MMSM 2014
Modellbasierte und modellgetriebene Softwaremodernisierung

Workshop Proceedings

http://akmda.ipd.kit.edu/mmsm

19. März 2014

Universität Wien

im Rahmen der Konferenz Modellierung 2014

www.modellierung2014.org

Organisatoren

Steffen Becker, s-lab, Universität Paderborn
Stefan Sauer, s-lab, Universität Paderborn
Benjamin Klatt, FZI Karlsruhe
Matthias Riebisch, Universität Hamburg
Thomas Ruhroth, TU Dortmund

Programmkomitee

Steffen Becker, s-lab, Universität Paderborn
Wilhelm Hasselbring, Universität Kiel
Elmar Jürgens, CQSE München
Benjamin Klatt, FZI Karlsruhe
Rainer Koschke, Universität Bremen
Claus Lewerentz, Universität Cottbus
Carola Lilienthal, C1 WPS Hamburg
Florian Matthes, TU München
Christof Momm, SAP Karlsruhe
Ralf Reussner, KIT Karlsruhe
Matthias Riebisch, Universität Hamburg
Thomas Ruhroth, TU Dortmund
Stefan Sauer, s-lab, Universität Paderborn
Hannes Schwarz, Universität Koblenz-Landau
Ulrike Steffens, HAW Hamburg
Detlef Streitferdt, TU Ilmenau
Andreas Winter, Universität Oldenburg
Christian Zeidler, ABB Forschungszentrum Ladenburg

Programm

Für die Vorträge sind jeweils 15min + 5min für direkte Fragen eingeplant. Am Ende jeder Session findet ein Session-Panel mit allen Sprechern von 20 Minuten zu dem jeweiligen Themenblock statt. In der letzten Session ist zusätzliche Zeit für eine offene Diskussion über aktuelle und zukünftige Fragestellungen und Forschung in der modellbasierten und modellgetriebenen Softwaremodernisierung vorgesehen.

Zeit	Inhalt
08:30 - 10:00	**Begrüßung** **Keynote: "Semi-automated Abstraction of Architectural Views throughout the Software Lifecycle"** - Univ. Prof. Dr. Uwe Zdun
10:00 - 10:30	Kaffeepause
10:30 - 12:00	Models to Code to Models **"Why Models and Code Should be Treated as Friends"** Mahdi Derakhshanmanesh, Jürgen Ebert and Gregor Engels **"Reverse-Modellierung von Traceability zwischen Code, Test und Anforderungen"** Harry Sneed **"Towards a Reference Architecture for Toolbox-based Software Architecture Reconstruction"** Ana Dragomir, Firdaus Harun and Horst Lichter **Session Panel**
12:00 - 13:30	Mittagspause
13:30 - 15:00	Quality Aspects **"Model-Driven Load and Performance Test Engineering in DynaMod"** Eike Schulz, Wolfgang Goerigk, Wilhelm Hasselbring, André van Hoorn and Holger Knoche **"Towards Quality Models in Software Migration"** Gaurav Pandey, Jan Jelschen and Andreas Winter **"A Tool-Supported Quality Smell Catalogue For Android Developers"** Jan Reimann, Martin Brylski and Uwe Aßmann **Session Panel**
15:00 - 15:30	Kaffeepause
15:30 - 17:00	Evolution and Migration **"Architectural Restructuring by Semi-Automatic Clustering to Facilitate Migration towards a Service-oriented Architecture"** Marvin Grieger, Stefan Sauer and Markus Klenke **"Why Determining the Intent of Code Changes Helps Sustaining Attached Model Information During Code Evolution"** Michael Langhammer and Max E. Kramer **Diskussion und Zusammenfassung**

Why Models and Code Should be Treated as Friends

Mahdi Derakhshanmanesh, Jürgen Ebert
University of Koblenz-Landau,
Institute for Software Technology
{manesh, ebert}@uni-koblenz.de

Gregor Engels
University of Paderborn,
Department of Computer Science
engels@upb.de

Abstract

Various approaches have been proposed to face the difficulties related to constructing and maintaining modern software systems. Often, they incorporate models in some part of the development or evolution process. Even the use of models at runtime seems to receive more and more attention as a way to enable the quick, systematic and automated application of change operations on software as it executes. Assuming that existing systems have been largely developed in code and that novel target architectures depend on – or even embed – models to some extent, the possible roles of models and code as well as their interaction and interchangeability need to be thoroughly examined. In this position paper, we attempt to initiate a discussion on why models and code should become closer friends.

1 Introduction

Recent approaches to the construction and maintenance of modern software systems rely on models to a large extent. The most popular approach is the Model Driven Architecture (MDA) where large parts of the system are expressed in domain-specific representations, i.e., in *models*. These models are *transformed* stepwise towards a technical solution. Furthermore, they can be changed quickly and the complex technology-specific artifacts can be regenerated. Hence, this approach supports the systematic adaptation of development artifacts. Whenever change is required, the transformation chain needs to be retraversed and a new version of the software is generated, compiled and deployed.

In addition, there are other approaches that tackle the need for quicker (and possibly smaller) change operations while the software is operating. In these works, models are not discarded before deployment but are shipped with the rest of the system. In fact, these *runtime models* [2] are integral parts of the system and they may even "drive" its execution. That is, models – such as behavioral models – are interpreted at runtime; no code is generated. The core idea can be referred to as *direct model execution*.

Depending on the followed engineering approach, the portion of used models and code varies. While we believe that the use of models at runtime can support software evolution by enabling adaptivity at different levels of granularity, we are also aware of the fact that code execution is still the established and better-performing method. Therefore, the roles of models and code as well as their dependencies need to be well-understood to utilize the advantages of each side when defining the target architecture in software development as well as modernization.

In Section 2, we give examples of five typical relationships between models and code. Implications for software design and development are covered in Section 3, before we conclude the paper in Section 4.

2 Model and Code Relationships

Reflecting on parts of our previous work, especially on the Graph-based Runtime Adaptation Framework (GRAF) [1], we may state that at least five relationship constellations will occur naturally during software development that is based on mixing models and code. The resulting *spectrum* of different cases is sketched in Figure 1.

This spectrum can be seen as describing common ways of using models and/or code, but it is also a spectrum of non-functional properties of the resulting software systems as it suggests the existence of tension between *efficiency* (performance) and *flexibility*. The underlying assumption is that software represented by compiled code is less expensive than interpreting (behavioral) models w.r.t. memory consumption and execution time [3]. Yet, models can be changed easier at runtime. Hence, they are more flexible. Each constellation in this spectrum is described subsequently.

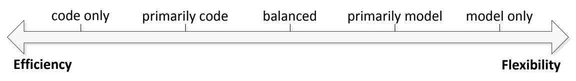


Figure 1: Model and Code Use in the Spectrum between Efficiency and Flexibility (Illustrative Sketch)

Code Only. The software consists of (application) code and there are no models. Obviously, in this case, no mediation between models and code is needed. The system's interfaces are implemented conventionally in code. Furthermore, administration tasks such as inspecting or modifying the software's state are realized in code, too. For instance, they may be implemented with the programming language's reflection capabilities if supported.

Primarily Code. The code is the *primary part* (e.g., w.r.t. the control flow) but it is extended by models in places which need to be specifically flexible. For instance, parts of the behavior that are expected to be in need for (frequent) adaptation are expressed in terms of UML activity diagrams, statecharts, or Petri nets. In this case, the control flow – and potentially any kind of additional data – is redirected from code to models.

Balanced. A combination of the two previously mentioned "pure" cases is applied. Data needs to be synchronized bidirectionally between model and code parts and each side can invoke behavior implemented in the counterpart. In this case, both sides are closely integrated and wired to each other. We can imagine further cases, e.g., where behavioral models implement interfaces defined in code.

Primarily Model. The code is seen as a library of functionality and the main flow is actually dictated by models. Actions implemented in code are orchestrated by the model. Models can be seen as the primary part, whereas code represents the secondary part. For example, a model interpreter may trigger actions that are realized in code related to drivers or existing middleware. The Service Oriented Architecture (SOA) with its business process models is another example.

Model Only. The model part exists and there is no code, at least no application code. In this case, model interpreters are available – and potentially even tightly integrated with the meta-models – making models capable of stand-alone execution. This is the most flexible solution, as any change to the models impacts the software's state and execution behavior immediately. The collection of models and their interpreters *is* the program.

3 Implications

In many traditional software engineering processes, models are seen as the predecessors of code. The final product is – in the end – still made up of code. Hence, existing systems may be developed via model-based or model-driven approaches but the final outcome is usually comparable to "code only" software. To date, models can be involved in all phases of the software life-cycle: while the use of models in the early phases supports documentation and communication (model-based development) and the generation of code enables more systematic and semi-automated construction (model-driven development) the use of models at runtime supports primarily the adaptation of software via *query/transform/interpret* operations [1].

When developing or evolving systems to meet the ever-growing demand for faster turn-around times and adaptation of a "life" system, models offer attractive benefits as initially sketched. As a prerequisite, the target architecture needs to be supported by an *infrastructure* that can manage the relationships between models and code. In contrast to *models@run.time* [2], these models are not necessarily reflective.

A common approach to realizing these capabilities is to encode *model semantics* in the form of a model interpreter that is developed in the same language as the host programming language, e.g., Java. Hence, type-conversion problems do not occur and model and code are finally executed by the same virtual machine. Models are not compiled and so no redundant data needs to be stored for their execution. In the scope of a modernization project, it seems to be a desirable future goal to derive executable models from parts of the code as a basis for further evolution.

4 Concluding Remarks

As *first class entities* of a software system, models can play different roles with different relationships to code. We claim that the relationships between models and code need to be revised in this new context. In order to unleash the full potential of models (e.g., abstraction, flexible adaptation) and code (e.g., stable tools, high performance) for realizing flexible software architectures, we need a *generic infrastructure* that facilitates the integration of models and code as equally valuable constituents of tomorrow's software systems. The recently accepted DFG project *Model-Integrating Self-Adaptive Components* (MoSAiC), conducted together by the authors, will tackle this challenge.

Acknowledgments. The authors thank Thomas Iguchi for fruitful discussions and proofreading.

References

[1] Mehdi Amoui, Mahdi Derakhshanmanesh, Jürgen Ebert, and Ladan Tahvildari. Achieving Dynamic Adaptation via Management and Interpretation of Runtime Models. *Journal of Systems and Software*, 85(12):2720 – 2737, 2012.

[2] Gordon Blair, Nelly Bencomo, and Robert B. France. Models@run.time. *Computer*, 42(10):22–27, 2009.

[3] Edzard Höfig. *Interpretation of Behaviour Models at Runtime - Performance Benchmark and Case Studies.* PhD thesis, Technical University of Berlin, 2011.

Reverse-Modellierung von Traceability zwischen Code, Test und Anforderungen

Harry M. Sneed
ANECON GmbH, Wien

Abstrakt: *In diesem Beitrag zur Modellierung der architektonischen Abhängigkeiten zwischen Code, Test und Anforderungen wird ein Reverse Engineering Prozess zur Wiederherstellung von Traceability zwischen Code-Bausteinen, Testfällen und Anforderungen über die Anwendungsfälle geschildert. Im Mittelpunkt des Prozesses steht ein Software-Repository, das dazu dient die aus dem Code und dem Test gewonnenen Abhängigkeiten zusammenzuführen. Das Ziel ist festzustellen, welche Testfälle welche Anforderungen testen und welche Code-Bausteine welche Anforderungen implementieren.*

1 Einführung

Ein wichtiges Ziel von Software Reengineering ist die Herstellung von Links zwischen dem reengineered Code, den Testfällen und den Anforderungen. Dies ist eine Voraussetzung für die modellgesteuerte Evolution des jeweiligen Systems. Erreicht wird dieses Ziel durch mehrere automatische Schritte von der Analyse des veränderten Codes bis zur Ergänzung der bestehenden Anforderungen. Als Grundlage der Vorgehensweise dient ein Software Repository in dem die Grundelemente der Architektur, die Testfälle und die Anforderungselemente in relationalen Tabellen gespeichert werden. Die Wiederherstellung von Traceability in bestehenden Software-Systemen hat eine lange Vorgeschichte, die bis zu den ersten Studien über Impact-Analyse Anfang der 90'er Jahre zurückreicht. Die hier beschriebene Forschungsarbeit basiert auf den vorausgegangenen Arbeiten von den Universitäten Benevento in Italien [ACDM2002], Bern in der Schweiz [GDS2006], Kentucky in den USA [Hayes2005] und an den Universitäten Ilmenau und Hamburg in Deutschland [Lehn2013].

2 Der Reverse-Engineering Prozess

2.1 Statische Source Code Analyse

Der erste Schritt auf dem Weg vom Code zum Architekturmodell ist die Source-Analyse. Hier werden die Grundelemente der Programmiersprache - die Klassen, Schnittstellen, Methoden, Attribute und Bedingungen - in dem Code erkannt und zusammen mit ihrer Beziehungen in eine relationale Tabelle überführt. Die Einträge in dieser Tabelle sind binäre Beziehungen zwischen einem Basiselement und einem Zielelement. Ein Basiselement kann z.B. ein Zielelement besitzen, benutzen oder aufrufen. Die Elemente und Beziehungsarten sind für jede Zielsprache festgelegt.

Mit dieser Beziehungstabelle wird das Repository gefüttert. Es entstehen mehrere relationale Tabellen, eine für jeden Beziehungstyp. Aus diesen Tabellen wird die statische Codestruktur abgebildet.

2.2 Dynamische Code Analyse

In dem zweiten Schritt wird der Code instrumentiert und dynamisch analysiert. Instrumentiert wird auf der Methoden- bzw. Prozedurebene. In jede Methode wird ein Trace-Punkt gesetzt der eine Monitorklasse aufruft. Die Monitorklasse registriert den Namen der Methode und die exakte Uhrzeit der Methodenausführung. Für jede Zielsprache gibt es eine eigene Monitorklasse, d.h. für COBOL, PL/I. C++, C# und für Java. Daraus entsteht eine Trace-Datei in der alle ausgeführten Methoden samt Ausführungszeit aufgelistet sind.

Zur gleichen Zeit werden vom Testwerkzeug, die Beginn- und Endzeiten von jedem angestoßenen Testfall festgehalten. Daraus geht eine Liste der ausgeführten Testfälle mit ihrer Beginn- und Endzeiten hervor. Diese Liste wird anschließend mit der Trace-Datei der ausgeführten Methoden abgeglichen. Jede Methode, die zwischen dem Startpunkt und dem Endpunkt eines Testfalls ausgeführt wird, liegt auf dem Pfad jenes Testfalls. Der Pfad eines Testfalls ist letztlich eine Kette Methodenausführungen. Jede Methode auf dem Pfad eines Testfalles liegt im Wirkungsbereich – Impact Domain – des Testfalles. Das Ergebnis ist eine binäre Beziehungstabelle in der die Testfälle auf die Methoden verweisen, die sie ausführen. Durch die Invertierung dieser Tabelle kommen die Beziehungen von den Methoden zu den Testfällen zum Vorschein. Jede Methode kann von einem oder mehreren Testfällen betroffen sein. Falls eine Methode von keinem Testfall überquert wird, ist dies ein Indikator, dass die Methode zumindest nicht in dieser Anwendung verwendet wird.

2.3 Verbindung zu den Anforderungen

In den dritten Schritt wird eine Verbindung zu den Anwendungsfällen und damit auch zu den Anforderungen hergestellt. Jeder Testfall bezieht sich auf einen bestimmten Anwendungsfall. In dem Moment, in dem ein Testfall spezifiziert wird, weiß der Tester, welchen Anwendungsfall er damit testet. Dies wird in der Testfallbeschreibung dokumentiert. Deshalb gibt es in jedem Testfall einen Verweis auf den Anwendungsfall, der damit getestet wird.

Andererseits erfüllt jeder Anwendungsfall eine oder mehrere Anforderungen. In der Beschreibung der Anwendungsfälle gibt es Verweise auf die Anforderungen, die sie erfüllen bzw. auf die Geschäftsregel, die sie implementieren. Daraus folgt eine Verbindung zwischen den Testfällen, den Anwendungsfällen und den Anforderungen. Ein Testfall testet einen bestimmten Anwendungsfall und dieser erfüllt wiederum mehrere Anforderungen. Über den Testfall kommt die Verbindung zu den Codeklassen zustande, da die Testfälle auf die Methoden in den Klassen verweisen.

3 Modellbildung

Nachdem die Tabellen der Codebausteine, Testfälle, Anwendungsfälle und Anforderungen vorliegen, wird mit der Modellbildung begonnen. Dies geschieht mit dem Tool *SoftRepo*, einem Tool mit einer Importschnittstelle, über die die Ergebnisse der anderen Werkzeuge importiert werden. Von dem statischen Analysator kommt die Tabelle mit dem Inhaltsverzeichnis der Code-Komponente, von dem dynamischen Analysator die Tabelle der Testfall zu Codebaustein-Beziehungen, von der Testfall-spezifikation die Tabelle der Testfall zu Anwendungsfall-Beziehungen, und von der Spezifikation der Anwendungsfälle die Verweise von dem Anwendungsfall auf die Anforderungen.

SoftRepo braucht diese Tabellen nur zu sortieren und einander zuzuordnen, um ein Beziehungsgeflecht von den Codebausteinen über die Testfälle zurück zu den Anforderungen zu bilden. Dargestellt wird das Beziehungsgeflecht bzw. das Systemmodell, in der Form einer Baumstruktur. Der Baum kann sowohl von unten – bottom-up – oder von oben – top-down – betrachtet werden. Von unten wählt der Betrachter einen bestimmten Codebaustein, z.B. eine Klasse oder eine Methode. Ausgehend von diesem Knoten wird der Baum nach oben aufgefaltet. Es folgen die Testfälle, die Anwendungsfälle und die Anforderungen, die diesen Codebaustein benutzen. Von oben wählt der Betrachter eine bestimmte Anforderung als Ausgangspunkt. danach wächst der Baum nach unten. Es folgen die Anwendungsfälle, die diese Anforderung erfüllen, die Testfälle, die jene Anwendungsfälle testen und die Codebausteine, die von diesen Testfällen ausgeführt werden. Der Kreis schließt sich. Aus dem Beziehungsnetz wird ein Beziehungsbaum.

4 Der Nutzen des Modells

Ein solches Systemmodell hat einen mehrfachen Nutzen, besonderes für die Reengineering der Architektur. Die drei Nutzarten sind:

- Gewinnung von Information
- Veränderung der Architektur
- Überprüfung der Reengineering Maßnahmen.

4.1 Gewinnung von Information

Zum einen kann es rein informativ verwendet werden. Das Wartungspersonal kann schnell erkennen, welche Codebausteine zu welchen Anwendungsfällen gehören. Sie können auch sehen, welche anderen Codebausteine von diesen abhängig sind. Wird z.B. eine Schnittstelle neu gemacht, kann der Entwickler sehen, welche Testfälle er wiederholen muss um die Schnittstelle zu testen.

4.2 Veränderung der Architektur

Das Modell kann auch konstruktiv verwendet werden, z.B. um Codebausteine innerhalb des Systems zu versetzen. Eine Klasse könnte einer anderen Komponente zugeordnet werden. Man hat die Möglichkeit den Baum zu editieren, in dem man Baumknoten nach oben oder nach unten rückt. Dabei ist zu erkennen, wie die Beziehungen von und zu diesem Knoten sich verändern. Der Benutzer kann auch neue Knoten hinzufügen, z.B. neue Codebausteine oder neue Anwendungsfälle. Auf dieser Weise ist es möglich, strukturelle Änderungen in dem Modell zu simulieren ohne den Code selbst anzufassen.

4.3 Überprüfung der Reengineering Maßnahmen

Schließlich wird das Modell verwendet um einzelne Komponente zu sanieren. Die sanierten Komponenten erfüllen die gleiche Funktionalität wie bisher und behalten die gleichen Schnittstellen zu den anderen Komponenten, werden jedoch intern überarbeitet. Methoden werden gespalten und Daten versetzt.

Durch einen automatisierten Vergleich der beiden Modelle lässt sich erkennen, welche Komponente sich wie verändert haben, welche Modellelemente hinzukommen und welche weggefallen sind. Die veränderten Knoten werden farblich von den alten unterschieden. Dieser Abgleich der alten und neuen Systemstrukturen ist der erste Schritt in der Verifikation der durchgeführten Reengineering Maßnahmen. Der nächste Schritt ist der Codeabgleich, bei dem die alten und neuen Komponenten anweisungsweise verglichen werden. Mit dem Strukturabgleich lassen sich die groben Reengineering Fehler erkennen. Der Modellabgleich soll für jedes neue Release wiederholt werden.

5 Schlussfolgerung

Ein dynamisches Modell der Beziehungen zwischen Testfällen, Anforderungen und Codebausteinen stellt Traceability her, dokumentiert die Testüberdeckung und trägt dazu bei, die Konsistenz eines Softwaresystems zu bewahren. Ein relationales Repository wie SoftRepo ist eine wichtige Voraussetzung dafür.

Literaturhinweise

[ACDM2002]Antoniol,G./Canfora,G./DeLucia,A./Merlo,E.: "Recovering traceability Links between Code and Documentation", IEEE Trans. on S.E., Vol. 28, No. 10, Oct. 2002, p. 970

[GDS2006]Greevy,O./Ducasse,S./Girba,T.:Analyzing software evolution through feature views", JSME, Vol. 18, No. 6, Dec. 2006, p. 425

[Hayes2005] Hayes,J./Dekhtyar,A./Sundarian,S.: "Improving after the fact Tracing and Mapping of Requirements", IEEE Software, Dec. 2005, p. 30

[Lehn2013] Lehnert, S./Farooq, Q./Riebisch, M.: Rule-based Impact Analysis for heterogeneous Software Artifacts", IEEE Proc.of CSMR2013, Genova, March 2013, p. 209

An Architecture for Toolbox-based Software Architecture Reconstruction Solutions

Ana Dragomir, M. Firdaus Harun, Horst Lichter
Research Group Software Construction
RWTH Aachen University
Ahornstr. 55
52074 Aachen, Germany
{ana.dragomir, firdaus.harun, horst.lichter@swc.rwth-aachen.de}

ABSTRACT

Although up-to-date architecture views outstandingly aid the understanding, meaningful evolution and evaluation of software systems, software architecture reconstruction tools are still not broadly employed in the industry. While this situation might seem contradictory, we have identified - based on our experience with two industry cooperation partners - some important improvement potentials that could increase the usefulness of the current state of the art reconstruction approaches and thus lead to a better industry acceptance. In this paper we present a general architecture for constructing tool-box-based software architecture reconstruction solutions that enable the flexible integration of various analysis plug-ins on a per need basis while addressing the improvement directions identified earlier. Furthermore, we present an instantiation of this architecture that enables the reconstruction of architecture behavior views on more abstraction levels. We conclude the paper with a tool implementation overview of the latter architecture.

1. INTRODUCTION

Up-to-date software architecture descriptions can be of enormous help, when software architecture migration and modernization activities need to be performed. Understanding the dependencies of the various software systems as well as the internals of each system in insolation is crucial and paves the way to more accurate architecture evaluations and goal-oriented evolution. In our previous work [1], we have described the current state of the practice of two of our industry cooperation partners with respect to the need and use of software architecture reconstruction tools to monitor and evaluate complex industrial software landscapes. In both companies, the architecture descriptions are elaborated manually, resulting into considerable, continuous effort invested into ensuring their consistency with the implemented architectures, on the one side, or the emergence of very implementation-close descriptions that lack an abstract view of the described systems or the existence of only early elaborated high-level views that can be hardly associated with the actual implementation, on the other side. None of the two industry partners employ architecture reconstruction tools, although the current state of the art would seem to improve their current situation. We have identified several reasons why this is the case: (1) while some current solutions do address the reconstruction of heterogeneous systems [3], they are not actively used and their setup can be rather complicated; (2) metrics that guide the architects to understand where to start the architecture improvement from are often missing; (3) the evolution of the reconstructed architectures is not properly supported, for example by enabling the architects to define evolution variants and identify the most convenient ones; (4) reconstruction tools often use their specific architecture description terminology (e.g., Sonargraph-Architect [4] allows only the definition of layers, layer groups, vertical slices, vertical slices groups and subsystems) which is often rejected by the architects, as it does not correspond to their understanding of their systems.

In order to address these issues we propose an **a**rchitecture **m**onitoring and **a**nalysis **i**nfrastructure (ARAMIS) that contains an extendable tool-box to which architecture monitoring and analysis plug-ins can be added on a per need basis.

2. ARAMIS - OVERVIEW AND EXAMPLE

In the following we first present the general architecture of ARAMIS and then we give an instantiation example that focuses on the architecture visualization and the analysis of architecture violations during run-time

2.1 ARAMIS - General Architecture

Using ARAMIS (Figure 1) architects should be able to extract architectural information from various, possibly heterogeneous systems in order to facilitate the recreation of more complex architecture landscapes. To address this, it should allow the easy addition and integration of technology-specific *architecture information collectors*. Consequently, if the extracted information is too heterogeneous (e.g., information extracted from procedural code vs. information extracted from object oriented code) an intermediate processing step that normalizes it to a common intermediate representation might be required. The collected information can then be stored in an *architecture repository* and/or can be used directly for further analyses. Based on the extracted information or on the intermediate representation, the architects should be able to define the systems' higher level abstractions, using exactly the same terminology that they are using in their day-to-day practice. This should be possible, even if the architects work with different abstractions in different projects (e.g., Cobol systems are organized in subsystems, Java systems are organized in components and the components are further organized in layers, etc.). The definition of the various architecture description languages that comprise these abstractions, as well as their relation to each other will thus constitute a so-called *architecture modeling language family*. Once the extracted information is mapped

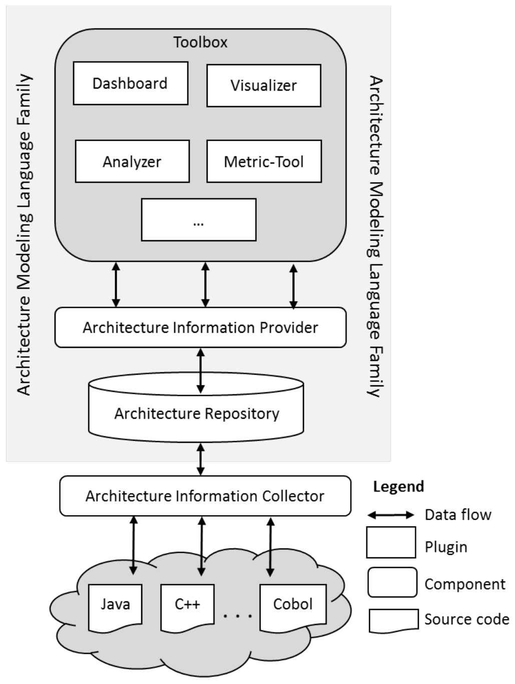

Toolbox
Dashboard
Visualizer
Analyzer
Metric-Tool
...
Architecture Modeling Language Family
Architecture Modeling Language Family
Architecture Information Provider
Architecture Repository
Architecture Information Collector
Java
C++
. . .
Cobol
Legend
Data flow
Plugin
Component
Source code

Figure 1: ARAMIS - General Architecture

on the abstractions chosen by the architects, various plug-ins can be employed to further facilitate the analysis of the results. E.g., based on the reconstructed views, metric plug-ins can be added to compute structural or behavioral metrics and/or to allow the definition and assessment of architecture evolution variants. To gain access to the architecture data necessary for their analyses, the *plug-ins* will register themselves to an *architecture information provider* which will consequently forward them relevant data whenever this is extracted and/or persisted in the architecture repository by any of the employed architecture information collectors.

2.2 ARAMIS - Instantiation Example

We have developed a first instantiation of ARAMIS, that aims to analyze the architecture of software systems during run-time (called ARAMIS-RT). The purpose of ARAMIS-RT is to generate real-time visualizations of data monitored during the run-time. The data should be visualizable on multiple abstraction levels (communication between layers, communication between components, etc.). In order to define these levels of abstraction, we are currently providing the architects with a very simple architecture modeling language that they can use to define their architecture model of interest. As shown in Figure 2, the meta-model simply depicts *architectural units* that can be further contained in other *architectural units*. An architecture unit can have a role (e.g. “layer”, “component”, “filter”, etc.) and can be associated with various software units, as an abstraction thereof. The software units are easily mappable on the data extracted at run-time and represent the entities (in this case - the class instances) between which the intercepted communication occurs. Furthermore, the architect can define architecture communication rules for the defined architecture levels, and then check if these are obeyed or violated during the execution of the system. The data is transmitted

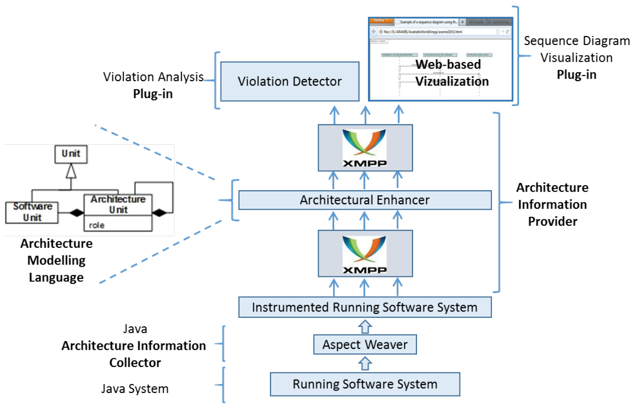

Sequence Diagram Visualization Plug-in
Web-based Vizualization
Violation Analysis Plug-in
Violation Detector
XMPP
Architectural Enhancer
XMPP
Architecture Information Provider
Unit
Software Unit
Architecture Unit
role
Architecture Modelling Language
Instrumented Running Software System
Java Architecture Information Collector
Aspect Weaver
Java System
Running Software System

Figure 2: ARAMIS-RT - An ARAMIS Implementation

in real-time via the XMPP communication protocol. Thus, it is even possible for the architects to trigger the execution using the user interface of the application and then observe almost instantly the way the various architectural elements interacted and if violations occurred. In this case, the XMPP server plays the role of the architecture information provider, as it uses a publish/subscribe mechanism to redirect the data collected during run-time to the two registered plug-ins. Because we only interested in real-time analyses, we did not include an architecture repository to provide data storage for post-mortem investigations. This will be however part of our future work. While ARAMIS-RT is still under development, the results of the first evaluations that we have performed look very promising [2].

3. CONCLUSION

In this paper, we have proposed a general architecture for creating toolbox-based software reconstruction solutions that aim towards improving the current state of the art and achieving a better industry acceptance. We have then presented an instantiation of this architecture that aims at the real-time monitoring of Java systems on more abstraction levels and at the analysis of violations occurring during system execution. In the future, we will further evolve and evaluate ARAMIS-RT presented in this paper and assess its industrial acceptance.

4. REFERENCES

[1] A. Dragomir, M. F. Harun, and H. Lichter. On bridging the gap between practice and vision for software architecture reconstruction and evolution – a tool perspective. SAERoCon Workshop 2014, Sydney, Australia, April 8, 2014.

[2] A. Dragomir and H. Lichter. Run-time monitoring and real-time visualization of software architectures. In *Proceedings of the 20th Asia-Pacific Software Engineering Conference*, December 2013.

[3] J. Ebert, B. Kullbach, V. Riediger, and A. Winter. Gupro: Generic understanding of programs – an overview. In *Electronic Notes in Theoretical Computer Science*, 2002.

[4] Sonargraph-Architect. `https://www.hello2morrow.com/products/sonargraph/architect`, 2013.

Model-Driven Load and Performance Test Engineering in DynaMod*

Eike Schulz[2], Wolfgang Goerigk[1], Wilhelm Hasselbring[2], André van Hoorn[3], Holger Knoche[1]

[1] b+m Informatik AG, D-24109 Melsdorf

[2] Software Engineering Group, Kiel University, D-24098 Kiel

[3] Reliable Software Systems Group, University of Stuttgart, D-70569 Stuttgart

Abstract

Defining representative workloads, involving workload intensity and service calls within user sessions, is a core requirement for meaningful performance testing. This paper presents the approach for obtaining representative workload models from production systems that has been developed in the DynaMod project for model-driven software modernization.

1 Introduction

Workload generation is essential to systematically evaluate performance properties of software systems under controlled conditions. For example, for load, stress, and regression testing or benchmarking it is necessary to expose the system to workload, i.e., to generate requests to provided services. Manual load generation is not practical for various reasons. Hence, automatic generation of synthetic workload is a common practice in performance evaluation [1, 2, 4] and different approaches have been proposed (e.g., [3, 4, 5, 9]). Established tools for generating requests from workload specifications exist, typically based on recorded traces or analysis models. However, one of the biggest challenges is still to obtain *representative* workload specifications similar to production usage.

In the software modernization context, the existing legacy system can be utilized for valid test case generation. An adequate domain (business) model of the legacy system is very useful—if not even necessary—for modernization. Together with usage profiles from production systems, it can be exploited for model-based test development as well.

This paper presents a systematic semi-automatic model-driven approach to obtain and execute representative workload specifications for session-based systems, based on use case specifications and refined by quantitative workload information, such as transition probabilities, think times, behavior mix, etc. The approach was developed as part of our DynaMod project for model-driven software modernization [8], which is depicted in Figure 1. In DynaMod, a combination of static and dynamic reverse engineering

*This work was funded by the German Federal Ministry of Education and Research under grant number 01IS10051.

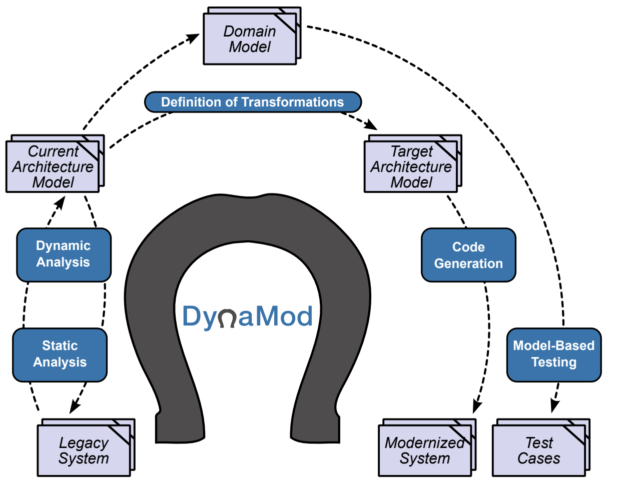


Figure 1: DynaMod software modernization approach

techniques is used to reconstruct architectural models of the legacy system. Employing model-driven techniques [7], these models are transformed into architectural models and executable artifacts of the modernized system. Along with the modernized system artifacts, performance tests are generated.

The latter activity forms the scope of this paper, and is summarized in Section 2. Section 3 summarizes the application of the approach to one of the DynaMod case study systems. Section 4 draws the conclusions and outlines future work. A more detailed description of the approach can be found in [6].

2 Workload Model Construction

The workload modeling formalism of our previous work [9] introduces *workload models* consisting of (i) an *application model* and (ii) a weighted set of *user behavior models*, which provides a probabilistic representation of user sessions. A user session is a sequence of related actions by the same user [5].

The workload model construction comprises four complementary steps, as depicted in Figure 2: (i) manual specification of the use cases and (ii) the application model, (iii) dynamic analysis of the production system (PS), and (iv) automatic extraction of user behavior models from the production data. The

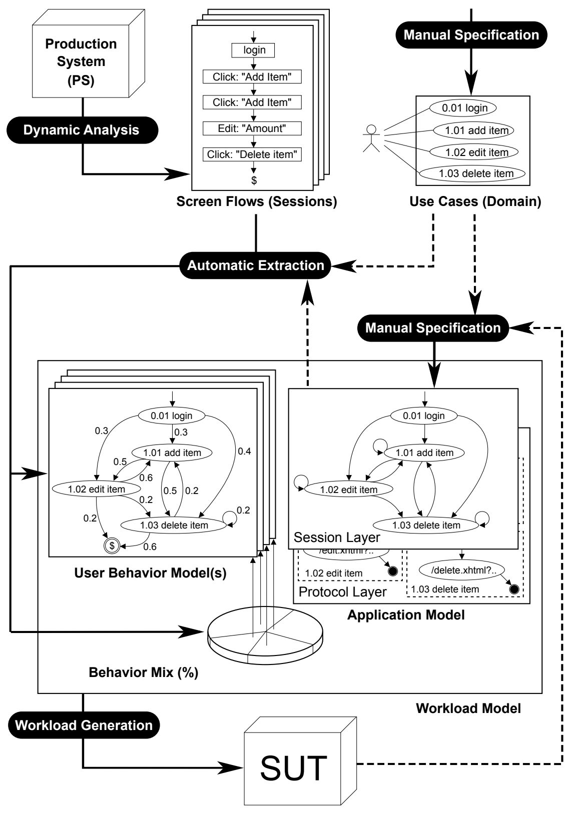


Figure 2: Overview of our approach

behavior and application models are used in a subsequent step to generate test plans to be executed by a suitable load driver.

A basic assumption in our approach is that the production system and the system under test (SUT) share a common domain model (cf. Figure 1), including the set of use cases that are specified manually. The application model is manually created based on the use cases, reflected on the application model's session layer. SUT-specific details are modeled on the protocol layer. Based on a dynamic analysis of the production system, user sessions are reconstructed. The set of user behavior models and its weighting function (behavior mix) are extracted from these sessions. The structure of the behavior models is given by the application model's session layer.

3 Case Study

We applied our approach to one of the DynaMod case study systems [8], namely AIDA-SH, which is an information management and retrieval system for inventory data of historical archives. The system is a client-server application based on the Visual Basic 6 platform and is being used in production by several German archives. As part of the DynaMod research project, a modernized prototype version of the system, called AIDA-Gear, was developed employing model-driven software development techniques [7], which served as the SUT. The set of use cases, from which we created the application model, was provided by domain experts. We obtained runtime traces from the PS instrumented with the Kieker monitoring tool [10]. The extraction of user sessions—in form of executed VB6 UI events—and user behavior models was automated using the tool support for our approach developed in the DynaMod project. The generated workload models were executable on the SUT using JMeter/Markov4JMeter [9].

4 Conclusions and Future Work

We sketched our approach to systematic and semi-automatic creation and execution of representative workloads for load tests of session-based systems based on dynamic analysis results of legacy systems. It is based on previous work on DynaMod [8] and on modeling and executing probabilistic and intensity-varying workloads [9]. Details are also provided in [6]. Future work will be to increase the degree of automation, e.g., by integrating the approach into a model-driven software development platform [7] and to assess the validity of the obtained workload models.

References

[1] Paul Barford and Mark Crovella. Generating representative web workloads for network and server performance evaluation. In *Proc. SIGMETRICS '98/PERFORMANCE '98*, 1998.

[2] Raj Jain. *The Art of Computer Systems Performance Analysis*. Wiley, New York, 1991.

[3] Diwakar Krishnamurthy, Jerome A. Rolia, and Shikharesh Majumdar. A synthetic workload generation technique for stress testing session-based systems. *IEEE TSE*, 32(11), 2006.

[4] Daniel A. Menascé. Load testing of web sites. In *IEEE Internet Computing*, 2002.

[5] Daniel A. Menascé, Virgilio A. F. Almeida, Rodrigo Fonseca, and Marco A. Mendes. A methodology for workload characterization of e-commerce sites. In *Proc. EC '99*, 1999.

[6] Eike Schulz. A model-driven performance testing approach for session-based software systems, 2013. Student research paper, Kiel University, Kiel, Germany.

[7] Thomas Stahl and Markus Völter. *Model-Driven Software Development – Technology, Engineering, Management.* Wiley & Sons, 2006.

[8] André van Hoorn, Sören Frey, Wolfgang Goerigk, Wilhelm Hasselbring, and Holger Knoche et al. DynaMod project: Dynamic analysis for model-driven software modernization. In *Proc. MDSM '11*, volume 708 of *CEUR Workshop Proc.*, 2011.

[9] André van Hoorn, Matthias Rohr, and Wilhelm Hasselbring. Generating probabilistic and intensity-varying workload for Web-based software systems. In *Proc. SIPEW '08*, 2008.

[10] André van Hoorn, Jan Waller, and Wilhelm Hasselbring. Kieker: A framework for application performance monitoring and dynamic software analysis. In *Proc. ICPE '12.* ACM, 2012.

Towards Quality Models in Software Migration

Gaurav Pandey, Jan Jelschen, Andreas Winter
Carl von Ossietzky Universität, Oldenburg, Germany
{pandey,jelschen,winter}@se.uni-oldenburg.de

Abstract

To preserve legacy systems in continuous software development and evolution, next to redevelopment, they can be migrated to new environments and technologies. Deciding on evolution and migration strategies early, requires predicting the quality of the migrated software systems depending on applied tools. There is a need for comparable measures, estimating the inner software quality of legacy and target systems.

Technically, software migration tools use a transformation-based toolchain using model-driven technologies. Therefore, quality measurement can be based on the underlying models representing input and output of applied migration tools.

This paper proposes a Software Migration Quality Model in order to provide support for quality-driven tailoring of utilized model-driven migration tools.

1 Motivation

Software Migration comes across as an important technique to evolve legacy systems into new environments and technologies without changing the system's functionality [4]. It continues the modernization, operation and development of software without dealing with the risk and cost of a complete redevelopment [8]. Each migration project requires an especially tailored toolchain [2], aiming at preferably automatically transferring legacy to target. Moreover, deciding between software migration and redevelopment as well as choosing the components of migration toolchain, requires reliable predictions regarding quality of migrated software. To achieve this, there is a need to measure and compare the quality of the legacy software, migrated software and the intermediate software stages.

Monitoring changes in software-quality during software development is supported by various incremental approaches: e.g. Teamscale [3] and SonarQube [9]. These approaches are restricted to a single implementation platform. Since language based software migrations, e.g. migrating from COBOL to Java, deal at least with two different development platforms, cross platform monitoring is needed. This challenges for providing metrics, which are applicable in both environments allowing comparison of quality issues across platforms.

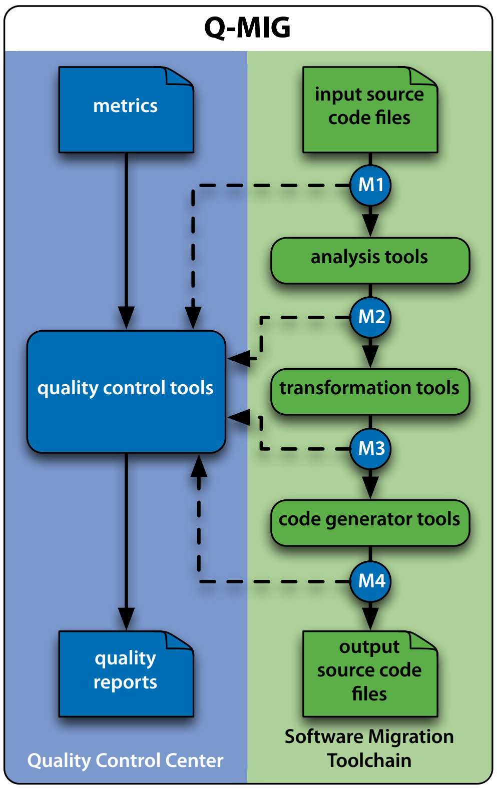


Figure 1: The Q-MIG integrated toolchain.

The Q-MIG-project[1] (Quality-driven software MIGration) aims at monitoring changes in software quality during migration and at supporting quality driven decisions on migration strategies and tooling [6].

2 Q-MIG

Q-MIG combines software migration toolchain [1] with a quality control center (cf. Figure 1). It allows for software quality management during the migration process including quality prediction during project planning and tooling. The monitoring points *(M1-M4)* allow to measure, monitor and compare the quality in the software toolchain. Here, migration and quality tools are integrated. Cross-platform quality comparison is achieved by the calculation of same metric at monitoring points. Moreover, calculation of

[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).*

different metrics at the monitoring points, that represent the same quality, also enables quality comparison. This is particularly useful when the implementations of the quality metrics differ across the monitoring points, but their interpretations are the same. Also, analyzing the quality of migrated software with respect to tools used, helps in determining the combination of components in migration toolchain.

The software migration toolchain in Figure 1 can technically be viewed as a combination of model-driven tools. As model driven environments can handle code and model in the same fashion, we define the internal representation of the two as a *codel*. At various monitoring points these codels are available for picking the artifact's quality prior and after each migration step.

As the migration toolchain is already model-driven, model-based approaches to measure the quality can be applied. Measurement of the quality of codels can be based on querying [7] which has been stressed as an important enabling technology in software evolution. So, quality measurement and monitoring in software migration can utilize the already existing model-driven query tools to calculate and to compare the quality of succeeding codels.

3 Software Migration Quality Model

Measuring and comparing quality of succeeding codels requires to align metrics, codels and the applied migration tools, which can be viewed as model transformations, in a *Software Migration Quality Model*. Figure 2 shows a conceptual view on this model.

The quality model for software migrations (*Q-MIG-Model*) aligns *Components* providing the required transformation services (*Transformations*). It also aligns the originating and resulting *Codels* to migration projects specific *QualityModels* which summarize all *Metrics* defining the project specific quality issues. For each *Codel* all relevant metrics-values are stored. These *Values* will be monitored during migration and knowledge on changing their values during migration will help to predict the quality of migration results.

Metrics are calculated by applying *Queries* to *Codels* resulting in the appropriate *Values*. Since the *Codels* conform to certain language definitions (either grammars or meta models) defining the codel's abstract syntax, the *Queries* also have to conform to the language definitions.

Migration steps can be viewed as services realized by *Components* according to the service-based tool integration approach SENSEI [5]. Separating the *Components* from the implemented *Transformations* allows for considering and comparing different migration tools like different COBOL-to-Java Translators.

Next steps in Q-MIG deal with specifying relevant metrics in COBOL-to-Java migration projects and applying these values to all codels in a given migration tool chain to provide an initial migration monitoring.

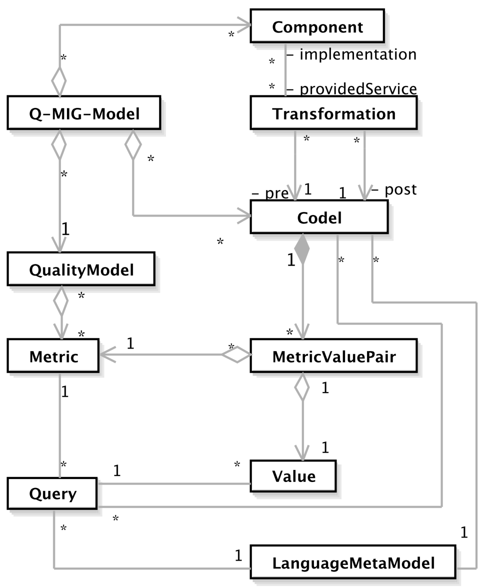


Figure 2: Software Migration Quality Model.

4 Summary

This paper presented the first steps in the Q-MIG project in providing a quality-driven support to software migration. The strongly model-driven foundation of Q-MIG was given by referring to Q-MIG's Software Migration Quality Model.

References

[1] C. Becker, U. Kaiser. Test der semantischen Äquivalenz von Translatoren am Beispiel von CoJaC. *Softwaretechnik-Trends*, 32(2), 2012.

[2] J. Borchers. Erfahrungen mit dem Einsatz einer Reengineering Factory in einem großen Umstellungsprojekt. *HMD*, 34(194): 77–94, Mar. 1997.

[3] CQSE GmbH. Teamscale, 2014. http://www.cqse.eu/en/products/teamscale/overview/.

[4] A. Fuhr, A. Winter, U. Erdmenger, T. Horn, U. Kaiser, V. Riediger, W. Teppe. Model-Driven Software Migration — Process Model, Tool Support and Application. In A. D. Ionita, M. Litoiu, G. Lewis, editors, *Migrating Legacy Applications: Challenges in Service Oriented Architecture and Cloud Computing Environments*. IGI Global, Hershey, PA, USA, 2012.

[5] J. Jelschen. SENSEI: Software Evolution Service Integration. In *CSMR-WCRE Software Evolution Week*, Antwerp, Belgium, 469–472, 2014. IEEE.

[6] J. Jelschen, G. Pandey, A. Winter. Towards Quality-Driven Software Migration. In *Proceedings of the 1st Collaborative Workshop on Evolution and Maintenance of Long-Living Systems, Kiel*, 8–9, 2014.

[7] B. Kullbach, A. Winter. Querying as an Enabling Technology in Software Reengineering. In *3rd European Conference on Software Maintenance and Reengineering*, 42–50. IEEE Computer Society, 1999.

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

[9] SonarSource. SonarQube, 2014. http://www.sonarqube.org.

A Tool-Supported Quality Smell Catalogue For Android Developers

Jan Reimann, Martin Brylski, Uwe Aßmann
Software Technology Group
Technische Universität Dresden
Dresden, Germany
jan.reimann|uwe.assmann@tu-dresden.de, martin.brylski@gmail.com

Usual software development processes apply optimisation phases in late iterations. The developed artefacts are optimised regarding particular qualities. In this sense *refactorings* are executed since the existing behaviour is preserved while the artefact improves its quality properties. In the area of mobile applications qualities (e.g. *energy* or *memory efficiency*) is crucial due to limited hardware resources. The problem is twofold. First, there is no unified set of potential problems which can cause an artefact to dissatisfy a quality requirement. Such a set must contain not only the indicators to search for and a relation to specific qualities, but it must also provide solutions to resolve them. Second, no tool support exists to enable developers focussing problems regarding specific qualities. To overcome these problems we introduce a new quality smell catalogue for Android applications in this paper and provide tool support.

1 Motivation

The *quality requirements* of applications may be specified explicitly in the requirements document, or become present in optimisation phases when deficiencies regarding the qualities are noticed, as e.g. that the battery of a mobile device drains too fast. What developers do when optimising w.r.t to such qualities is refactoring [2]. They manually detect relevant artefacts containing structures being responsible for not satisfying the particular quality requirements. Fowler calls such structures *bad smells* which indicate candidates for applying refactorings to improve qualities while preserving the internal behaviour.

Against this background we correlated the concepts *bad smell*, *quality* and *refactoring*, and introduced the term *quality smell*. A quality smell is a bad smell with regard to a specific quality expressing that this bad smell negatively influences the given quality of a model. In particular, the identified quality smell can be resolved by a concrete model refactoring [5]. We implemented the concept of quality smells within our generic model refactoring framework[1] [6] for enabling developers to detect and resolve quality smells in early development phases already and focus specific qualities explicitly. Our tool is based on the Eclipse Modeling Framework and seamlessly integrates into existing model-driven setups.

In contrast to Fowler's bad smells and refactorings (being universally applicable), quality smells are very concrete because satisfying a particular quality requirement has different meanings or granularities in distinct contexts. Thus, the principle of a quality smell is universal in a particular domain but the concrete instance is specific because it refers to a precise setting, as e.g. the use of a concrete framework. Because of this we decided to focus the context of mobile development since quality requirements play an essential role in this area. We chose Android since it is publicly available. The problem in mobile development is that developers are aware of quality smells only indirectly because their definitions are informal (best-practices, bug tracker issues, forum discussions etc.) and resources where to find them are distributed over the web. It is hard to collect and analyse all these sources under a common viewpoint and to provide tool support for developers.

To overcome these limitations we compiled a unified catalogue for Android. It contains 30 possible quality smells, explaining which qualities they influence, and potential refactorings to resolve them.

2 Quality Smell Catalogue

We have created a catalogue containing 30 quality smells. In the following we present only one quality smell but the whole catalogue can be found here:

http://www.modelrefactoring.org/smell_catalog/

Based on the catalogues from Brown et al. [1], Fowler [2] and Gamma et al. [3] we derived a scheme in Table 1 which each quality smell conforms to.

[1] http://www.modelrefactoring.org/

Table 1: Scheme of a quality smell

Concept	Description
Name	unique and descriptive identifier
Context	categoric relation (e.g. UI, sensors, etc.)
Affected Qualities	lists qualities negatively influenced by this quality smell
Description	detailed description of the specific problem including an example
Refactorings	explains refactorings being able to resolve this quality smell
References	further (web) resources containing more information regarding this quality smell
Related Quality Smells	list of similar or related smells

As an example the quality smell *Data Transmission Without Compression* is presented in the following.[2]

Name Data Transmission Without Compression

Context Implementation, Network

Affected Qualities Energy Efficiency

Description In [4] Höpfner and Bunse discussed that transmitting a file over a network infrastructure without compressing it consumes more energy than with compression. More precisely, energy efficiency is improved in case the data is compressed at least by 10%, transmitted and decompressed at the other network node.

The example in Listing 1 shows file transmission implemented with the Apache HTTP Client Library.[3] With the help of JaMoPP[4] we can refer to Java code in terms of a model.

```
1  public static void main(String[] args) throws Exception {
2      HttpClient httpclient = new DefaultHttpClient();
3      HttpPost httppost = new HttpPost("http://some.url:8080/
           servlets-examples/servlet/RequestInfoExample");
4      FileBody bin = new FileBody(new File(args[0]));
5      StringBody comment = new StringBody("A␣binary␣file");
6      MultipartEntity reqEntity = new MultipartEntity();
7      reqEntity.addPart("bin", bin);
8      reqEntity.addPart("comment", comment);
9      httppost.setEntity(reqEntity);
10     System.out.println("executing␣request␣" + httppost.
           getRequestLine());
11     HttpResponse response = httpclient.execute(httppost);
12     HttpEntity resEntity = response.getEntity();
13     EntityUtils.consume(resEntity);
14 }
```

Listing 1: File transmission without compression before refactoring

[2]For the missing properties *References* and *Related Quality Smells* we refer to the online catalogue.

[3]The example was taken and adapted from http://archive.apache.org/dist/httpcomponents/httpclient/binary/httpcomponents-client-4.2.4-bin.zip.

[4]http://www.jamopp.org

In line 4 one can see that the passed `File` object in this constructor is transmitted without compression.

Refactorings The refactoring *Add Data Compression to Apache HTTP Client based file transmission* adds a compression method. Then it passes the `File` parameter of the constructor in line 4 of Listing 1 to this method. Thus, the file is transmitted with compression. In Listing 3 the result of this refactoring is depicted. Line 3 contains the invocation of the compression method `gzipFile(File uncompressedFile)`.

```
1  public static void main(String[] args) throws Exception {
2    // ...
3    FileBody bin = new FileBody(gzipFile(file));
4    // ...
5  }
6  private static File gzipFile(File file){
7    File gzFile = File.createTempFile(file.getName(), "gz");
8    FileInputStream fis = new FileInputStream(file);
9    GZIPOutputStream out = new GZIPOutputStream(new
           FileOutputStream(gzFile));
10   byte[] buffer = new byte[4096];
11   int bytesRead;
12   while ((bytesRead = fis.read(buffer)) != -1){
13     out.write(buffer,0, bytesRead);
14   }
15   fis.close();
16   out.close();
17   return gzFile;
18 }
```

Listing 2: File transmission with compression after refactoring

3 Conclusion

In this paper we motivated the need for a quality smell catalogue. As a representative the exemplary quality smell *Data Transmission Without Compression* is presented. Our tool can be used to detect the smell by searching for uncompressed data transmission implemented with the Apache HTTP Client framework. Beyond that it provides a refactoring to resolve the quality smell which adds file compression to the identified class. The whole catalogue can be seen online.

References

[1] W. H. Brown, R. C. Malveau, and T. J. Mowbray. *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis.* Wiley, 1st edition, 1998.

[2] M. Fowler. *Refactoring: Improving the Design of Existing Code.* Addison-Wesley, 1999.

[3] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns. Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995.

[4] H. Höpfner and C. Bunse. Towards an Energy-Consumption Based Complexity Classification for Resource Substitution Strategies. In *22nd Workshop "Grundlagen von Datenbanken 2010"*, 2010.

[5] J. Reimann and U. Aßmann. Quality-Aware Refactoring For Early Detection And Resolution Of Energy Deficiencies. In *4th International Workshop on Green and Cloud Computing Management*, 2013.

[6] J. Reimann, M. Seifert, and U. Aßmann. On the reuse and recommendation of model refactoring specifications. *Software & Systems Modeling*, 12(3):579–596, 2013.

Architectural Restructuring by Semi-Automatic Clustering to Facilitate Migration towards a Service-oriented Architecture

Marvin Grieger, Stefan Sauer
Universität Paderborn, s-lab – Software Quality Lab
Zukunftsmeile 1, 33102 Paderborn
{mgrieger, sauer}@s-lab.upb.de

Markus Klenke
TEAM GmbH
Hermann-Löns-Straße 88, 33104 Paderborn
mke@team-pb.de

1 Introduction

Enterprises are nowadays increasingly faced with the fact that their information systems have become legacy. A common solution to this problem is to migrate the existing application to a new environment, e.g. to a new platform. Modern platforms often employ the architectural style of service-oriented architecture (SOA), in which an application is composed of loosely coupled entities that provide functionality as a service. The challenge in migrating towards a service-oriented architecture is to identify these entities in the legacy application and restructure its architecture accordingly.

In this paper we describe a semi-automatic process which combines hierarchical and partitioning clustering in order to improve an initial service design. The result is used to restructure the legacy application during the migration process. The purpose of the approach is to create a maintainable, service-oriented architecture. It is being developed in the context of an industrial project in which a model-driven software migration process and tool chain is built [1]. It will be applied on a set of enterprise legacy applications.

The paper is structured as follows: The case study is briefly introduced in Section 2. In Section 3 we describe our approach using an example. Section 4 discusses related work in this area. Thereafter we present preliminary results and outline future work in Section 5.

2 Case Study

The case study is based on a platform migration from Oracle Forms to Oracle ADF. On the source platform, the main concept of an application is to provide masks as user interfaces to interact with an underlying database. Applications consist of multiple modules, where each module is implemented separately. A module consists of module components, for which we assume that each includes exactly one mask as well as related data connections and business logic. In migrating towards the target platform, services have to be identified. As the user realizes business processes by traversing masks, a single module provides functionality that is related to one or more sequential activities in the underlying process. Thus, a module can be seen as a *business service*, according to the classification of service types given in [2].

Mapping modules to *services* results in applications that are hard to maintain, since many fine-grained services emerge. Therefore, *composite services* need to be identified, which aggregate and orchestrate the *business services*. In addition, since the capabilities of the legacy platform for software reuse were quite limited, code clones are omnipresent in every application we examined. Removing these clones by extracting the functionality and moving it into one single service will again increase the maintainability of the resulting application and additionally reduce the effort of manual reimplementation that is necessary.

3 Restructuring Process

The general idea of the approach is illustrated in Figure 1. As described in Section 2, there exist fine-grained *business services* as an initial service design. Additionally, we assume the existence of a dedicated *composite service*, which is externally callable as an entry point for an end user. The initial service design can be seen in Figure 1 (I).

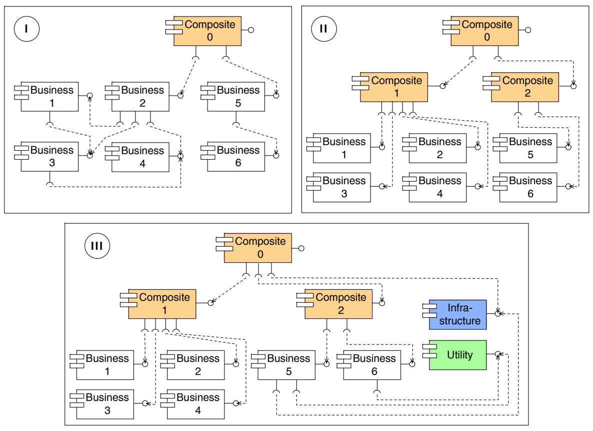


Figure 1: Illustrative example

The process related to our approach is shown in Figure 2. Based on the initial service design, hierarchical and partitioning clustering are performed to improve it gradually. These two activities consist of multiple steps which are described in detail subsequently.

3.1 Hierarchical Clustering

The first step of the restructuring process is to perform *hierarchical clustering* based on the dependencies between the services. As the dependencies constitute navigation flows, the resulting clusters are expected to aggregate services that are related in terms of the underlying business process. The clusters are then implemented by introducing additional *composite services*, as can be seen in Figure 1 (II).

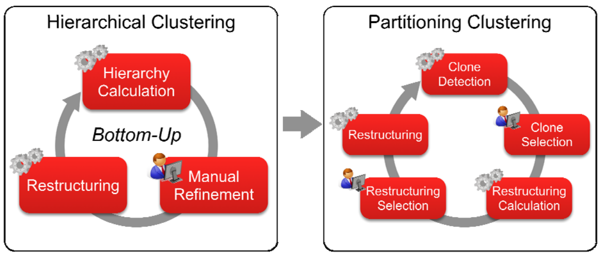


Figure 2: Restructuring process

As illustrated in Figure 2, hierarchical clustering consists of three activities, which are performed repeatedly. Each iteration starts by calculating new clusters for a hierarchy layer. This is performed automatically using a clustering algorithm. The result gets refined manually since it may not be optimal. This is due to the fact that clustering is performed on navigation flows rather than the business processes itself. The calculated clusters are then realized by introducing *composite services*. For that, predefined restructuring rules are executed. Depending on the size of the application, further iterations may be performed.

3.2 Partitioning Clustering

The second step of the restructuring process is to detect and remove software clones. This is achieved by partitioning the functionalities of multiple services into individual and common parts. Common parts are cut out and moved to a single service. This might be an existing one or a newly created *utility* or *infrastructure service*. The purpose of these types of services is to provide cross-cutting domain or technical functionalities that are used across different services. Therefore, these services do not belong to a specific layer calculated during the hierarchical clustering. The result of this step is shown in Figure 1 (III).

The *partitioning clustering* consists of five activities that are executed repeatedly, as shown in Figure 2. First, automatic clone detection is performed to suggest candidates of software clones. In order to do this, a strategy has to be defined where to search for clones, i.e., which services are being compared and for which functionalities clones are being detected. The reengineer thereafter selects clones that are to be removed. Based on this information, an algorithm determines restructuring possibilities, i.e., where to implement the functionality that should be cut out. As a result, it may either suggest the reengineer to move the functionality to an existing service or to create a new one. After the reengineer performs a decision, the restructuring is applied by executing a set of predefined restructuring rules.

4 Related Work

Migration towards SOA is an active area of research [3][4][5]. In contrast to most of the existing approaches, we assume that an initial service design exists, based on a mapping of existing legacy structures. As a result, we focus on identifying *composite services* instead of *business services*. In addition, we consider the use of clone detection for further refinement of the service design. To the best of our knowledge, no approach has combined these techniques in the context of a migration towards SOA.

5 Preliminary Results and Future Work

This approach is being developed in an industrial context. It has already been applied in the migration of an enterprise legacy system. Based on the described process, systematic restructuring steps have been derived which were manually applied on an application. We were able to show that the restructured service design was better in terms of increased maintainability and reduced effort for manual reimplementation, compared to a migration without restructuring.

The initial service design is improved by introducing new services and moving functionality between existing ones. In practice, a merge operation for small services is desired which may complement the approach. We are currently working on a tool that supports the activities described in the process. Automation will reduce the effort of restructuring and reduce the risk of manual errors. Afterwards, we will evaluate the approach with multiple applications.

6 Literature

[1] Grieger, M.; Güldali, B.; Sauer, S.: Sichern der Zukunftsfähigkeit bei der Migration von Legacy-Systemen durch modellgetriebene Softwareentwicklung. In Softwaretechnik-Trends 32(2):37-38, 2012.

[2] Alahmari, S.; Zaluska, E.; De Roure, D.: A service identification framework for legacy system migration into SOA. In Proc. IEEE Intl. Conf. on Service Computing (SCC 2010), pp. 614-617, 2010.

[3] Fuhr, A.; Horn, T.; Riedinger, V.: Using dynamic analysis and clustering for implementing services by reusing legacy code. In Proc. 18th Working Conf. Reverse Engineering (WCRE 2011), pp. 275-279, 2011.

[4] Zhang Z.; Liu, R.; Yang, H.: Service identification and packaging in service oriented reengineering. In Proc. Intl. Conf. Software Engineering and Knowledge Engineering (SEKE 2005), pp. 620-625, 2005.

[5] Matos, C.; Heckel, R.: Migrating legacy systems to service-oriented architectures. Electronic Communications of the EASST, vol. 16, 2008.

Determining the Intent of Code Changes to Sustain Attached Model Information During Code Evolution

Michael Langhammer, Max E. Kramer
Karlsruhe Institute of Technology
{michael.langhammer, max.e.kramer}@kit.edu

Abstract

If code is linked to models with additional information that cannot be represented in the code, changes in the code may have unwanted effects on these models. In such scenarios, the desired effect of code changes may be unclear and impossible to determine regardless of the used change recording or propagation mechanism. Existing round-trip engineering tools do not solve this problem as they just support models that contain only information that can be regenerated from the code or drop such information. In this position paper, we propose an approach for controlled code evolution that automatically maps unambiguous code changes to well-defined operations and that blocks ambiguous changes until the user clarified his intent. We present a case study for Java code and component-based architectures to show how such an approach would only permit code changes with unambiguous effects on the architecture. To stimulate discussions at the workshop, we argue why such an approach is necessary and describe benefits and drawbacks of such a solution.

1 Introduction

In Model-Driven Software Development (MDSD), a software system may be described by its source code and with models that represent a part of the system from a specific perspective for a specific task. The used modelling languages and model instances may be tailored to specific development and analysis tasks so that they display only the required information. If the same information is required for several tasks, then information may be spread across various models. In such cases, all artefacts have to be mutually synchronized to avoid differences between code and models (drift) and inconsistencies (erosion) during software evolution. There are several ways to bypass this need for synchronization: Redundant information can be completely disallowed or strict refinement can be employed so that every information has a unique origin and redundant elements in other artefacts can always be regenerated [3]. A problem arises, however, if attached models contain information that can neither be represented nor be computed from the code. In such cases, code changes may occur for which the intended impact on attached models is unclear.

2 Approach

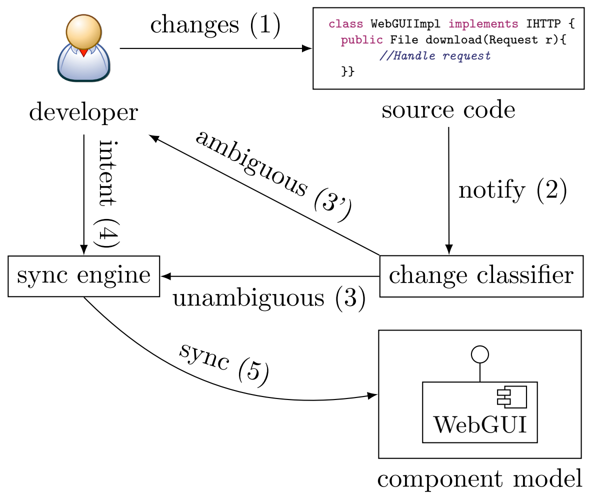
We propose a co-evolution approach that differentiates between unambiguous and ambiguous code changes with respect to a linked model. We use the Java Model Printer and Parser (JaMoPP) [4] to obtain a syntax tree model so that we can directly work on code model changes instead of textual changes.

Unambiguous changes are changes where the impact to the linked model and the intent of the user are clear. For example, when the effect on the linked model has been specified upfront in a transformation. This means a change is unambiguous when the corresponding operations on the linked models are clear.

Ambiguous changes, however, are changes where the intent of the developer cannot be determined automatically. In our approach, these changes are blocked to let the developer clarify his intent in a way that unambiguously induces an operation on the attached models. If a developer moves, for example, a method from one class to another, additional information on design rationale, which is not represented in the code but in attached models, may become obsolete for the new system. As the developer has no access to this information in the code editor, it is unclear which effect on this information he desired for this code change. The desired effect of his change cannot be determined with confidence without any further information or assumptions regardless of the used change recording or propagation mechanism. For our method move example, this means, that the developer can specify whether the former design rationale for the method is still valid. After this, the source code change is mapped to an operation on the syntax model and can be propagated to the linked models using model-to-model transformations that use the additional clarification information.

If a developer cannot or does not want to clarify the impact on linked models, we may obtain inconsistencies that have to be resolved later or by others.

The benefit of our approach is that models, which contain additional information with unclear code-correspondence, are kept synchronized with the source code. Hence the effort for manually synchronization between the model and source code is reduced to the provision of clarifying information. Since we do not



```
class WebGUIImpl implements IHTTP {
  public File download(Request r){
        //Handle request
  }}
```

Figure 1: The proposed approach applied to Java source and a component-based architecture model.

regenerate the linked model, additional information that is not represented in the source code is sustained in the models. Furthermore, developers are notified when their changes affect linked models, which gives developers the possibility to influence the impact on linked models. A drawback of our approach is that the workflow of developers may get interrupted when the changes they made are blocked until clarification.

We will embed the proposed approach into a view-centric engineering framework [5]. Change implications will be specified with declarative metamodel mappings and instance level correspondences between code regions and linked models will be automatically managed in a tracing model. Using this information we will check whether the current code change affects code elements that are linked to another model.

3 Application scenario

An application scenario of the proposed approach is to synchronize Java source code and component-based architecture models (see Figure 1) with additional performance information from the Palladio simulator [2]. Palladio models consist of components, interfaces and signatures, which are represented in the source code e.g. in terms of interfaces, classes and method declarations (cf. [5]). If a developer changes (1), for example, the name of an architecture relevant interface, then this change is unambiguous (2,3) and the corresponding interface in the component model can be renamed automatically (5). If a developer renames (1), for example, a class method that implements a signature of an architecture relevant interface, then the impact on the component architecture is not clear (2). The developer has to be asked (3') whether he wants to change the signature of the component interface or whether the changed method should no longer be linked to the architecture interface (4). If he chooses the first option, this will also influence other parts of the code: all corresponding methods of classes that realize components that provide the corresponding architecture interface will be renamed as well (5).

4 Related Work

The Software Model Extractor (SoMoX) [3] is able to generate component models from source code. The architecture is, however, not updated incrementally and additional information added manually to the generated architectureis lost during regeneration. UML-Lab[1] supports round-trip engineering for UML class diagrams and source code. It does, however, not support additional information in the class diagrams which can not be represented in the source code. The Orthographic Software Modeling (OSM) [1] approach proposes the use of a Single Underlying Model (SUM) that contains all information of a particular software system. This approach does not need to synchronize information between models because there is no redundant information in the SUM. It is, however, an open question how a SUM for object-oriented code and component-based architectures can be obtained.

5 Conclusion

In this position paper, we have presented an approach for the co-evolution of source code and models that contain additional information. We have also briefly described an application scenario using Java source code and component-based architecture models. In the future, the presented approach could be the foundation for a round-trip engineering approach in which code and models can evolve in parallel even if they partly contain independent information.

References

[1] Colin Atkinson, Dietmar Stoll, and Philipp Bostan. "Orthographic Software Modeling: A Practical Approach to View-Based Development". In: *Evaluation of Novel Approaches to Software Engineering*. Vol. 69. 2010, pp. 206–219.

[2] Steffen Becker, Heiko Koziolek, and Ralf Reussner. "The Palladio component model for model-driven performance prediction". In: *Journal of Systems and Software* 82.1 (2009), pp. 3–22.

[3] Steffen Becker et al. "Reverse engineering component models for quality predictions". In: *Software Maintenance and Reengineering (CSMR), 2010 14th European Conference on*. IEEE. 2010.

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

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

[1]http://www.uml-lab.com/