provides several ideas how to improve the performance of PDG-based code clone detection techniques. We suggest an efficient way to handle the subgraph isomorphism problem without losing precision and present an algorithm that avoids the unnecessary matching effort for subclones of larger clones.*

1 Introduction

To reach a high recall in code clone detection, PDGs (Program Dependency Graphs) can be used as an intermediate representation of the analyzed code. Komondoor and Horwitz [1] have introduced this approach. Clone pairs are calculated by matching subgraphs along two synchronously constructed slices. The main advantage of PDG-based clone detection is that it abstracts from several structural code differences which are semantically irrelevant, i.e., from reordering independent statements. PDG-based clone detection is able to recognize many more semantically equivalent clones that are not structurally equivalent than structural approches like token-based or AST-based clone detection [2]. To express program semantics in detail directly in the graph structure, Krinke [3] has suggested fine-grained PDGs. The nodes of fine-grained PDGs are at a granularity level similar to 3-address-code statements. But a fine granularity results in PDGs with many nodes so that performance issues become relevant. Because of the subgraph isomorphism problem, graph matching is principally NP-complete. Krinke tackles this problem with an approximation that groups similar paths through the graph into equivalence classes. The tradeoff is a loss of precision.

Our intention is to construct a PDG-based clone detector that provides high precision along with acceptable performance. A high detection precision is important to recognize clones that are sematically equivalent and not only similar. The contributions of this paper are to suggest a precise high-performance approach to handle the subgraph isomorphism problem specifically in the context of PDG-based clone detection and to provide an algorithm that avoids unnecessary matching effort for clones that are part of another clone.

2 Handling of the Subgraph Isomorphism Problem

During the PDG matching process two nodes v_1 and v_2 are compared based on their node types and node attribute values. If v_1 and v_2 are equal, their outgoing edges are matched. We assume that both nodes have n outgoing edges. Without further distinguishing the edges there are n! possibilities to map the outgoing edges of v_1 to the outgoing edges of v_2. Each step to an adjacent node provides a similar multitude of possibilities. To check all these possibilities requires a backtracking algorithm with exponential complexity. But NP-completeness does not necessarily make an algorithm unusable, if the problem size is small.

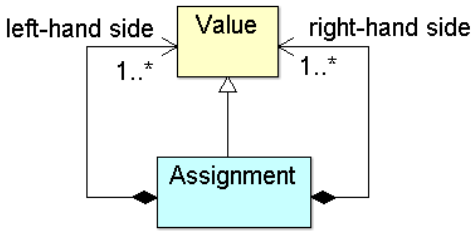


Fig. 1. Data dependency categories of assignments

In a PDG the edges represent dependencies of different types. The main categories are data dependencies and control dependencies. And there are many subcategories of data dependencies for different purposes specifically for different node types. It does not make sense to mix up these categories. E.g., an assignment is data dependent on the expression for the new value on the right-hand side and on a l-valued expression on the left-hand side that determines where the new value is stored. These two dependencies have completely different purposes. Fig. 1 shows the situation as a composite pattern in a UML class diagram. Further examples are the operands of non-commutative operators and the data dependencies of Φ-nodes. If the outgoing edges of each node are grouped into sections for the different categories, the matching process can handle each section separately and has to take into account only the inner permutations of the sections. Let k be the number of sections and n_i the number

of outgoing edges in section i: $n_1!+n_2!+...+n_k! < n!$, $\Sigma n_i=n$, $n_i>0$. Based on this observation the size of the PDG matching problem can be reduced significantly. Our approach is a generalization of the equivalence classes suggested by Krinke [3].
A special case is program code that does not use commutative operators and does not dereference pointers. In this case, $n_i=1$ for all sections and just one unique mapping is possible. We use a PDG variation that combines the PDG approach with SSA form by integrating Φ-nodes into the PDG. In conventional PDGs, $n_i>1$ at all join points, even without commutative operators and dereferenced pointers. The operands of commutative operators, like add or multiply operators, have the same purpose and are therefore grouped into the same section. Dereferencing pointers may cause n_i values greater than 1 as pointers usually have multiple target objects. Each object in the target set establishes a data dependency. To keep the n_i values small, it is important to use a precise pointer analysis as a basis of the clone detection.
The handling of sections with multpile entries can be further improved by introducing a partial order on expressions [4]. E.g., for two binary add operations x and y their summands are sorted according to the partial order on expressions:
(x_1, x_2), $x_1 \leq x_2$ and (y_1, y_2), $y1 \leq y2$. If the summands can be ordered in such a way, no permutation is needed and x_1 is uniquely mapped to y_1 and x_2 to y_2.
If the performance is still not sufficient, the entries of a section can be merged and not distinguished any more in the further process, at the cost of reduced precision, as suggested in Krinke's approach [3]. But this is not necessary in general. A performance improvement is already gained by grouping dependencies into sections, without any loss of precision.

3 Avoiding Subclones

Another important issue is the selection of suitable start nodes for the subgraph matching. A naive approach would start a comparison at each node with every other node. A consequence of this quadratic scheme is that comparisons are started at nodes which are inside of larger clones. We call the resulting clones subclones of the encompassing clones. But usually one is interested in the maximal clones only. Several clone detection techniques filter subclones in a postprocessing step, e.g., the AST-based technique of Baxter [2]. In contrast, our approach avoids a repeated matching of many subclones in advance.
At first, we start with any node $r_1 \in V$ that represents an output value at the exit of a procedure. The backward slice S_1 spanned by this node is matched against the backward slices $S_{2,i}$ spanned by all nodes $r_{2,i}$ of the same node type that have not been processed yet. Then we mark r_1 as processed. The result of a succesful match between S_1 and $S_{2,i}$ is a clone $C = (C_1, C_{2,i})$ with $C_1 \subseteq S_1$, $C_{2,i} \subseteq S_{2,i}$ and a relation $M \subset V \times V$ that describes which node matches which other node. A matching starting with any $v_1 \in C_1$, $v_1 \neq r_1$ and $v_{2,i} \in C_{2,i}$, $v_{2,i} \neq r_{2,i}$ with $M(v_1, v_{2,i})$ provides a subclone of C. The unnecessary matching process starting at the root nodes v_1 and $v_{2,i}$ is skipped. Instead, the next match attempt starts with each $q_1 \in S_1$, $q_1 \notin C_1$, where q_1 is a direct successor of a node in C_1. q_1 is processed in the same way as described above. After all nodes reachable from r1 are processed, the algorithm continues with the next procedure-output-value node. The set of output values encompasses return values, output parameters, and writing side effects. Nodes that are not processed, in the end, indicate dead code.
Although the worst case complexity is still quadratic, we expect the average performance of this algorithm to be much below that.

4 Conclusion

Although algorithms with exponential complexity are often viewed as unusable, in the context of PDG matching several performance improvements are possible. As PDGs provide a chance to push clone detections further towards the detection of semantic clones, it should be examined in more detail. A prototypical implementation of the ideas presented in this paper is currently under construction.

References

[1] Raghavan Komondoor and Susan Horwitz, "Using Slicing to Identify Duplication in Source Code," in Proc. of the 8th International Symposium on Static Analysis (SAS '01), London, UK, Springer-Verlag, 2001

[2] Chanchal Kumar Roy and James R. Cordy, "A survey on software clone detection research," technical report, Queen's University, Canada, 2007.

[3] Jens Krinke, "Identifying Similar Code with Program Dependence Graphs," in Proc. Eight Working Conference on Reverse Engineering (WCRE 2001), *S*tuttgart, Germany, pp. 301-309, October 2001

[4] Torsten Görg and Mandy Northover, "A canonical form of Arithmetic and Conditional Expressions," in Proc. of the 16th Workshop Software Reengineering & Evolution (WSRE 2014), Bad Honnef, Germany, 2014

Model Based Development

Klaus Müller and Bernhard Rumpe – A Methodology for Impact Analysis Based on Model Differencing

Dilshodbek Kuryazov and Andreas Winter – Towards Model History Analysis Using Modeling Deltas

Domenik Pavletic and Syed Aoun Raza – Multi-Level Debugging for Extensible Languages

A Methodology for Impact Analysis Based on Model Differencing

Klaus Müller, Bernhard Rumpe
Software Engineering
RWTH Aachen University
mueller@se-rwth.de, rumpe@se-rwth.de

1 Introduction

A software system typically has to be changed frequently to adapt the system to new or changing requirements or due to bug fixes. One crucial problem is that every kind of change can introduce severe errors into the software system and that it is difficult to predict in what way which parts of a software system are potentially affected by a change. Impact analysis approaches cope with this problem by trying to identify the potential consequences of changes [1].

In model-based software development, models are usually transformed into concrete implementations [2]. Even though code generators can automatically generate essential parts of a software system, it is usually still required to create and maintain further handwritten artifacts such as source code. These artifacts have to be integrated into the generated parts of the software system and, thus, they sometimes heavily depend on the generated artifacts. For example, a code generator might generate the database schema based on a UML class diagram. If developers introduce a handwritten source code file which contains SQL queries that access this database, the source code file depends on the generated database schema and consequently also on the UML class diagram. Hence, model changes can have tremendous impact on the handwritten artifacts.

In this extended abstract, we discuss a methodology for developing and applying a model-based impact analysis approach, in which explicit impact rules can be specified in a domain specific language (DSL). These impact rules embody what kind of model changes have what kind of impact. Based on such impact rule specifications, impact rule implementations are generated, which check the specified conditions and output the defined impact.

The main advantage of defining explicit impact rules is that they allow formalizing knowledge about known dependencies and characteristics of a software system. As the impact rules describe the impact of model changes, it is possible to create a checklist that informs developers about the impacts of all model changes that have been detected in a model differencing step. The resulting checklist, thus, contains concrete hints about the development steps that are (potentially) necessary to adapt the system to the model changes. Due to this, the checklists can simplify the evolution process, as developers can work through the checklists. The motivation for creating such checklists is that developers might forget to perform certain development steps that are necessary after specific model changes. This particularly holds in a complex software system.

In the next section, we elaborate on the methodology for developing and appyling the approach. Results of a case study dealing with the impacts of UML class diagram changes can be found in [3].

2 Methodology to Generate Checklists

Our methodology proposes an impact analysis approach that is composed of two steps: the identification of model differences and the application of impact rules on these differences. As a result, the approach produces a checklist which can be ticked off.

Subsequently, Subsection 2.1 outlines the steps that have to be performed to set up this impact analysis approach. After that, Subsection 2.2 outlines the steps that have to be carried out to apply the approach.

2.1 Setting up the Impact Analysis Approach

At first, a model differencing tool has to be chosen to be able to perform model differencing. If the chosen model differencing tool expects input models of a certain type, but the original input models have another type, a model converter has to be implemented or an existing one has to be integrated into the tool chain to allow for differencing the input models.

One problem that has to be considered in the context of model differencing is that a completely automatic approach to model differencing cannot infer the differences correctly in all cases [4]. Because of this, we propose to allow users to integrate knowledge of how specific model elements changed in so-called user presettings. Hence, user presettings have to be derived that fit to the corresponding input model type. Furthermore, the model differencing tool needs to be extended to be able to process user presettings [4].

These two steps are the only required steps to be able to calculate the model differences. Next, the steps that are necessary to set up the impact analysis part of the approach are sketched.

The impact analysis approach that results from applying the proposed methodology relies on impact rules capturing the consequences of changes in particular types of models. In an impact rule the user is free to define what kind of change leads to what kind of impact. To improve the comprehensibility of an impact rule, the methodology proposes a simple DSL, in which it can be specified which conditions have to be fulfilled by a model difference so that a certain checklist hint is created.

```
1  impactRule "IRExample" {
2    description = "Example description"
3    severity = critical
4    relevantFor = "mueller@se-rwth.de"
5
6    impact {
7      renamedClass() =>
8        "Implement data migration."
9    }
10 }
```

Figure 1: Simple impact rule example

A very simple example of an impact rule written in the DSL is illustrated in Figure 1. At first, a description indicating what the impact rule is used for (line 2) is denoted, then it is defined how critical violations against the impact rule are (line 3) and for which persons the hints are relevant for (line 4).

In the subsequent part, it is defined which conditions have to be fulfilled by a model difference to result in the creation of the subsequently given checklist hint (line $7-8$). According to Listing 1, a checklist would inform the developer about the necessity to implement a data migration if a class was renamed. An impact rule can contain zero or multiple blocks of such condition parts and according checklist hints. Moreover, the condition part can consist of multiple conditions that can be combined using the logical operators && and || known from Java. For each type of model change which can be found in the model differencing step and which is relevant for the impact analysis, we propose to derive a condition which checks whether the particular type of model change occurred. For instance for UML class diagrams, there would be conditions such as `renamedClass` (see line 7 of Listing 1) or `addedAssociation` [3]. These different conditions need to be implemented in the checklist tool so that the conditions can be referenced in the condition part of the impact rule DSL.

As soon as a first set of conditions has been implemented, concrete impact rules can be defined using the impact rule DSL. An impact rule generator will generate implementations of the impact rules out of the impact rule specifications written in the DSL. In some situations it can be necessary to extend this generated implementation and to add handwritten parts to a handwritten subclass [3].

2.2 Applying the Impact Analysis Approach

After having executed the steps listed in the previous subsection, the tool chain contains the required parts to identify the impacts of model changes. The workflow to apply this impact analysis approach to generate checklists is outlined in the following.

If not all impact rules should be executed when creating the checklist, the checklist tool first needs to be configured by defining which impact rules should (not) be invoked in the checklist generation. By default, all impact rules are taken into account.

Afterwards, the developers have to decide for which pairs of input models the checklists should be generated. If a model converter had been integrated into the tool chain, the next step is the invocation of the model converter to produce models that can be processed by the model differencing tool. Finally, the model differencing tool is invoked for the potentially converted pairs of input models.

Next, the checklist generator is called for the resulting difference model. Every difference contained in the difference model is passed to the impact rules. Each impact rule then analyzes the current model difference and creates a list of hints at further (potential) development steps, if the difference is regarded as relevant. These different hints are finally merged into a checklist, together with further information such as a list of detected model differences.

Before performing the development steps that are contained in the resulting checklist, developers need to verify that the reported model differences are correct. If wrong model differences have been reported, developers have to provide user presettings to fix these problems. In this case, the model differencing tool has to be invoked again and the subsequent steps have to be repeated.

Remarks This extended abstract discusses a generalization of previous work [3].

References

[1] S. A. Bohner. *A graph traceability approach for software change impact analysis*. PhD thesis, George Mason University, Fairfax, VA, USA, 1995.

[2] R. France, B. Rumpe. Model-Driven Development of Complex Software: A Research Roadmap. In *Proc. Future of Software Engineering (FUSE'07)*. Pp. 37–54. 2007.

[3] K. Müller and B. Rumpe, "A Model-Based Approach to Impact Analysis Using Model Differencing," *in Proc. International Workshop on Software Quality and Maintainability (SQM'14), ECEASST Journal*, vol. 65, 2014.

[4] K. Müller and B. Rumpe, "User-Driven Adaptation of Model Differencing Results," *International Workshop on Comparison and Versioning of Software Models (CVSM'14), GI Softwaretechnik-Trends*, vol. 34, no. 2, pp. 25–29, May 2014.

Towards Model History Analysis Using Modeling Deltas

Dilshodbek Kuryazov and Andreas Winter
Carl von Ossietzky Universität, Oldenburg, Germany
{kuryazov,winter}@se.uni-oldenburg.de

1 Motivation

The evolving complex software models are designed and maintained by a team of designers using *collaborative modeling tools* with a support of *version control*. Collaborative modeling tools provide a teamwork of several designers on a shared modeling artifact, whereas model version control is used to store, manage and handle the histories of that model. During the evolution and maintenance process of models, model designers feel a need for *history analysis* feature for tracing and comprehending the change history of a complete model or its particular artifacts.

In order to analyze the histories or trace a particular element of an evolving model, designers need to determine answers to several questions such as (1) How often does an element change? (2) When is an element created? (3) When is an element deleted? (4) Which elements are constantly changing? (5) How does the history of an element look like? (6) How was the state of a whole model in earlier versions? (7) What are the differences between any two versions of a model? etc. These analysis questionnaires are also partly defined in [3]. For answering these questions, the change histories of modeling artifacts have to be identified and stored in appropriate ways for further analysis and manipulation. To this end, this paper presents early status of history visualization to model history analysis using modeling deltas.

The differences between subsequent model versions are represented in difference documents, also referred to as *Modeling Deltas* [2]. Modeling Deltas are executable sequence of modification operations which transform a model from one state to another. Modeling deltas represent information about the whole history of a model. Thus, modeling deltas are essential for building and developing various services and components for version control, history analysis and collaborative applications on top of them. It is quite essential to reuse and exploit the model differences in further analysis and manipulations i.e. only difference representation is useless if difference information is not reusable.

The general *Delta Operations Language (DOL)*, meta-model generic and operation-based approach is introduced in [2] to model difference representations. Conceptually, DOL is a set of domain-specific languages for model difference representation in terms of operations. A specific DOL for a specific modeling language is derived from the meta-model of a modeling language. A specific DOL is fully capable of representing model differences conforming the given meta-model in terms of DOL operations. Only changed elements between model versions are identified and represented in Modeling Deltas. Each modeling delta consists of the semantic differences between subsequent model versions. DOL-based modeling delta representation is applied to model history analysis in this paper.

The remainder of the paper is structured as follows: Section 2 gives a motivating example of DOL-based difference representation. Model history analysis using modeling deltas is discussed in Section 3. Section 4 draws some conclusions.

2 Modeling Delta Representation

In order to present the idea behind the DOL-based approach to model history analysis, this section explains a simplified example of model difference representation in terms of DOL operations.

Figure 1 depicts three subsequent versions of the same UML activity diagram. The example model illustrates the case of *ordering system*. Each concept of the model is assigned to a *persistent identifier*. The first model version has one *Receive* action. In the second version, a new action *Fill Order* and control flow `g7` are created, the name of the existing action is changed to *Receive Order*, and the target of the control flow `g4` is also changed to the new action. Then, the target of control flow `g5` is reconnected back to the final node and the created action `g6` and the control flow `g7` are deleted in the third version.

Each of the modeling concepts can be *created*, *changed* or *deleted* during the evolution process. Thus, the DOL-based approach considers only these three basic operations for representing all kind of model changes ([2], [1]).

The differences between subsequent versions of that model are represented in terms of delta operations. In order to be independent from the underlying implementation technique, the most recent version (*version 3*) of the model is also represented by DOL operations. Eventually, there are two *modeling deltas* for representing the difference between three subsequent versions

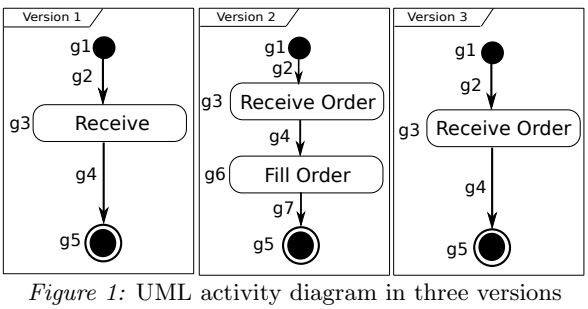


Figure 1: UML activity diagram in three versions

and one so-called *active modeling delta* for representing the recent model version. The active delta only consists of *creation* operations (Figure 2) which results in most recent version of the model.

```
1  g1=createInitialNode();
2  g3=createOpaqueAction("Receive Order");
3  g5=createActivityFinalNode();
4  g2=createControlFlow(g1,g3);
5  g4=createControlFlow(g3,g5);
```

Figure 2: Active delta

The differences delta between the third and the second versions consists of three DOL operations for creating one action (`g6=createOpaqueAction("Fill Order");`), one control flow (`g7=createControlFlow(g6,g5);`) and changing the target of `g4` (`g4.changeTarget(g6);`). In the same vein, the difference delta between the second and the first versions contains three operations changing the target of `g4` (`g4.changeTarget(g3);`), deleting `g6` (g6.delete();) and deleting `g7` (`g7.delete();`).

The approach represents differences in directed modeling deltas (*backward delta*) which are precisely executable descriptions of differences i.e. deltas are applicable to models and applying results in other version (older version in this example) of the model.

3 Model History Analysis

Analyzing model histories is the best aid in comprehending and understanding what changes are made by designers or to know how a model evolves. Also, observing the model history and its evolution process assists the users in making important decisions in further steps.

The entire set of modeling deltas in a repository represents the complete history of the model. The DOL approach also provides several DOL-services for reusing and manipulating the DOL-based modeling deltas [2]. One of these services is the *change tracer* which allows to trace the change history of a specific modeling artifact and gather required information about it. The model history analysis is built on top of the change tracer DOL-service.

The change tracer receives a list of modeling deltas and looks through a chain of modeling deltas. It seeks change information of a requested model element based on its persistent identifier by concatenating the given set of modeling deltas. The outcome of the change tracer service is a report about change history in an appropriate form. For example, the change tracer service is employed for the example in Section 2. It receives three modeling deltas (one active and two differences deltas) as input and it is requested to return history reports for the control flow `g4`. The resulting list of changes is depicted in Figure 3.

```
1  g4=createControlFlow(g3,g7);
2  g4.changeTarget(g6);
3  g4.changeTarget(g5);
```

Figure 3: History information of Control Flow `g4`.

Finally, the detected change history information can be used in further analysis by different visualizations. This approach uses a tabular view to visualize change information, but difference information can be visualized in any other forms, like model, tree, graph or even textual. Importantly, most of the questions stated in Section 1 can be answered in the current status of the visualization.

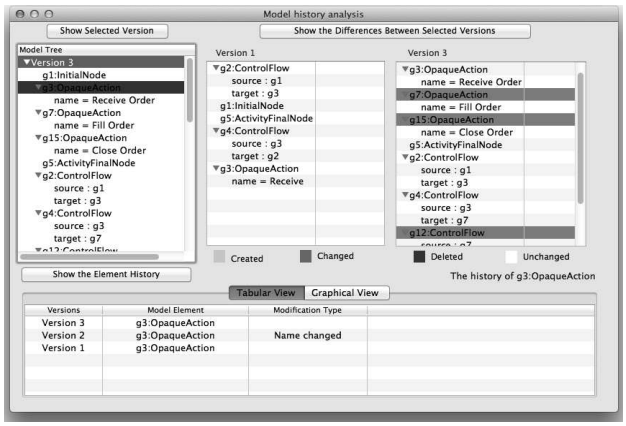


Figure 4: History Analysis User Interface

The screen shot in Figure 4 displays the example model in Section 2. The user interface shows the model tree on the left, including all versions with their elements. One model version can be selected, and a whole version can be seen by clicking *Show Selected Version* button or any two versions can be compared in the tabular view on the right side highlighting different kinds of changes with different colors. To see the history information of any model element, it can be selected from the model tree. The change history information is listed on the table on bottom.

4 Conclusion

This paper has addressed model history analysis using the DOL-based modeling deltas to model difference representations. The DOL representation is an appropriate approach for difference representation as, it makes the data representation efficient and allows suitable data structure for data processing. Modeling deltas embody all necessary information about the complete change history of a model. These information can easily be extracted and reused by the DOL-services in further analysis.

Implementation of the analysis tool is planned to be finalized in near future and the whole set of analysis questions will be covered by this tool.

References

[1] Dilshodbek Kuryazov. Delta operations language for model difference representation. In Plödereder et. al., editor, *44. Jahrestagung der Gesellschaft für Informatik*, volume 232, Stuttgart, 2014. GI.

[2] Dilshodbek Kuryazov and Andreas Winter. Representing model differences by delta operations. In Reichert et al., editor, *18th International Enterprise Distributed Object Oriented Computing Conference, Worshops and Demonstrations (EDOCW)*, pages 211–220, Ulm, 2014. IEEE.

[3] Sven Wenzel. How to trace model elements? In *9. Workshop Software-Reengineering (WSR'07)*, Bad Honnef, May 2007.

Multi-Level Debugging for Extensible Languages

Domenik Pavletic[1] and Syed Aoun Raza[2]

[1]itemis AG, pavletic@itemis.de
[2]itemis AG, raza@itemis.de

Abstract

Multi-level debugging of extensible languages requires lifting program state to the extension level while translating stepping commands to the base-level. Implementing such bi-directional mappings is feasible for languages with a low abstraction level (e. g., C). However, language workbenches support language stacking with a bottom-up approach from low- to high-level (e. g., domain-specific) languages. This way, generation of code written with these high-level languages is incremental. However, languages can have more than one generator, which is selected depending on the execution environment. On the other hand, provision of such flexibility makes multi-level debugging much harder. In this paper, we present an approach on how to enable debugging for such multi-staged generation environments. The approach is illustrated by mbeddr, which is an extensible C language.

1 Introduction

In domain-specific and model-driven software development several different abstractions come into play, where each involved entity might not want to deviate from their abstraction level. Mixed-language environments or environments with language extensibility fulfill this requirement i.e., each entity can use notations from its respective abstraction level and is therefore not forced to implement everything with a low-level language.

We discussed the significance of extensible debuggers for extensible languages in [1] with an implementation based on mbeddr [1]. Further, we described the requirements and the architecture of the debugger: it is designed for extensibility and supports debugging of mixed-language programs. However, in mbeddr users can build multiple levels of language extensions. As shown in Figure 1, programs written with these extensions are incrementally generated to some base language (e. g., C).

Currently, the mbeddr debugger supports single-level debugging of programs written with language extensions. Furthermore, debugger extensions are always implemented relative to the extension- and base-level. However, if at any level a generator for language extension is changed, then the corresponding debugger extension must be changed as well. Changes to generators happen on a frequent basis due to bug fixes or implementation of additional requirements.

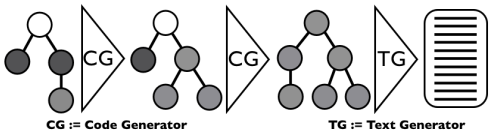


Figure 1: Incremental generation from extension- to base-level

This paper contributes an approach on how to enable multi-level debugging and reducing the effort to support generator changes. We illustrate the approach with examples based on mbeddr.

2 Requirements

The flexibility to switch between generators is an important feature of language workbenches which allow language extension. Accordingly, they must also support debuggers to provide debugging functionality for such language extensions. This introduces further requirements in addition to those defined in [1]. After analysing the application scenario, we have come to the following further requirements:

GR1 Multi-Level Debugging: Because of the semantical gap, some errors cannot be analyzed on the extension-level. Debugging the generated code is possible, however, this involves more effort because of the missing semantical richness. Hence, debugging support on different extension-levels is essential.

GR2 Seamless Integration Support: Languages can have different generators. The debugger must provide capabilities for integrating corresponding debugger extensions.

GR3 Scalability: An arbitrary number of generators can be involved during code generation. This can slow down debugging experience, however, there should not be a restriction on how many code generators can be involved.

[1]mbeddr is an extensible language [2], build with the Meta Programming System (MPS).

3 Implementation Proposal

To support multi-level debugging of extensible languages, we propose an incremental approach based on the work described in [1]. In this approach, we propose lifting program state bottom-up, whereas stepping commands are translated top-down. Figure 2 illustrates the approach: the white box represents our initial mixed-languages program, which is stepwise translated by different generators to intermediate programs (grey boxes) until it finally results in a pure base language program (black box). This representation is in contrast to our initial approach [1], which required significant re-implementation in debugger extensions if any of the generators is replaced. Because debugger extensions are always implemented in correspondence to the extension- and base-level.

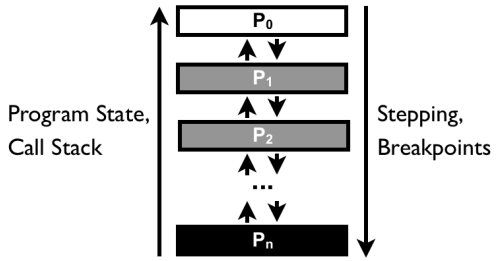


Figure 2: Bi-directional flow of debugging and generation information

Each generation step will have a corresponding debugger extension, which provides program state and propagates stepping commands with necessary information to lower debugger extensions. This way, the approach facilitates multi-level debugging (GR1). The framework will provide APIs for easily plugging in new debugger extensions (GR2). In order to construct program state or translate stepping commands, the approach will have to traverse all related debugger extensions. This way, number of extensions will only be limited by the number of generators involved during transformation (GR3).

4 Discussion

In below listings we show a multi-level transformation of a language extension that sums up a range of numbers. This extension is stepwise translated to C. Listing 1 contains the code of the high-level language extension for summarizing numbers from 0 to 10.

```
1 void main() {
2   int sum = 0;
3   sum = [0 to 10];
4 }
```

Listing 1: First Level

Listing 2 shows the generated code after the first transformation to a *loop* language extension.

```
1 void main() {
2   int sum = 0;
3   loop [0 to 10] { sum += it; }
4 }
```

Listing 2: Second Level

The listing 3 shows the complete unrolled form of `loop` after the final transformation step as a pure C program (the base language).

```
1 void main() {
2   int sum = 0;
3   sum += 0;
4   ...
5   sum +=10;
6 }
```

Listing 3: Base Level

For supporting multi-level debugging for the above described scenario a debugger extension is required for each language. In this example, if a user wants to *step over* the `sum statement` in listing 1, it involves the following steps: first, `sum statement` debugger extension sets a breakpoint on the `loop` in listing 2. Next, `loop` debugger extension sets a breakpoint on the second statement of listing 3. Finally, this information is propagated to the base-level debugger (here, `gdb`).

The previously described scenario clearly defines that it is possible to debug on different levels in a multi-level transformation. Further, this is accomplished by mapping debug information stepwise. The combination of such debugger extensions will provide multi-level debugging from highest to the base language (GR1). Additionally, it is possible to scale this approach by combining an arbitrary amount of transformation levels (GR3).

Generating a different structure requires introducing a new generator, but also a new debugger extension for this generator (GR2). Nevertheless, changes to existing generators only require re-implementation in the respective debugger extension.

5 Conclusion

This paper discusses the requirements and an implementation strategy for supporting multi-level debugging for extensible languages in mbeddr. Depending on the language workbench there can be additional requirements. However, we have discussed the general requirements which are necessary to implement basic functionality of multi-level debugging.

6 Future Directions

In the future we will investigate how debugger extensions can be implemented inside generators. Also we will analyze how much information (e. g., variable names) can be reused this way. Finally, we will investigate to which extent debugger extensions can be derived from transformation rules.

References

[1] D. Pavletic, S. A. Raza, M. Voelter, B. Kolb, and T. Kehrer. Extensible debuggers for extensible languages. *Softwaretechnik-Trends*, 33(2), 2013.

[2] M. Voelter. Language and IDE Development, Modularization and Composition with MPS. In *GTTSE 2011*, LNCS. Springer, 2011.

Migration

Johannes Meier, Dilshodbek Kuryazov, Jan Jelschen and Andreas Winter – A Quality Control Center for Software Migration

Tilmann Stehle and Matthias Riebisch – Establishing Common Architectures in a Process for Porting Mobile Applications to new Platforms

Harry Sneed – Namensänderungen in einem Reverse Engineering Projekt

Werner Teppe – Data reengineering and migration to prepare a legacy application platform migration

A Quality Control Center for Software Migration

Johannes Meier, Dilshodbek Kuryazov, Jan Jelschen, Andreas Winter
Carl von Ossietzky Universität, Oldenburg, Germany
{meier,kuryazov,jelschen,winter}@se.uni-oldenburg.de

Abstract

Software Migration, as transformation of legacy software into new software implemented in a different programming language, is motivated by selected quality goals like higher maintainability of the migrated software. To check which quality goals were reached, the inner quality of legacy and migrated software has to be determined and compared.

To investigate the inner quality of migrated software, this paper introduces a Software Migration Quality Control Center (QCC), which allows comparing the quality of legacy and migrated software systems. To this end, this paper discusses requirements for a QCC and their implementation in the Q-MIG project.

1 Motivation

Software migration is a means to translate existing legacy systems into another programming language without changing the functionality [1]. The motivation for such migrations is a system in a new programming language reaching quality goals like allowing better maintenance and improving functionality without redeveloping the complete software.

In order to achieve successful migrations by anticipating the impact of migration on the software quality, the quality of legacy and new software systems have to be observed and compared, so that the expected inner quality in terms of evolvability can be estimated.

Q-MIG[1] (Quality-driven software MIGration) aims at creating strategies and tools to investigate quality issues of software migrations from COBOL to Java [2]. To investigate the inner quality systematically, a toolchain for COBOL to Java migration [3] is combined with a Quality Control Center (QCC) with the following use cases:

1. *Compare quality of legacy and migrated software.* This use case compares the internal quality of both legacy and migrated software and helps *customer consultants* to check the final quality of the migrated software.
2. *Compare quality of migrated software using different migration tools.* This use case compares the quality of systems migrated using different tools with the goal of determining the quality of those tools. This helps *researchers* to rate and improve existing migration tools.
3. *Predict quality of migrated software before executing the migration.* This use case helps customer consultants to discuss the final quality of the migrated software before migration execution by only analyzing the legacy software. Doing this for different migration tools, allows selecting of the migration tool which produces the best quality.

The paper is structured as follows: Section 2 sketches the approach and identifies requirements for the QCC in Section 3. Applications are demonstrated in Section 4. Section 5 draws conclusions.

2 Requirements

To identify and define the inner quality of the migration results, the quality *characteristics maintainability* and *transferability* are selected from the ISO quality standard [4]. Several *metrics* relevant to the selected characteristics are chosen to directly, objectively and automatically measure the software quality [2]. The basic idea behind predicting the software quality is to estimate and use quality models, using automatically calculated metrics as independent values, and expert rated characteristics as dependent values. The measured metric values are used to predict the new values of the characteristics. The new quality characteristic values of the software can be used to satisfy the goals and use cases of Q-MIG in the QCC, for which some requirements are identified:

- *Quality Measurement (RQ1).* Metrics have to be calculated on COBOL and Java, and software experts have to rate characteristics for COBOL and Java systems.
- *Data Management (RQ2).* The QCC has to support data management tasks like importing data into the QCC, and creating detailed traceability links between legacy and migrated software.
- *Quality Analysis (RQ3).* The QCC enable quality comparisons of legacy COBOL and migrated Java systems for Use Cases 1 and 2.
- *Quality Prediction (RQ4).* The QCC has to support training quality models and predicting new characteristic values for new target systems reusing these models (Use Case 3).

3 Quality Control Center

In the upper part, Figure 1 depicts the used migration toolchain. *CobolFE* produces COBOL abstract syntax trees (ASTs) at measurement point $M2$, the *COBOL Transformator* transforms COBOL ASTs to Java ASTs ($M3$), and *JGen* produces Java source code at $M4$. The lower part displays the main components

[1]Q-MIG is a joint venture of *pro et con Innovative Informatikanwendungen GmbH, Chemnitz* and *Carl von Ossietzky University's Software Engineering Group. It is funded by Central Innovation Program SME of the German Federal Ministry of Economics and Technology – BMWi (KF3182501KM3).*

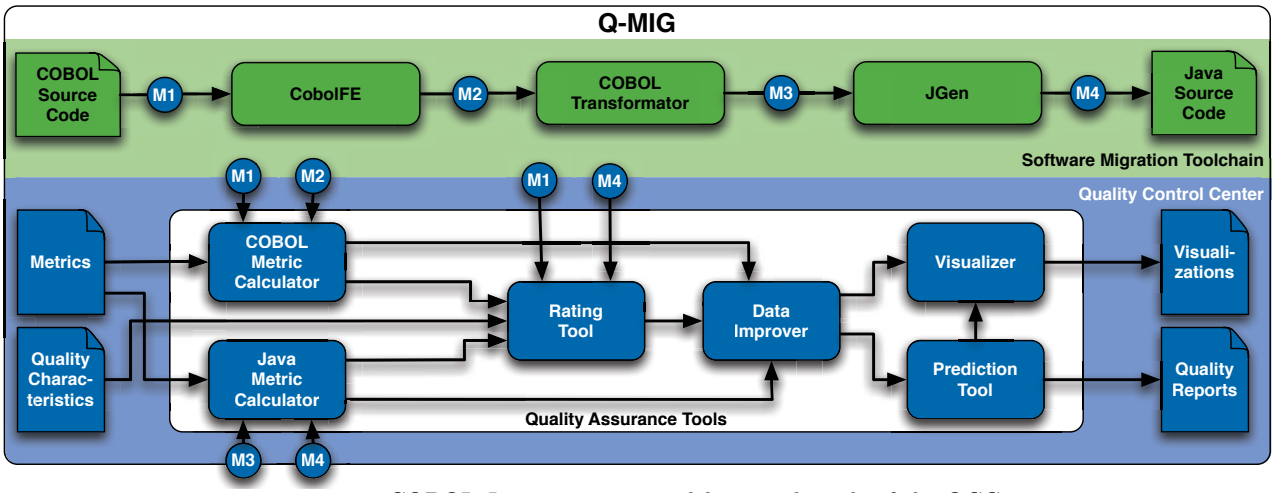


Figure 1: COBOL-Java-migration toolchain with tools of the QCC

of the QCC, which fulfills the requirements presented in Section 2. The *COBOL Metric Calculator* and the *Java Metric Calculator* calculate a given set of metrics on COBOL ($M1, M2$) and Java ($M3, M4$) source code and AST, respectively. Internally, the Java MetricCalculator uses GReQL graph queries [5] for metric implementation. The *Rating Tool* is created to support quality experts rating characteristics for COBOL and Java structures like classes and methods extracted by the metric calculators (RQ1).

The component *Data Improver* fulfills the data management tasks such as creating traceability links between the legacy COBOL and migrated Java software for later comparisons, and stores all data about the software systems and their characteristics and metric values in a central repository using TGraphs (RQ2).

The *Visualizer* creates different types of visualizations, such as HTML reports and line, bar, and scatter charts allowing to compare and analyze the quality of software systems before and after migration, as well as different migration tools (RQ3).

The *Prediction Tool* provides different machine learning algorithms including *artificial neural networks* and *multiple linear regression* for training and predicting the quality characteristics. It allows the estimation of prediction models by training samples of metrics and expert-rated characteristics. Later on, these prediction models are used to predict quality characteristics for future migrations. Quality models can be trained once and used multiple times (RQ4).

4 Application

The QCC is realized in Q-MIG and applied to three real-world examples of the project partner. After having measurement results of the metrics for COBOL (*COBOL Metric Calculator*) and Java (*Java Metric Calculator*) systems, and ratings of the software quality (*Rating Tool*), the *Visualizer* creates HTML reports, enabling both researchers and customer consultants to compare the quality of legacy and new software (Use Case 1). Additionally, further graphics like line and bar charts visualize detailed quality aspects.

The HTML reports display views for comparing different migration tools by showing Java systems migrated from the same COBOL system using different migration tools (Use Case 2). Some more visualizations are currently under development.

In order to predict the quality of prospective Java software which will be the result of future migrations, the *Prediction Tool* operates on existing quality data from previous executed migrations (Use Case 3). These prediction results help customer consultants to asses the viability of migrating the legacy COBOL software.

The QCC serves as a knowledge base to improve the quality of predictions, and will be enriched by the statistical data about further migrations which helps both researchers and customer consultants for effective quality prognosis.

5 Conclusion

This paper discussed a Quality Control Center (QCC) with several use cases for software migration quality analysis and prognosis. The QCC is realized in Q-MIG and applied to real-world applications. The QCC is valuable for researchers and customer consultants for analyzing and comparing quality of migrated software systems and making future decisions based on that knowledge. In the long term, the QCC gathers statistical data about the quality of different migrated software systems and makes prognoses on how the quality changes in prospective migrations.

References

[1] H. M. Sneed, E. Wolf, and H. Heilmann, *Softwaremigration in der Praxis: Übertragung alter Softwaresysteme in eine moderne Umgebung.* Heidelberg: Dpunkt.Verlag GmbH, 2010.

[2] G. Pandey, J. Jelschen, D. Kuryazov, and A. Winter, "Quality Measurement Scenarios in Software Migration," in *Softwaretechnik Trends*, vol. 34, no. 2. Gesellschaft für Informatik, 2014, pp. 54–55.

[3] U. Erdmenger and D. Uhlig, "Ein Translator für die COBOL-Java-Migration," *Softwaretechnik-Trends*, vol. 31, no. 2, 2011.

[4] ISO/IEC, "ISO/IEC 25010 - Systems and software engineering - Systems and software Quality Requirements and Evaluation (SQuaRE) - System and software quality models," Tech. Rep., 2010.

[5] J. Ebert and D. Bildhauer, "Reverse engineering using graph queries," in *Graph Transformations and Model-Driven Engineering*, ser. Lecture Notes in Computer Science. Springer, 2010.

Establishing Common Architectures for Porting Mobile Applications to new Platforms

Tilmann Stehle, Matthias Riebisch
{stehle, riebisch}@informatik.uni-hamburg.de
University of Hamburg, Department of Informatics

1 Introduction

Currently the market of mobile operating systems is divided between several platforms and developers have to target more than one in order to achieve a large number of users [4]. Reimplementing an existing application for a second platform is no trivial task, though. Developers need to learn the second platform's API, concepts and paradigms such as an app's life cycle, the platform's caching mechanisms and the like. On the one hand, different technologies and frameworks ease cross-platform development, such as Xamarin [8], PhoneGap and many others [5]. On the other hand, reimplementing an existing app with such a framework and discarding the original one is likely to break the maturity level and the users' acceptance in consequence. Additionally there are reasons not to use tools and languages other than those supported by the operating system producers: The native API is maintained continuously and the IDEs are strictly aligned to the latest technology.

This paper introduces a porting approach that aims at easing the maintenance of the original and the emerging implementation for the target operating system. Furthermore it contributes to the preservation of the flexibility of native development and the maturity level of the existing original app. This is achieved by restructuring the first app in order to subject most of the code to a semi-automated porting. It establishes a common architecture of the original and the ported implementation. This way it reduces the maintenance effort compared to a complete reimplementation. The following sections introduce this process and shortly specify related work. Finally, the currently conducted evaluation and future work is sketched.

2 The Mobile Porting Process

The goal of the process proposed in this section is to establish a common code base or at least a common structure for as much platform independent code as possible. This way, developers can conduct future maintenance tasks in a structurally equivalent way for both implementations and an experience portability [7] is established.

In the first step, both platforms are analyzed with regard to the API they provide for OS functionality. Considering Android as the original platform, the classes *Activity* as well as *AsyncTask* are exemplary parts of this platform API. They are used and subclassed to provide user interaction and multithreading.

Secondly, the original app's architecture is analyzed in order to find dependencies such as usage and inheritance relationships to the platform API as well as to external libraries, that are not available for the target platform. Furthermore, one has to assess, which functionality is intended to behave differently on the target platform. An example of such inherently platform specific functionality regularly is the user interface, since users of different platforms are used to different interaction and design concepts. The necessity of the identified platform dependencies is assessed and the effort for removing or isolating them from platform independent functionality is estimated in the third step.

During the fourth step, avoidable or even inappropriate platform dependencies are removed from the original implementation, for example by replacing a superfluous external library from the class path or by using a local variable instead of the platform's configuration mechanism.

In the subsequent separation step, one extracts as much platform independent code from code with platform dependencies into new, portable artifacts. This is done by building a common interface for the operating system functionality, which is implemented by platform specific code for both platforms. The platform independent codeparts have to be moved to new platform independent artifacts, which only have dependencies to the newly created interface. The interface can be implemented by adapters that either use or inherit from the specific platform classes. Depending on the considered structure one can argue, if inheritance or delegation is the right choice ([3], p.20; [6]). As a result of this separation, more platform independent functionality is freed from platform dependencies. At the same time, platform dependent code artifacts are isolated as recommended by [7]. It is important, that existing tests and documentation artifacts are adapted during the conducted changes in order to protect the existing functionality. A tool support for the refactorings can help to avoid the introduction of bugs in this phase.

After this separation, the platform independent code is transformed to the target language in the sixth step. In order to save effort for reimplementation and maintenance, it is highly desirable to use automatic code transformation tools, if available. These can also be used to translate future code changes that are conducted during maintenance. The least tool support that should be used is to

extract structural models such as class diagrams from the original code and to create class skeletons in the target language automatically.
After the (semi-)automatic translation, the isolated platform specific code is reimplemented for the target platform. The new implementation provides the common interfaces developed in the separation phase.
In order to use platform specific functionality in platform independent code (e.g. to use multithreading in business logic), instances of platform independent classes must get access to platform dependent ones. Corresponding instantiation statements can be added to the transformed independent code, or dependency injection mechanisms are introduced to both implementations in order to extract the wiring of both code categories to a single platform dependent component. As a third option, platform dependent classes can instantiate and pass platform specific objects to platform independent ones, that they create. In consequence of this wiring step, the target platform implementation becomes runnable. A comparison of those wiring concepts is subject to future works.
As a final step, traceability links between the original code artifacts, corresponding diagram elements and their target platform pendants are established to represent dependencies in an explicit way, as an important part of design information. These links may as well be creates automatically during code translation.
The process can be applied incrementally to single features of the application, thereby augmenting the target implementation run by run. This way, general problems concerning the later process steps (e.g. inappropriate translations) will become visible, the second platform version becomes testable and the customer is able to see results much earlier.

3 Related Work

In [1], common abstract architectures among several platform implementations are discussed the separation of the user interface and data manipulation components from others is suggested. The authors do not address porting though.
[9] describes a process for porting desktop applications to an iOS-Device. In contrast to the process described here, it states the necessity of using a common programming language and consequently does not cover the benefits of a common structure of two implementations. Furthermore it does not discuss an incremental application of the process.

4 Evaluation and future work

Currently, the process is applied to test projects in order to find proper decoupling mechanisms that separate platform dependent from platform independent code. Additionally, a case study is undertaken, which applies it to a real world Android application that is being ported to iOS. In this contexts, an MVVM-like pattern (cf. [2], similar to MVC) has been applied by extracting presentation logic from *Activity* classes into *ViewModel* classes and business logic further down into model classes. The iOS-version reimplements the user interface, which is wired to the ported *ViewModel*. The asynchronous initialization of *ViewModels* has been subject to the separation of technical OS-functionality (multithreading) from business logic (model initialization) in this context.
Further research has to develop decoupling mechanisms for other commonly used framework functionality and to suggest corresponding refactorings leading to a common structure. These have to be analyzed with regard to the effects they have on possible future maintenance strategies. Additionally there is a need for refactoring tools that ease the establishment of that structure. Prototypes of those tools will be developed to show the applicability of the process in a larger scale. A categorization of dependencies with regard to the effort that is needed to remove or isolate them, is desirable as well.

References

[1] ALATALO, P., JRVENOJA, J., KARVONEN, J., KERONEN, A., AND KUVAJA1, P. Mobile application architectures. In *Product Focused Software Process Improvement*, M. Oivo and S. Komi-Sirvi, Eds., vol. 2559 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2002, pp. 572–586.

[2] FURROW, A. Introduction to mvvm. http://www.objc.io/issue-13/mvvm.html, 6 2014.

[3] GAMMA, E., HELM, R., AND JOHNSON, R. E. *Design Patterns. Elements of Reusable Object-Oriented Software.*, 1st ed., reprint. ed. Addison-Wesley Longman, Amsterdam, 1995.

[4] GARTNER. Gartner says worldwide traditional pc, tablet, ultramobile and mobile phone shipments on pace to grow 7.6 percent in 2014, 01 2014.

[5] HEITKÖTTER, H., HANSCHKE, S., AND MAJCHRZAK, T. Evaluating cross-platform development approaches for mobile applications. In *Web Information Systems and Technologies*, J. Cordeiro and K.-H. Krempels, Eds., vol. 140 of *Lecture Notes in Business Information Processing*. Springer Berlin Heidelberg, 2013, pp. 120–138.

[6] KEGEL, H., AND STEIMANN, F. Systematically refactoring inheritance to delegation in java. In *Proceedings of the 30th International Conference on Software Engineering* (New York, NY, USA, 2008), ICSE '08, ACM, pp. 431–440.

[7] MOONEY, J. Strategies for supporting application portability. *Computer 23*, 11 (Nov 1990), 59–70.

[8] PETZOLD, C. *Creating Mobile Apps with Xamarin. Forms*. Microsoft Press, 2014.

[9] SCHMITZ, M. *Strategie für die Portierung von Desktop-Business-Anwendungen auf iOS-gestützte Endgeräte*. BestMasters. Springer Fachmedien Wiesbaden, 2014.

Namensänderung in einem Reverse Engineering Projekt

Harry M. Sneed
SoRing Kft. H-1221 Budapest
Harry.Sneed@T-Online.de

Abstrakt: In diesem Beitrag wird das Problem der verstümmelten Daten- und Prozedurnamen im alten Code für Reengineering Projekte angesprochen. In den meisten Legacy-Systemen sind diese Namen mnemotechnische Abkürzungen die keiner mehr versteht. Dies verhindert, dass Diagramme und andere Dokumente, die aus dem Code gewonnen werden, verständlich sind. Sollte der Code transformiert werden, z.B. von COBOL in Java, bleibt auch der Java Code unleserlich und wird von zuständigen Entwicklern abgelehnt. Um erfolgreiche Reverse- oder Re-Engineering Projekte durchzuführen muss etwas mit den Namen im Code geschehen. Hier wird eine Lösung beschrieben, die für ein Mainframe-Migrationsprojekt in einer Landeverwaltung angewandt wurde. Mit Hilfe dieser Lösung konnten verständliche Dokumente aus altem VisualAge Code gewonnen werden, die als Basis für eine Re-Implementierung dienen.

Schlüsselwörter: Migration, Reverse Engineering, Reengineering, Datennamen, VisualAge, Java, Wiederverwendung, Umbenennung.

1 Hintergrund der Reverse Engineering

Eine Landesverwaltung in Österreich hat ein altes Personalabrechnungssystem aus den 90er Jahren, das sie erneuern möchte. Das System um das es sich handelt läuft auf einem Mainframe-Rechner und ist teils in VisualAge und teils in PL/I implementiert. Die Frontend Online-Programme sind in VisualAge, die Backend Batch-Programme sind in PL/I. Der VisualAge Teil soll als erster mit Java re-implementiert werden. Die Benutzeroberflächen werden neu gestaltet. Um den neuen Java Code zu schreiben müssen die Entwickler wissen was in dem alten Code steckt. Vor allem die Entscheidungslogik, bzw die Geschäftsregel, soll verstanden werden. Deshalb wurde entschieden eine Redokumentation der alten VisualAge Prozeduren durchzuführen.

Das Ziel der Reverse Engineering war es, drei Sichten auf jede Prozedur sowie einen Prozedurbaum für jedes Programm zu erstellen. Die drei Sichten sind:

- Ein Struktogramm
- Eine Datenverwendungstabelle und
- Einen Aufrufbaum

Diese drei Sichten sollten dazu dienen die Prozedurlogik verständlicher zu machen. Der Prozedurbaum sollte einen Überblick über die Zusammenhänge aller Prozeduren in einem Programm vermitteln. Es sind insgesamt mehr als 7000 Prozeduren in 196 Programmen mit 577296 Anweisungen .

Die einzelnen Prozeduren sind in der CSP Sprache von IBM geschrieben, ein Kreuz zwischen PL/I und COBOL. Der Code ist an sich gut strukturiert, die Daten- und Prozedurnamen aber verschlüsselt. Das macht den Code schwer zu lesen. (siehe Beispiel 1)

```
MOVE ' ' TO TXA05U.TXTEOD;
MOVE 1 TO QX05W01.QX05SCZ;
WHILE QX05W01.QX05SCZ <= 20;
   IF QX05W12.PNRINT[QX05SCZ] ^= ' ';
```

Die ersten Diagramme, die aus dem Code mit einem Dokumentationswerkzeug namens ***SoftRedoc*** generiert wurden, erwiesen sich als wenig brauchbar für jemanden, der mit den verschlüsselten Namen nicht vertraut ist. Demzufolge wurde entschieden die Namen in dem ursprünglichen Code zu verändern, ehe mit der Nachdokumentation weiter gemacht wird.

2 Die Gewinnung der Prozedur- und Datennamen aus den Kommentaren

In VisualAge ist der Code in verschiedenen Source-Typen aufgeteilt:

- Datendefinitionen
- Satzdefinitionen
- Maskendefinitionen
- Tabellendefinitionen und
- Prozeduren.

Zu jedem Datenelement, jedem Datensatz, jeder Maske, jeder Tabelle und jeder Prozedur gibt es ein eigenes Source Member. Dort befindet sich in der Regel, aber nicht überall, neben dem CSP-Namen des Elementes auch ein Kommentar - desc, das beschreibt was der Name bedeutet. (siehe Beispiel 2)

```
:item  name     = TXTEOD
       date = '09/28/2004' time = '23:36:16' type = CHA
       bytes    = 00001    decimals = 00    evensql = N
       desc     = 'Kennzeichen End of Data'
```

Ein Tool wurde entwickelt um die Source-Code Dateien durch zu scannen und die Namen zusammen mit den Kommentaren herauszuschneiden. Es wurde unterschieden zwischen drei Namensarten:

- Datenelementnamen
- Datensatznamen und
- Prozedurnamen

Für jede Namensart wurde eine Excel Tabelle erzeugt mit dem originalen Kurznamen, das Kommentar und der neue Java Name. Der neue Name ist eine Konkatenierung des Kurznamen und das modifizierte Kommentar. In dem Kommentar sind ungültige Sonderzeichen und Leerstellen mit Unterstichen ersetzt. Der ursprüngliche Kurzname wird als Präfix beibehalten um die Verbindung zum alten Namen zu behalten und die Eindeutigkeit zu gewährleisten, denn viele der Kommentare sind dupliziert. Die Excel Tabellen werden anschließend in eine relationale Datenbank geladen, wo sie abgefragt und mit einander verknüpft werden können. (siehe Beispiel 3)

```
CHA ;1;0;1;QPENDVRP   ;lfd_Nr_Dienstv_Pensionist
CHA ;2;0;1;URLAT2     ;Arbeitstage_2
CHA ;1;0;1;TXTEOD     ;Kennzeichen_End_of_Data
CHA ;1;0;1;QLSMKZ     ;KZ_ob_Zeile_gefuellt_ist
CHA ;13;0;1;QLGHZULMIN1 ;minZULErh1
CHA ;1;0;1;QUEVTR1    ;Trennungszeichen
```

Insgesamt wurden 16,384 elementare Datennamen, 1,590 Satznamen und 7,459 Prozedurnamen in die Namensdatenbank gespeichert.

3 Die Veränderung der Namen im Code

Nachdem die Namenstabellen aufgebaut sind, wird in einem zweiten Durchlauf durch den Code die Kurznamen in den Anweisungen durch die Langnamen ersetzt. Wenn ein Kurzname vorkommt wird er in der entsprechenden Tabelle gesucht und durch den langen Namen ersetzt. Bei Datendeklarationen ist dies eine einfache Überschreibung aber bei den Prozeduren werden die Textzeilen verlängert. Dort wo die maximale Zeilenlänge überschritten wird muss die Zeile zerlegt und eine neue Zeile eingeschoben werden. So wächst die Größe der Source-Members im Verhältnis zur Anzahl neuer Zeilen.

Wichtig dabei ist die Konsistenz der Namen. Die aufgerufenen Prozeduren müssen überall gleich heißen, ebenfalls die referenzierten Daten. Da der veränderte Code nur zu Dokumentationszwecken verwendet wird, ist es nicht so schlimm wenn Fehler im Codetext vorkommen, dennoch wurde versucht den Code so genau wie möglich zu reproduzieren. Schließlich soll der veränderte Code benutzt werden, die Dokumente zu generieren. (siehe Beispiel 4)

```
MOVE ' ' TO TXA05U. TXTEOD_Kennzeichen_End;
MOVE 1 TO QX05W01_LWS. QX05SCZ_Zaehler
_Anzahl_Kinder;
WHILE QX05W01_LWS.QX05SCZ_Zaehler_
Anzahl_Kinder <= 20;
 IF      QX05W12.PNRINT_lfd_Nr_Person[QX05SCZ
_Zaehler_Anzahl_Kinder] ^= ' ';
```

4 Generierung der Programmdokumente

Die Erstellung der Programmdokumente und der Programmbeziehungstabelle wurde mit dem veränderten Code durchgeführt. Aus jeder der 7000 CSP-Prozeduren wurde ein Struktogramm, der Entscheidungslogik, ein HIPO Diagramm der Ein- und Ausgabendaten sowie ein Protokoll aller Unterprogrammaufrufe abgeleitet und als PDF-File gespeichert. Danach wurden die Programme analysiert und für jedes Programm, bzw. Anwendungsfall, einen Baum der dazugehörigen Prozeduren erstellt. Dieser Prozedurbaum erfüllt für prozedurale Programme den gleichen Zweck wie das Sequenzdiagramm für objekt-orientierte Programme. Aus den Datensatz-Zuweisungen wird ein Datenbaum produziert mit den Sätzen, Tabellen und Feldern als Knoten. Dieser Datenbaum erfüllt den gleichen Zweck wie ein Klassendiagramm. Schließlich wird aus den Maskendefinitionen, bzw. ein statischer Oberflächenprototyp im HTML Format erzeugt, damit der Java Entwickler sehen kann, wie die Oberfläche aussieht ohne das System auf dem Mainframe dynamisch ausführe*n zu müssen.

Zum Schluss werden alle Einzeldokumente in einem einzigen großen PDF Dokument zusammengefasst, in dem alles von der Oberfläche bis zu den if-Anweisungen erfasst ist. Der Entwickler muss nur einen Prozedurknoten anklicken und er bekommt den dahinterliegenden Code in Struktogramm-Format zu sehen. Über das Baumdiagramm kann der Java Entwickler durch das alte System navigieren, zumindest was innerhalb eines Anwendungsfalles passiert.

5 Grenzen der Nachdokumentation

Der Ansatz zur Nachdokumentation eines bestehenden Softwaresystems, der hier in einem Verwaltungsprojekt verfolgt wurde ist ein Bottom-Up Ansatz. Man beginnt mit der Analyse einzelner Codebausteine und fügt die Information die man gewonnen hat zusammen. Mit dieser Vorgehensweise kommt man nur bis zur Ebene der einzelnen Dialog-, bzw. Batchprogramme. Der Überbau, bzw. der Geschäftsprozess und die übergeordnete Architektur bleiben außer Sichtweite. Es ist nicht möglich zu erkennen, in welcher Reihenfolge und unter welchen Bedingungen die einzelnen Anwendungsfälle ausgeführt werden. Diese Information kann aus dem Code nicht gewonnen werden.

Die Qualität der Nachdokumentation kann nur so gut sein wie die Qualität des Codes selbst und diese hängt im Wesentlichen von der Benennung ab. Ohne sprechende Namen haben die Programmdokumente, egal in welcher Form immer, nur einen begrenzten Wert. In dem hier beschriebenen Projekt konnten neue Namen aus den Kommentaren gewonnen werden um die alten Namen zu ersetzen. Dies hat viel zur Lesbarkeit der Dokumente beigetragen. In künftigen Projekten dieser Art wird diese Benennungstechnik weiter verfeinert, aber man stoßt hier sehr bald auf die Grenzen der Automation. Darüber hinaus müssen die Menschen eingreifen.

Literaturhinweise:

Data Reengineering, Evolution and Migration to Prepare a Legacy Application Platform Migration

Werner Teppe
Amadeus Germany GmbH
Email: wteppe@de.amadeus.com

Abstract: Langlebige Softwaresysteme erfahren während ihrer Lebenszeit vielfältige Änderungen und Anpassungen. So werden Fehler behoben und kleinere Anpassungen durchgeführt (Maintenance). Massive Erweiterungen auf Grund von Kundenanforderungen können an die Grenzen der anfänglichen gewählten Architektur gehen. Das gleiche kann bei Anwendungsrückbauten auftreten, Außerdem kann sich das Applikationsumfeld ändern: neue Technologien kommen auf bei Hardware, Software, Middleware usw. In jedem der letztgenannten Fälle gilt es zu entscheiden, ob man zu einer "Standardsoftware" wechseln soll, die Anwendung neu entwickeln oder migrieren soll. Wenn der Funktionsumfang der Anwendung nahezu unverändert bleiben kann, bietet die Migration Vorteile (Kosten, Risikominimierung u.a. [2, [3]).

Auf früheren Workshops wurde über ARNO - ein großes industrielles Migrationsprojekt – berichtet.
In diesem Projekt haben wir erfolgreich eine Onlinetransaktions-Applikation von BS2000 nach Solaris migriert. Die aus mehr als 6 Millionen Lines of Code bestehende Applikation wurde von SPL (ein PL1 Subset) nach C++, die mehr als 5000 Jobs von SDF nach Perl und das hoch performante Filehandling-System von rund 800 Dateien nach Oracle migriert.

Um die Komplexität der Migration zu beherrschen, entschieden wir damals, die Datenmigration einfach zu halten. Daher wurden aus Datensätzen im BS2000 nun einfache Relationen in Oracle. Sie bestehen nur aus einem Index und aus einem langen Feld (BLOB - Binary Large Object).
So konnten wir erreichen, dass die in der Anwendung enthaltene Navigation auf den Daten nur wenig geändert werden musste.

Um die Weiterentwicklung der Anwendungen zu erleichtern, wird nun die Datenhaltung auf ein „echt" relationales System umgestellt. Über die Herausforderungen, die angestrebten Lösungen und das Vorgehen, die in diesem konkreten Praxisfall anstehen, wurde auf der WSRE 2014 berichtet. In der Zwischenzeit wurde das Projekt, das erneut einen Wechsel auf eine modernere Hardwarearchitektur und einen Betriebssystemwechsel vorbereiten und durchführen soll, fortgeführt. Neben der „rein technischen" Migration gilt es, die betroffenen Mitarbeiter einzubinden (Migration of the people).

In dem Vortrag auf dem WSRE 2015 werden die angewendeten Methoden, die verwendeten Werkzeuge und die erreichten Zwischenziele vorgestellt sowie ein Ausblick auf die nächsten Projektschritte gegeben.

Literatur

[1] Werner Teppe: Redesign der START Amadeus Anwendungssoftware. Softwaretechnik-Trends 23(2) (2003)

[2] Werner Teppe: The ARNO Project: Challenges and Experiences in a Large-Scale Industrial Software Migration Project; Proceedings European Conference on Software Maintenance and Reengineering (CSMR), pp. 149-158, 2009

[3] Werner Teppe: Teststrategien in komplexen Migrationsprojekten. Softwaretechnik-Trends 29 (2009)

[4] Werner Teppe: Wiedergewinnung von Informationen über Legacy-Systeme in Reengineeringprojekten. Softwaretechnik-Trends 30 (2010)

[5] Werner Teppe: Ein Framework für Integration, Build und Deployment bei Maintenance- und Reengineering-Prozessen. Softwaretechnik-Trends 31(2) (2011)

[6] Christian Zillmann, Andreas Winter, Alex Herget, Werner Teppe, Marianne Theurer, Andreas Fuhr, Tassilo Horn, Volker Riediger, Uwe Erdmenger, Uwe Kaiser, Denis Uhlig, Yvonne Zimmermann: The SOAMIG Process Model in Industrial Applications. CSMR 2011: 339-342

[7] Werner Teppe: Migrationen - (K)eine Alternative für Langlebige Softwaresysteme? Softwaretechnik-Trends 33(2) (2013)

[8] Uwe Kaiser, Uwe Erdmenger, Denis Uhlig, Andreas Loos: Methoden und Werkzeuge für die Software Migration. In: Proceeding of: 10th Workshop Software Reengineering, 5-7 May 2008, Bad Honnef

[9] Uwe Erdmenger, Denis Uhlig: Konvertierung der Jobsteuerung am Beispiel einer BS2000-Migration. Softwaretechnik-Trends 27(2) (2007)

[10] Uwe Erdmenger: SPL- Sprachkonvertierung im Rahmen einer BS2000 Migration. Softwaretechnik-Trends 26(2) (2006)

[11] Werner Teppe: Data Reengineering and Evolution in (industriellen) Legacy Systemen. Softwaretechnik-Trends 34(2) (2014)

Quality

Nils Göde – Quality Control in Action

Jens Borchers – Software-Qualitätsmanagement im Rahmen von Application Management Services

Martin Brandtner, Philipp Leitner and Harald Gall – Profile-based View Composition in Development Dashboards

Quality Control in Action*

Nils Göde

CQSE GmbH
Lichtenbergstr. 8, 85748 Garching bei München
goede@cqse.eu

Zusammenfassung

Trotz der großen Zahl statischer Analysewerkzeuge, die Qualitätsdefizite im Quelltext aufzeigen, nimmt die Qualität in den meisten Systemen kontinuierlich ab. Der Schritt von der reinen Beobachtung zu einer echten Verbesserung stellt in der Praxis immer noch eine große Hürde dar. Dieser Beitrag beschreibt unsere Erfahrungen bei der schrittweisen Einführung eines Quality-Control-Prozesses für ein Java-System in der Versicherungsbranche. Unsere Ergebnisse zeigen, dass allein durch den Einsatz von statischer Analyse noch keine Qualitätsverbesserung erzielt wird. Durch einen begleitenden Quality-Control-Prozess lässt sich die Qualität allerdings kontinuierlich steigern.

1 Quality Control

Das Ziel von Quality Control ist die Reduktion der Qualitätsdefizite („Findings“) im Quelltext und die damit einhergehende langfristige Sicherung der Wartbarkeit. *Quality Control* fasst dabei alle Maßnahmen zusammen, die für die Verbesserung der Qualität erforderlich sind. Eine ausführlichere Beschreibung des Vorgehens findet sich in [5].

2 Setup

Bei dem beobachteten Java-System handelt es sich um ein betriebliches Informationssystem aus der Versicherungsbranche mit einer Historie von mehreren Jahren. Der Quelltext wird kontinuierlich durch das Analysewerkzeug *Teamscale*[3, 4] auf Qualitätsdefizite untersucht. Die Ergebnisse stehen den Entwicklern durch ein Web-Interface sowie direkt in der IDE zur Verfügung.

Zusätzlich finden monatliche Retrospektiven statt in denen wir als externe Experten der CQSE die Entwicklung der Qualität mit den Entwicklern diskutieren. Auf dieser Basis erstellen wir für ausgewählte Findings konkrete Arbeitsaufträge („Tasks“), die im Issue-Tracker eingestellt und bei der Sprintplanung berücksichtigt werden.

*Das diesem Artikel zugrundeliegende Vorhaben wurde mit Mitteln des Bundesministeriums für Bildung und Forschung unter dem Förderkennzeichen **Q-Effekt, 01IS15003A** gefördert. Die Verantwortung für den Inhalt dieser Veröffentlichung liegt bei den Autoren.

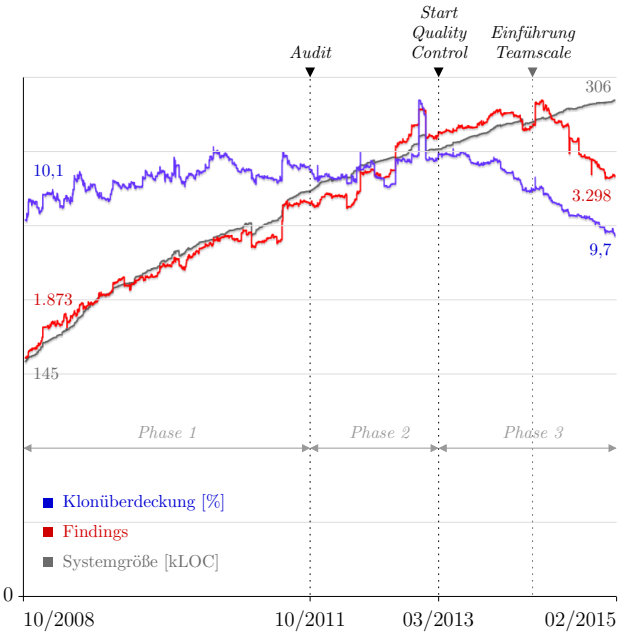


Abbildung 1: Trend zentraler Metiken

3 Evolution

Abbildung 1 zeigt die Entwicklung der zentralen Metriken vom Beginn der Versionsverwaltung bis heute. Diese beinhalten die Größe des Systems, die Klonüberdeckung und die Zahl der Findings. Die Abbildung zeigt, dass sich die Systemgröße im beobachteten Zeitraum etwa verdoppelt hat. Die Zahl der Findings ist auch deutlich gestiegen, während die Klonüberdeckung leicht gesunken ist.

Die Evolution lässt sich grob in drei Phasen unterteilen. Der Übergang von der ersten zur zweiten Phase besteht aus einem umfassenden Audit des Quelltextes durch die CQSE und der Einführung von *ConQAT*[1] zur kontinuierlichen Analyse des Quelltextes. Der Übergang zur dritten Phase ist charakterisiert durch den Beginn des Quality-Control-Prozesses. Im Laufe der dritten Phase wurde *ConQAT* zudem durch das Werkzeug *Teamscale* abgelöst.

Während die Systemgröße in jeder Phase etwa gleichbleibend zunimmt, nimmt die Zahl der Findings und die Klonüberdeckung in den ersten Phase deutlich zu. Auch in der zweiten Phase nimmt die Qua-

lität weiter ab – trotz der Tatsache, dass der Audit den Entwicklern einen umfassenden Überblick über die Qualitätsdefizite gegeben hat und der Quelltext kontinuierlich durch *ConQAT* analysiert wurde. Erst in der dritten Phase, nach der Einführung des Quality-Control-Prozesses, ist eine Trendwende erkennbar, die durch die Einführung von *Teamscale* noch einmal deutlich verstärkt wird.

4 Beobachtungen

Dieser Abschnitt fasst unsere im Laufe des Projektes gesammelten Beobachtungen zusammen, die gleichzeitig die Ausgangsbasis für zukünftige Verbesserungen des Quality-Control-Prozesses bilden.

4.1 Externer Quality Engineer (QE)

Es hat sich bewährt die Rolle des QE außerhalb des Entwicklungsteams zu besetzen. Anderenfalls besteht die große Gefahr, dass das Thema Qualität permanent zugunsten des Tagesgeschäftes vernachlässigt wird. Durch die externe Besetzung ist sichergestellt, dass das Team regelmäßig mit dem Thema Qualität konfrontiert wird.

4.2 Interner Ansprechpartner

Zusätzlich zum QE hat es sich bewährt, einen Ansprechpartner aus dem Team zu ernennen, der unter anderem die Kommunikation mit dem QE führt. Die Hauptaufgabe dieser Person ist dafür zu sorgen, dass die vom QE erstellten Tasks und Themen Einzug in die Team-interne Sprintplanung erhalten.

4.3 Tasks in Sprintplanung

Um die Bearbeitung der vom QE erstellten Tasks zu gewährleisten, müssen diese gleichberechtigt mit anderen Änderungen in der Entwicklungsplanung berücksichtigt werden. Hierfür sollte der interne Ansprechpartner aus dem Team verantwortlich sein. Bevor dies praktiziert wurde, sind Qualitätsverbesserungen häufiger im Zuge vieler anderer Änderungen „vergessen“ worden.

4.4 Pfadfinderregel

Das Befolgen der Pfadfinderregel – den Code qualitativ besser oder zumindest gleichwertig zu hinterlassen wie man ihn vorgefunden hat – wird in der Praxis leider noch nicht konsequent befolgt. Das Problem an dieser Stelle ist zum einen die Frequenz der Analyse (erst nach dem Commit stehen die neuen Findings zur Verfügung). Zum anderen stellt sich die Behebung mancher Findings als sehr aufwändig heraus, so dass das Verhältnis des Aufwands zwischen der eigentlichen Änderung und der Qualitätsverbesserung nicht gerechtfertigt scheint.

4.5 IDE-Integration

Ohne die Integration der Findings in die IDE ist eine Qualitätsverbesserung kaum zu erzielen. Der Kontextwechsel von der IDE in z.B. eine Web-basierte Ansicht ist nicht praktikabel. Erst mit der Einführung des *Teamscale*-Plugins ist die Zahl der Findings, die im Rahmen anderer Änderungen entfernt wurden spürbar gestiegen.

4.6 Web-Dashboard

Auch wenn die Web-basierte Ansicht im Vergleich zur einfachen Ansicht der Findings in der IDE eine Fülle weiterer Informationen bietet, so wird diese von Entwicklern fast nicht berücksichtigt. Andererseits ist die Web-basierte Ansicht für den QE zwingend erforderlich, da nur hier die notwendigen Delta-Informationen [2] und die Evolution der Findings sichtbar ist.

4.7 Know-How Transfer

Durch die monatlichen Retrospektiven, bei denen ausgewählte Qualitätsdefizite im ganzen Team diskutiert werden, entsteht ein nicht zu unterschätzender Know-How-Tansfer. Dies betrifft sowohl Wissen über die Java-Programmierung im Allgemeinen als auch Wissen über das System im Speziellen. Zudem werden unter Umständen übergreifende Probleme diskutiert, die sich aus der lokalen Sicht einzelner Entwickler nicht lösen lassen.

4.8 Ressourcenbedarf

Um Qualitätsverbesserung in der Praxis zu erreichen, müssen dem Thema entsprechende Ressourcen zur Verfügung gestellt werden. In diesem konkreten Projekt wird in jedem Sprint ein festgelegter Prozentsatz des Entwicklungsbudgets für nicht-fachliche technische Qualitätsverbesserungen reserviert. Dadurch wird vermieden, dass Qualitätsverbesserungen dem Tagesgeschäft zum Opfer fallen.

5 Zusammenfassung

Der Trend in Abbildung 1 zeigt, dass durch Quality Control eine kontinuierliche Qualitätsverbesserung erzielt wird. Aufgrund der überwiegend positiven Erfahrungen und Ergebnisse wird das Verfahren zukünftig auf weitere Projekte ausgeweitet.

Literatur

[1] CQSE GmbH. ConQAT. `www.conqat.org`.

[2] N. Göde and F. Deissenboeck. Delta analysis. *Softwaretechnik-Trends*, 32(2), 2012.

[3] N. Göde, L. Heinemann, B. Hummel, and D. Steidl. Qualität in Echtzeit mit Teamscale. *Softwaretechnik-Trends*, 34(2), 2014.

[4] L. Heinemann, B. Hummel, and D. Steidl. Teamscale: Software quality control in real-time. In *Proceedings of the 36th International Conference on Software Engineering*, 2014.

[5] D. Steidl, F. Deissenboeck, M. Poehlmann, R. Heinke, and B. Uhink-Mergenthaler. Continuous software quality control in practice. In *Proceedings of the 30th International Conference on Software Maintenance and Evolution*, 2014.

Software-Qualitätsmanagement im Rahmen von Application Management Services

Jens Borchers
Sopra Steria Consulting Hamburg
Email: jens.borchers@soprasteria.com / jensborchers@acm.org

Abstract: *Application Management Services sind als Outsourcing-Modell für den Betrieb und die Wartung von Softwaresystemen mittlerweile weit verbreitet. Dabei werden häufig beide Stränge, der „run the business" (RTB) und „change the business" (CTB) –Teil, vom selben Dienstleister betreut. Dieser hat damit wie der Auftraggeber selbst ein vitales Interesse an einer stabilen Software, die außerdem in der Wartung nicht zu unnötig Kosten führt. Dazu ist es notwendig, die Qualität der Anwendungssysteme nicht nur in fachlicher Richtung (durch entsprechende Tests) abzusichern, sondern auch die nicht-funktionalen und Produktivitätsmetriken aktiv zu messen und für eine optimierte Steuerung der Wartungsressourcen einzusetzen. Dieser Beitrag beleuchtet den Einsatz von Software-Qualitätssicherung im Rahmen von Application Management Services aus Sicht eines AMS-Dienstleisters.*

1 Einführung

1.1 Application Management Services

Wie in [1] dargestellt, ist das Outsourcing von IT-nahen Aufgaben eine seit vielen Jahren etablierte Strategie, mit der Unternehmen sowohl Kosten- als auch andere strategische Ziele umsetzen. Dabei sind auch sog. Application Management Services (AMS) zunehmend anzutreffen. Beim AMS übernimmt ein Dienstleister den gesamten Betrieb und i.d.R. auch die Wartung eines oder mehrerer Anwendungssysteme. Für AMS sind i.d.R. umfangreiche Vertragswerke erforderlich, in denen die zu erbringenden Leistungen sehr detailliert vereinbart werden, und zwar auch in Form sog. „Service Level Agreements" (SLA), vgl. [1], die auch Qualitätsvorgaben umfassen.

1.2 Software-Qualitätssicherung

Die Qualitätssicherung von Software ist eine seit vielen Jahren bekannte Disziplin innerhalb der Entwicklung und Wartung von Softwaresystemen (vgl. [2]).
Im Rahmen von AMS spielt dabei für beide Vertragspartner eine gemeinsam abgestimmte Ermittlung der Software-Qualität eine essentielle Rolle. Dabei geht es neben den fachlichen Aspekten auch um nicht-funktionale Anforderungen. Diese basieren i.d.R. auf den ISO-Standards der ISO/IEC 250xx-Reihe ([5]).
In den folgenden Kapiteln wird dargestellt, welche Qualitätssicherungsaspekte im Rahmen von AMS eine Rolle spielen und wie sie in der Praxis umgesetzt werden können.

2 Einsatz von Qualitätsmessungen im AMS-Lebenszyklus

Die Hauptphasen eines AMS-Engagements stellen sich – vereinfacht dargestellt – in der Regel wie folgt dar:

a. Ausschreibung der AMS-Leistungen ;
b. Ermittlung der wesentlichen Angebotsfaktoren durch potentielle Auftragnehmer;
c. Auswahl von einem bis (i.d.R. max.) drei Anbietern für Vertragsverhandlungen;
d. Ausarbeitung und Unterzeichnung des detaillierten Vertragswerks;
e. Überführung der Systeme in die Hoheit des Auftragnehmers;
f. Betrieb und Wartung der Anwendungssysteme über den vereinbarten Zeitraum;
g. Rückführung der Systeme an den Auftraggeber oder einen anderen Dienstleister bei Ende des AMS-Vertrags, die sog. „Reverse Transition".

Wo kommen nun in diesen Phasen Qualitätsmessungen zum Einsatz und in welchem Umfang?

Vertragsanbahnungs-Phasen (a – b)
Bereits der Auftraggeber kann in seinen Ausschreibungsunterlagen Aussagen zu wesentliche Kennzahlen der auszulagernden Systeme machen. Diese können aber in der Detailtiefe stark variieren. Wenn es um die Einschätzung des Risikos und die Entscheidung über die Investition in ein Gesamtportfolio von Anwendungssystemen geht, kann diese Entscheidungsbasis mit Hilfe einer sog. Rapid Portfolio Analysis (RPA) hergestellt werden.

Vertragsdetaillierungs-Phasen (c – d)
Hier wird häufig der Begriff der „Due Diligence" verwendet eine mit „gebotener Sorgfalt" durch-

geführte Risikoprüfung“ der zu übernehmenden Systeme in Bezug auf die durch den Dienstleister zu übernehmenden Verantwortlichkeiten umfasst. Erst nach einer entsprechenden Due Diligence kann der Anbieter seriös ein bindendes Vertragsangebot erstellen.

Transition- und Betriebs-Phasen (e – f)
Nach dem Vertragsabschluss mit dem ausgewählten Dienstleister beginnt die sog. „Transition“, in der alle Systeme und zugehörigen Prozesse vom Auftraggeber in die Verantwortung des AMS-Dienstleisters übergehen. Für die eigentliche Betriebs- (RTB-) und Wartungs- (CTB-) Periode, die ja normalerweise mehrere Jahre dauert, bilden heute Qualitätsstandards eine wesentliche Säule der sog. Service Level Agreements zwischen Auftraggeber und Dienstleister.

Transition- und Betriebs-Phase (g) – Reverse Transition
Sofern der AMS-Vertrag am Ende der Laufzeit nicht verlängert werden soll, wenn also entweder der Auftraggeber die Systeme wieder in die eigene Verantwortung überführen will („In-Sourcing“) oder aber den Vertrag einem anderen Dienstleister übertragen will, bilden die Qualitätsmessungen aus Basis des letzten erstellten Releases den Nachweis für die nicht-funktionale Qualität der betreuten Anwendungssysteme.

3 Aktives Qualitätsmanagement im Rahmen von AMS

In den meisten AMS-Engagements spielt die CTB-Organisation eine wesentliche Rolle und kann den Aufwand für den reinen Betrieb, also die RTB-Seite, deutlich übersteigen. Die CTB-Organisation kümmert sich um die Wartung und Weiterentwicklung der Anwendung. Sie ist damit auch Hauptnutzer von Qualitätssicherungs-maßnahmen im Rahmen von AMS. Die CTB-Prozesse für die fachliche Weiterentwicklung gliedern sich normalerweise in die übliche Fertigungskette. Nach der Fertigstellung von Softwarekomponenten durchlaufen diesedie Qualitätsprüfungen, um die fachliche und grundlegende technische Eignung für den Produktionsbetrieb nachzuweisen. Neben den aus der Software selbst ermittelten Qualitätsbewertungen spielen als weitere wesentliche Dimension die betrieblichen Kennzahlen eine Rolle für den Nachweis einer leistungsfähigen AMS-Organisation.

Neben Software-Metriken spielt auch die Messung der Produktivität eine wesentliche Rolle. Daher sind im Rahmen von AMS neben den reinen Qualitätsmaßen auch die fachliche Größe der Anwendung und Aufwandszahlen zu erfassen, um die Produktivität der Wartungs- und Entwicklungs-prozesse bewerten und steuern zu können. Zum Schätzen des Entwicklungsaufwands wurde er von Albrecht [3] bereits Ende der siebziger Jahre die „Function Point-Methode“ eingeführt. Mit der internationalen Standardisierung durch die Object Management Group (OMG) ist ein einheitliches Regelwerk zur nachträglichen Ermittlung der Function Points und damit die Basis für Produktivitätsmessungen geschaffen worden (vgl. [4]).

4 Fazit

Im Rahmen von Application Management Services sind die Messung der inhärenten Qualität aller Softwarekomponenten und die Korrelation mit betrieblichen Kennzahlen eine wesentliche Aktivität für die Steuerung der Wartungs- und Entwicklungsprozesse im Rahmen des CTB-Teils. Neben der Qualitätsmessung spielt auch die objektive Ermittlung der Leistungsfähigkeit der AMS-Entwicklungsorganisation eine wesentliche Rolle, um neben der Qualität auch die Wirtschaftlichkeit aller Maßnahmen bewerten zu können.

5 Literatur

[1] Jens Borchers: Industrialisierung von Application Management Services auf Basis von Standards wie ISO 20000, 35. WI-MAW-Rundbrief, FB Wirtschaftsinformatik der Gesellschaft für Informatik, Jahrgang 19, Heft 1, April 2013, ISSN 1610-5753

[2] Ernest Wallmüller: Software-Qualitäts-management in der Praxis, Hanser, 2001, ISBN 978-3-446-21367-8

[3] Allan J. Albrecht: Measuring Application Development Productivity, Proc. of IBM Application Development Symposium, October 1979, pp. 83-92.

[4] OMG: Automated Function Points, January 2014, http://www.omg.org/spec/AFP/1.0/, Zugriff am 08.08.2014

[5] http://www.iso.org, Zugriff am 08.08.2014

Profile-based View Composition in Development Dashboards

Martin Brandtner, Philipp Leitner, Harald Gall

University of Zurich, Switzerland
{brandtner, leitner, gall}@ifi.uzh.ch

Abstract

Continuous Integration (CI) environments cope with the repeated integration of source code changes and provide rapid feedback about the status of a software project. However, as the integration cycles become shorter, the amount of data increases, and the effort to find information in statically composed CI dashboards becomes substantial. We want to address the shortcoming of static views with a so-called Software Quality Assessment (SQA) mashup and profiles. *SQA-Mashup* describes an approach to enable the integration and presentation of data generated in CI environments. In addition, *SQA-Profiles* describe sets of rules to enable a dynamic composition of CI dashboards based on stakeholder activities in tools of a CI environment (e.g., version control system).

1 Introduction

A fundamental aspect of Continuous Integration (CI) according to Fowler is visibility: "*[...] you want to ensure that everyone can easily see the state of the system and the changes that have been made to it.*"[4]. The integration of a modern CI environment within the development process of a software project is fully automated, and its execution is triggered after every commit. With each integration run, modern CI tools generate a bulk of data. However, this data is scattered across the entire CI environment and analyzing it, for instance, to monitor quality of a system, is a time consuming task. This can delay the rapid feedback cycles of CI and one of its major benefits is then not realized.

The need for an integration and the tailoring of data generated by the tools used in a CI environment is expressed in studies that address the information needs of software developers. Questions are, for example, *What has changed between two builds [and] who has changed it?* [5].

In this work, we briefly revisit our approaches to support the answering of such questions. We cherry-picked parts of our previous research, which was published at the International Conference on Software Analysis, Evolution, and Reengineering [2] and the Information and Software Technology Journal [1].

2 SQA-Mashup

The goal of the SQA-Mashup approach [1] is to integrate the information scattered across CI tools into a single view. The information is presented according to the information needs of different stakeholders.

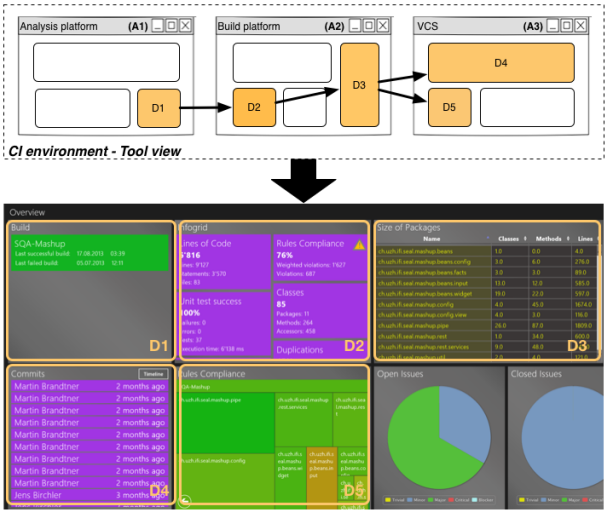


Figure 1: SQA-Mashup: Integrated view

We aim for a fast and easy way to analyze and communicate the actual state, the current health, and the most recent changes of a system. Figure 1 depicts the integration of data from a CI environment to present it to different stakeholders. The *CI environment – Tool view* in Figure 1 represents the information gathering process of, for example, a software tester during a quality measurement review. Starting from a quality analysis platform (*A1*), where the tester detects an increase of convention violations, the developer navigates to the build platform (*A2*) and from there to the associated source code commits (*A3*) to locate the causes of the violations. However, during a development or testing task, such as bug fixing, only some data of each CI tool is relevant to solve the task. The highlighted elements (*D1-D5*) in the single tools indicate the data accessed by the software tester during his review. Our proposed mashup integrates the single pieces of data from different CI tools into a condensed model to dynamically represent the data according to the information needs of a stakeholder.

To evaluate our model-based approach, we de-

signed and conducted a controlled user study with 16 participants who had to solve nine software maintenance tasks. The control group had to use state-of-the-art CI tools and a social coding platform: Jenkins-CI and SonarQube as the CI tools and GitHub as the social coding platform. The experimental group had to use SQA-Mashup, which is a tool that was implemented according to the equally named model.

Overall, we found evidence that the participants of the experimental group solved the tasks of our study faster (57%) and with a higher correctness (21.6%). When analyzing these differences on the level of the individual tasks we found major insights in the benefits of monitoring the CI process and the capabilities of existing CI tools. On the one hand, regarding single aspects of software quality, such project size or code complexity, existing CI-tools provide already good support. On the other hand, we found evidence that monitoring software quality during CI can substantially benefit from an integrated view.

3 SQA-Profiles

The aim of the SQA-Profiles approach [2] is the profiling of stakeholders within a group of software project stakeholders. An example for such a stakeholder group is the project management committee (PMC) in Apache projects.[1] In comparison to other groups, such as committers, the covered spectrum of tasks is broader in a PMC. For example, committers work on issues and contribute source code changes. PMC members actively contribute issues and source code as well, but the PMC is additionally in charge of project and community management. The management of the project incorporates tasks, such as monitoring or gatekeeping. A benefit of using a committee compared to a single manager is the ability to share tasks among the different committee members. For example, some PMC members might focus on source code integration, whereas others taking care of the issue management and gatekeeping. However, the resulting different focus of the PMC members requires a different view on the data presented in dashboards as well [3]. With SQA-Profiles we tried to extract the different focus of PMC members from a collection of Apache projects and to describe them in a project-independent and rule-based manner. We extracted following activity attributes: commits, merges, issue status changes, issue comments, issue assignee changes, and issue priority changes. We found four distinct profiles. One of these profiles is the so-called *Integrator* profile. Any PMC member with this profile has a high merge and commit activity in the according Apache project. The thresholds for a high, medium, and low activity in a certain attribute get computed automatically and individually for each project.

To evaluate our proposed approach, we studied the activity data of PMC members from 20 Apache projects that use Java as main language between September 2013 and September 2014.

Overall, we found evidence that activity data mined from the VCS and the issue tracking platform can reflect the tasks of stakeholders within a certain group. The evaluation results (precision: 0.92, recall: 0.78) showed that the automatic SQA-Profile approach performs almost as good as the semi-automatic baseline, which requires project-dependent threshold parametrization. Additionally, we were able to show that an assigned role in a software project does not necessarily reflect the actual activities of a stakeholder. For example, in some projects normal contributors take care of PMC tasks.

[1] `http://www.apache.org/foundation/how-it-works.html`

4 Bridge the Gap

SQA-Mashup and SQA-Profiles address different aspects to bridge the gap between the information available and the information needed by a stakeholder. The first provides a model to integrate and present CI data and the second enables a categorization of stakeholders based on their activity. To finally bridge the gap and to enable a profile-based view composition, a consolation of research in the field of information needs is required to determine a mapping between the profiles and the available CI data to satisfy the individual information needs.

5 Conclusion

In this work, we revisited two approaches that can enable the integration, tailoring, and presentation of scattered data collected from modern CI environments. We shortly highlighted the potentials of the approaches and described the remaining challenges to bridge the gap between the information available and the information needed by a stakeholder.

References

[1] M. Brandtner, E. Giger, and H. Gall. Sqa-mashup: A mashup framework for continuous integration. *Information and Software Technology*, 2014.

[2] M. Brandtner, S.C. Müller, P. Leitner, and H.C. Gall. Sqa-profiles: Rule-based activity profiles for continuous integration environments. In *Proc. SANER*, pages 301–310, 2015.

[3] R.P.L. Buse and T. Zimmermann. Information needs for software development analytics. In *Proc. ICSE*, pages 987–996, 2012.

[4] M. Fowler. Continuous integration. 2006. `http://www.martinfowler.com/articles/continuousIntegration.html`.

[5] T. Fritz and G.C. Murphy. Using information fragments to answer the questions developers ask. In *Proc. ICSE*, pages 175–184, 2010.

Tool Demos

Nils Göde	Teamscale
Arne Wichmann	Kuestennebel
Dilshodbek Kuryazov	Q-MIG

Architecture

A Co-evolution Approach for Source Code and Component-based Architecture Models*

Michael Langhammer
Karlsruhe Institute of Technology
Karlsruhe, Germany
michael.langhammer@kit.edu

Klaus Krogmann
FZI Research Center for Information Technology
Karlsruhe, Germany
krogmann@fzi.de

Abstract

During the lifecycle of a software system, the software needs to evolve, e.g, through new features or necessary platform adaptions. If architecture and source code are not kept consistent during this software evolution, well-known problems, such as architecture drift and architecture erosion, can occur.

To solve these problems, existing approaches usually focus on the consistency between UML class diagrams and code, or use approaches where the architecture model can completely be generated from the code.

In this paper, we present a fully integrated co-evolution approach for component-based architecture and source code based on Vitruvius. We also present initial, extendable mapping rules from component-based architecture to source code.

1 Introduction

Architecture drift and architecture erosion are two well known problems, which can occur during software evolution [7]. They can occur e.g., if code evolves independently from the system's architecture.

In this paper, we present a co-evolution approach that helps software architects and developers to prevent architecture drift between source code and component-based software architecture. This approach is based on a view-centric engineering approach called Vitruvius[5] [6]. Vitruvius (see Figure 1) can be used to keep heterogeneous models consistent during the development of a system. It is based on the idea of having all information that to a software system stored within a SUM (Single Underlying Model)[2]. The access to this SUM is only possibles via well-defined views. While a SUM eases accessing a single underlying information source, it is hardly applicable in practice since it requires a single world model. To reuse existing meta-models and models within Vitruvius we have introduced the idea of a so called VSUM (Virtual Single Underlying Model)[5], which orchestrates all individual used models without extending or changing models or meta-models.

*Acknowledgment: This work funded by the German Research Foundation in the Priority Programme SPP1593

PCM element	Source code element
Repository	three **packages**: main, contracts, datatypes
Component	Package in main **package** and public component realization **class** within the package
Interface	**Interface** in contracts package
Signature&parameters	**Operation**&**parameters**
Datatype	**Class** with getter and setter for inner types
Required role	**Member** typed with required interface and **constructor parameter** for member
Provided role	Main class of providing component **implements** the provided interface

Table 1: Initial mapping between architectural model elements (PCM) and source code elements (Java)

Within Vitruvius, mapping rules describe the overlap between heterogeneous models of the VSUM. This can be done either by using the MIR (Mapping Invariant Response) language, which we are currently developing, or using Xtend[1]. The MIR or Xtend rules transform changes among models. Therefore, Vitruvius monitors all views respectively editors acquire atomic changes. These atomic changes are used as input for the consistency preserving transformations, which translate changes element by element.

2 Co-evolution approach

The current focus of our work is the application of Vitruvius to component-based software architecture and code (see Figure 1). Vitruvius solely operates on models. As a component-based architecture

[1] http://eclipse.org/xtend/

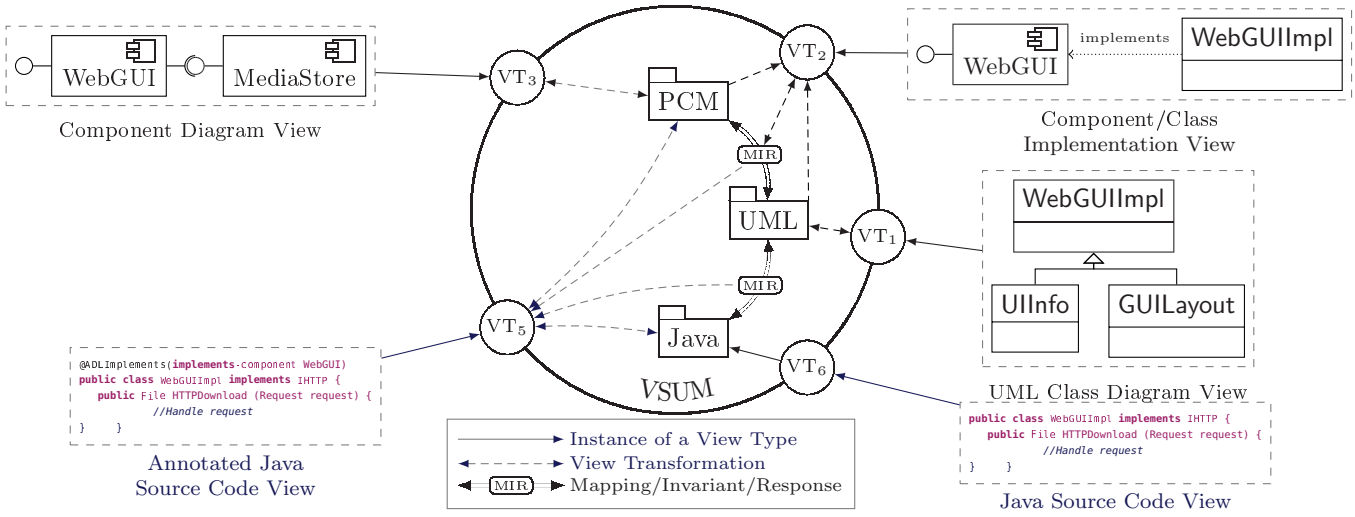


Figure 1: Application of VITRUVIUS to component-based software engineering with multiple views. For our first prototype we only use PCM and JaMoPP as meta-models within the VSUM (Virtual Single Underlying Model) and the standard component views for PCM and the source code view for JaMoPP.

model, we use the the PCM (Palladio Component Model)[3]. The PCM offers users the creation of a component-based architecture in terms of components and interfaces.To get a model representation of the source code, we use JaMoPP (Java Model Parser and Printer)[4], which extracts an EMF-based (Eclipse Modelling Framework) representation of Java code.

To use VITRUVIUS we have implemented a monitor for the Eclipse Java code editor and for EMF-based models [6]. We also have defined initial mapping rules (see Table 1) from component-based architecture models to Java source code. The current prototype is able to round-trip architecture and code for the above-mentioned mapping rules. Further mapping rules to Eclipse Plugins and a dependency injection framework will be realised in our ongoing work.

3 Related work

Existing approaches, e.g., IBM Rational Rhapsody[2], support round-trip engineering but rely on source code as the single information source and generate other representations, such as class diagrams from it. Other approaches, e.g., UMLLab[3], ensure consistency between UML class diagrams and source code, but not between component diagrams and source code. ArchJava[1] includes architectural constructs (e.g., ports) into the source code itself while our approach is not invasive.

4 Conclusion

In this paper, we presented our co-evolution approach for component-based software architecture and source code, which is based on the VITRUVIUS approach. Our co-evolution approach helps software developers and architects evolving their software system by keeping the architecture and the source code consistent during the evolution of a software system.

[2]http://www-03.ibm.com/software/products/de/ratirhapfami
[3]http://www.uml-lab.com/

References

[1] J. Aldrich, C. Chambers, and D. Notkin. “ArchJava: connecting software architecture to implementation.” In: *Proceedings ICSE 2002*. IEEE. 2002, pp. 187–197.

[2] C. Atkinson, D. Stoll, and P. Bostan. “Orthographic Software Modeling: A Practical Approach to View-Based Development”. In: *Evaluation of Novel Approaches to Software Engineering*. Springer, 2010.

[3] S. Becker, H. Koziolek, and R. Reussner. “The Palladio component model for model-driven performance prediction”. In: *Journal of Systems and Software* 82.1 (2009), pp. 3–22.

[4] F. Heidenreich et al. “Closing the gap between modelling and java”. In: *Software Language Engineering*. Springer, 2010, pp. 374–383.

[5] M. E. Kramer, E. Burger, and M. Langhammer. “View-centric engineering with synchronized heterogeneous models.” In: *Proceedings of the 1st Workshop on VAO*. ACM. 2013.

[6] M. E. Kramer et al. “Change-Driven Consistency for Component Code, Architectural Models, and Contracts.” In: *Proceedings of the 18th International ACM Sigsoft Symposium on CBSE*. accepted, to appear. ACM, 2015.

[7] D. Perry and A. Wolf. “Foundations for the study of software architecture”. In: *ACM SIGSOFT Software Engineering Notes* (1992).

Architecture-based Analysis of Changes in Information System Evolution*

Robert Heinrich[1], Kiana Rostami[1], Johannes Stammel[2], Thomas Knapp[1], Ralf Reussner[1]
[1]Karlsruhe Institute of Technology (KIT), {heinrich,rostami,reussner}@kit.edu
[2]andrena objects ag, johannes.stammel@andrena.de

Abstract

Software is subject to continuous change. Software quality is determined by large extent through architecture which reflects important decisions, e.g. on structure and technology. For sound decision making during evolution change impacts on various system artifacts must be understood. In this paper, we introduce a new evolution scenario (replacing the database) to an established demonstrator for information system evolution. We demonstrate the application of an architecture-based approach for change impact analysis to identify artifacts affected by the scenario.

1 Introduction

Software-intensive systems, such as information systems, are frequently operated over decades. In industrial practice these systems face diverse changes, e.g. due to emerging requirements, bug fixes, or adaptations in their environment, such as legal constraint or technology stack updates [6]. As a result, the systems change continually which is understood as software evolution [5]. The software architecture is one of the central artifacts of software-intensive systems and is crucial in evolution. Software development and operation involve a variety of organizational and technical roles covering different responsibilities and knowledge. Thus, coordinating and implementing changes is difficult. Although these roles are very heterogeneous, they all use artifacts which are tightly related to software architecture. Reflecting changes in software architecture models helps to identify maintenance tasks for associated artifacts like source code or test cases.

In this paper, we introduce a new evolution scenario "replacing the database" to the *Common Component Modeling Example* (CoCoME) [7] which serves as a common case study on information system evolution. An overview of CoCoME is given in Sec. 2. We use the tool-supported approach *Karlsruhe Architectural Maintainability Prediction* (KAMP) [3] for change impact analysis based on change requests in an architecture model. Sec. 3 introduces KAMP. In Sec. 4, we demonstrate how to apply KAMP to the evolution scenario for identifying artifacts affected by the change request. The paper concludes in Sec. 5.

*This work was partially supported by the DFG (German Research Foundation) under the Priority Programme SPP1593: Design For Future – Managed Software Evolution.

2 CoCoME – A Case Study on Information System Evolution

CoCoME represents a trading system as it can be observed in a supermarket chain handling sales. This includes processing sales at a single store as well as enterprise-wide administrative tasks like ordering products or inventory management. The CoCoME system is organized as a three-layer software architecture which allows for distributing the system on server nodes and for remote communication. Detailed description is given in [7]. CoCoME has been set up as a common demonstrator in a Dagstuhl research seminar. Since CoCoME has been applied and evolved successfully in various DFG and EU research projects, several variants of CoCoME exist, spanning different platforms and technologies, such as plain Java code, service-oriented or hybrid Cloud-based architectures. Various artifacts from development and operation are available, such as requirements specification, design decisions, source code, architecture models, or monitoring and simulation data, that evolved over time.

The new evolution scenario "replacing the database" refers to the plain Java variant of CoCoME. In the scenario, CoCoME faces performance issues. In order to avoid them the company which operates CoCoME decides to replace the existing database. They shift away from a relational database (e.g., MySQL) to a non-relational database (e.g., CouchDB).

3 Architecture-based Change Impact Analysis

KAMP [3] explicitly includes formal architecture descriptions by means of meta-models for identifying change impacts. The approach relies on the following assumptions: (a) All artifacts of system development and operation must be considered. Focusing only on code is not sufficient. (b) Changes are initialized through evolution scenarios, resulting into predictable requirements on changes. (c) It is easier to identify the effort of fine-grained maintenance tasks, e.g. adding, deleting, or modifying architecture elements, than of coarse-grained maintenance tasks.

KAMP consists of two phases – *preparation phase* and *analysis phase* – and is followed by an *interpretation phase*. In the preparation phase, an architecture model for each design alternative to be compared is created. For this, meta-modeled architecture description languages are applied. Starting with a given evolution scenario, e.g. replacing a middleware technology or replacing the database, the considered change request(s) are identified by a human software architect. A change request among other things includes initially affected architecture elements, such as a particular software component or an interface, that are already known by the architect. In the analysis phase, artifacts affected by initially changed architecture elements are identified first. Then, lists of maintenance tasks (i.e. work plans) specific to the affected artifacts are created for each architecture alternative. This is performed automatically by the KAMP tooling for each architecture alternative and change request. In the interpretation phase, change efforts are estimated and architecture alternatives are compared by the architect based on the lists of maintenance tasks identified by the KAMP tooling. KAMP basically comprises three contributions [3]: (i) meta-models to describe system parts and their dependencies, (ii) a procedure to automatically identify system parts to be changed for a given change request defined manually as well as (iii) a procedure to automatically derive required maintenance tasks from a given change request to simplify the identification of a change effort and, by that, the maintainability estimation. Furthermore, KAMP is proposed to be applied for automated software project planning [2] and for deriving work plans to solve performance and scalability issues [1].

4 Applying KAMP to a CoCoME Evolution Scenario

Next, we describe how to apply KAMP to the CoCoME evolution scenario "replacing the database". Replacing a relational database by a non-relational database raises certain consequences. For example, because JDBC has just been developed to provide a connection to relational databases, the interface has to be replaced, too. KAMP is applied to identify such consequences of changing the database and to find out all affected components. While in the following only an overview is given, further details on the application of KAMP to the scenario and the affected architecture elements is available in [4].

In the preparation phase, the architect creates the architecture model of CoCoME and annotates it with additional information regarding building, deployment, and testing.

In the analysis phase, a copy of the current architecture model is created first. Second, the architect executes the structural changes in the architecture model. S/he deletes the old Database component in the architecture and adds another one. Moreover, s/he knows that the interface will not be usable any more and removes it and adds a new one. After the architect has marked all changes in the model copy, s/he triggers the calculation of the differences between the original and the copied model. The KAMP tooling recognizes that the Database component and its interface have been removed and another component and interface have been added. KAMP maps this information to maintenance tasks and builds up a first draft of a work plan which contains all tasks to realize the changes.

Next, possible side effects of changing single components are analyzed by investigating connections to other components. In the given scenario, the Data component (cf. [4]) is affected by changing the Database component. KAMP recognizes that Data is a composite component consisting of several sub-components which are included in the analysis and affected subcomponents are added to the work plan.

After all affected architecture elements have been mapped to tasks additional tasks for building, deployment, and testing are considered. For example, given that the architecture model has been enriched with test case information, for every test connected to the Data and Database component a *modify* and *run* task is suggested. This procedure results in comprehensive work plans suitable to implement and estimate changes.

5 Conclusion

We described the application of KAMP for change impact analysis in the new CoCoME evolution scenario "replacing the database". In the future, CoCoME will be further modified to create new and evolve existing artifacts by new evolution scenarios such as the introduction of mobile clients.

References

[1] C. Heger and R. Heinrich. Deriving work plans for solving performance and scalability problems. In *EPEW*, pages 104–118. Springer, 2014.

[2] O. Hummel and R. Heinrich. Towards automated software project planning - extending Palladio for the simulation of software processes. In *KPDAYS*, pages 20–29. CEUR Vol-1083, 2013.

[3] K. Rostami et al. Architecture-based assessment and planning of change requests. In *11th Intl. Conference on the Quality of Software Architectures*. ACM, 2015 (accepted to appear).

[4] T. Knapp. KAMP analysis applied to CoCoME. In *Seminar thesis, SDQ Chair, KIT*, 2012.

[5] M. M. Lehman and L. A. Belady, editors. *Program Evolution: Processes of Software Change*. Academic Press Professional, Inc., 1985.

[6] R. Heinrich et al. Integrating run-time observations and design component models for cloud system analysis. In *MRT*, pages 41–46. CEUR Vol-1270, 2014.

[7] S. Herold et al. CoCoME – the common component modeling example. In *The Common Component Modeling Example*, pages 16–53. Springer, 2008.

Modularity – Often Desired, Too Often Failed

Jens Knodel, Matthias Naab, Balthasar Weitzel
Fraunhofer IESE
Fraunhofer-Platz 1, 67663 Kaiserslautern, Germany
{jens.knodel, matthias.naab, balthasar.weitzel}@iese.fraunhofer.de

***Abstract*—"Everything should be modular" is an exalted goal stated by almost every architect – but is it really possible to achieve this goal? In this experience paper, we share our lessons learned across a number of restructuring projects that went modular. We discuss typical business motivations, restructuring efforts starting with good intentions, and reconstruction reality striking back. In retrospective, we analyze typical pitfalls to be circumvented. Examples illustrate our findings and support a truism too often ignored by architects: everything has its price, and more often than not, the price for modularity is a lot higher than initially estimated.**

Keywords—architecture, modularity, reconstruction, reverse engineering, experience report

I. INTRODUCTION

Modularity is a design goal that is often desired by various stakeholders in development organization, especially if systems have historically grown and problems in maintaining them emerge. Modularity is considered to be the silver bullet in such cases: restructuring the system towards building blocks of manageable size with defined functionality and adequate quality. To achieve modularity sounds straightforward, but in practice it is not. We have anecdotal evidence from basically any of our industrial partners starting modularization initiatives ending in vain. The projects start with a collection of high-level modularization goals, come up with an idealistic architecture of the system and fail when reality strikes back – restructuring the system reveals a lot of technical constraints not being thought of in advance. However, there are as well success stories: the risk of failure can be significantly reduced by avoiding common pitfalls and applying best practices.

In this paper we present consolidated experiences from a number of past modularization projects. We introduce an overview of typical stakeholder goals to be achieved by modularity. We then present a way for redesigning the system in order to achieve these goals, followed by the actual reconstruction. We discuss typical findings and complement them with best practices.

II. WHY MODULARITY

The requirements of the different product stakeholders collected at the beginning of a modularization project can be divided into modularity goals and product requirements that should be supported by the modularized product. In terms of product requirements our main experience was that the system basically should support the same processes with slight changes. Modularization projects were often also used to get rid of unused features or realize adaptations that had lower priority for a longer time. Since the details vary a lot in the different projects, we want to focus on the actual modularity or decoupling requirements of the different stakeholders:

Product manager
- React on new requirements faster
- Shorter release cycles, also for parts of the product
- Keep investments of the past
- Avoid changes in the UI to not confuse customers

Development manager
- Increase internal productivity by parallelization
- Use external teams
- Use new developers with different or less skills

Sales representative
- Sell parts of existing product separately
- More combination options, also external products
- Additional value as a compensation if changes to current installations at customer's site are required

Architect
- React on new requirements easier
- Local changes in manageable parts
- Clear assignment of responsibilities to modules
- Flexible usage of new technologies

Developers and tester
- Less coordination effort by working more local
- More guidance from the systems structure
- Keep proven technologies and complex algorithms

Customizer
- Separate customizations clearly from product code
- Reuse parts of the system
- Keep existing customizations

Support staff
- Reconstruct problem situations faster
- Locate problems easier

III. APPROACHING MODULARITY

In our past projects a combination of two approaches for identifying potential components was applied [1]. A top-down strategy identifies ideal modules and relationships based on communication needs of the supported business processes, considering both product and modularization requirements. In a second step the existing implementation is taken as a starting point, trying to identify current modules, their boundaries and relationships. Both approaches are iteratively combined in order to come up with a potential to-be architecture that can be realized within the project constraints and is still able to fulfill the decoupling and product requirements. It might be necessary to negotiate some of the product requirements

This paper has been partially sponsored by the German Ministry of Education and Research, grant number 01IS12053

during redesign if it turns out that the cost-benefit is not appropriate. In parallel a migration plan is created, capable of moving the system from the current state towards the intended to-be architecture. In the final reconstruction step this migration plan is conducted and the system transformed into a more modular structure.

IV. Discussion

Based on our experience with our industry partners we collected several pitfalls often found in modularization projects.

Goal overloading: Modularity gets considered as a silver bullet for a number of problems that have been observed around the system. Even if the modularization project is a success, not all of the issues are likely to be solved, so disappointment is inevitable. Distinguishing between product and decoupling requirements is a step in the right direction to reduce this risk.

Everything should be modular: Achieving full modularity is impossible, so a general modularity goal for the overall system is not sufficient [2]. Some modules have to be connected to others, if there are dependencies in the supported business processes. Thus it is the main challenge to anticipate areas of potentially changing requirements and explicitly make those parts of the system modular that are impacted by these changes. It requires finding the right level of modularity for every part of the system, keeping the balance between cost for creation and maintenance of the modularity and the savings gained by that modularity.

Modularity as an end in itself: Modularity itself does not automatically guarantee the fulfillment of the implicitly desired flexibility goals. These goals need to be elicited thoroughly and the adequacy of the to-be architecture has to be checked with respect to these change scenarios.

Simplification of functional dependencies: If business processes supported by the system are not analyzed enough or over-simplified dependencies that are inevitable tend to get forgotten. This results in inadequate module definition and compromising of the to-be architecture during reconstruction.

Disregarding as-is architecture: Another common pitfall is to base decisions about the to-be architecture only on the ideal architecture without analyzing the current as-is architecture in detail. This ultimately leads to unexpected issues during reconstruction when reality strikes back and the intended module interfaces are not sufficient.

Getting lost in details: Trying to recover the complete as-is architecture to a high level of detail consumes significant effort and does not reduce the complexity. A more efficient approach is to let experienced developers do a tool based request driven analysis where identified dependencies are rated and only important ones are refined.

Not using existing knowledge: Not involving developers that created the system makes it hard to get the rationales of past design decisions, which is especially valuable if these decisions should be revised.

Perception over analysis: There are often assumptions about the system that have been said so often so that they are considered as reality. More often than not it turns out that they are not exactly true. Basing decisions on these "ensured assumptions" is critical, checking them by analyzing the code, for example, reduces risk significantly.

Neglecting iterative nature of redesigning: Another pitfall is not being prepared for performing several iterations in the redesign process, thus wasting effort by creating detailed documentations of intermediate to-be architectures.

New technology considered as savior: New technology tends to look "shiny", having many advantages over the current one. Disadvantages of the current technology can be easily observed, its advantages compared to the new one seem to be small. The source for this conception is the missing knowledge of the actual behavior of the new technology in the actual environment. A comparison based on sound prototypes often reveals that the differences are not as evident as expected. A cost-benefit comparison that includes the replacement effort is often debunking the new technology.

Changing fundamental architectural decisions: Trying to change fundamental architectural decisions that have been identified as being suboptimal during modularization is tempting. In such cases the actual impact of such a change is often underestimated. Starting with a more reachable intermediate goal is a much more risk-aware approach.

Unmanaged reconstruction: The to-be architecture is only valuable if it is also realized as it was intended. To achieve that, a continuous monitoring of the reconstruction work in terms of a fact-based tracking is required. Regular architecture compliance assessments are an efficient way of achieving that.

Wrong expectation management: Admitting that some expectations of the project are not realistic is hard. Procrastinating to tell the truth results in disappointment and significant loss of credibility.

Modularization as a disguise: In most systems there are some technical modernization tasks that are considered as important by architects or developers, but not by those who decide about budget allocation. Hiding such tasks in a modularization project is a risky approach since it will result in a significant loss of credibility if this disguise gets public.

No clear project goal: As in any project it is necessary to clearly define its goals upfront, so that an evaluation of its success is possible. In case of modularization this best practice gets often ignored, especially if no clear requirements have been elicited at the beginning of the project.

Gold plating: A similar pitfall like on project level can also happen on task level. Especially clean-up tasks that are common in such reconstructing projects need a measurable stop criteria. Otherwise developers tend to make good things even better and not seeing other, more pressing tasks.

V. Conclusion

We gave an overview of typical modularization goals and common pitfalls when aiming at them. A positive conclusion is that for every one of them strategies are available to avoid them, in most cases just by making them explicit.

References

[1] Naab, M.; Weitzel B.; et. al: *Isolation modularer Technologie-komponenten aus smart FIX*; IESE-Report; Kaiserslautern; 2013.

[2] Naab, M.: *Enhancing architecture design methods for improved flexibility in long-living information systems*; PhD Theses; Fraunhofer-Verlag; Kaiserslautern; 2011.

Project Support

Jan Jelschen, Johannes Meier and Andreas Winter – SENSEI Applied: An Auto-Generated Toolchain for Q-MIG 23

Marvin Grieger and Masud Fazal-Baqaie – Towards a Framework for the Modular Construction of Situation-Specific Software Transformation Methods

Hakan Aksu and Ralf Lämmel – API-related Developer Profiling

SENSEI Applied: An Auto-Generated Toolchain for Q-MIG

Jan Jelschen, Johannes Meier, Andreas Winter
Carl von Ossietzky Universität, Oldenburg, Germany
{jelschen,meier,winter}@se.uni-oldenburg.de

1 Introduction

Large software evolution, migration, or reengineering endeavors require integrated tooling to support their specific goals [1]. While some functionality is project-specific, for many standard software evolution tasks, tools are readily available. Those tools usually provide little means for interoperability, making integration a tedious and error-prone struggle. Furthermore, software evolution projects must usually follow iterative processes – even with fully elicited requirements, subjected legacy systems, being large, complex, and undocumented, obscure the view to a clear path through a project. Rigid, ad-hoc tool integration impedes experimentation, encumbers adaption and extension, and overall slows down the project.

SENSEI (*Software EvolutioN SErvices Integration* [2]) is a conceptual framework developed to ease the toolchain-building process, by combining service-oriented, component-based, and model-driven techniques. SENSEI provides the means and structures to describe required functionality and their interplay as *services* and *orchestrations*, respectively, and enables automatic mapping to appropriate implementing *components* and auto-generation of integration code.

The utility of SENSEI has been put to the test by using it to (re-)build the toolchain for the Q-MIG [3] research project. This paper aims to demonstrate SENSEI's advantages by explaining its application, and comparing it to "manual" toolchain-building. To this end, Section 2 gives a brief overview over the goals of Q-MIG. Section 3 outlines the key principles of SENSEI, and how it is practically applied, using Q-MIG as example. Section 4 exemplifies the utility of SENSEI regarding *flexibility*, *reusability*, and *productivity* using tooling-related issues that arose during the project. The paper concludes with a summary in Section 5.

2 The Q-MIG Project

Q-MIG investigated quality dynamics of software under language migrations from COBOL to Java. Its objectives were to *measure*, *compare and visualize*, as well as *predict* software migration quality. Therefore, a *Quality Control Center* has been developed: a toolchain to complement an existing migration toolchain in support of *a*) researchers studying migration quality, *b*) experts rating the inner quality of systems, and *c*) software migration consultants projecting post-migration quality to improve migration tools, provide insights to clients, and choose migration strategies and tooling according to quality goals. The required tools and toolchains had been developed conventionally, first, as the SENSEI tooling had not been ready when the project commenced.

3 The SENSEI Approach Applied

The key aspects of SENSEI can be summarized along the utilized principles of *service-oriented*, *component-based*, and *model-driven* paradigms, consolidated by *capabilities* in an integrated meta-model. More detail is given in the following, explaining the steps of applying SENSEI (*defining services*, *designing orchestrations*, *adapting and registering components*, and *generating toolchains*), using Q-MIG as example.

Defining Services. First, the required functionality has to be identified and described as services. In SENSEI, a service consists of a name and description of its intended function, consumed inputs and produced outputs with associated (abstract) data structures, and *capability classes* (explained in the following).

Services can either be defined *top-down* or *bottom-up*. The former approach identifies them from relevant publications and diverse software evolution projects [5], to create a catalog of generic, standardized services. If available, services can be picked from the catalog instead of being created for the project. Otherwise, services can be created *bottom-up*, only for a project's required functionalities, giving full control over service design, but potentially leading to project-specific services with lower reuse value. This can be used, though, to fill a catalog incrementally, and refine and generalize its services in the process.

Lacking a comprehensive catalog, the bottom-up approach was chosen for Q-MIG. Services were identified for *parsing*, *calculating metrics*, *visualizing*, *learning and predicting*, as well as *extracting* and *consolidating* data. All services were fitted with input and output parameters, e.g. the parsing service's input is *source code*, and its output is a corresponding *abstract syntax tree*. To be able to refine what kind of source code can be parsed, the service also got a capability class named *programming language*, with COBOL and Java among the possible values. The parameter's types can be restricted according to a particular capability, e.g. Java requires Java source code as input.

Designing Orchestrations. Once all service descriptions are ready, they can be instantiated in an orchestration. A graphical editor [4] is available to support this task, so that service instances can be wired up to define control and data flows. Service instances can be nested in control structures to model concurrent execution, or loops, for example.

The orchestration for Q-MIG's quality measurement starts with parsing. Next, a service to calculate a metric is invoked once for each metric specified in the input, using a loop, while concurrently, a service extracts sub-system nesting information. Finally, the data is consolidated and returned. The orchestration supports COBOL and Java *both*, which is specified declaratively using *required capabilities*; no branching was modeled.

Adapting and Registering Components. To be usable with SENSEI, components have to conform to their services' interfaces. The service-to-component mapping is not one-to-one: for example, the parsing service is implemented by two different components, one for parsing Java, and one for COBOL. Data extraction and consolidation, as well as metric calculation are implemented in a single component.

Generating Toolchains. Lastly, the toolchain generator *SCAffolder* (targeting *SCA*: "Service Component Architecture" as component framework) is fed with the artifacts resulting from the previous steps. It will try to find matches for each service instance in the orchestration. It may select multiple components to realize a single service with different capabilities, and generate appropriate branching logic. SCAffolder leverages service capabilities and associated data type restrictions on them to invoke the right implementation at runtime based on concrete input data.

4 Evaluation

This section compares the conventional toolchain-building approach with SENSEI, again using Q-MIG as example. The comparison is structured according to the criteria *flexibility*, *reusability*, and *productivity*. Three exemplary issues that arose during Q-MIG have been picked to illustrate relevance and particular advantages of SENSEI: (**1**) To integrate a clone detector for the *number of cloned lines* metric within the toolchain, technical interoperability issues became a major selection criterion. A Java tool was selected, because it was the easiest to integrate. (**2**) Some project members had less experience in object-oriented programming, leading to architecture violations. (**3**) Due to legal restrictions, parts of the toolchain had to be run by, and on the premises of, the project's industry partner. The distributed part could not be automated, introducing manual steps, communication overhead between the partners, and a rigorous release process.

Flexibility. SENSEI enables a technology-independent choice of existing tools (1), as it abstracts from interoperability and implementation issues. The target platform SCA offers support for different implementation languages. This helps less experienced developers (2) to create components with familiar techniques, strictly isolated from other parts of the toolchain. The high abstraction level also enables non-programmers (e.g. data scientists, analysts) to partake in designing toolchains. And it abstracts from deployment concerns, easing toolchain distribution (3).

Reusability. Reuse is facilitated through SENSEI by building up a library of components, with interfaces standardized through a service catalog. With SENSEI, adapters will rest with the tools, whereas in Q-MIG (1), it was natural to keep them somewhat "buried" and mixed in within the metric calculation code.

Productivity. SENSEI decreases development effort partly through automation (code generation), and by avoiding redundant developments through added flexibility and easier reuse. E.g., while external tools (1) still have to be adapted to SENSEI's infrastructure, it only has to be done once (possibly even by the tool vendor). The application to Q-MIG has shown that small changes can sometimes be implemented more quickly without SENSEI's imposed structure, but their accumulation may lead to declining evolvability of the toolchain.

The inability to create a gapless, fully integrated toolchain in Q-MIG (3) highlights its importance, as the manual procedures lead to misunderstandings, and markedly slowed down turnarounds. While SENSEI does not currently support distributed toolchains, it can be extended towards it, providing full toolchain control without the need of human intervention. Here though, it remains unclear whether full integration would have been permissible from a legal point of view.

5 Summary

SENSEI structures and partly automates toolchain building to support (not only) software evolution project processes. It facilitates flexibility and reuse, and can thereby help save time and effort. Its application to a concrete project is a first proof of viability.

Achieving the same advantages building toolchains "conventionally", e.g. by adhering to principles like loose coupling and encapsulation, or by "only" using a particular component framework requires more foresight, very disciplined development, and additional implementation effort – something that is hard to keep up under the pressures of a time- or budget-constraint project and evolving conditions and requirements. SENSEI enforces the required structures, and reduces the overall effort through automated integration code generation. The overhead of defining services and adapting component interfaces required at the outset is set off by integration automation. In the long term, it is expected to pay off due to increased reusability.

References

[1] S. E. Sim, "Next generation data interchange: Tool-to-tool application program interfaces," in *WCRE*, 2000, pp. 278–280.

[2] J. Jelschen, "SENSEI: Software Evolution Service Integration," in *Software Evolution Week (CSMR-WCRE)*. Antwerp: IEEE, Feb. 2014, pp. 469—-472.

[3] G. Pandey, J. Jelschen, D. Kuryazov, and A. Winter, "Quality Measurement Scenarios in Software Migration," in *Softwaretechnik Trends*, vol. 34, no. 2. Bonn: GI, 2014, pp. 54–55.

[4] J. Meier, "Editoren für Service-Orchestrierungen," master's thesis, University of Oldenburg, 2014.

[5] J. Jelschen, "Discovery and Description of Software Evolution Services," *Softwaretechnik-Trends*, vol. 33, no. 2, pp. 59–60, May 2013.

Towards a Framework for the Modular Construction of Situation-Specific Software Transformation Methods

Marvin Grieger, Masud Fazal-Baqaie
Universität Paderborn, s-lab – Software Quality Lab
Zukunftsmeile 1, 33102 Paderborn
{mgrieger, mfazal-baqaie}@s-lab.upb.de

Abstract

Software transformation methods are enacted during a migration project to perform the technical transition of a legacy system to a new environment. A critical task of each project is to construct a situation-specific transformation method. In this paper, we categorize current Situational Method Engineering (SME) approaches that support the construction of situation-specific transformation methods according to their degree of controlled flexibility. Based on the findings, we introduce a method engineering framework that enables the modular construction of software transformation methods.

1 Introduction

If an existing software system does not realize all of its requirements but is still valuable to ongoing business, it has become legacy. This might be due to the fact that the underlying technology restricts the fulfillment of new requirements that arose over time. As new development is risky and error-prone, a proven solution is to migrate the existing system into a new environment, which is performed in the context of a migration project. The technical transition of a system is achieved by enacting a *transformation method* which defines activities to perform, artifacts to create, tools to use, roles to involve and techniques to apply. As those methods are used to perform some kind of reengineering [1], they are instances of the well-established horseshoe-model [2].

Constructing a situation-specific transformation method when migrating a software system is critical, as it influences the efficiency and effectiveness of the overall migration project. Efficiency in this context relates to properties of the process to perform the transformation, e.g. the effort required, while effectiveness refers to properties of the migrated system, e.g. its software quality. Besides being a critical task, the construction of a transformation method is also a complex one. Consider for example that source and target environment differ in terms of the programming language. Performing a migration using a horseshoe-based process requires, among other things, to determine the abstraction level to use. A transformation on a syntactical level can be efficient, but ineffective. On the one hand it only requires to develop parsers, code generators and a mapping between the syntactic elements of the languages involved, and enables to transform large parts of the system automatically. On the other hand however, as the amount of information on a syntactical level is limited, it might not be possible to adapt the system to the new environment, e.g. in case of a migration from a monolithic to a layered architecture, making the method ineffective [3]. Using a higher level of abstraction can increase the effectiveness, e.g. by determining for each part of the system to which architectural layer it belongs. This will, however, reduce the efficiency of the method, as sophisticated program analysis techniques are required. Beside the abstraction level to use, a decision needs to be made on whether to automate the transformation at all. If an automatic transformation would be either inefficient or ineffective, a guided manual transformation is a possible alternative. In general, transformation methods are semi-automatic, using multiple abstraction levels jointly. Thus, constructing for a given project an efficient and effective, that is, situation-specific transformation method remains a critical but complex task in every migration project.

2 Situation-Specific Engineering of Transformation Methods

Situational Method Engineering (SME) approaches support the construction of situation-specific methods by providing reusable methods that have been successfully applied in practice, as well as guidance on how to adapt them to a new situation encountered. In [4] a categorization is introduced by which SME approaches are categorized based on the degree of *controlled flexibility* that describes the extent to which a situation-specific adaptation is possible. The adaptation needs to be *controlled* to ensure a high quality of the resulting method. Figure 1 illustrates this categorization applied on SME approaches that support the construction of software transformation methods.

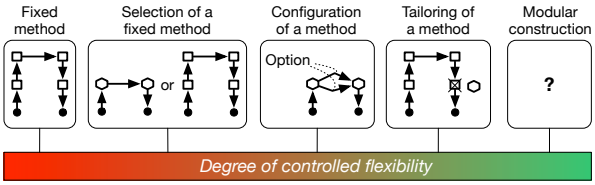


Figure 1: Categorization of method engineering approaches according to their degree of controlled-flexibility, based on [4]

The reuse of *fixed* methods can be seen as an edge case, SME approaches belonging to this category have the lowest degree of flexibility. *Fixed* methods are those who do not foresee any situation-specific adaptation as they describe a static set of activities to perform, tools to use, artifacts to generate, roles to include or techniques to apply in order to transform a legacy system. Such a method can either be described specifically, making it

only applicable to few situations, or generically, requiring situation-specific concretization. In any case the assumed situational context is often only described implicitly. If the situational context for a set of fixed methods is made explicit, it allows a *selection* of the most suitable one, which can be seen as a more flexible SME approach to perform situation-specific adaptation. But, since the resulting method will be fixed, the adaptability of approaches belonging to this category is still limited. In contrast, the definition of a base method that allows *configuration* or *tailoring* provides a higher degree of flexibility. While approaches of the former category aim at configuring foreseen variation points, approaches of the latter category can allow performing arbitrary, but well-defined change operations to the base method. This is achieved by providing a formal description and corresponding tools. However, if many changes to the base method are required, e.g. with novel content, or if no guidance is given on how to assure the quality of the resulting method, constructing a situation-specific method becomes complex and error-prone.

These drawbacks are addressed by SME approaches that enable the *modular construction* of transformation methods by assembling predefined method parts. In addition to the method parts, approaches of this category need to provide assembly guidelines and quality assurance capabilities for constructed methods. Unfortunately, approaches for the modular construction of transformation methods are hardly available. As transformation methods are a specific kind of reengineering methods, one could argue that software reengineering frameworks can be used to construct transformation methods in a modular manner. Although these frameworks are useful to implement tools that are part of transformation methods, they fall short in providing guidance on how to systematically construct the method itself and in assuring its quality.

3 Towards a Framework for the Modular Construction of Transformation Methods

We address this problem by developing a method engineering framework that enables the modular construction of transformation methods and thereby provides a high degree of controlled flexibility. An overview of the intended method engineering process is shown in figure 2.

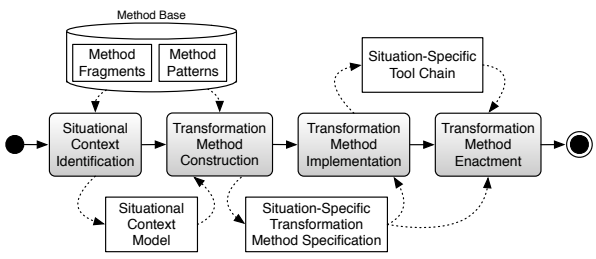


Figure 2: Method engineering process for the modular construction of situation-specific transformation methods

In the beginning, the situational context of the migration project needs to be identified, examples being characteristics of the legacy system, the change of the environment, or goals of stakeholders. The knowledge about the context is an essential prerequisite in order to construct a situation-specific transformation method. Using this knowledge, the transformation method is constructed before related tools are implemented. The construction is supported by a method base, which is a repository that contains predefined transformation method parts and transformation method patterns. While method parts are building blocks of a method of any granularity, method patterns describe methodological or quality aspects that shall be incorporated into the method [5]. For this purpose, a method pattern defines constraints over the method parts, e.g., by defining which ones to use. Adhering to these constraints during the construction allows to ensure properties of the resulting method. In addition, the patterns provide a guideline for the construction of a transformation method, since they are required to be selected, configured and integrated during the construction process. As a last step, the quality of the resulting method is validated. After the transformation method is constructed, related tools are implemented, i.e., the corresponding tool chain is initialized where necessary. As a last step, the transformation method is enacted to perform the transformation.

4 Conclusion and Future Work

In this paper, we characterized *Situational Method Engineering* (SME) approaches that support the construction of situation-specific transformation methods according to their degree of *controlled flexibility*. We concluded that current approaches have some shortcomings which we address by introducing a method engineering framework that enables the modular construction of transformation methods. We aim to develop and apply the framework in the MoSAiC project, which is supported by the Deutsche Forschungsgemeinschaft (DFG) under grants EB 119/11-1 and EN 184/6-1.

5 Literature

[1] E. J. Chikofsky and J. H. I. Cross, “Reverse engineering and design recovery: a taxonomy,” IEEE Software, vol. 7, no. 1, pp. 13–17, 1990

[2] R. Kazman, S. G. Woods, and S. J. Carrière, “Requirements for Integrating Software Architecture and Reengineering Models: CORUM II,” in Proc. of WCRE 1998, pp. 154– 163.

[3] F. Fleurey, E. Breton, B. Baudry, A. Nicolas, and J. Jézéquel, “Model-Driven Engineering for Software Migration in a Large Industrial Context,” in Proc. of MODELS 2007, pp. 482–497.

[4] Frank Harmsen, Sjaak Brinkkemper, and Han Oei. “Situational method engineering for information system project approaches.” In Proc. of CRIS 1994, pp. 169–194.

[5] Masud Fazal-Baqaie, Markus Luckey, and Gregor Engels. “Assembly-Based Method Engineering with Method Patterns.” In Proc. of SE 2013, pp. 435–444.

API-related Developer Profiling
(Extended Abstract)*

Hakan Aksu and Ralf Lämmel
Software Languages Team, University of Koblenz-Landau, Germany

Abstract

We analyze the version history of software projects to determine API-related profiles of software developers. To this end, we identify API references in source-code changes and aggregate such references through suitable metrics that provide different views on the API usage per developer so that certain conclusions regarding developer experience or comparisons between developers become feasible. We apply this approach in a case study for the open-source project JHotDraw.

1 Motivation

Knowledge of developer profiles (or experience or skills) is clearly valuable, e.g., for hiring or program managers. For instance, a hiring manager who is filling a position in a specific project may want to match the developer skills with the skills required for the project. A program manager who is reassigning sparse resources across several projects may want to validate that developers are reassigned in a way that all projects are still sufficiently staffed in terms of required skills.

Developer skills may be determined, in principle, by means of interviews, questionnaires, assignments, or analysis of available social (coding) network information (such as GitHub, topcoder, or StackOverflow). We describe an approach that analyzes API usage in source-code changes over the timeline of a project so that API-related developer profiles can be aggregated on the grounds of suitable metrics. The individual APIs in a project may also be mapped to more abstract domains [4] (such as GUI or XML or database programming), thereby permitting a discussion of developer profiles at a higher level of abstraction.

2 API-usage analysis

Without loss of generality, our approach has been implemented for Java as the source-code language and subversion as the version control system. We use the metamodel of Fig. 1 for data extraction. For what it matters, it is implemented as a relational database (relying on *Java DB*). The metamodel shows how APIs consists of API packages, how these packages declare certain API elements (e.g., classes and methods), how APIs are associated with domains, how repositories consist of files, which files have changed, which specific lines have changed, and how these changes are associated with APIs and elements thereof on the grounds of analyzing the changed lines.

The analysis relies on several extractors. The *SVNRepositoryExtractor* iterates over all commits and extracts *Version* information like developer name, revision number, and commit message, the *ChangedFile* information, and the *ChangedLine* information. As a result, every changed line can be associated with a specific developer. Data cleaning is applied so that, for example, bulk moves are excluded, as such changes would otherwise lead to severely imprecise results.

The *ClassExtractor* extracts API packages and elements from a given *.jar* file for an API. In our implementation, as a concession to scalability, we only maintain information about the latest version of an API, which may affect precision and recall.

The *APIUsageExtractor* extracts API package imports from changed files and 'potentially' referenced API elements from changed lines. A lexical approach is used in that changed lines are tokenized and the extracted names are intersected with API elements from imported API packages, as known due the *ClassExtractor*. In this manner, the underlying software projects do not need to be built, which is often difficult for larger projects and specifically older versions thereof. However, the lexical approach also challenges the precision of the analysis; this is not a severe problem in the case study that we performed.

3 Profiling metrics

For brevity, we only mention a few per-developer metrics, also without motivating them deeply:

- Given an API, the number of distinct API elements that are referenced by changed lines, over all commits by the developer.
- Given an API, the largest number of methods changed in a single commit by the developer and referencing elements of the API.
- The number of APIs of which elements were referenced in changed lines, over all commits by the developer.

*A comprehensive description of this research is being prepared [1, 2] and made available online: `http://softlang.uni-koblenz.de/apidevprof/`

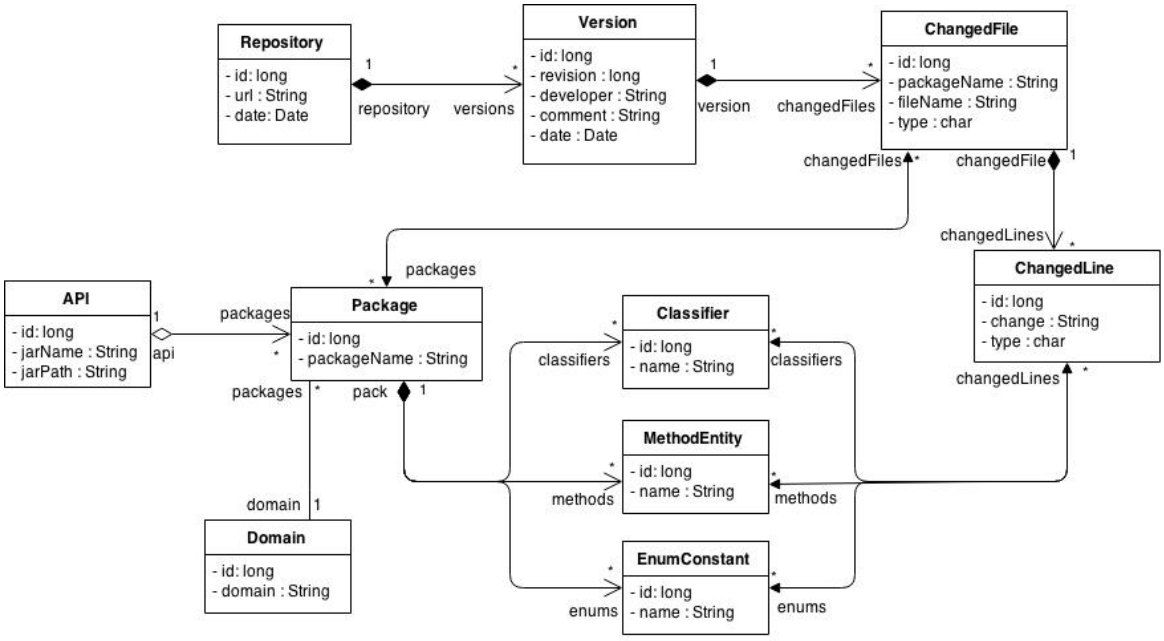


Figure 1: Metamodel for the underlying API-usage analysis.

The idea is that developers are to be compared in terms of these metrics. Clearly, these metrics need to be configured with thresholds and normalized in certain ways to permit useful comparisons.

4 Related Work

Software analysis related to APIs has received much attention by research in the last few years. For example, our team and collaborators have analyzed API usage to understand what API facets are used to what extent in what parts of a project and also the combination of APIs involved [4]. Analyzing API usage may also be useful to complement API documentation [6]. Evolution-aware analyses are also common. For instance, the evolution of an API may be analyzed to guide the implied migration work on projects that use an API [5].

Our current work is particularly concerned with linking API usage to developers and aggregating profiling information regarding experience or skills. Thus, our work is closely related to any effort on mining software repositories that takes properly into account developers. For instance, there is research on analyzing interactions between distributed open-source software developers and leveraging data mining techniques so that developer roles can be derived [7]. Another approach [3] applies statistical topic modeling to source code, thereby providing a basis for determining developer competencies, developer similarity, and others.

5 Case study

We analyzed the subversion repository of *JHotdraw* with its version history of 15 years (2000-2015) and 800 versions. There are more than 17K changed files and more then 650K changed lines. We identified 47 APIs and grouped them in 18 programming domains. We associated changes with 11 developers. We determined the profiling metrics such as those mentioned above. These metrics provide API-related quantitative insight into developer activities, ultimately profiling ('sorting') the developers in terms of their API-related skills; see [1, 2] for details.

References

[1] H. Aksu. Evolution-aware API analysis of developer skills, Mar. 2015. Master's thesis. University of Koblenz-Landau. Computer Science Department.

[2] H. Aksu and R. Lämmel. API-related Developer Profiling. Draft. Unpublished. To be submitted., 2015.

[3] E. Linstead, P. Rigor, S. K. Bajracharya, C. V. Lopes, and P. Baldi. Mining Eclipse Developer Contributions via Author-Topic Models. In *Proc. of MSR 2007*, page 30. IEEE, 2007.

[4] C. D. Roover, R. Lämmel, and E. Pek. Multi-dimensional exploration of API usage. In *Proc. of ICPC 2013*, pages 152–161. IEEE, 2013.

[5] W. Wu, B. Adams, Y. Guéhéneuc, and G. Antoniol. ACUA: API Change and Usage Auditor. In *Proc. of SCAM 2014*, pages 89–94. IEEE, 2014.

[6] T. Xie and J. Pei. MAPO: mining API usages from open source repositories. In *Proc. of MSR 2006*, pages 54–57. IEEE, 2006.

[7] L. Yu and S. Ramaswamy. Mining CVS Repositories to Understand Open-Source Project Developer Roles. In *Proc. of MSR 2007*, page 8. IEEE, 2007.