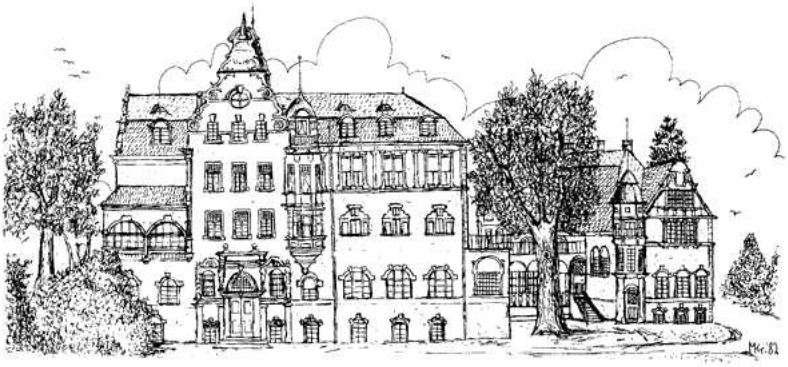
6. Workshop Software Reengineering (WSR 2004)

Bad Honnef, 3.-5. Mai 2004

Jürgen Ebert, Volker Riediger, Andreas Winter
(Universität Koblenz-Landau)
Franz Lehner (Universität Regensburg)



Vorwort

Der *Workshop Software-Reengineering* (WSR) (`http://www.uni-koblenz.de/ist/wsr`) bietet ein deutschsprachiges Forum zum Thema *Software-Wartung und Reengineering*. Ziel ist es, sowohl Praktiker als auch Wissenschaftler, die an Softwaretechnik und Wirtschaftsinformatik interessiert sind, zusammenzubringen, um Probleme, Ideen und Lösungsansätze aus ihrer aktuellen Arbeit in Forschungsprojekten und praktischer Tätigkeit zu diskutieren.

Der sechste *Workshop Software-Reengineering* fand vom 3.–5. Mai 2004 wieder im Physikzentrum in Bad Honnef statt. Die Unterstützung des Workshops durch die GI-Fachgruppen Softwaretechnik (SE) und Software Produktmanagement (WI-PrdM) reflektiert auch die Positionierung des WSR als übergreifende Veranstaltung von Softwaretechnik und Wirtschaftsinformatik.

Der Workshop wurde von 50 Teilnehmern besucht. In 35 Vorträgen wurde ein Überblick über die aktuellen Reengineering-Aktivitäten in Deutschland, Österreich und den Niederlanden gegeben. Die Themen des Workshops spiegeln die Breite des Fachgebiets wider: *Software Evolution, Praxis- und Erfahrungsberichte, Methoden und Werkzeuge, Reengineering-Prozesse, Architekturerkennung, Programmanalyse, Entwurfsmuster, Programmvisualisierung, Softwarekomponenten und Toolintegration*, sowie *Sprachen, Grammatiken und Transformationen*.

Trotz des vollen Programms kamen Diskussionen und Gespräche auch in diesem Jahr nicht zu kurz. Hierzu hat sicherlich auch die diskussionsintensive Atmosphäre im Physikzentrum und der gut gefüllte Weinkeller beigetragen. Auch der inzwischen zur Tradition gewordene Spaziergang zur Bad Honnefer Eisdiele bot eine gute Gelegenheit, über Begriffe, praktische Probleme und theoretische Lösungsansätze zu diskutieren.

In diesem Jahr durften wir erstmals den neuen Hörsaal nutzen. Hier ist es dem Physikzentrum hervorragend gelungen, eine zeitgemäße Präsentationsumgebung zu schaffen, in der es Spaß macht, vorzutragen und zu diskutieren. Die Unterbringung und Versorgung im Physikzentrum war auch beim 6. WSR wieder perfekt. Stellvertretend für das gesamte Personal sei hierfür Herrn Gomer und Frau Viehöfer gedankt. Herzlicher Dank gebührt auch Hans Becker, Universität Koblenz, für seine Unterstützung bei der administrativen Workshop-Organisation.

Inzwischen hat sich der *Workshop Software-Reengineering*, der als "Low-Cost"-Workshop ohne eigenes Budget durchgeführt wird, als zentrale deutschsprachige Reengineering-Konferenz etabliert. Der WSR hat auch maßgeblich zur Bildung der deutschsprachigen "Reengineering-Community" beigetragen. Im Rahmen des Workshops wurde daher auch beschlossen, die bisherigen Aktivitäten im Rahmen einer *Fachgruppe Software-Reengineering* der Gesellschaft für Informatik weiterzuführen. Mit der Gründung dieser Fachgruppe wurden Andreas Winter, Universität Koblenz, als Sprecher und Rainer Gimnich, IBM Global Services, Frankfurt, als stellv. Sprecher beauftragt. Weitere Informationen zur Fachgruppe und zum Themengebiet Software-Reengineering können der Reengineering-Mailingliste (Anmeldung unter `http://mailhost.uni-koblenz.de/mailman/listinfo/reengineering`) entnommen werden.

Die Kurzfassungen der Vorträge des Workshops Software-Reengineering sind im folgenden zusammengestellt. Sie sind auch — wie die Vortragszusammenstellungen der Vorjahre — unter `http://www.uni-koblenz.de/ist/wsr` abrufbar.

Die nächsten *Workshops Software-Reengineering* sind für den 2.-4. Mai 2005 und den 3.-5. Mai 2006 ebenfalls im Physikzentrum Bad Honnef geplant.

Mai 2004

Jürgen Ebert
Franz Lehner
Volker Riediger
Andreas Winter

Programm

1. **Michael Burch**
   (Universität des Saarlandes, Saarbrücken)
   Stephan Diehl, Peter Weißgerber
   (Katholische Universität Eichstätt-Ingolstadt)
   EPOSee: A Tool for Visualizing Software Evolution Patterns

2. **Thomas Zimmermann, Andreas Zeller**
   (Universität des Saarlandes, Saarbrücken)
   Data Mining Version History

3. **Dharmalingam Ganesan, Jean-Francois Girard**
   (Fraunhofer Institut für Experimentelles Software Engineering, Kaiserslautern)
   M-Track: A Metric Tool Framework for Monitoring the Evolution of Object-Oriented Systems

4. **Harry Sneed**
   (Universität Regensburg)
   Beleg Reengineering

5. **Jens Borchers**
   (CC GmbH, Wiesbaden)
   Genauigkeit von Aufwandsschätzungen in Reengineering-Projekten am Beispiel einer großen Sprachumstellung von Assembler nach COBOL

6. **Simon Giesecke, Andre Marburger**
   (RWTH Aachen)
   E-CARES research project: Interactive, stakeholder-tailored re-engineering

7. **Uwe Erdmenger**
   (pro et con, Innovative Informatikanwendungen, Chemnitz)
   Der pro-et-con Migration Manager - Ein Werkzeug für die Migration von Host-Anwendungen auf UNIX-Plattformen

8. **Werner Teppe**
   (Amadeus, Bad Homburg)
   Unterstützung von Reengineering Projekten durch eine moderne, gemeinsame Softwareentwicklungsumgebung (Praxisbericht)

9. **Silvia Breu**
   (Universität Passau)
   Case Studies in Aspect Mining

10. **Rainer Gimnich**
    (IBM Global Services, Frankfurt)
    The IBM Legacy Transformation Offering

11. **Peter Schützendübe**
    (Suss MicroTech Lithography, Asslar)
    Stand des Software-Reengineering in der SMTL

12. **Stefan Opferkuch, Jochen Ludewig**
    (Universität Stuttgart)
    Software-Wartung – eine Taxonomie

13. **Urs Kuhlmann, Andreas Winter**
    (Universität Koblenz)
    Softwarewartung und Prozessmodelle in Theorie und Praxis

14. **Arie van Deursen, Leon Moonen**
    (CWI & Delft University of Technology)
    Christine Hofmeister
    (Lehigh University)
    Rainer Koschke
    (University of Stuttgart)
    Claudio Riva
    (Nokia Research Center, Helsinki)
    Viewpoints in Software Architecture Reconstruction

15. **Rainer Koschke, Daniel Simon**
    (Universität Stuttgart)
    Symphony Fallstudie: Hierarchische Reflexion Modelle

16. **Markus Bauer, Mircea Trifu**
    (FZI Forschungszentrum Informatik, Karlsruhe)
    Combining Clustering with Pattern Matching for Architecture Recovery of OO Systems

17. **Michael Müller-Wünsch**
    (myToys.de, Berlin)
    Wartung von Standard-Software-Systemen am Beispiel von `myToys.de`

18. **Martin Moro**
    (Universität Regensburg)
    Heidelberger Eye Explorer, Die technologisch Neuausrichtung,
    Erfahrungsbericht aus einem Reengineering Projekt

19. **Udo Borkowski**
    (Aachen)
    C^4D oder Wie ich lernte, mit Code Clones zu leben

20. **Holger Cleve, Andreas Zeller**
    (Universität des Saarlandes, Saarbrücken)
    Experimental Program Analysis

21. **Rene Witte**
    (Concordia University, Montreal)
    Ulrike Kölsch
    (T-Systems, Wörth)
    Supporting Reverse Engineering Tasks with a Fuzzy Repository Framework

22. **Rainer Schmidberger**
    (Universität Stuttgart)
    Reverse-Engineering durch Identifikation von Eingabedaten-Äquivalenzklassen aus Programmabläufen

23. **Jochen Kreimer**
(Universität Paderborn)
Adaptive Erkennung von Entwurfsmängeln in objekt-orientierter Software

24. **Adrian Trifu, Olaf Seng, Thomas Genssler**
(FZI Forschungszentrum Informatik, Karlsruhe)
Automatisierte Behebung von Strukturproblemen in objekt-orientierten Systemen

25. **Jörg Niere**
(Universität Siegen)
Recovering Design Elements in Large Software Systems

26. **Lothar Wendehals**
(Universität Paderborn)
Specifying Patterns for Dynamic Pattern Instance Recognition with UML 2.0 Sequence Diagrams

27. **Holger Eichelberger, Jürgen Wolff von Gudenberg**
(Universität Würzburg)
JTransform, a Tool for Source Code Analysis

28. **Jens Krinke**
(FernUniversität Hagen)
Textual vs. Graphical Visualization of Fine-Grained Dependencies

29. **Dierk Ehmke**
(Darmstadt)
Rechnergestützte Diagnose in Software-Entwicklung und Test

30. **Thomas Haase**
(RWTH Aachen)
Die Rolle der Architektur im Kontext der a-posteriori Integration

31. **Uwe Zdun**
(Abteilung für Wirtschaftsinformatik, Wien)
Komponierung, Konfiguration und Adaptierung von heterogenen Software Komponenten

32. **Jens Knodel**
(Fraunhofer Institut für Experimentelles Software Engineering, Kaiserslautern)
On Analyzing the Interfaces of Components

33. **Ralf Lämmel**
(Vrije Universiteit & CWI, Amsterdam)
Evolution of Language Interpreters

34. **Wolfgang Lohmann**
(Universität Rostock)
Two co-transformations of grammars and related transformation rules

35. **Niels Veerman**
(Vrije Universiteit, Amsterdam)
Experiences with lightweight checks for mass-maintenance transformations

1 EPOSee: A Tool For Visualizing Software Evolution Patterns

Michael Burch
Saarland University
michael@cs.uni-sb.de

Stephan Diehl
Catholic University Eichstätt-Ingolstadt
diehl@acm.org

Peter Weißgerber
Catholic University Eichstätt-Ingolstadt
peter.weissgerber@ku-eichstaett.de

Software archives contain historical information about the development process of a software system. Using data mining techniques patterns can be extracted from these archives. In this paper we present our tool EPOSee [1] that allows to interactively explore these patterns. For this particular application domain, it extends standard visualization techniques for association rules [1] and sequence rules [2] to also show the hierarchical order of items. Clusters and outliers in the resulting visualizations provide interesting insights into the relation between the temporal development of the system and its static structure.

1.1 Introduction

During the life time of a software system many versions will be produced. Analyzing the source code of these versions, as well as documentation and other meta-information can reveal regularities and anomalies in the development process of the system at hand.

Industrial, as well as open source projects keep track of versions and changes using configuration management systems [3] like RCS and CVS. Other tools keep track of additional information, e.g. bug databases or e-mails. The information stored by a configuration management system and related tools is called a software archive. The software archive provides the history of a software system.

Previously we have used data mining to extract association rules from such archives to characterize the development process [4] or to support programmers [5]. In this paper we discuss the visualization techniques that we implemented to analyze association and sequence rules (which in the following we call software-evolution patterns) and show some kinds of insights that can be gained about the evolution of a software system by visualizing these patterns. To interactively explore the mining rules extracted from software archives we developed EPOSee (see Figure 1.1) which provides the following visualizations:

- Visualization of Association Rules
  - Pixelmap (overview, context)
  - 3D Bar Chart (of selected rules, focus)
- Visualization of Sequence Rules
  - Parallel Coordinate View (overview, context)
  - Decision Tree (overview, context)
  - 3D Branch View (of selected rules, focus)
  - Rule Detail Window (of selected rule, focus)
- Histogram (distribution of confidence and support)

In addition, rules can be filtered according to their support and confidence, searched for keywords, and various schemes can be used for color-coding. All visualizations shown in this paper have been produced with EPOSee.

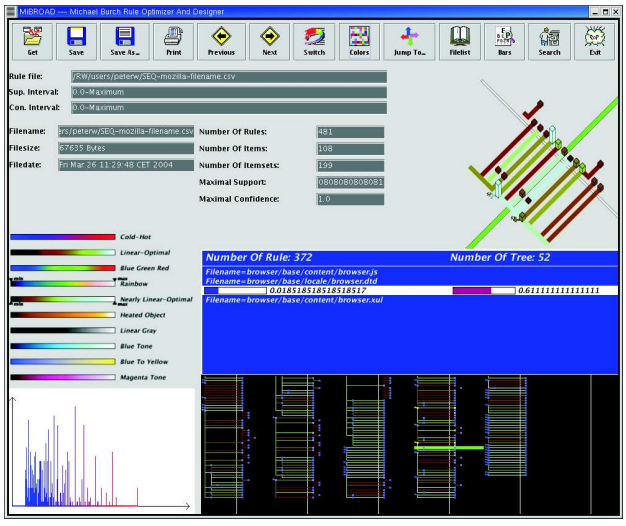


Figure 1.1: EPOSee

The items in the software-evolution patterns are software artifacts like files, classes, or methods for the case of software archives. In the visualization we use a total order derived from a hierarchy stemming from the application domain, e.g. methods are contained in classes, classes are

[1] Evolution Patterns of Software

contained in files, files are contained in directories, and directories are contained in other directories.

1.2 Visualizing Association Rules

To detect relations between items we first look at how often two items have been changed together, i.e. how often have they been checked into the software archive at the same time. As a result, we obtain a table $S : I \times I \to N$ of *change counts* where I is the set of items.

This table can be read as follows: Item i and item j have been changed together $S_{i,j}$ times. We call the matrix S the *support matrix* as it indicates how much evidence is there for each dependency. In particular, $S_{i,i}$ is the total number of times item i was changed.

Next we compute the strength of each dependency, i.e. the number of changes of a pair of items relative to the number of changes of a single item. As a result we get the *confidence matrix* C: $C_{i,j} = \frac{S_{i,j}}{S_{i,i}}$. Given a support matrix S, we can easily compute C by dividing every row by its element on the diagonal. In contrast to S, C is not symmetric.

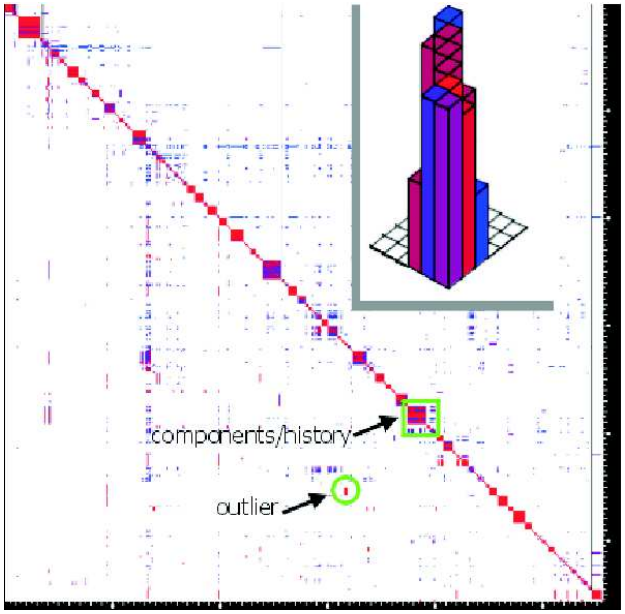


Figure 1.2: Pixelmap of the confidence matrix of MOZILLA

The pixelmap in Figure 1.2 shows the associations of the files in the `/browser` subdirectory of the CVS archive of the MOZILLA project. As the files are ordered hierarchically one can see that files which are next to each other, i.e. those that are in the same part of the hierarchy, are stronger related than others. Thus clusters typically extend along the diagonal of the pixelmap and very much correspond to the hierarchical structure of the system.

Software developers are mainly interested in the outliers, i.e. those pixels representing couplings between files in different directories. This kind of coupling, we call it evolutionary coupling, is based on the simultaneous changes of files rather than on one referencing the other. Outliers can be a sign of aspects orthogonal to the system hierarchy, but also a sign of a bad system architecture. In other words, if in the pixel map we do not find rectangular areas nicely aligned along the diagonal, then it might be a good idea to restructure the system.

The 3D bar chart shown at the corner of the pixelmap is a zoom of any part of the map and illustrates both support (height of the bar) and confidence (color of the bar) at the same time.

1.3 Visualizing Sequence Rules

Next, we would like to know in what temporal order changes typically occur. To this end we compute and visualize sequence rules. Both the antecedent and the consequent of a sequence rule are sequences of items. This gives the antecedent and consequent a time component.

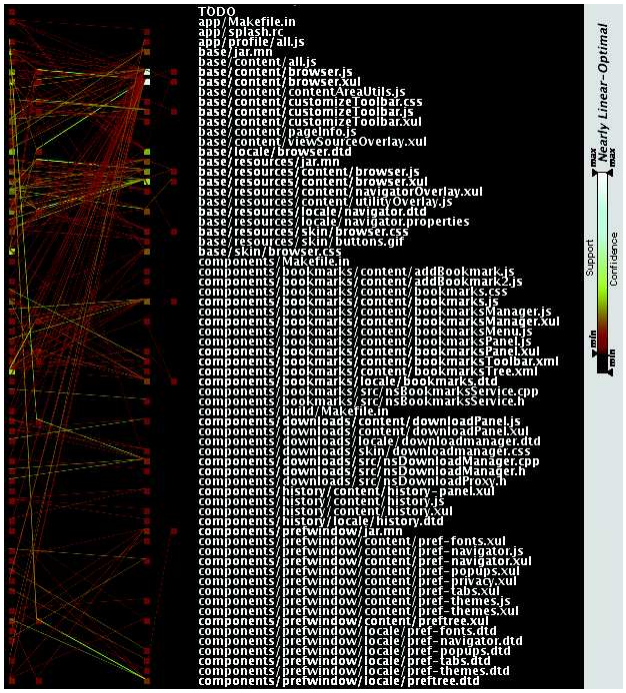


Figure 1.3: Parallel Coordinate View of MOZILLA

For example the sequence rule $a_1 \to a_2 \to a_3 \Rightarrow b_1 \to b_2$ means that if a_1 is changed before or at the same time as a_2 and a_2 before or at the same time as a_3, then it is likely that some time later b_1 and simultaneously or later b_2 will be changed. Figure 1.3 shows a parallel coordinate view of the `/browser` directory. Every sequence rule is displayed by connecting the node in the n-th column representing the n-th item in the sequence with the node in the $n+1$-th column representing the $n+1$-th item. The color of the nodes indicates the weighted sum of the support values of the subsequences ending at this node of all rules which share this node, while the color of the edges indicates the weighted sum of the confidences. As the nodes are ordered with respect to the hierarchical order of the items, we see multiple clusters consisting of many edges which only relate items in the same subdirectory. We also see that the files `base/content/browser.js` and `base/content/browser.xul` are related in a very

interesting way to almost all Javascript respectively XUL files: they are often changed after one of these other files has been changed.

In contrast to the parallel coordinate view in which one edge can belong to multiple rules, the decision tree visualization (see the bottom right corner of Figure 1.1) allows to have a deeper look at the *single* rules. Due to the color coding it is easily possible to find strong rules. Furthermore, one can see the structure of the rules, e.g. the length of the antecedents and consequents of the rule set, or the number of consequents for one given antecedent.

Bibliography

[1] R. Agrawal, T. Imielinski, and A. Swami. Mining association rules between sets of items in large databases. In *Proceedings of the ACM SIGMOD Conference on Management of Data*. 1993.

[2] R. Agrawal and R. Srikant. Mining sequential patterns. In *Eleventh International Conference on Data Engineering*.

[3] R. Conradi and B. Westfechtel. Version Models for Software Configuration Management. *ACM Computing Surveys*, 30(2), 1998.

[4] T. Zimmermann, S. Diehl, and A. Zeller. How history justifies system architecture (or not). In *Proc. International Workshop on Principles of Software Evolution (IWPSE 2003)*, 2003.

[5] T. Zimmermann, P. Weißgerber, S. Diehl, and A. Zeller. Mining version histories to guide software changes. In *Proceedings of International Conference on Software Engineering ICSE 2004*, 2004.

---

2 Data Mining Version Histories

Thomas Zimmermann
Andreas Zeller
Lehrstuhl für Softwaretechnik, Universität des Saarlandes, Saarbrücken, Germany
{tz,zeller}@acm.org

Abstract

Program analysis long has been understood as the analysis of source code alone. A modern software product, though, is more than just program code; it contains documentation, interface descriptions, resource data—all of which must be maintained and organized. In this paper, we propose a novel approach to maintain such non-program entities: By learning from the development history of the product, we can determine coupling between entities: "Programmers who changed *ComparePreferencePage.java* typically also changed *plugin.properties*". As a first proof of concept, our ROSE plug-in for ECLIPSE automatically guides the programmer along related changes.

2.1 Learning from History

Shopping for a book at Amazon.com, you may have come across a section that reads "Customers who bought this book also bought…", listing other books that were typically included in the same purchase. Such information is gathered by *data mining*— the automated extraction of hidden predictive information from large data sets. We have applied such data mining to the *version histories* of large open-source software systems. This results in rules like the following:

Coupling between entities: *"Programmers who changed the* `fkeys[]` *field always also changed the* `initDefaults()` *function"*. The `initDefaults()` function initializes new elements of the `fkeys[]` field; whenever `fkeys[]` was extended by a new element, `initDefaults()` was extended by a statement that initialized the element.

Coupling between programs and documentation: *"In 8 out of 10 cases, Programmers who changed the embedded SQL statement in line 47 of status.py changed the JPEG image igordb.jpg"*. The JPEG image is part of the product documentation and is a view of the database schema; whenever the schema changed, the SQL statements were changed, too, and the documentation was updated.

Such rules can reveal invariants of the development process (such as updating documentation); they can reveal factual coupling through common changes; and they can be put to use for actual programmers. Figure 2.1 shows our ROSE plug-in for the ECLIPSE programming environment, actually working on the ECLIPSE source code: As soon as the programmer makes a change to `fkeys[]`, ROSE suggests further related changes as listed above.

2.2 Mining Rules

Figure 2.2 shows the basic information flow through ROSE. The *ROSE Server* first extracts the *transactions* from the CVS archive—changes that were committed by the same

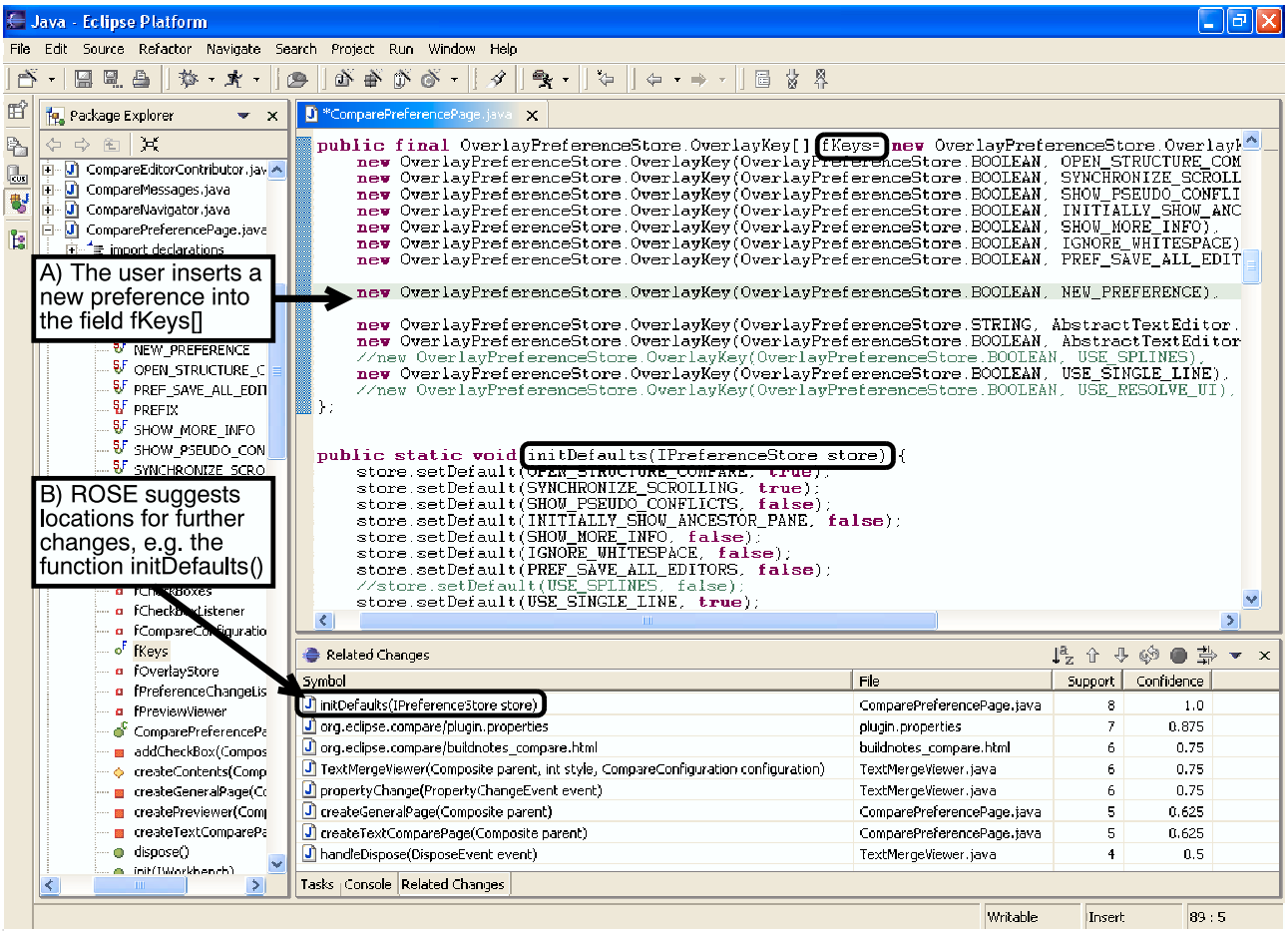


Figure 2.1: After the programmer has made some changes to the ECLIPSE source (above), ROSE suggests locations (below) where, in similar transactions in the past, further changes were made.

programmer with the same rationale in a short time window. The ROSE server then stores the transactions in a database. This representation is independent from the concrete version system used and can be used for arbitrary analyses. A unique feature of ROSE is that it maps the changes to *syntactic entities* such as methods, attributes, or sections [1]. ROSE is thus able to detect coupling at a much finer granularity than, say, files or directories.

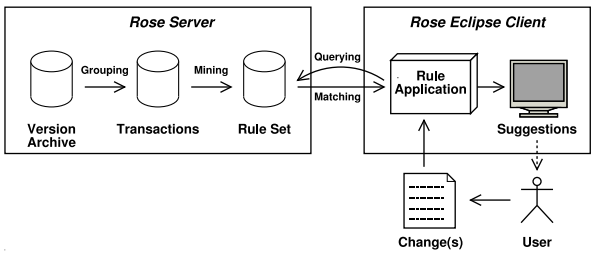


Figure 2.2: The data flow through ROSE.

The ROSE client makes the database accessible to programmers: As soon as the programmer makes a change, ROSE mines the database for possible related changes and presents these to the programmer in a list at the bottom of the screen (Figure 2.1). This is a very efficient process, taking at most 0.5 seconds; the programmer can examine these suggested locations simply by clicking on them.

Do ROSE's recommendations make sense? Yes and no. ROSE is not able to predict every single change. An evaluation on eight large open source projects [2] shows a recall of only about 15%, meaning that only a sixth of the actually changed entities could be predicted by ROSE. On the other hand, the recommendations have a high likelihood to be correct: the topmost three suggestions contain a correct location with a likelihood of 64%. Thus, if ROSE suggests something, it had better be taken seriously.

2.3 Some Perspectives

Our work with ROSE opens interesting new research perspectives in program analysis. Traditionally, program analysis has been concerned with source code alone. Leveraging the development history of the product obviously allows to reveal coupling that would otherwise be inaccessible to source code analysis—because we can detect coupling between programs and documentation, or between entities that are not even programs.

Yet, we have only begun to scratch the surface of what may be hidden in version archives and other process artifacts. In the future, we plan to exploit log messages and problem databases (which often are synchronized with changes); mailing lists or developer forums may be other sources of gathering knowledge. Of course, all of this information is fuzzy and insecure, especially when compared with the hard facts that source code analysis can extract. On the other hand, when it comes to understanding a program, a good hint may be better than no hint at all—and we're afraid that most of our programs are of such complexity that any hint may be precious.

More information about this and related work can be found on our web site

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

Bibliography

[1] Thomas Zimmermann and Peter Weißgerber. Preprocessing CVS data for fine-grained analysis. In *Proc. Intl. Workshop on Mining Software Repositories (MSR)*, Edinburgh, Scotland, May 2004.

[2] Thomas Zimmermann, Peter Weißgerber, Stephan Diehl, and Andreas Zeller. Mining version histories to guide software changes. In *Proc. 26th International Conference on Software Engineering (ICSE)*, Edinburgh, Scotland, May 2004.

---

3 M-Track: A Metric Tool Framework for Monitoring the Evolution of Object-Oriented Systems

Dharmalingam Ganesan
Jean-François Girard
Fraunhofer Institute for Experimental Software Engineering Sauerwiesen 6, 67661 Kaiserslautern, Germany
{ganesan,girard}@iese.fraunhofer.de

Abstract

This article reports about M-Track, a metric tool framework for tracking the evolution of Object-Oriented (OO) systems. It tracks the evolution using metrics reflecting cohesion, coupling, inheritance, and size. M-Track was applied for analyzing the evolution of a product family of systems in the domain of stock market.

3.1 Introduction

Maintaining the existing software effectively is an important activity for any organizations that develop systems that heavily depend on software. As software systems become older, changeability, understandability and testability start decreasing unless maintainers take active measures to prevent it.

One promising approach to control the maintenance of legacy OO software system is monitoring its evolution. It is well known that systems that adhere to the principles of low-coupling and high-cohesion are easier to maintain. But due to time to market pressure, these principles are often not followed, decreasing the maintainability of the system. On the other hand, without tools practitioners often only have a vague feeling about the degradation of coupling and cohesion between previous two releases. *M-Track helps addressing this problem.*

3.2 Applying M-Track

We use M-track's metrics to focus the attention of developers, designers and managers on the classes[1] with extreme metric values and extreme value changes. The idea is that such classes are more likely to cause problems than other. Many coupling and cohesion metrics have been proposed for OO systems in the literature. Many of them capture the same underlying concepts with more or less success depending on the system and its context. Because analyzing many partially redundant metrics requires too much effort from experts we select a subset of the available metrics. We used principal component analysis [Dun89] to select the subset of metrics that capture most of the underlying concepts.

3.3 Goals & Design of the M-Track Framework

The following are the major goals that drive us to the design the M-Track framework and the solutions we have selected.

- OO language Independent: The goal is to minimize the modification effort to apply M-track to a different oo language.
  This goal is achieved by decoupling the fact extraction from the metrics computation and by using a different fact extractor for each language. Each

[1] The metrics can also be applied at higher level of abstraction like namespace or packages.

fact extractor produces a standard fact representation (RSF) from which the metrics are computed.

- Extendable : M-Track should be extendable in order to introduce new metrics easily and also be able to customize the evolution analysis to support the needs of the analyst. We made M-Track easier to extend by separating the infrastructure needed by multiple metrics and by putting the specific part of each metric in its own module.
- Efficient and Scalable: The main target of M-Track is analyzing the evolution of large industrial systems. There scalability is the main concern.
  To achieve an efficient and scalable implementation, we applied three strategies. Firstly, we identified intermediate results needed by multiple metrics and made sure that they would be computed only once. Secondly, we preprocessed the relations in the fact based, so that indirect relations could be access directly. Thirdly we optimized the infrastructure to query the fact base.
- Portable: To offer short feedback cycle on the evolution and avoid delay, M-Track should run where the code is produced. Since different industry partners use different platform M-Track should be portable.
  To achieve this portability we implemented M-Track in Perl.
- Applicable to different level of granularity: In large industrial systems, analyzing coupling and cohesion metrics at different levels of abstraction helps to get a high-level overview, then to focus on the details of extreme cases.
  To support this analysis, we defined hierarchical version of the class metrics.

M-Track computes and visualizes OO metrics with the following steps.

1. Extract the containment information about files[2], classes and method as well the relations among them for the current and previous version of the system; then store these facts into RSF files.
2. Compute metric for both versions of the system.
3. Count the number of lines added, changed and deleted for each file appearing in both systems.
4. Prepare evolution report combining the metrics and the code change. Export the results as CSV format from which Microsoft Excel generate charts to provide an overview for the people analyzing the evolution.

3.4 Case Study and Lessons Learned

We applied M-Track to monitor the evolution of a product family from stock market domain implemented in Java [GVG04]. From the time where we started applying M-Track, we introduced new metrics and refined existing metrics to apply them at the package level. Our experience shows that new metrics can be easily introduced. Furthermore, metrics computation takes only few seconds for systems that contains more than 300 Kloc of code and with around 2000 classes in it.

One key lesson learned during this case study is that it is important to keep the presentation of the results flexible. As the developers, designers, and managers of a system perform multiple workshops using the metrics, they discover new ways to analyze the results. It is important to quickly adapt the reports and the visualization of the results to make the workshops most effective.

3.5 Conclusion and Future Work

This paper reports on a tool framework called M-Track, for monitoring the evolution of object-oriented systems. M-Track is customizable to the interest of industry partners for monitoring the evolution. This customizability was achieved by computing the metrics from a language-independent intermediate model, decoupling metrics computation from the metrics visualization, and also by implementing the entire framework in a portable language (Perl). Scalability is achieved by taking advantage of the commonality among the definition of the metrics itself.
This work will be extended in three directions in future.

1. Extending the M-Track to monitor dynamic aspects of the system.
2. Identifying relationships between static and dynamic measures.
3. Choosing an appropriate visualization approach for doing useful analysis on the static and dynamic measures at different level of abstraction.

References

[Dun89] G.Dunteman, "'Principal Component Analysis"', Sage Publication, 1989.

[GVG04] J.Girard, M.Verlage and D.Ganesan. "'Monitoring the Evolution of an OO system with Metrics: an Experience from the Stock Market Software Domain"', Submitted for publication

[2]For different languages higher abstraction are also captured (e.g. packages for Java, namespace for C++).

Praxis

4 Beleg Reengineering

Harry M. Sneed
Institut für Wirtschaftsinformatik an der Universität Regensburg
Harry.Sneed@t-online.de

Zusammenfassung

Dieser Beitrag zum Thema Software Reengineering befaßt sich mit dem Reengineering von alten Druckdateien. Das Ziel ist es, die Inhalte der bestehenden Druckdateien in XML Dokumente zwecks der weiteren Verarbeitung sowie zum Abgleich mit den neuen Druckausgaben zu vergleichen. Dabei werden die XML Schema Sprache und die XSLT Transformationssprache herangezogen. Das Tool zur Transformation der alten Belege in XML wurde im Rahmen eines Migrationsprojektes bei der Österreichischen Wirtschaftskammer entwickelt und eingesetzt, um die neuen Grundumlagen Aussendungen gegen die der alten Anwendung auf dem Host zu validieren.

Keywords: Reengineering, Belege, XML, XSD, XSLT

4.1 Hintergrund

Legacy-Software-Systeme produzieren bekanntlich viele Berichte bzw. Listen, die an einzelne Fachdienststellen versendet werden. Gerade im Öffentlichen Dienst sind Papierausgaben bzw. Belege sehr weit verbreitet. Viele dieser Belege, z.B. Steuerbescheide und Zahlungsaufforderungen, gelangen direkt an die Kunden. Es ist also keineswegs einfach, diese Art Benachrichtigung abzulösen.

Andererseits stehen Betriebe und Behörden unter Druck, ihre Informationstechnologie zu modernisieren. Auch die Papierbelege, die sie an ihre Anwender versenden, sollten durch andere Medien, z.B. E-Mails, ersetzt werden. Auch wenn die Belege weiterhin ausgedruckt werden, empfiehlt es sich, dies an einem lokalen Ort zu tun. D.h., die Beleginhalte werden elektronisch verschickt und erst an einer dezentralen Stelle ausgedruckt. Dies hat den zusätzlichen Vorteil, daß das Belegformat lokalisiert werden kann. Es bekommt nicht jeder den gleichen Beleg, sondern einen für ihn angepaßten Beleg.

Um zu diesem höheren Grad an Flexibilität zu gelangen, müssen die bisherigen Druckausgaben in eine andere Form versetzt werden. Dies kann auf zweierlei Weise geschehen. Entweder werden die alten Programme umgeschrieben, um das neue Ausgabeformat zu produzieren, oder die alten Ausgaben werden abgefangen und in das neue Format umgesetzt. Im ersten Fall handelt es sich um Program-Reengineering, im zweiten Fall um Data-Reengineering.[1]

Es besteht demzufolge ein dringender Bedarf, alte Druckbelege in neue Formate zu versetzen und zwar möglichst vollautomatisch. [2]

4.2 Problematik

Die Schwierigkeit beim Reengineering von Computerausdrucken liegt in deren Unregelmäßigkeit. Im Gegensatz zu Programmen, die eine fest definierte Grammatik haben, haben Belege keine Standardgrammatik. Sie können jede beliebige Form annehmen. Physikalisch gesehen gibt es Seiten, Zeilen und Spalten. Logisch gesehen gibt es Literale, Variable und Füllfelder.

Eine physikalische Seite besteht aus n Zeilen mal m Spalten. Die Anzahl Spalten bzw. die Seitenbreite ist in der Regel fix. Alte Mainframe-Berichte hatten oft eine Zeilenlänge von 133 Zeichen, weil dies der Druckerzeilenlänge von 132 plus einem Drucksteuerungszeichen entsprach. Neuere Berichte haben Zeilen mit einer Länge bis zu 160 Zeichen. Die Länge einer Seite hängt ebenfalls vom Zieldrucker ab. Je nach Druckerart könnte sie von 60 bis 100 Zeilen lang sein. Endlose Listen haben keine Seiteneinteilung, sondern nur eine endlose Anzahl Zeilen. Die Zeilen haben aber dann ein festes Format.

Wenn es darauf ankommt, Belege zu beschreiben, muß als erstes das physikalische Format festgelegt werden, d.h. die Zeilenlänge und die Anzahl Zeilen pro Blatt. Damit wird der Rahmen für die weitere Verarbeitung des Beleges gesetzt.

Ein Beleg, bzw. seine Zeilen, enthält außer Leerraum (Spaces) Titelfelder, Datenfelder und Füllfelder. Füllfelder mit irgendwelchen Sonderzeichen, die das Listenlayout ansehnlicher gestalten sollen, tragen zum Reverse Engineering eines Beleges nicht bei, es sei denn, um davor- oder danach stehende Datenwerte zu lokalisieren. Titel sind wichtig, um die Datenwerte zu identifizieren. Sie können als Tags verwendet werden. In Listen mit festen Datenspalten stehen die Spalten Titel in einer oder mehreren übergeordneten Zeilen, die leicht erkennbar sind. Sie können dort entnommen werden, um die Spalten zu identifizieren. In Listen mit festem Format stehen die Titel an bestimmten Stellen auf jedem Blatt. Sie können anhand ihrer Position identifiziert werden.

In frei formatierten Belegen können die Titel an einer beliebigen Stelle stehen. Weder die Zeile noch die Spalte ist bestimmt. In diesem Falle können die Titel nur anhand

ihrer Bezeichnung erkannt werden. Dies setzt allerdings voraus, daß die Bezeichnung ein einheitliches Muster hat. Da das fast immer der Fall ist, gibt es kaum Belege, die sich nicht interpretieren lassen.

Was für die Titel zutrifft, trifft ebenso für die Datenwerte zu. In Listen mit festen Spalten werden sie immer eine feste Position haben, z.B. die Spalte 21. In fest formatierten Belegen werden sie zwar immer an einer bestimmten Spalte beginnen, aber die Zeile ist variabel. D.h., es wird nötig sein, die Zeile anhand gewisser Titel oder Füllfelder zu erkennen, was auch ohne weiteres möglich ist.

In frei formatierten Belegen ist die Identifizierung der Datenwerte nur relativ zum Titel und den Füllfeldern möglich. D.h., es werden zuerst bestimmte Titeltexte und/oder Füllfelder identifiziert. Dann läßt sich, ausgehend von dieser Position, entweder davor, danach oder darunter der Datenwert erkennen. Diese relative Erkennung ist zwar schwierig zu beschreiben, dennoch durchaus möglich.

Schließlich gibt es Werte, die in Vektoren vorkommen. Hier wird eine unbestimmte Anzahl Werte des gleichen Datentyps hintereinander aufgelistet. Das Problem hier ist, zu erkennen, welcher der letzte Datenwert in der Reihe ist. Dies kann nur über eine Typ Überprüfung geschehen. Das soll verhindern, daß der nächste Datenwert, der eine völlig andere Bedeutung hat, als letzter Datenwert des Vektors angenommen wird. Hierfür braucht man ein Unterscheidungskriterium.

4.3 Lösung

4.3.1 XML-Schema als Beschreibungsformat

Entschieden wurde, die XML Schemasprache zu verwenden, um die Belege zu beschreiben. Die Schemasprache hat schon eingebaute Attribute, die auch zu den Belegfeldern passen z.B. der Feldtyp “String” , das Attribut Name für den Feldbezeichner, und das “occurs” Attribut für wiederholte Felder und Feldgruppen. Es fehlen allerdings Attribute, um die genauen Koordinaten eines Belegfeldes zu spezifizieren. Deshalb wurde die Schemasprache hier um zwei zusätzliche Attribute ergänzt

Line und
Col.

Line identifiziert die Zeile, in der ein Feld vorkommt, und Col identifiziert die Spalte, in der das Druckfeld beginnt. Falls das Feld in einer beliebigen Zeile vorkommt, dann heißt das row- Attribut “any”. [3]

Mit der XML-Schemasprache, ergänzt durch die zusätzlichen Attribute, ist es möglich fast alle bestehenden Belege zu spezifizieren und somit auch alle Felder in den Belegen zu identifizieren. Die XML-Schemabeschreibung ist die Grundlage für die Transformation der Belege in XMLDokumente.

4.3.2 XSLT Transformationssprache zur Identifikation der Felder

Für die Identifikation der Druckfelder wird die Style Sheet Sprache XSLT herangezogen. [4] Normalerweise ist XSLT gedacht, XML Dokumente oder XSL Style Sheets in andere Zieldokumente umzusetzen, z,.B. in HTML, CSS oder XHTML. Das heißt, XSLT ist konstruiert um XML, bzw XSL zu interpretieren. Hier wurde die Sprache ergänzt um einfache Text Eingaben zu verarbeiten. Das wesentliche dabei ist die Einführung einer String Verarbeitung. Strings im Text werden an Hand ihrer Anfangsposition “Col” und ihrer Länge erkannt. Die Test Anweisung

```
Test = '(col[30:7] == "Gebuehr")'
```

bedeutet, die Zeichenfolge ab Spalte 30 in der Länge 7 wird mit dem Literal “Gebuehr” verglichen. Bei Gleichheit ist der Test `true`, sonst ist er `false`.

Die Werte werden ebenfalls anhand ihrer Position und Länge herausgeschnitten. Hierzu wird die XSLT copy Anweisung verwendet:

```
<xsl:copy Zahlung = 'col[40:10]'>
```

Dadurch wird die Zeichenfolge in der jeweiligen Zeile ab Spalte 40 in der Länge in das XML Datenelement namens Zahlung heraus kopiert.

Diese String Operationen werden durch zwei weitere nützliche Attribute ergänzt:

Delimited by <Zeichen> und
Prelimited by <Zeichen>

Die delimited by Klausis beended eine String Copy an dem genannten Zeichen, z.B. hier an dem ersten Leerzeichen.

```
<xls:copy
Payment = 'col[40:8] delimited by " "'>
</xsl:copy>
```

Mit dieser Anweisung wird die Zeichenfolge ab Spalte 40 in der Länge 8 heraus kopiert bis zum ersten Leerzeichen.

Die prelimited by Klausis startet eine String Copy nach dem genannten Zeichen, z.B. hier nach dem ersten Komma.

```
<xls:copy
Payment = 'col[40:16] prelimited by ","'>
</xsl:copy>
```

Mit dieser Anweisung wird die Zeichenfolge ab Spalte 40 bis zur Länge 16 nach dem ersten Comma heraus kopiert.

In der Regel können Felder in Druckzeilen nur über eine Abfrage bestimmter Textelemente erkannt werden, d.h., die eigentlichen Werte sind nur über ihre Titel erkennbar. Zur Auswahl von Felder in einer Textdatei werden drei XSLT Anweisungen benutzt:

xsl:if,
xsl:choose und
xsl:for-each

Wenn es um eine wahr/falsch-Entscheidung geht, wird eine *if*-Abfrage verwendet. Wenn es um eine Auswahl nach alternativen Textfragmenten geht, wird eine *choose*-Anweisung benutzt. Die *for each*-Schleifenanweisung

wird dann herangezogen, wenn Felder bzw. Feldgruppen mehrfach, aber in einer unbestimmten Anzahl, auf der gleichen Zeile vorkommen, z.B. wenn ein Kunde mehrere Adressen hat.

4.3.3 Belegtransformationsprozeß

Sind die Belege einmal mit einem XML-Schema beschrieben und deren Transformation mit einer XSLT Spezifikation spezifiziert, ist der Rest voll automatisiert. Das Werkzeug "Repo2XML" kann in einem Lauf mehrere Belege umsetzen. Für jeden Beleg wird zuerst das XML-Schema zu diesem Beleg geholt und geparst. Daraus wird eine interne Datenbeschreibungstabelle für den Beleg generiert. Als zweites wird die XSLT Spezifikation gelesen und eine Umsetzungsprozedur daraus generiert. Dann wird die entsprechende Druckdatei von der generierten Prozedur gelesen und Zeile für Zeile, Feld für Feld in eine XML-Datei umgesetzt.

Die Feldgruppen bilden die XML-Objekte, die auch im Schema beschrieben sind, z.B. Mitglieder, Forderungen und Kammer. Jedes Objekt wird zunächst in einer eigenen Zwischendatei gespeichert, in der pro Instanz eine XML-Datengruppe gebildet wird. Anschließend werden alle Objekte zusammen mit dem Schema in einer einzigen XML-Datei mit dem neuen Schema am Anfang zusammengeführt und sortiert. Das Ergebnis ist eine XML-Datei mit einem Schema als Kopf und n mal m Objekten als Rumpf. Das Schema wird von einer Standardschemadatei entnommen.

4.4 Test der XML Dateien

Ein Test ist stets ein Test gegen etwas. [5] Bei Entwicklungsprojekten wird gegen die Anforderungsspezifikation getestet. Bei Migrationsprojekten wird gegen das alte System getestet. Mit der Belegtransformation durch Repo2Xml wird die Möglichkeit geschaffen, die neuen XML-Dateien mit den alten Druckdateien abzugleichen, denn beide sind jetzt im gleichen Format. Das Werkzeug TestComp liest die alte XML-Datei und speichert die Feldinhalte in einer SQL-Datenbank. Anschließend liest es die neue XML-Datei und paart die Instanzen aufgrund der spezifizierten Suchbegriffe. Entspricht eine Instanz der neuen XML-Datei einer Instanz in der alten XML-Datei, werden die Werte aller Attribute verglichen und die nicht passenden Attribute ausgewiesen. Ist eine neue Instanz in der alten XML-Datei nicht zu finden, gilt dies als Ergänzung.

4.5 Erfahrung mit Beleg Reengineering

Die Methode des Beleg-Reengineering, die in diesem Beitrag beschrieben wurde, ist in einem Migrationsprojekt für die Österreichische Wirtschaftskammer entstanden. Dort war es erforderlich, ein neues .NET-Anwendungssystem gegen das alte CICS-System auf dem Host zu testen. Dazu gehörte neben dem Abgleich der SQLServer- Datenbanktabellen mit den VSAM-Dateien auch der Abgleich der neuen XML-Ausgabedateien mit den alten Druckdateien vom Host. Dies war der Anlaß, die bisherigen Druckdateien in XML umzusetzen.

Insgesamt wurden 5 verschiedene Belegtypen in XML umgesetzt

- die Aussendungen,
- die Mahnungen,
- die Exekutionen,
- die Lohndaten und
- die Buchungsbelege.

Dies ist in allen Fällen mit dem Tool Repo2XML gelungen trotz einzelner Unterschiede zwischen den Ländern. Die generierten XML-Dokumente konnten ohne weiteres mit den neuen XMLDokumenten verglichen werden. Auf diese Weise wurden einige Fehler in den neuen Ausgaben aufgedeckt, die sonst unbemerkt geblieben wären. Die Umsetzung hat sich also gelohnt.

Literaturhinweise

[1] Aiken, P.: Data Reverse Engineering, Addison- Wesley, Reading, MA., 1998.

[2] Sneed, H.: Objektorientierte Softwaremigration, Addison-Wesley, Bonn, 1999.

[3] Fitzgerald, M.: Building B2B Applications with XML, John Wiley & Sons, New York, 2001.

[4] Box, D./Skonnard, A./Lam, J.: Essential XML, Addison-Wesley, München, 2001, S. 179.

[5] Sneed, H./Winter, M.: Test objektorientierter Software, Hanser Verlag, München, 2001.

5 Genauigkeit von Aufwandsschätzungen in Reengineering-Projekten am Beispiel einer großen Sprachumstellung von Assembler nach COBOL

Jens Borchers
CC GmbH Wiesbaden
jens.borchers@caseconsult.com

5.1 Einführung

Die SPARDAT, die ausgelagerte IT-Service-Organisation der österreichischen Sparkassen, hat sich im Jahre 2002 entschieden, ihre bestehende Wertpapiertransaktionsverarbeitung technisch grundlegend zu renovieren. Als ein Teil dieser umfassenden Renovierung wurde auch die Umstellung von ca. 400 mehr oder weniger komplexen /370 Assembler-Programmen (unter IBM z/OS) erforderlich. Als Zielsprache wurde dabei COBOL (auf Basis des ANSI-Standards von 1985) definiert. Es soll an dieser Stelle nicht weiter darauf eingegangen werden, warum diese Zielsprache - und nicht modernere gewählt wurde, gute Argumente dafür finden sich aber in z.B. [1]. In [2] sind die üblichen Problemfelder eine Assembler-COBOL-Umstellung dargestellt, der dort ebenfalls propagierte Ansatz unter Nutzung der Spezikationssprache WSL hat sich aber für das vorliegende Projekt aber als nicht anwendbar erwiesen; es wurde mit einem „direkten“ Assembler-COBOL-Converter gearbeitet.

Bei dieser Sprachumstellung handelt es sich also um ein gut definiertes und verhältnismäßig homogenes Reengineering-Projekt, das sich daher auch gut abschätzen lassen sollte. Im weiteren Beitrag wird kurz dargestellt, was überhaupt alles zu schätzen war und wie die bisherigen Erfahrungen im Verhältnis zum tatsächlichen Aufwand stehen. Das Projekt steht kurz vor dem Abschluß, alle Zahlenwerke liegen daher im Moment noch nicht vor, werden aber dann Gegenstand weiterer Untersuchungen sein.

5.2 Die Projektaufgabe

Bei dem Reengineering-Projekt handelt es sich um eine reine Sprachumstellung, d.h. alle anderen technischen Randbedingungen wie Dateien, Datenbanken, Oberflächen bleiben unverändert. Einzige weitere Änderung ist die Ersetzung von Aufrufen der alten Assembler-orientierten Zugriffsroutinen durch die entsprechenden für COBOL. Außerdem sind alle kurzen Assembler-Namen durch entsprechende COBOL-Namen zu ersetzen (was nicht so trivial ist, wie es sich auf den ersten Blick vielleicht darstellt).

Der Umfang des Projekts stellt sich wie folgt dar:

- 400 Assembler-Programme mit ca. 400.000 Lines of Code
- weit über 1.000 genutzte Macros und Copybooks, über 1 Million LoC

Wie bereits oben beschrieben, werden im Rahmen des Projekts ausschließlich technische Anpassungen durchgeführt, d.h. die betriebswirtschaftliche Funktionalität bleibt unverändert. Dieser Nachweis ist durch entsprechende Regressionstests zu führen. Da es sich um eine Anwendung im Bankenbereich handelt, in der praktisch jedes zweite Feld einen Geldwert repräsentiert, sind die Anforderungen an die Testabdeckung entsprechend hoch. In den Originalprogramme erkannte Fehler, die (nur) im Rahmen jeder Umstellung ans Licht kommen, wurden gesondert behandelt.

5.3 Die Projektabwicklung

Das Projekt wurde nach dem von CC entwickelten „Factory“-Ansatz für Reengineering-Projekte abgewickelt, und zwar mit Einsatz von Offshore-Ressourcen. Dieser Ansatz ist ausführlich in [3]und [4] dargestellt und hat sich seitdem mehrfach bewährt und wurde weiter optimiert.

Dabei sind folgenden Haupt-Phasen und -Aufgaben - auch im Sinne einer Aufwandsschätzung relevant:

- Setup-Phase
  - Aufbau der Projektinfrastruktur (Konfigurations-Management, Testumgebungen)
  - Erstellen des Umstellungs-"Kochbuchs", Anpassen der Conversion Tools
- Umstellung eines ersten Pakets als Pilot, Optimierung der Abläufe und Regelwerke
- Paketorientierte Umstellung der Anwendungsprogramme, insgesamt wurde 7 Pakete (+Pilot) definiert
  - Eigentliche maschinelle und manuelle Konversion der Programme und zugehörigen Komponenten
  - Durchführung von Referenzläufen mit Original-Programmen (ungenau immer als Referenz-Tests"bezeichnet) zur Gewinnung von Testdaten
  - Durchführung der Regressions-Tests, bis zum Nachweis der identischen Funktionalität
  - Statische und dynamische (primär Testabdeckungsgrad) Qualitätsmessungen
- Übergabe der neuen Programme g zu normalen Integrationstests und Produktionsaufnahme

5.4 Die Projektschätzansätze

Für die Schätzung von Software-Reengineering-Projekten gibt es diverse Ansätze, die primär auf dem Umfang und der Komplexität der zu bearbeitenden Komponenten basieren (vgl. z.B. [5]). Prominentester Vertreter ist ein abgewandeltes COCOMO II-Verfahren, welches auch von

Sneed beschrieben wurde und bei dem die Zahl der Einflußparameter von den ursprünglich 100 auf 12 reduziert wurde. Trotzdem können auch diese 12 Faktoren die Schätzungen noch signifikante Größenordnungen beeinflussen.

Es ist deshalb in [6] vorgeschlagen worden, während der Durchführung eines Reengineering-Projekts nach jedem abgeschlossenen Arbeitspaket eine Rekalibrierung des Schätzansatzes vorzunehmen. Dieses hat sich auch in diesem Projekt als sinnvoller Ansatz erwiesen, wenngleich es nach unserer Einschätzung nicht mit dem dort vorgeschlagenen und verhältnismäßig aufwendigen Verfahren passieren muß.

Neben den Schätzverfahren, die auf Eigenschaften der umzustellenden Komponenten abheben, gibt es natürlich immer den rein heuristischen Ansatz, bei dem auf der Basis von Erfahrungswerten mit vereinfachten Formeln gerechnet werden kann. Eine entsprechende Erfahrungsbasis vorausgesetzt, kann auch dieser Ansatz zu Schätzungen führen, die nicht wesentlich ungenauer als die oben angesprochenen sind.

Es ist außerdem festzustellen, daß die komponentenorientierten Schätzungen andere wesentliche Blöcke aus dem im vorhergehenden Abschnitt beschriebenen Hauptaufgaben gar nicht berücksichtigen können. So ist z.B. die Erstellung von Referenzdaten in der Praxis von ganz anderen Parametern abhängig als man sie sich z.B. vorstellen könnte (z.B. Zahl der in einer Komponente genutzten Ein-/Ausgabebestände). Hier haben sich „rule of thumbs“ („wir brauchen x Projekttage für ein Batchprogramm und y Projekttage für ein Onlineprogramm“) als völlig ausreichend erwiesen. Sie spiegeln auch keine „Pseudogenauigkeit“ vor wie andere Ansätze.

Der Aufwand für die Regressionstests kann im allgemeinen proportional zum Aufwands für die eigentliche Komponentenbearbeitung geschätzt werden. Detailliertere Schätzansätze (z.B. mit Berücksichtigung der Logik-Komplexität der Zielprogramme) erscheinen nur vordergründig genauer.

Im vorliegenden Projekt wurde folgende Schätzansätze kombiniert:

- Eine detailliertes, toolbasiertes Assessment der Assemblerprogramme, um eine Einordnung in Komplexitätsklassen zu ermöglichen
- Heuristische Schätzung auf Basis der Anzahl, des Umfangs und der Komplexitätsklasse der umzustellenden Komponenten; hier konnte das Projekt auch aus der Erfahrung von 4 vorhergehenden Umstellungen gleicher oder zumindest sehr ähnlicher Art (alle Assembler nach COBOL) schöpfen
- Abschätzung von Referenz- und Regressionstest als abgeleitete Größen der obigen Basisschätzungen
- „Normale“ Schätzansätze für alle Aktivitäten, die primär von der Projektlaufzeit abhängig sind (wie z.B. Problem- und Konfigurationsmanagement) bzw. einmalige Aufgabenblöcke (wie z.B. Setup) darstellen

5.5 Die Projekterfahrungen mit der Schätzung

Das Projekt hat eine Gesamtlaufzeit von 16 Monaten, von denen ca. 3 Monate für die Setup-Phase genutzt wurden und ca. 1 Jahr für die eigentliche paketorientierte Umstellung. Im Mittel waren 20 Mitarbeiter mit dem Projekt beschäftigt, die sich auf drei Standorte (Kunde, CC Deutschland, CC India) verteilt haben.

Von Kundenseite wurden primär über die gesamte Projektlaufzeit die Bereiche Referenzdatenerstellung (2 Mitarbeiter + ztw. zusätzliche Experten) und Problembearbeitung (unklare Umsetzungsregel für spezielle Komponenten, Fehler im Originalprogramm etc.) abgewickelt.

Insgesamt haben sich die ursprünglichen Schätzungen bis zum aktuellen Stand des Projekts als sehr gut erwiesen und liegen deutlich unterhalb der Grenzen, die z.B. in [6] als noch tolerierbare Abweichungen genannt werden. Es wurde jeweils nach Abwicklung der einzelnen Pakete eine Rekalibrierung der Produktivität (basierend auf der Umsetzung von jeweils 500 LoC Assembler) vorgenommen. Diese hat erwartungsgemäß die Schätzgenauigkeit für die letzten Pakete verbessert.

Es hat sich gezeigt, daß Erfahrungswerte und darauf basierende vereinfachte Schätzansätze durchaus mit aufwendigeren Ansätzen mithalten können.

Ganz ausschließen kann man Ausreißer nach oben oder unten ohnehin nie, da sich bestimmte Komplexitäten in bestehenden Programmen erst dann zeigen, wenn man sie konkret bearbeitet. Problematisch ist dabei, daß die üblichen Analysetools derartige Problemzonen nicht finden können und sich selbst die Entwickler dieser Programme dieser nicht mehr bewußt sind.

Außerdem beeinflussen andere Projektereignisse wie z.B. eine zunächst nicht geplante Überarbeitung des Conversion Tools oder überraschende Abgänge von erfahrenden Projektmitarbeitern auch die beste Schätzung.

5.6 Das weiteres Vorgehen

Wie bereits oben angemerkt, befindet sich das Projekt derzeit in seiner letzten Phase. Während dieses Projekts sind detaillierte Zahlen für alle möglichen Einflußfaktoren und natürlich auch die tatsächlichen Aufwendungen (bis auf Programmebene) erfaßt worden.

Wir werden (im Sinne von CMMI 4) diese statistisch nach allen Gesichtspunkten auswerten und versuchen, eine Korrelation herzustellen. Inwieweit das wirklich gelingt, wir sich zeigen.

Literatur

[1] Terekhov, C. Verhoef; *The Realities of Language Conversions*, IEEE Software, November/December 2000

[2] M.P. Ward; *The FermaT Assembler Reengineering Workbench*, ICSM 2001, Florenz, 6.-9. November 2001

[3] J. Borchers, *Erfahrungen mit dem Einsatz einer Reengineering Factory in einem großen Umstellungsprojekt*, HMD Nr. 194, März 1997

[4] J. Borchers, *Software Evolution Enabling - IT-Sicherung auf Basis bestehender Systeme*, 4. Workshop Software-Reengineering, Bad Honnef, 29.-30. April 2002

[5] Andrea De Lucia et. al.,*Empirical Analysis of Massive Maintenance Processes,* CSMR 2002, Budapest, März 2002

[6] M.T. Baldessarre, D. Caivano, G. Vissagio; *Software Renewal Estimation Using Dynamic Calibration*, ICSM 2003, Amsterdam, 22.-26. September 2003

ch gesichertes Zahlenmaterial vorgelegt werden kann.

---

6 E-CARES research project: Interactive, stakeholder-tailored re-engineering

Simon Giesecke
Dept. of Computer Science III, RWTH Aachen University, 52056 Aachen, Germany
giesecke@i3.informatik.rwth-aachen.de

André Marburger
Dept. of Computer Science III, RWTH Aachen University, 52056 Aachen, Germany
marand@cs.rwth-aachen.de

6.1 Introduction

The E-CARES project, which was initiated in 1999, is committed to researching reverse and re-engineering approaches of telecommunications software systems based on a graph transformation infrastructure. Ericsson's AXE10 mobile-services switching center serves as a case study during the project. The progress of the E-CARES research project has been presented to this forum in the previous years. In this paper, we first briefly present the current status of the project (Section 6.2), in particular the developments in the last two years. Afterwards, we point to several ideas for future directions of the project in Section 6.3, particularly the use of reverse engineering repositories for communication, explorative re-engineering, and improved interoperability of reverse and re-engineering tool suites.

6.2 Current state of the project

First results from state machine extraction, which was proposed in [5], have been obtained by analyzing more than two million lines of PLEX code spread over more than a hundred compilation units (blocks). These results show that certain assumptions regarding the structure of the state machines are often violated, so that we had to partly revise our extraction algorithm.

Further progress has been achieved concerning static link chain analysis exploiting combinations of static and dynamic analyses, as well as visualizing signal traces [6].

In the beginning of the E-CARES project, only the PLEX programming language was supported. In the meantime, support for the C programming language was added. As part of the process of adding C support, the whole prototype was restructured, so that it is now possible to support a variety of languages with little effort. If one accepts that language-specific features will be neglected, it is possible to add support for a new language without changing the underlying graph schema, as long as it adheres to an imperative paradigm. It is then only necessary to provide a parser generating compatible output.

In the near future, Ericsson's experts themselves will be able to use and evaluate the tool at their site. We expect improved feedback on the adequacy of the features provided by the E-CARES prototype.

6.3 Future directions

It would be desirable to complete the existing reverse engineering tool suite towards a re-engineering tool suite, as already planned in the starting phase of E-CARES. However, in order to gain new insights into the problems of re-engineering, an approach is taken that is in some way substantially different from the numerous previous approaches. Being built by means of the high-level graph transformation language PROGRES [7], the existing prototype stands out from the majority of reverse engineering tools in the dimension of the underlying specification language. It is interesting in itself to investigate the advantages and disadvantages of this approach when extending the tools towards re-engineering support. Other specifics of the intended approach are presented below.

Communication through reverse engineering tools. There are approaches to software architecture focusing on the use of architectural descriptions as a means of communication between different stake-holders [2]. Different stake-holders have differing background knowledge, and different interests in the software system and its description documents. Similarly, the descriptions of a software system regained by reverse engineering techniques can be used for communication. Reverse engineered descriptions

are commonly not at a genuine architectural level. In many organizational contexts, however, they might be the only descriptions that are apt to be used, since original architectural descriptions might not be available at all or too outdated to be useful. Communication through these descriptions might be realized with less intrusion to an organization's processes than that caused by direct migration to an architecture-centered process. Thus, this approach might have more potential for real impact on industrial practice.

One possible usage scenario of communication through reverse engineering tools involves testers and system architects as stake-holders. Testers are able to provide typical signal traces, and may use the reverse engineering system themselves to visualize the traces in the context of the actual system structure. System architects may use these traces for identifying communication hot spots, which may help in deciding where to split a subsystem, for example. If this kind of information is placed in the reverse engineering repository, it may be easily used by all relevant stake-holders.

Interactive, explorative re-engineering. Based on our reverse engineering tool prototype, we plan to investigate methods for interactive and explorative re-engineering in the domain of telecommunications systems. It is difficult and not sustainable to provide a tool pursuing a fixed re-engineering method, which can be applied to a software system once. Instead, small re-engineering steps may be performed throughout time. These steps may correspond to recurring tasks, but probably there will be substantial variations in the experts' requirements. Therefore, we want to give domain experts a tool at hand, using which they can create re-engineering methods tailored for specific situations. These may not apply to the system as a whole, but to multiple parts of the system. Thus, it is desirable that a re-engineering method may be recorded using one example, reworked into a general set of transformation rules and replayed at other parts of the system. It will have to be examined how a specialization of the PROGRES language, a specific method of use of PROGRES, or a language built on top of PROGRES will be helpful in guiding this process. In the latter case, atomic PROGRES transformations must be identified. It is obvious that the less effort is put into manual reworking, the less general the resulting method will be. Here, it is interesting to find a balance which makes it possible for domain engineers to use the tool after an acceptably short learning period, and which is expressive enough to be of sustainable utility.

An explorative approach to re-engineering has been taken by Jahnke et al. [4]. However, their approach is concerned with database re-engineering, which is related to, but substantially different from re-engineering software, let alone process-centered software systems. They concentrate on the integration of external changes in the subject system. On the other hand, we want to consider the exploration of different re-engineering approaches to a given subject system.

Re-engineering frameworks interoperability. In order to reduce the tendency of re-implementation of functionally equivalent components of re-engineering frameworks, interoperability between different frameworks should be improved [1]. Such efforts have already been pursued in the re-engineering community (e.g. [3]). The use of a common repository interchange format, like GXL [8], provides a starting point for such work, but is not sufficient for most uses. For example, a query language like GReQL or Grok [3] would be a useful facility in the context of E-CARES. The PROGRES language could be seen as a query language as well, but we consider it too generic and complex to be used by a re-engineer. On the other hand, other re-engineering frameworks (e.g. the GUPRO project) could benefit from the interactive visualization and processing framework provided by the E-CARES prototype. It should be possible to exchange parsers between the frameworks, which requires not only a common interchange format, but also a common ground of its semantics and pragmatics. Integration of a language like C++ into the E-CARES graph schema would be feasible, but writing an adequate parser would require a major effort.

Acknowledgments

The work presented here has been generously supported in part by Ericsson Eurolab Deutschland GmbH (EED) and the Deutsche Forschungsgemeinschaft (GK 643).

Literaturverzeichnis

[1] S. Ducasse and S. Tichelaar. Dimensions of reengineering environment infrastructures. *Journal of Software Maintenance*, 15(5):345–373, 2003.

[2] Hasso-Plattner-Institut Potsdam. Initiative "Kommunikation in der Software-Entwicklung". `http://fmc.hpi.uni-potsdam.de/index.php?cat=research&subcat=KommSE/overview-german`. visited 2004-03-23.

[3] R. C. Holt, A. Winter, and J. Wu. Towards a Common Query Language for Reverse Engineering. Fachberichte Informatik 8–2002, Universität Koblenz-Landau, Universität Koblenz-Landau, Institut für Informatik, Rheinau 1, D-56075 Koblenz, 2002.

[4] J. H. Jahnke, W. Schäfer, J. P. Wadsack, and A. Zündorf. Supporting iterations in exploratory database reengineering processes. *Sci. Comput. Program.*, 45(2-3):99–136, 2002.

[5] A. Marburger and D. Herzberg. E-CARES Research Project: Extraction of State Machines from PLEX Code. In *Proc. 4th Workshop Software Reengineering*, pages 21–23. Fachberichte Informatik Universität Koblenz: Koblenz, Germany, 2002.

[6] A. Marburger and B. Westfechtel. Tools for Understanding the Behavior of Telecommunication Systems. In *Proc. 25th Intl. Conference on Software Engineering (ICSE 2003)*, pages 430–441, Portland, Oregon, USA, May 2003. IEEE Computer Society: Los Alamitos CA, USA.

[7] A. Schürr. Programmed graph replacement systems. In G. Rozenberg, editor, *Handbook of Graph Grammars and Computing by Graph Transformation. Vol. I: Foundations*, chapter 7, pages 479–546. World Scientific, 1997.

[8] A. Winter, B. Kullbach, and V. Riediger. An overview of the GXL graph exchange language. In S. Diehl, editor, *Software Visualization: International Seminar*, number 2269 in Lecture Notes in Computer Science, pages 324–. Springer, 2002.

Methoden und Werkzeuge

7 Der pro et con Migration Manager Ein Werkzeug für die Migration von Host-Anwendungen auf Unix-Plattformen

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

Derzeit sind in vielen Unternehmen Bemühungen zu erkennen, Anwendungen von hostbasierten Systemen in die Unix-Welt zu übernehmen. Die erhoffte Kostenersparnis dürfte der Hauptgrund für diese Entwicklung sein.

Im folgenden Beitrag soll ein Werkzeug vorgestellt werden, welches diese Migration unterstützt und so weit wie möglich vereinfacht.

7.1 Einleitung und Motivation

Obwohl sich die heute vorhandenen, auf Großrechnern basierenden Systeme bewährt haben, gibt es gegenwärtig verstärkt Bemühungen, diese Hardwareplattformen zu ersetzen. Neben der erwarteten *Kostenersparnis* sind die Menge der günstig oder *frei für Unix verfügbaren Programme* und auch der *auslaufende Support* der älteren Hardwaresysteme (HP stellt die Unterstützung seiner K-Serie der NonStop-Server Ende des Jahres ein) ein Grund für diese Tendenz.

Dabei stellt sich die Frage, was bei einem Wechsel der Hardwareplattform aus den vorhandenen Softwarelösungen wird. Neben Neuentwicklung und Einsatz von Standardsoftware (SAP) bietet sich dazu die Migration der Programme an. Das ist meist kostengünstiger und risikoärmer als die Alternativen.

Migrationsprojekte werden häufig komplett als Auftrag an externe Firmen vergeben. Dieses Verfahren hat jedoch auch Nachteile:

1. Das ausführende Unternehmen hat meistens keinen oder nur ungenügenden Einblick in die wirtschaftlichen oder technischen Vorgänge, welche die Software unterstützt. Das erschwert den Test der migrierten Pakete erheblich.
2. Die Migration kann gleich als Einarbeitung in die, meist vorher unbekannte, Zielplattform genutzt werden. Das ist natürlich nur der Fall, wenn die eigenen Mitarbeiter in die Migration eingebunden sind.

Die Alternative dazu ist die Migration mit Hilfe eines Werkzeugs im eigenen Unternehmen. Der Migration Manager (**MigMan**) der Firma pro et con ist ein solches Werkzeug, welches in der derzeitigen Ausbaustufe die Migration von HP NonStop-Cobol-Programmen nach MicroFocus-Cobol auf Unix unterstützt. Darüber hinaus bietet pro et con einen Translator, welcher die in der proprietären Sprache TAL erstellten Programme in portable C-Programme umsetzt sowie ein Werkzeug zur Unterstützung der Konvertierung von ScreenCobol-Masken nach HTML an. Mit Hilfe dieser Werkzeuge wird das gesamte Spektrum einer Migration von HP NonStop nach Unix abgedeckt und diese wesentlich erleichtert.

7.2 Quelltext-Migration

MigMan besteht aus einer Menge von Tools, welche die eigentliche Migration realisieren sowie einer in Java erstellten integrierenden Oberfläche. Im Folgenden soll eine Migration eines in HP NonStop-Cobol erstellten Programmpakets beschrieben werden.

MigMan verwaltet einzelne Migrationsprojekte. Am Beginn einer Migration steht also das Erstellen eines neuen Projektes. Das ist vergleichbar mit dem Erstellen eines Projektes mit Microsoft Visual Studio. In einem Dialog werden Verzeichnisname des Projektes, Quell- und Zieldialekt usw. abgefragt. MigMan erstellt daraus ein Projektverzeichnis mit diversen Unterverzeichnissen für Quell- und Zielcode sowie verschiedene Informationsdateien.

Der nächste Migrationsschritt ist die Bereitstellung des originalen Quellcodes vom Host im dafür vorgesehenen Unterverzeichnis. Das ist vom Anwender in Handarbeit zu erledigen. Dieser Schritt verursacht bei hinreichend

großen, gewachsenen Softwareprojekten einen nicht zu unterschätzenden Aufwand.

Nun müssen noch die Namen der Programme und Unterprogramme bekannt gegeben werden, damit das Tools sie von den Copybooks unterscheiden kann. Dazu wird eine File-Selection-Box verwendet. MigMan führt ein aktuelles Programm mit, auf welches sich weitere Befehle beziehen.

Im nächsten Schritt sollte der Anwender nun testen, ob wirklich alle von den angegebenen Programmen verwendeten Files vorhanden sind. Dies geschieht vollautomatisch durch ein in Perl geschriebenes Tool entweder für das aktuelle Programm oder auch gleich für alle Programme. Die Migration eines Programms ist nur möglich, wenn alle verwendeten Files vorhanden sind.

Hierbei sind noch Namensänderungen zu beachten. Ein Copy-Befehl sieht auf dem HP-Host wie folgt aus:

```
COPY SELECT-BENZ-DATEI OF $DATA3.KABADDLS.GESSLCT.
```

wobei `$DATA3.KABADDLS.GESSLCT` der eigentliche Filename ist. Unter einem Unix-System wird dieser Filename anders aussehen. Daher wird vom Projekt noch eine Tabelle von „Namensübersetzungsregeln“ verwaltet. Diese sind mit Hilfe von regulären Ausdrücken realisiert. Enthält diese Tabelle beispielsweise die Regel

```
$(\w*).(\w*).(\w*) ---> \1/\2\3
```

wird nach `DATA3/KABADDLS/GESSLCT` gesucht.

Werden einige Copybooks nicht gefunden, so wird der Anwender aufgefordert, diese noch bereitzustellen bzw. die „Namensübersetzungsregeln“ zu aktualisieren, damit sie gefunden werden.

Als nächster Schritt folgt die eigentliche Konvertierung. Der Konverter ist ebenfalls ein Perl-Tool und realisiert die folgenden Quelltextänderungen:

- *Entpacken von Bibliotheken*: Auf dem Host kann ein Copy-File in einzelne Sektionen unterteilt sein, die separat eingezogen werden können. Der gezeigte Copy-Befehl zieht z.B. nur die Sektion `SELECT-BENZ-DATEI` aus dem File ein. Ein entsprechendes Konzept gibt es bei MicroFocus nicht. Daher müssen die Sektionen in einzelne Files aufgeteilt werden.
- *Formatanpassungen*: Bei HP NonStop-Cobol ist 1 die Indikatorspalte und Code kann bis Spalte 132 stehen. Bei MicroFocus sind das die Spalten 7 und 72. Der Code muß entsprechend verschoben und evtl. umgebrochen werden.
- *COPY-Befehle anpassen*: Hier sind die neuen Namen zu verwenden. Aus dem obigen Befehl wird dann:

  ```
  COPY
  "DATA3/KABADDLS/GESSLCT/SELECT_BENZ_DATEI.cpy".
  ```

- *Anpassung der Datenbank-Befehle*: Wird auf dem Host noch ein proprietäres Datenbanksystem, wie z.B. Enscribe bei HP NonStop, verwendet, so müssen die entsprechenden Befehle (`OPEN`, `READ`, ...) und Deklarationen (`SELECT`, `FD`, ...) entfernt und dafür semantisch äquivalente SQL-Befehle bzw. Hostvariablendeklarationen eingefügt werden.
- *Anpassung an den Transaktionsmonitor*: Dieser steuert die Zusammenarbeit mehrerer Programme und Anzeigemasken. Die dazu nötigen Befehle im Quelltext unterscheiden sich für verschiedene Transaktionsmonitore und müssen angepasst werden.
- *Weitere Anpassungen*: z.B. Bezeichner `NAME` ist bei MicroFocus ein Schlüsselwort und muß umbenannt werden, `SOURCE-COMPUTER`- und `OBJECT-COMPUTER`-Klauseln müssen entfernt werden, ...

Bei der Konvertierung entstehen die neuen Quelltexte in einem separaten Verzeichnis, wobei die File- und evtl. vorhandenen Unterverzeichnis-Namen beibehalten werden. Wurde ein Copybook bereits bei der Konvertierung eines anderen Programms migriert, so erfolgt eine Rückfrage, ob es erneut migriert werden oder beibehalten werden soll.

Nach der Migration kann der entstandene Quelltext compiliert werden. Der Compileraufruf ist in die Oberfläche integriert. Das dabei entstehende Listing wird hinsichtlich Fehlermeldungen ausgewertet und diese werden angezeigt. Die Fehler können auch sofort in einem integierten Editor behoben werden, wobei automatisch an die betreffende Stelle positioniert wird. Damit sind sehr schnelle Korrektur-Test-Zyklen möglich.

Wurde ein bestimmter Stand erreicht, ist es wünschenswert, diesen zu archivieren. Zu diesem Zweck wird CVS benutzt. Es ist möglich, den aktuellen Stand mit einem Label versehen zu speichern, alle Stände zu einem Quelltextfile anzuzeigen und archivierte Stände wieder herzustellen.

7.3 Datenmigration

Neben der Migration der Quelltexte ist auch eine Übernahme des Datenbestandes notwendig. Diese umfasst die Punkte:

- *Export der Daten auf dem Host*: Die Datensätze werden dabei in ASCII-Files kopiert, die auch auf dem Zielsystem ausgewertet werden können.
- *Anlegen der Datenbank auf dem Zielsystem*: Dazu werden vor allem die entsprechenden CREATE-Statements für Tabellen und Indizes benötigt.
- *Import der Daten auf dem Zielsystem*: Das geschieht mit Hilfe von Scripten, welche die Daten aus den vom Host kopierten ASCII-Files in die entsprechenden Tabellen laden.

MigMan unterstützt alle drei Schritte. Ausgangspunkt ist die Datensatzstruktur im Cobol-Format (FD-Struktur). Mit Hilfe eines speziellen Tabelleneditors kann die Zuordnung der einzelnen Datenfelder zu den Spalten der Tabelle und deren SQL-Typ festgelegt werden, wobei MigMan zuerst eine funktionierende Zuordnung vorschlägt. Nachdem diese angepasst wurde, generiert MigMan daraus folgende Files:

1. *Ein HP NonStop-Cobol-Programm für den Datenexport*: Dieses muß hinsichtlich Filenamen angepasst,

auf den Host kopiert, dort compiliert und ausgeführt werden. Dabei entsteht das ASCII-Datenfile.
2. *Ein SQL-Script mit CREATE-Statements zum Anlegen der betreffenden Tabelle*
3. *Ein Loader-Controlfile speziell für den Oracle-Loader*: Wird der Loader mit diesem File aufgerufen, werden die Daten aus dem ASCII-File in die entsprechende Oracle-Tabelle geladen.

Damit wird der Prozess der Datenübernahme wesentlich erleichtert.

7.4 Zusammenfassung uns Ausblick

Der vorgestellte aktuelle Entwicklungsstand des MigMan unterstützt die Migration von HP NonStop-Applikationen und dabei insbesondere die Quelltext- und Datenmigration. MigMan automatisiert damit ca. 70 % der Migrationsarbeit.

In den folgenden Versionen ist die Integration eines derzeit separaten Werkzeugs zur Unterstützung der Migration von ScreenCobol-Masken nach HTML und des TAL-nach-C-Translators der Firma pro et con geplant.

---

8 Unterstützung von Reengineering-Projekten durch eine moderne, gemeinsam nutzbare Softwareenwicklungsumgebung (Praxisbericht)

Werner Teppe
Amadeus Germany GmbH, Marienbader Platz 1, D 61348 Bad Homburg
wteppe@amadeus.net

Reengineering bzw. Migration von Legacysystemen wird oftmals dadurch erschwert, dass die beteiligten Personen aus „alter“ und „neuer“ Welt andere Fachtermini verwenden, ja jeweils eine eigene Sprache haben. Dies hängt stark damit zusammen, dass unterschiedliche Betriebssysteme, Middleware, Programmiersprachen und Tools eingesetzt werden. Aus dieser in mehreren Projekten gewonnenen Erfahrung, beschlossen wir bei Amadeus Germany die Werkzeuge der verschiedenen Entwickler zu vereinheitlichen, um so eine „Wanderung zwischen den Welten“ zu erleichtern. Als neue Plattform setzen wir insbesondere auf die Open-Source-Plattform ECLIPSE, die wir um geeignete PlugIns für unsere Zwecke erweitert haben.

---

9 Case Studies in Aspect Mining

Silvia Breu
Lehrstuhl für Softwaresysteme, Universität Passau
breu@fmi.uni-passau.de

9.1 Motivation

A major problem in software re-engineering based on aspect-oriented principles lies in finding and isolating crosscutting concerns. This task is called *aspect mining*. The detected concerns can be re-implemented as separate aspects. This reduces the complexity, and improves the comprehensibility of software systems. Thus, aspects facilitate software maintenance and extension. Aspect mining can also provide us with insights that enable us to classify common aspects which occur in different software systems, such as logging, timing, and communication.

Several approaches based on static program analysis techniques have been proposed for aspect mining [vDMM03]. Our approach [BK03] is the first dynamic program analysis approach. It mines for aspects based on program traces that are generated during program execution, and monitor the run-time behaviour of a software system. These traces are then investigated for recurring execution relations. Different constraints specify when an execution relation is “recurring”, such as the requirement that the relations have to exist more than once or even in different calling contexts in the program trace. The dynamic analysis approach has been chosen because it is a very powerful way to make inferences about a system: It dynamically monitors actual program behaviour (run-time behaviour) instead of potential behaviour—as static program analysis does.

The approach has been implemented in a prototype called DynAMiT (*Dyn*amic *A*spect *Mi*ning *T*ool) and evaluated in several case studies over systems with more than 80 kLoC. As we will see, the technique is able to identify *automatically* both seeded and existing crosscutting concerns in software systems.

9.2 Case Study "AspectJ example telecom"

A case study has been conducted in order to verify how successful the developed analysis approach can be applied to a new field: Can DynAMiT also detect crosscutting concerns in Java programs which are already extended by aspects written in AspectJ?

For that purpose the telecom example (included in the distribution of AspectJ) has been chosen: There, people can make telephone calls with different connection types (local and long-distance). The simulation can be executed at three different levels: `BasicSimulation` just performs the calls with the basic functionality needed for making phone calls (call, accept, hang up etc.) `TimingSimulation` is the extension with a timing aspect which keeps track of a connection's duration and cumulates a customer's connection durations. `BillingSimulation` is a further extension with a billing aspect that adds functionality to calculate charges for phone calls of each customer based on connection type and duration. Due to space limitations we describe only some of the detected aspect candidates.

Analysis Results for BasicSimulation. This analysis provides us with crosscutting concerns as well as some insights in the usual sequence of actions in phone calls. The application of the execution relation constraints tells us that the simulation visualises the steps a customer is doing, such as calling someone, answering the phone or hanging up. When someone calls another person, the addition of the call to the pipeline of the customer is done as last thing. The same applies when a called person picks up the phone—the call is added to his pipeline. Another detected crosscutting concern reveals that after a customer has hung up, the call has to be removed from his pipeline.

Analysis Results for TimingSimulation. The resulting sets of aspect candidates in the `TimingSimulation` include those already detected in the `BasicSimulation` (except for some re-namings). The analysis also discovers functionality added by the application of the timing aspect. For instance, the introduction of new fields is discovered: A timer which keeps track of connection times is needed. Before the timer can be started, stopped (or asked for the time), one has to get hold of the timer belonging to the correct connection. Additionally, the timer is needed after a connection is completed or dropped, and when caller and receiver of a connection are determined.

Analysis Results for BillingSimulation. In the third simulation (which includes the timing aspect and a billing aspect on top of that) additional crosscutting concerns introduced by the billing aspect are found. We find, for instance, that before the call rate (local or long distance) for a connection or the receiver of a connection is determined, the current time of the timer is needed. Furthermore, the analysis tells us that—after the correct call rate for a connection is determined—the connection's payer has to be found out. After the paying customer is identified, the charge for the phone call is added to that customer.

9.3 Case Study "Graffiti"

Graffiti[1] (written in Java) is an industrial-sized editor for graphs and a toolkit for implementing graph visualisation algorithms. The software system currently has about 450 interfaces and (abstract) classes, over 3.100 methods and comprises about 82.000 lines. The results of the analysis applied to the Graffiti traces are huge. Only some interesting findings get a closer look as a complete interpretation of all results would be too elaborate. Besides, it would be nearly equivalent to re-factoring the software system.

First of all, DynAMiT has detected a typical, well-known crosscutting concern: logging. Several calls to a method `format` of class `SimpleFormatter` as first and/or last call inside several set- and add-methods were found. A code investigation reveals that all executions of those methods are logged in a log-file. For that, a logger provided by Java's class `Logger` is used. Although we have not traced Java API classes, we know that the logger uses a formatter to transform the system's log messages into human readable messages. For this purpose, Graffiti provides a class `SimpleFormatter` which implements method `format`. Therefore, the analysis detects the formatting of the log-messages, and thus provides us with the information that logging exists. The crosscutting logging functionality is discovered and can be encapsulated into an aspect in a re-engineering process.

The before-aspect candidate desribed in the following is more of an informative nature. Graffiti can easily be extended with graph algorithms by writing plugins. Every algorithm has to implement method `getName` of interface `Algorithm`. If a user wants to use a certain algorithm, e.g. Dijkstra's algorithm, this algorithm has to be added following the plugin principle of Graffiti. In general, every plugin a user wants to have available is registered on startup or (dynamically) later on when loaded. Thus, for every used algorithm, an appropriate plugin has to be added. To be able to determine the kind of the plugin, and in order to have a unique string for each algorithm plugin, the algorithm's name is used, which is gained by calling method `getName` of the corresponding algorithm. Thus, DynAMiT discovers that `getName` in the appropriate algorithm class is always preceded by an execution of `getAlgorithms` of class `GenericPluginAdapter`.

[1] Graffiti version November 2003; now renamed to Gravisto. `http://www.gravisto.org`

Some other retrieved first- and last-aspect candidates tell that method `isSessionActive` of class `MainFrame` is the first method executed inside methods `isEnabled` in each of the classes `FileCloseAction`, `ViewNewAction`, and `RunAlgorithm`. We also observe that this very call is the last one inside `isEnabled` in those three classes. An investigation of Graffiti's source code confirms these analysis results: In the system's architecture it can be seen that class `FileCloseAction`, class `ViewNewAction` as well as class `RunAlgorithm` extend abstract class `GraffitiAction`. Therefore, the question arises why this functionality has not been encapsulated into `GraffitiAction` following object oriented design principles. The reason for that is quite simple: There are a lot more classes extending class `GraffitiAction` which do not have the same functionality, e.g. class `EditUndoAction` or class `ExitAction`. So, the detected pattern (either first- or last-aspect candidate) is a distinct crosscutting concern and thus a candidate for encapsulating this functionality into an aspect.

Method `isSessionActive` was also found as first-aspect candidate in a method called `update` in class `EditRedoAction` as well as in class `EditUndoAction`. A look into the code tells that `EditRedoAction` and `EditUndoAction` both extend abstract class `GraffitiAction`. So the question is again why the developers did not choose a better design. But the analysis algorithm has detected this pattern in only those two but not all of the classes which extend `GraffitiAction`. A further investigation of the source code confirms that result. This suggests that the developers have not been able to provide a different design by encapsulating this concern into the superclass without overriding methods in subclasses (which would be considered bad practice). The introduction of more inheritance levels would not cure the problem either. There is no perfect solution in the sense of OOP, especially in Java as it is not designed to provide multiple inheritance.

9.4 Conclusions

In summary we can say that the results for the three telecom simulations clearly show that the presented approach identifies basic functionality and the functionality added by the two different aspects. The analysis is performed automatically, and did not produce any false positives.

In Graffiti it can be seen that a lot of the functionality is crosscutting its architecture, e.g. actions like to open, save or edit a file or a graph, respectively. It is worth thinking about encapsulation using aspects to satisfy the need for a proper design. Of course, this decision remains to the programmer.

Acknowledgements

Thanks to Jens Dörre for his valuable comments.

Bibliography

[BK03] Silvia Breu and Jens Krinke. Aspect Mining Using Dynamic Analysis. 5. Workshop Software-Reengineering, Bad Honnef. (Published in: GI-Softwaretechnik-Trends, Mitteilungen der Gesellschaft für Informatik, 23(2), pp. 21-22), May 2003.

[vDMM03] Arie van Deursen, Marius Marin, and Leon Moonen. Aspect Mining and Refactoring. In *First International Workshop on REFactoring: Achievements, Challenges, Effects (REFACE)*, 2003.

Reengineering Prozeß

10 The IBM Legacy Transformation Offering

Rainer Gimnich
IBM Global Services, BCS Financial Services, Wilhelm-Fay-Str. 30-34, D-65936 Frankfurt
gimnich@de.ibm.com

In 2003, IBM has launched a new set of offerings to help customers transform their application portfolios. With the need to reduce maintenance costs and to align the IT strategy to future requirements, many companies are looking for suitable methods and tools to leverage their application portfolios.

Application Portfolio Management, Consolidation and Migration, Application Integration, Web-enablement, and Application Renovation are the major building blocks to support application evolution towards flexible, integrated, on-demand operating environments.

10.1 Why transform legacy applications?

Legacy systems typically support the core business processes of an organization. They are essential for the company's economic success. Yet, these systems often require changes, mostly due to 'external' requirements (law, busi-

ness strategy), sometimes due to 'internal', IT-initiated issues (e.g. reorganization, database migration).

In the context of mergers and acquisitions, new operational interfaces are required, resulting in integration and consolidation projects [4].

With cost pressure on one hand, and the need to reposition the IT portfolio for new requirements (e.g. on-demand business) on the other, there is a need for mature methods and tools to analyze existing portfolios, support transformation decisions and perform the transformation work.

These needs have motivated IBM to build a new, comprehensive Legacy Transformation offering, along with a related Application Portfolio Management offering.

Within IBM, these offerings are led by Application Management Services (AMS), who closely cooperate with Business Consulting Services (BCS), the Software Group and IBM Business Partners to provide the optimal solution for the respective customer.

10.2 Application Portfolio Management

Application portfolio management (APM) [2] involves a comprehensive assessment of a company's application landscape, taking into account business, industry and technology priorities. The assessment entails collecting application portfolio information, analyzing the information in the context of business and technical objectives, and identifying transformation opportunities.

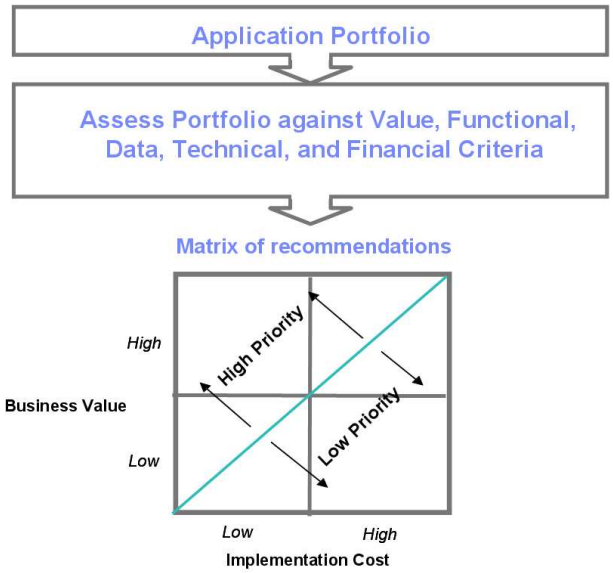


Abbildung 10.1: Application Portfolio Assessment

By using filter criteria during the portfolio assessment, the analysis of the collected data can be taken to the appropriate level of detail. These data, along with additional criteria specified by the customer, lead to target opportunities (TOs), which are then analyzed more thoroughly. Preliminary recommendations (which may include transformations) will be stated at a level that can be input to the IBM ROI (return on investment) toolkit. The estimated efforts and benefits become vital parts of the final recommendations, e.g. to retire, replace, restructure, reprioritize or relocate applications.

10.3 Legacy Transformation

The primary goal of legacy transformation (LT) is to unlock the business value in existing applications by

- supporting enterprise-wide sharing of business and customer data to ease the business/IT linkage;
- incrementally transforming legacy business logic and functionality for improved responsiveness to business change; and
- returning funding to the business through more efficient application maintenance and operation.

The LT offering [3] consists of 4 major capabilities:

Consolidation and Migration

These activities are often required when redundant or functionally overlapping products and solutions are in use, in particular after mergers or acquisitions, but also after in-house organizational changes. These may include technology changes, e.g. the move from IMS/DB to DB2 for data management.

Consolidation and migration may also involve the creation of shared repositories, application analysis for (re-)documentation and/or restructuring, replacement, and major rearchitecting in order to reduce complexity and thus maintenance costs. From the experience so far in various industries, the cost savings due to consolidation and migration can be tremendous.

Application Integration

This capability generally deals with connecting various business functions from different platforms in a technically solid and secure way.

Portals may be used for integration at user interface level. However, Enterprise Application Integration (EAI) activities are largely data-centered and involve the analysis, cleansing, transformation, 'bridging' and aggregation of data. Current integration technologies mostly rely on message-queuing (MQ) technologies: MQ integration, MQ workflow.

'Hub-and-spoke' architectures and midtier applications help to implement application integration without changing the legacy applications.

Web Enablement

Moving legacy applications (or components of them) to the Internet can provide significant benefits in customer services and cost savings. Examples are well-known Web portals relying on legacy functions, e.g. for procurement services, or for insurance information and contract initiation. Other examples use XML wrappers around legacy code and provide access to underlying transactions and data by using callable APIs.

Within LT, Web enablement will also rely on portal server technology, content management, usability engineering and presentational architecture support.

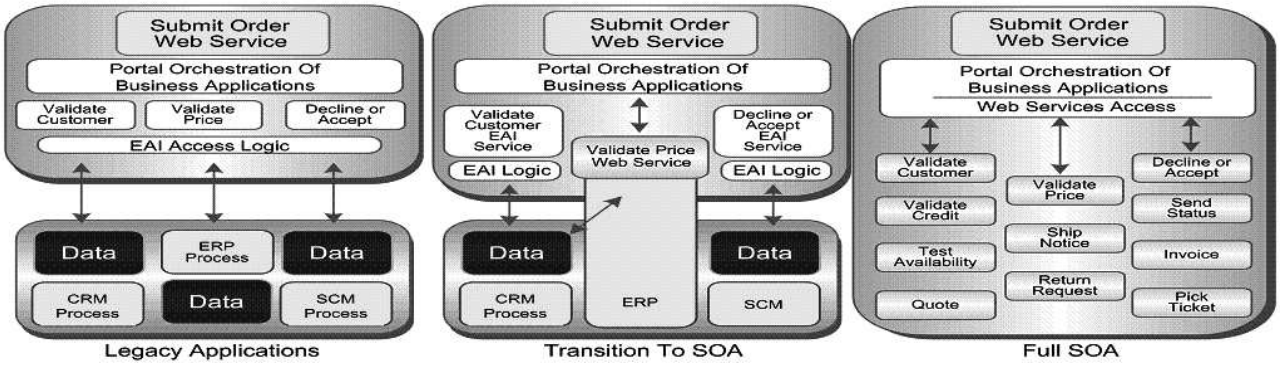


Abbildung 10.2: Logical Transition to SOA, according to [1]

Application Renovation

These are major reengineering tasks to ease application management and the reuse of application functions. Here, the well-known redocumenting, extraction and restructuring techniques merge with new approaches to 'modularize' or 'componentize' applications as a basis to achieve more flexible and maintainable software architectures.

One example [1] is the evolutionary transformation of a legacy landscape into a serviceoriented architecture (SOA) via EAI-based intermediary stages (see Fig. 10.2 below).

10.4 Summary

The Legacy Transformation offering presents a major investment in methods, architecture concepts, products and tools to support present transformation purposes. Use of the LT technology, especially in the Financial Services industry, has proved encouraging so far.

Also, IBM Corporation provides a legacy transformation reference of its own, after consolidating its data and computing centers, and transforming and reducing its 16.000 applications worldwide.

References

[1] Aberdeen Group Inc., Legacy Applications: From Cost Management to Transformation. Executive White Paper. Boston, March 2003.

[2] IBM, Application Portfolio Management Services. `http://www-1.ibm.com/services/ams/apm.html`

[3] IBM, Legacy Transformation Services. `http://www-1.ibm.com/services/ams/legtran.html`

[4] William Ulrich, Legacy Systems Transformation Strategies. Prentice-Hall PTR, Upper Saddle River, NJ, 2002.

---

11 Stand des Software-Reengineering in der SMTL

P. Schützendübe
Suss MicroTec Lithography GmbH Asslar
`P.Schuetzenduebe@web.de`

Erfahrungsgemäß existiert zwischen Theorie und Praxis eine mehr oder weniger große Differenz. Anliegen dieses Vortrages ist es, den aktuellen Stand des Software-Reengineering in der Industrie an praxisrelevanten Beispielen unserer Firma darzustellen.

Nach einer Gegenüberstellung der allgemeinen Arbeitsbedingungen in der Industrie mit den Bedingungen in einem Forschungsinstitut wird eine konkrete Analyse der Industrieerfahrungen vorgestellt.

In einem kurzem Resümee wird ein Überblick über die Arbeit des Softwareingenieurs in der Suss MicroTec Lithography GmbH gegeben. Da ein sehr großer Teil der Software zur Steuerung maschineller Vorgänge benötigt wird, spielt die Hardware der Maschine eine nicht zu unterschätzende Rolle. Dies wird am Beispiel des Maskaligners Ma6 erläutert.

Diskutiert werden dabei die Besonderheiten bei der Auftragsentstehung sowie Probleme durch eine nicht mitgewachsene Softwarearchitektur. Auch auf die Erweiterung fertiger SW-Projekte mit neuen Modulen und die Besonderheiten der Test- und Debuggingmöglichkeiten wird eingegangen. Es wird das aktuelle Know How diskutiert und ein Ausblick auf zukünftige Ziele im Softwarebereich gegeben.

12 Software-Wartung – eine Taxonomie

Stefan Opferkuch
Jochen Ludewig
Abteilung Software Engineering, Institut für Softwaretechnologie, Universität Stuttgart
www.iste.uni-stuttgart.de/se

12.1 Einführung

Das Thema „Software-Wartung“ ist schon so alt wie das Thema „Softwareentwicklung“. Trotzdem fällt es Softwareingenieuren an den Hochschulen und in der Industrie auch heute noch schwer, sich über das Gebiet der Software-Wartung zu verständigen.

Dagegen gibt es kaum Verständigungsschwierigkeiten zwischen Softwareingenieuren über die Entwicklung von Software. Man kann sich sofort darüber abstimmen, in welchem Teilgebiet der jeweilige Gesprächspartner tätig ist. Analyse, Spezifikation, Entwurf, Implementierung, Test, dies sind alles Tätigkeiten, unter denen sich Softwareingenieure, bei allen Differenzen im Detail, etwas vorstellen können.

Und bei der Software-Wartung? Da wird die Verständigung plötzlich schwierig. Da fällt es schwer, überhaupt die Grenzen der Wartung von Software zu benennen. Was ist noch Erstentwicklung, was schon Wartung? In welche Gebiete lässt sich die Software-Wartung aufteilen?

Viele Schwierigkeiten sind auf eine fehlende oder unzureichende Taxonomie für dieses Gebiet zurückzuführen. Im Nachfolgenden soll ein kleiner Beitrag dazu geleistet werden, dass zukünftig nicht immer wieder erneut die grundlegenden Begriffe geklärt werden müssen und dass die Kommunikation über Software-Wartung vereinfacht wird.

12.2 Abgrenzung des Wartungsbegriffs

Der Wartungsbegriff ist alt und kann darum nicht einfach frei definiert werden. Welche Konnotationen sind üblicherweise mit der Wartung verbunden?

Bei technischen Geräten und Einrichtungen findet die Wartung in der Regel statt, um den **Verschleiß** zu kompensieren. Autos, Flugzeuge und andere Maschinen werden darum regelmäßig gewartet. Diese Bedeutung spielt aber in der Software-Wartung keine Rolle, weil Software immateriell ist und daher keinem Verschleiß unterliegt.

Während der Verschleiß vorhersehbar, die entsprechende Wartung also planbar ist, lassen sich überraschende Defekte und Änderungen der Anforderungen nicht vorhersehen. Die dadurch erforderliche Wartung ist also im Detail nicht planbar; man kann höchstens Vermutungen darüber anstellen, welcher Aufwand für diese Art der Wartung einzuplanen ist. Hier ist zu betonen, dass die Defekte in der Software wohl überraschend auftreten, aber natürlich vorher bereits latent vorhanden sind.

Der von Lehman und Belady [1] postulierte Entropie-Anstieg, der bei Software-Systemen als ein Effekt der Wartung eintritt, hat eine gewisse Ähnlichkeit mit dem Verschleiß. Entsprechend kann man bei einem laufend eingesetzten Softwaresystem abschätzen, wann eine Strukturverbesserung (ein Re-Engineering) notwendig wird. Wir halten es daher für sinnvoll, das Software-Re-Engineering als eine von der Software-Wartung verschiedene Tätigkeit zu behandeln. Es scheint zweckmäßig, einen Oberbegriff einzuführen („**Software-Pflege**“), der Wartung und Re-Engineering einschließt.

Wir halten daher fest: Software-Wartung ist im Einzelnen **nicht vorhersehbar**, mit ihr ist ein **Überraschungsmoment** verbunden. Wir sehen als weiteres Merkmal, dass die Software-Wartung im unmittelbaren Interesse der Benutzer liegt. Daher definieren wir:

Software-Wartung ist jede Arbeit an einem bestehenden Software-System, die nicht von Beginn der Entwicklung an geplant war oder hätte geplant werden können und die unmittelbare Auswirkungen auf den Benutzer der Software hat.

Die Wartung findet in der Regel nach Abschluss der Entwicklung statt, sie kann aber auch bereits während der Entwicklung nötig werden.

Umgekehrt endet die **Entwicklung** nicht automatisch mit der Inbetriebnahme der Software. Wenn ein System in einer Folge von Ausbauschritten realisiert wird, handelt es sich nicht um Wartung, es sei denn, dass es beim Ausbau zu überraschenden Änderungen der Planung kommt.

Auch die **Komposition** bestehender Komponenten oder Systeme zu einem neuen System ist keine Wartung, soweit nicht die einzelnen Bestandteile des neuen Systems angepasst oder erweitert werden müssen.

12.3 Arten und Aspekte der Wartung

Wartungsaktivitäten kann man nach verschiedenen Kriterien klassifizieren. Als Kriterien bieten sich an:

- der **Anlass** der Wartung
- der **Ausgangszustand** der Wartung
- die **Art der Anforderungen**, die die Wartung erforderlich machen
- der **Prozess**, nach dem die Wartung abläuft

Abb. 12.1 zeigt eine Übersicht der Taxonomie für die Software-Wartung, sowie deren Verhältnis zu Software-Pflege und Software-Re-Engineering.

12.3.1 Der Anlass der Wartung

Zu Beginn der Wartung einer Software sind vor allem Korrekturen erforderlich. Diese erfolgen in der Regel **reaktiv**, d.h. nach Auftreten von Fehlern oder Problemen.

Abbildung 12.1: Taxonomie der Software-Wartung

Wenn man die Wartung über den gesamten Lebenszyklus einer Software betrachtet, so wird deutlich, dass der überwiegende Teil der Wartungsaktivitäten aus Anpassungen oder Erweiterungen besteht, also **proaktiv** bearbeitet wird.

Teilweise werden auch Korrekturen *proaktiv* durchgeführt, wenn Fehler oder Probleme vorhergesehen oder vermutet werden.

Beim Jahr-2000-Problem wurde ganz überwiegend versucht, die Probleme proaktiv zu beheben. Ebenso geht man meist proaktiv ans Werk, wenn man bei einer Änderung der Umgebung bestimmte Inkompatibilitäten vorhersieht.

In der Literatur wird an Stelle von „proaktiv“ das Wort „präventiv“ verwendet. Wir folgen dem nicht, weil die Prävention durch die Medizin eine ganz andere Konnotation hat. Prävention soll Defekte (Krankheit) verhindern, während sich proaktive Wartung mit Defekten befasst, bevor sie Schaden angerichtet haben.

12.3.2 Der Ausgangszustand der Wartung

Wird in der Software ein Fehler auffällig, so besteht ein Widerspruch zwischen Anforderungen und Implementierung, das System ist **inkonsistent**. Hier ist eine **Korrektur** fällig; die Spezifikation bleibt unverändert, die Implementierung wird verändert, um Konsistenz herzustellen.

Ändern sich die Anforderungen (z.B. durch eine Portierung auf ein anderes Betriebssystem), so ist der Ausgangszustand (prinzipiell) **konsistent**. Wir sprechen in diesem Fall von einer **Anpassung**: Die Veränderung der Spezifikation wird in der Implementierung nachvollzogen, das System wird von einem konsistenten Zustand in einen *anderen* konsistenten Zustand überführt.

12.3.3 Die Art der Anforderungen, die die Wartung nötig machen

Eine Wartung ist dann und nur dann notwendig, wenn eine **Diskrepanz zwischen Anforderungen und Realisierung** aufgetreten oder absehbar ist. Handelt es sich um funktionale Anforderungen, so sprechen wir von einer (funktionalen) **Korrektur** oder einer (funktionalen) **Erweiterung**. Handelt es sich um nicht-funktionale Anforderungen, die die Gebrauchsqualität betreffen, so ist das Ziel der Wartung eine **Verbesserung** des Systems (für den Benutzer oder Kunden).

Offensichtlich kann es sich bei der Verbesserung auch um eine Korrektur handeln, beispielsweise dann, wenn die Antwortzeiten des Systems oder seine Wartbarkeit nicht den Anforderungen entsprechen.

12.3.4 Der Prozess, nach dem die Wartung abläuft

Wartung kann ganz unterschiedlich durchgeführt werden: In vielen Fällen übernehmen die Entwickler selbst, soweit sie noch verfügbar sind, die Wartung neben anderen (Entwicklungs-)Arbeiten. Dies bezeichnen wir als **Wartung durch die Entwickler**. Eine weitere Möglichkeit ist, spezielle Personen mit der Wartung zu betrauen. Wir sprechen dann von einer **Wartung durch Wartungsingenieure**.

Software-Re-Engineering wurde zu Beginn als eigene, von der Wartung verschiedene Aktivität definiert. Wir bezeichnen als Re-Engineering Änderungen an einer Software, die ausschließlich einer Verbesserung hinsichtlich von Wartungsqualitäten dienen. Wartungsqualitäten sind Eigenschaften der Software, die direkt keinen Einfluss auf die Qualität aus Sicht eines Anwenders haben, sondern nur die Entwickler und Wartungsingenieure betreffen.

In der Praxis werden Re-Engineering und Wartung oft vermischt; das ist in der Regel nicht sinnvoll, weil das reine Re-Engineering durch Regressionstests sehr gut überwacht werden kann; sobald funktionale Änderungen dazukommen, ist die klare Regel „Alles muss nachher funktionieren wie vorher.“ aufgehoben.

Literaturverzeichnis

[1] LEHMAN, M.M. und L.A. BELADY: *Program evolution: processes of software change*. Academic Press Professional, Inc., San Diego, CA, USA, 1985.

13 Softwarewartung und Prozessmodelle in Theorie und Praxis

Urs Kuhlmann
Andreas Winter
Universität Koblenz-Landau, Institut für Softwaretechnik, D-56016 Koblenz
(kuhlurs|winter)@uni-koblenz.de

13.1 Motivation und Wartungsbegriff

Sowohl der Begriff der *Wartung* als auch der des *Reengineerings* sind spätestens seit den 1980er Jahren im Bereich der Softwaretechnik in der Literatur und in der Praxis etabliert. Allerdings gibt es bisher keine einheitlichen und allgemein gebräuchlichen Begiffsdefinitionen.

Wartung wird in den meisten Definitionen ausdrücklich als Aktivität *„nach Auslieferung des Softwareprodukts“* verstanden. Die meisten Autoren betonen aber auch, dass die Planung von Wartungsaktivitäten mit der Planung der Softwareentwicklung beginnen soll. Wartungsaktivitäten sind daher nicht losgelöst von Software-Entwicklungsprozessmodellen zu betrachten, sondern sollten als zentrale Bestandteile der Software-Entwicklung verstanden werden.

Diese Arbeit beschreibt die Einbettung der Wartung in Prozessmodelle der Theorie und fasst sechs Fallstudien zusammen, die die Integration der Wartungsaktivitäten in die betriebliche Praxis zeigen. Eine Zusammenfassung mit einem Ausblick über mögliche Lösungsansätze zur verbesserten Etablierung der Wartung in der Softwareentwicklung beendet diese Kurzfassung.

13.2 Prozessmodelle

In den Prozessmodellen der Softwareentwicklung sind Wartungsaktivitäten sehr unterschiedlich integriert. Dieser Abschnitt liefert eine Übersicht über die Einbettung von Wartungsaktivitäten in verschiedenen Prozessmodellen.

- Im *Wasserfallmodell* [1], als Beispiel für lineare Modelle, sind Wartungsaktivitäten in einer separaten, nämlich der letzte Phase zusammengefasst. Diese Phase wird als Betrieb und Wartung an die eigentliche Entwicklung angehängt.
- Der *IEEE Standard for Software Maintenance* [3] liefert eine Beschreibung für die Vorgehensweise bei Wartungsprojekten. Er beschreibt detailliert linear angeordnete Wartungsaktivitäten. Problematisch ist hier die Tatsache, dass nur ein einziger, sehr umfangreicher und grober Rahmen geliefert wird. Kleinere, speziellere Wartungsaufgaben werden aufgrund der Allgemeinheit dieses Raster nicht ausreichend unterstützt.
- In *Extreme Programming* [2], als Beispiel für die agile Software-Entwicklung, wird nur erwähnt, dass Wartung als Normalzustand eines XP Projektes angesehen werden kann. XP ist als generelles Prozessmodell für Wartung im Allgemeinen nur beschränkt einsetzbar. Auch beschränken Projektgröße und -ziele den Einsatz von XP im Reengineering [6].
- Im *(Rational) Unified Process* [4], einem inkrementellen Modell, sind wartungsrelevante Aktivitäten zwar in vielen Workflows enthalten. Jedoch wird Wartung nicht explizit unterstützt und genutzte Aktivitäten sind eher auf Neuentwicklung ausgerichtet. Wichtige analytische Wartungsaktivitäten wie z. B. das Programmverstehen und die Impact-Analyse sind im RUP nicht enthalten.

Wartung wird nur im Wasserfallmodell explizit erwähnt und ist in anderen Modellen i.d.R. implizit enthalten. Die meisten Aktivitäten werden nicht unter Wartungsaspekten betrachtet, reine Wartungsaktivitäten fehlen. Moderne Prozessmodelle decken Wartung gar nicht oder nur ungenügend ab [5].

13.3 Fallstudien

Eine Befragung von sechs Firmen zum Thema Softwarewartung [5] ergab, dass auch in der Praxis kein einheitlicher Wartungsbegriff verwendet gibt. Die Begriffsverwendungen sind im Vergleich zur Literatur sogar noch verschwommener, was deutlich ausdrückt wird durch Aussagen wie *„Wir warten keine Software, wir entwickeln sie weiter“* und *„Wartung ist der Weg vom Ist zum Soll“*.

Gängige Praxis zur Definition von Wartung ist die Eingrenzung über den Aufwand, der zur „Änderung“ aufgewandt wird. Wartung umfasst z. B. alle Änderungen mit einem Aufwand von weniger als 15 oder 30 Personentagen. Jedoch wird dadurch Wartung hauptsächlich auf korrektive Wartung beschränkt. Wartungskategorien wie adaptive, perfektive und preventive Wartung werden (wenn überhaupt) als von (korrektiver) Wartung unabhängig betrachtet. Wartung umfasst jedoch mehr als nur das Beheben von Fehlern. So werden spezifische Wartungsaktivitäten, besonders analytische Aktivitäten, auch bei beliebigen Anpassungen bestehender Programmen an geänderte Umgebungen benötigt.

Wartung ist nur selten in die Entwicklungsprozesse der Firmen integriert. Meist werden Wartungsaktivitäten separat betrachtet und in getrennten Wartungsprojekten durchgeführt. Die ausgeführten Aktivitäten sind dabei teilweise identisch, aber zumindest sehr ähnlich zu denen der Entwicklung. So wird die Aktivität „Implementierung“ in der Entwicklung als auch in der Wartung durchgeführt. Es werden aber auch Aktivitäten benötigt, die in reinen Entwicklungsprojekten nicht vorkommen.

Wesentliche Faktoren, die Wartungsprozesse beeinflussen, sind die Firmenkultur und das zu wartende System selbst. Auch die Umsetzung von Programmierrichtlinien bei der Entwicklung und in früheren Wartungsprojekten sind entscheidend. Weiterhin wurden die Fachkenntnisse

der Beteiligten und eine angemessene Werkzeugunterstützung hervorgehoben.

Auch wenn sich die Befragten gute Noten für die Qualität ihrer Wartungstätigkeiten geben, erhoffen sie sich Qualitätsverbesserungen durch Veränderung ihrer Prozesse. Verbesserungen werden durch ausgereiftere und besser integrierte Wartungswerkzeuge erwartet. Ursachen für die bisher geringe Werkzeugnutzung liegen in den Besonderheiten zu wartender Software und dem Anpassungsaufwand der vorhandenen Werkzeuge. Aufgrund eher schrumpfender Wartungsbudgets wird allerdings nicht erwartet, dass die erhofften Änderungen in naher Zukunft umgesetzt werden.

Die wesentlichen Ergebnisse der Fallstudien sind, dass der Wartungsbegriff auch in der Praxis nicht klar definiert ist, Wartungsaktivitäten nur bedingt in betriebliche Softwareentwicklungsprozesse integriert sind und dass ein wesentlicher Änderungswunsch die Werkzeugunterstützung für den Bereich der Wartung betrifft [5].

13.4 Zusammenfassung und Ausblick

Aus der Untersuchung des Wartungsbegriffs in Theorie und Praxis lassen sich einige Problembereiche für die unzureichende Wahrnehmung von Fragestellungen der Wartung ableiten. Im Folgenden werden diese Problembereiche sowie einige Lösungsansätze zur besseren Etablierung der Wartung in der Software-Entwicklung skizziert [5].

Terminologie

Die Begriffsinhalte der Wartung sind weder in der Literatur noch in der Praxis eindeutig definiert und werden daher vielseitig verwendet. Auch eine Abgrenzung von Neuentwicklung und Wartung ist schwierig und kaum möglich. Es gibt keinen klaren Übergang von Entwicklung zu Wartung, denn Wartungsaktivitäten müssen bereits bei der Entwicklung berücksichtigt und eingeplant werden. In gemeinsamen Projekten ist es daher notwendig eine *gemeinsame Definition* als Arbeitsgrundlage zu finden.

Statistische Daten

Statistische Daten über Wartung liegen nur aus den 1970er und 1980er Jahren vor. Aktuelle Angaben über die Kosten der Wartung und die Aufteilung auf verschiedene Wartungskategorien sind nicht verfügbar. In den meisten Firmen erfolgt keine Abgrenzung zwischen Entwicklungs- und Wartungsbudgets. Zur Ermittlung der Bedeutung von Wartung sind *empirische Untersuchungen* zur Sammlung von aktuellen statistischen Daten notwendig.

Bewusstsein für Wartung

Wartung wird in der Literatur zwar oft als ein wichtiges Gebiet bezeichnet, aber der Schwerpunt der Aufmerksamkeit liegt auf der Neuentwicklung. Auch in der Praxis ist die Wartung nur das unbeliebte Stiefkind, dem keine Priorität gegeben wird. Es wird, im Gegensatz zu Empfehlungen aus der Literatur, auch oft als Trainingsbereich für neue Mitarbeiter eingesetzt, die bei entsprechender Leistung „vom Wartungspersonal zu Entwicklern aufsteigen können“. Daher ist es notwendig, den Entscheidungsträgern in Firmen die *Bedeutung und Folgen* von (mangelhafter) Wartung zu verdeutlichen und bei Mitarbeitern für Wartungsprojekte zu werben.

Einbettung der Aktivitäten in Prozessmodelle

Wartungsaktivitäten werden von existierenden Prozessmodellen nur ungenügend abgedeckt. Auch viele Firmen haben nur einen beschriebenen Entwicklungsprozess und keinen klar definierten und praktizierten Wartungsprozess. In diesem Bereich ist es nötig, wartungsspezifische Aktivitäten zu identifizieren und zu beschreiben. Zur Berücksichtigung in Software-Entwicklungs- und -Wartungsprozessen müssen diese in bestehende Prozessmodelle eingebettet oder durch ein *Referenzprozessmodell* für Softwarewartung zur Verfügung gestellt werden. Dieses kann z. B. durch Zusammenfassung der für die Wartung relevanten Aktivitäten in einen Wartungsworkflow des RUP erfolgen.

Werkzeugunterstützung

Eine umfangreiche Werkzeugunterstützung wird in der Literatur als kritischer Faktor für die Erhöhung von Qualität und Effizienz genannt. Obwohl eine stärkere Unterstützung in der Praxis gewünscht wird, ist sie nur selten anzutreffen. Ein mögliches Referenzprozessmodell für die Softwarewartung sollte um passende *Werkzeugunterstützung* erweitert werden, damit die Wartungsaktivitäten effizient und mit hoher Qualität durchgeführt werden können.

13.5 Fazit

Zusammenfassend bleibt festzustellen, dass sich die Software-Wartung als *zentrale Disziplin der Software-Entwicklung* noch etablieren muss. Die klare Definition des Wartungsbegriffes und die Abgrenzung zur Entwicklung ist hierbei aber nicht das Wesentliche. Vielmehr ist die Einbettung von Wartungsaktivitäten in Prozessmodelle zur Software- (Weiter-) Entwicklung und die konsequente werkzeugbasierte Anwendung dieser Aktivitäten der wesentliche Erfolgsfaktor für die zukünftige Softwareerstellung.

Literaturverzeichnis

[1] W.W. Agresti. The Conventional Software Life-cycle Model: Its Evolution and Assumptions. *New Paradigms for Software Development*, pp 2–5, 1986.

[2] K. Beck. *Extreme programming explained: embrace change*. Addison-Wesley, Upper Saddle River, 2000.

[3] IEEE Standard for Software Maintenance, IEEE Std 1219-1998. The Institute of Electrical and Electronics Engineers, 1998.

[4] P. Kruchten. *The Rational Unified Process - An Introduction*. Addison-Wesley, Upper Saddle River, 2nd edition, 2000.

[5] U. Kuhlmann. Maintenance Activities in Software Process Models: Theory and Case Study Practice. Universität Koblenz, 2004.

[6] C. Poole and J.W. Huisman. Using Extreme Programming in a Maintenance Environment. *IEEE Software*, 18(6):42–50, 2001.

Architektur-Erkennung

14 Viewpoints in Software Architecture Reconstruction

Arie van Deursen
CWI & Delft Univ. of Technology, The Netherlands
Arie.van.Deursen@cwi.nl

Christine Hofmeister
Lehigh University, USA
hofmeister@cse.lehigh.edu

Rainer Koschke
University of Stuttgart, Germany
koschke@informatik.uni-stuttgart.de

Leon Moonen
Delft Univ. of Technology & CWI, The Netherlands
Leon.Moonen@computer.org

Claudio Riva
Nokia Research Center, P.O. Box 407, FIN-00045, Helsinki, Finland
claudio.riva@nokia.com

14.1 Introduction

Many software engineering tasks are hard to conduct without relevant architectural information (e.g., migrations, auditing, application integration, or impact analysis). Unfortunately, architectural information, if available at all, is often outdated, incorrect, or inappropriate.

Software architecture reconstruction is the process of obtaining a documented architecture for an existing system. Although such a reconstruction can make use of any possible resource (such as available documentation, stakeholder interviews, domain knowledge), the most reliable source of information is the system itself, either via its source code or observations on its execution.

It is widely accepted that architectures must be described by multiple views. A *view* is a representation of a whole system from the perspective of a related set of concerns [3]. Prominent views are the 4+1 views by Kruchten [4] or the Siemens views [2]. The recent book by Clements and colleagues [1] and the *IEEE Recommended Practice for Architectural Description of Software-intensive Systems* [3] give a larger catalog of architectural views.

Previous research in architecture reconstruction has focused on recovering a single architectural view or a few preselected views. The application of these techniques usually involves three steps: extract raw data from the source, apply the appropriate abstraction technique, and present or visualize the information obtained. These steps are specific to the views to be reconstructed.

Unfortunately, there is no set of "standard views"that fits all purposes of an architectural description, so that the applicability of these techniques is limited in scope. In recognition that views depend upon the specific purpose of an architectural description and these purposes may be very diverse in practice, an architecture reconstruction method should treat views as first-order elements.

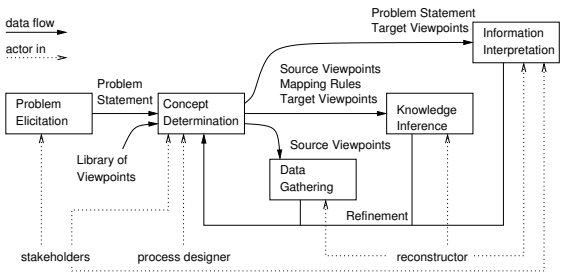


Figure 14.1: Interaction during reconstruction design.

Filling this gap, in this paper we describe Symphony, a process framework that has an explicit step for the discovery of the views that should be reconstructed in order to solve the problem at hand. Symphony is view-based

in recognition of the importance of multiple architectural views not only in presenting architecture but more fundamentally in defining the reconstruction activities. Symphony[1] is the result of a systematic analysis of our own experiences in software architecture reconstruction, cases conducted by close colleagues, and the various approaches that have been published in the literature. Symphony provides a conceptual framework that helps researchers by providing a unified approach to reconstruction, with consistent terminology and a basis for improving, refining, quantifying, and comparing reconstruction processes and case studies.

14.2 The Symphony Framework

Symphony consists of two stages. The first stage (Problem Elicitation and Concept Determination) produces a repeatable and reusable reconstruction strategy that creates the views necessary to address the original problem. This procedure may be useful beyond the scope of the current reconstruction: it can play a role in continuous architecture conformance checking and in future reconstructions. Although not an ultimate goal, the problem-dependent types of views created or refined in the Concept Determination phase are another reusable output of this stage.

The second stage of Symphony concerns the execution of the reconstruction strategy. Its outcome is the foundation for addressing the problem for which the particular reconstruction is carried out. A secondary outcome is the sequence of mappings from the source views (those extracted from the system's artifacts) to the target views (those that address the problem at hand). This sequence allows one to trace back the information in the views to the artifacts from which they were derived.

Typically the two stages are iterated: Reconstruction execution reveals new reconstruction opportunities, which lead to a refined understanding of the problem and a refined reconstruction design. The underlying types of source and target views and the mapping rules evolve throughout the process.

Reconstruction Design The Reconstruction Design is devided in two steps: *Problem Elicitation* analyzes the problem triggering the reconstruction and involves all stakeholders. Once the problem is understood, the Concept Determination step is used to determine the architectural information needed to solve the problem and the way to derive this information. In this step, the architect is a process designer, defining the architectural reconstruction that will take place in the Reconstruction Execution.

The reconstruction activities are defined in terms of the views they deal with: A *source view* is a view of a system that can be extracted from artifacts of that system, such as source code, build files, configuration information, documentation, or traces. A *target view* is a view of a software system that describes the as-implemented architecture and contains the information needed to solve the problem for which the reconstruction process was carried out.

Views are specified by so-called viewpoints. In IEEE 1471, a *viewpoint* describes the rules and conventions used to create, depict, and analyze a view based on this viewpoint [3]. A view conforms to a viewpoint. While a view describes a particular system, a viewpoint specifies the kind of information that can be put in a view and is independent of any particular system.

In the Concept Determination activity, the viewpoints for the target and source views are selected or defined, and the mapping rules from source to target views are designed. The mapping rules are ideally a formal description of how to derive a target view from a source view. Realistically, parts will often be in the form of heuristics, guidelines, or other informal approaches. If a mapping can be completely formalized, the reconstruction can be fully automated. This is not typically possible for software architecture, thus we expect the mapping to contain both formal and informal parts.

Reconstruction Execution The *Reconstruction Execution* stage (cf. Figure 14.2) operates only at the level of views constrained by the viewpoints created before.

The goal of the *Data Gathering* step is to collect the data that is required to recover selected architectural concepts from a system's artifacts through static or dynamic analyses. *In Knowledge Inference*, the reconstructor applies the mapping rules to populate the target views by condensing the low-level details of the source view and abstracting them into architectural information. The mapping rules and domain knowledge are used to define a map between the source and target view.

In the *Information Interpretation*, conclusions are drawn from the reconstructed views. These conclusions then lead to measures to be taken to remedy the problem. To this end, the target views need to be made accessible both physically and mentally to all stakeholders.

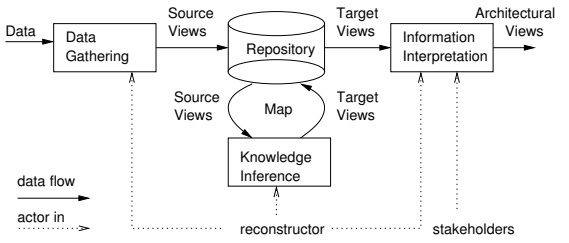


Figure 14.2: Reconstruction execution interactions.

14.3 Concluding Remarks

Symphony incorporates the state of the practice, where reconstruction is problem-driven and uses a rich set of architecture views. It has been applied by the authors in academic and industrial case studies and unifies other existing reconstruction techniques and methods.

Viewpoint selection and definition is an important part

[1] The name Symphony reflects that a successful reconstruction is the result of the interplay of many different instruments. Moreover, the authors' collaboration in the area of software architecture reconstruction started in the music room of Castle Dagstuhl in Germany.

of the Symphony process. Using viewpoints to specify the input and output of an activity allows us to decompose the reconstruction process systematically and to review the outcome of each activity. In addition, we can reuse an activity as a building block to compose new reconstruction processes.

In addition, Symphony provides a common reference framework that can be used when classifying and comparing various techniques described in the literature. It helps us to find and demarcate research problems in software architecture reconstruction. For example, Symphony's viewpoint emphasis calls for a catalog of reconstruction methods, techniques, and experiences organized by viewpoints. Moreover, it raises the question what reconstruction-specific viewpoints exist. Symphony's inclusion of mappings between source and target views suggests finding a systematic way to discover and describe such mappings as a key research question. Problems like these are hard to tackle. Symphony makes it possible to address them on a case-by-case basis, offering its process model as a way to classify and compare results.

Bibliography

[1] P. Clements, F. Bachmann, L. Bass, D. Garlan, J. Ivers, R. Little, R. Nord, and J. Stafford. *Documenting Software Architectures: Views and Beyond*. Addison-Wesley, 2002.

[2] C. Hofmeister, R. Nord, and D. Soni. *Applied Software Architecture*. Object Technology Series. Addison Wesley, 2000.

[3] IEEE P1471-2000. IEEE recommended practice for architectural description of software-intensive systems, 2000.

[4] Phillipe Kruchten. The 4+1 view model of architecture. *IEEE Software*, 12(6):42–50, November 1995.

---

15 Symphony Fallstudie: Hierarchische Reflexion Modelle

Rainer Koschke
Daniel Simon
University of Stuttgart, Universitätsstraße 38, 70569 Stuttgart, Germany
{koschke,simondl}@informatik.uni-stuttgart.de

15.1 Einleitung

In diesem Bericht stellen wir eine Fallstudie vor, die dem Prozessmodell für Symphony [3] folgt. Wir führen in diesem Zusammenhang eine Sichten getriebene Architekturrekonstruktion durch, bei der das hierarchische Reflexion Modell [4] zum Zuge kommt. Im folgenden werden die einzelnen Schritte von Symphony für die Validierung von zwei Open-Source Compiler gegen eine Compiler Referenzarchitektur aus der Literatur mit konkretem Inhalt gefüllt.

15.2 Symphony Designebene

Problem Elicitation Als erster Schritt wird bei Symphony die Aufgabe des Reengineering Projekts definiert. In unserem Fall ist dies der Vergleich einer Compiler Referenzarchitektur mit den Architekturen der beiden Open-Source C Compiler `sdcc` [8], einem Compiler für Mikrocontroller, und `cc1` [2], einem Teil der GNU Compiler Collection ist. Die Beteiligten an dem Prozess waren wir selbst; als Nebeneffekt erhofften wir uns eine Evaluation unserer Erweiterung des Reflexion Modells.

Concept Determination Bei der Bestimmung der benötigten Konzepte für die Architekturvalidierung wurden als nützliche Sichten die Modulsicht auf den jeweiligen Compiler, die konzeptuelle Sicht mit der hypothetischen Architektur sowie die Reflexion Modell Sicht ausgewählt. Der gewählte *Target Viewpoint* war in der ersten Iteration die Kombination aus Modul und Reflexion Sicht, in der zweiten Iteration haben wir die Modulsicht mit der hierarchischen Reflexion Sicht kombiniert. Die *Source* und `Target Viewpoints` sind in Abbildung 15.1 zusammengefasst. Bemerkenswert ist im Zusammenhang mit Symphony, dass bei der Anwendung der Reflexion Modelle auch explizit hypothetische Sichten benutzt werden.

Source Viewpoint	Target Viewpoint
dir *contains* dir	module *convergence* module
dir *contains* module	module *divergence* module
module *contains* declaration	module *convergence* module
declaration *depends-on*$_a$ declaration	module *contains* module
	module *depends-on*$_a$ module
	module *depends-on*$_h$ module

Abbildung 15.1: Viewpoints bei der Compiler Validierung

Iterationen Auf der Designebene von Symphony führten die Mängel des ursprünglichen Reflexion Modells zu einer Iteration, in der wir unsere Erweiterung des Reflexion Modells zum Einsatz brachten.

Hierarchisches Reflexion Modell Das ursprüngliche Reflexion Modell, das von Murphy and Notkin [5, 6, 7]

vorgeschlagen wurde, erlaubt es dem Reengineer, ein abgeleitetes oder vorgeschriebenes Architekturmodell eine Softwaresystems gegen ein automatisch aus den Quelltexten abgeleitetes Modell zu validieren. Dabei werden zunächst die Entitäten im Quellmodell auf das hypothetische Architekturmodell abgebildet. Im Anschluss daran werden die Diskrepanzen zwischen Quell- und Architekturmodell automatisch berechnet und können dem Benutzer in geeigneter Weise präsentiert werden.

Da das Modell keine hierarchischen Architekturen unterstützt, ist es für die Untersuchung großer Systeme nur bedingt geeignet. In [4] haben wir das Modell von Notkin u.a. um die Möglichkeit erweitert, auch mit hierarchischen Architekturen umzugehen.

15.3 Symphony Ausführungsebene

Data Gathering Das Aufsammeln der Daten gestaltet sich in diesem Fall als Extraktion von statische Abhängigkeiten aus den Quellen der Compiler. Die Extraktion wurde automatisiert mit Hilfe der Bauhaus Toolsuite [1] durchgeführt; unter anderem haben wir globale Deklarationen (von Typen, Variablen und Routinen), Module und Verzeichnisse sowie die in Abbildung 15.2 aufgezählten Beziehungen automatisch ermittelt.

Knowlegde Inference und Information Interpretation Sowohl die Berechnung des hierarchischen Reflexion Modells als auch der Abgleich der gewonnen Informationen über die Quellen kann interaktiv mittels der graphischen Benutzeroberfläche von Bauhaus durchgeführt werden. Das Reflexion Modells lässt sich auf natürliche Art durch einen Graph visualisieren.

Iterationen Auf der Ausführungsebene von Symphony gab es eine ganze Reihe von Iterationen. Diese wurde zum einen verursacht durch die Tatsache, dass beim Erstellen der Referenzarchitektur Fehler gemacht wurden. Als Konsequenz wies das Reflexion Modell unerwartete Diskrepanzen auf; Ergänzungen der Referenzarchitektur behoben diese Problem.

Weitere Iterationen rührten von der Tatsache, dass die Abbildung von Quellentitäten auf hypothetische Module zum Teil von Dateien auf kleinere Einheiten verfeinert werden musste. Eine Iteration verfeinerte das Quellmodell, in dem zusätzlich zu direkten Funktionsaufrufen auch Aufrufe von Funktionen über Zeiger einfließen. Dazu ist allerdings eine Zeigeranalyse notwendig, deren (notwendigerweise konservative) Ergebnisse noch manuell geprüft werden mussten.

15.4 Erfahrungen mit dem Reflexion Modell

Das Validieren einer Referenzarchitektur macht bei großen System nur Sinn, wenn man hierarchische Reflexion einsetzt. Das manuelle Erstellen der Referenzarchitektur ist aufwändig, ebenso braucht man viel Erfahrung mit dem untersuchten System, um die Abbildung von Quellentitäten auf Modellentitäten durchzuführen. Beides sind allerdings Aufgaben, die sich viel einfacher gestalten, wenn sie bereits bei der Entwicklung der Softwaresysteme mit berücksichtigt werden.

Die anschließende Anwendung der Reflexion zum Zwecke der Validierung und Kontrolle der Implementierung ist automatisierbar und bietet auch Möglichkeiten, Fortschritte über mehrere Quellversionen zu verfolgen.

Referenz Typ	Beschreibung
static call	statically bound call of function
dynamic call	call through function pointer
access	use, set, or address-taken of a variable or record component
r-access	address-taken of a function
signature	type occurs in function signature
of-type	type of a variable or record component
local-var-of-type	function has local variable of type
based-on-type	one type uses another type for its declaration

Abbildung 15.2: Extrahierte Referenz Typen.

Literaturverzeichnis

[1] Bauhaus. `http://www.bauhaus-tec.com/`, Mai 2004.

[2] Free Software Foundation. The GNU Compiler Collection. `http://gcc.gnu.org/`.

[3] R. Koschke. View-Driven Software Architecture Reconstruction. In *Proc. of the WSR*, May 2004.

[4] R. Koschke and D. Simon. Hierarchical Reflexion. In *Proc. of the WCRE*, pages 36–45, Victoria, B.C., Nov. 2003.

[5] G. C. Murphy and D. Notkin. Reengineering with Reflexion Models: A Case Study . *IEEE Computer*, 30(8):29–36, Aug. 1997.

[6] G. C. Murphy, D. Notkin, and K. Sullivan. Software Reflexion Models: Bridging the Gap Between Source and High-Level Models. In *Proc. of the Third ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 18–28, New York, NY, 1995.

[7] G. C. Murphy, D. Notkin, and K. J. Sullivan. Software Reflexion Models: Bridging the Gap between Design and Implementation. *IEEE TSE*, 27(4):364–380, Apr. 2001.

[8] *sdcc*, a C Compiler for small devices. `http://sourceforge.net/projects/sdcc/`, June 2003.

16 Combining Clustering with Pattern Matching for Architecture Recovery of OO Systems

Markus Bauer
Mircea Trifu
FZI Forschungszentrum Informatik, Karlsruhe, Germany
{bauer,mtrifu}@fzi.de

16.1 Introduction

This work is concerned with recovering the architecture of an object-oriented software system based on information extracted from the source code. Over the years, several automatic techniques to recover subsystem decompositions have been proposed, such as clustering techniques or pattern-matching techniques, however none of them has the desired precision. The resulting decompositions are either not meaningful to a software engineer or they cover only pieces of the whole system ([3], [4]).

16.2 The Approach

In this paper, we propose an approach that combines clustering with pattern-matching techniques to recover complete and meaningful decompositions. Pattern-matching is used to identify architectural clues — small structural patterns that provide semantic information about the dependencies found between a system's entities. These clues are then used to compute an adaptive inter-class similarity measure which is then used by a clustering algorithm to produce the final system decomposition. In essence, the proposed approach tries to capture as much as it can from the original structure and then fill in the rest of the puzzle by imposing a suitable structure so as to minimize the coupling between the resulting subsystems and maximize their internal cohesion. Coupling and cohesion are expressed in terms of inter-class relationships and usage.

Our approach consists of five phases: *Fact extraction*, *Architectural clue gathering*, *Couplings adaptation*, *Compaction* and *Clustering*.

The purpose of the *Fact extraction* phase is to construct an object model of the source code in order to ease access to the information contained in it. The underlying meta-model[1] contains all the major syntactic elements and the interactions specifiable in a typical OO language such as: classes, methods, attributes, inheritance, aggregation, access, call, etc.

In the second phase, called *Architectural clue gathering*, the source model is decorated with semantic information. The information is incorporated in the model as annotations called *architectural clues*. One must point out that the information added in this phase is not essentially new. It is extracted from the already constructed source model by a set of *structural pattern recognizers*. As architectural clues we use *method types* — a classification of methods based on their semantic role, *library classes*, as well as seven GoF design patterns: *Template method*, *Abstract factory*, *Strategy*, *Composite*, *Proxy*, *Adapter* and *Facade*.

According to their semantic role, methods are classified based on three criteria: *Kind* (Abstract, Constructor, Constant, Empty, Accessor, Template, Factory, Delegating, Alias, Normal), *Inheritance Statute* (Implementing, Extending, Overriding, Adding, New) and *Usage* (Initialization, Public Interface, Protected Interface, Implementation).

Detection of library classes followed by coupling adaptation is our solution to the problem of omnipresent entities faced by other clustering approaches. We recognize library classes based on the number of clients that use them.

Recognizing design patterns can provide invaluable information about subsystem structure. Their presence usually points to a group of classes that belong together. For example, the presence of a *Composite* shows a strong coupling between the composite class and the aggregated component class.

In the *Couplings adaptation* phase, the annotated source model is reduced to a multigraph structure having classes as nodes and *coupling metrics* as edge values. For each of the syntactic interactions extracted in the first phase, a specific coupling metric is computed to show the strength of that particular type of interaction. Architectural clues are used to put each interaction in a wider context and adapt its corresponding metric value according to its semantic role in that context. There are six types of couplings that we consider: *Inheritance coupling*, *Aggregation coupling*, *Association coupling*, *Access coupling*, *Call coupling* and *Indirect coupling*. Following our previous example, the presence of a *Composite* pattern results in higher coupling metric values for the inheritance, aggregation as well as all the delegating calls between the composite class and the aggregated component class.

Indirect coupling expresses the coupling given by common usage. If two classes are constantly used together, it is likely that they are somewhat related even if no other direct relationship exists between them. To determine if two classes are used together, we consider only the calls to their public methods. If a method body contains calls to methods belonging to several classes, then between each pair of called classes, there is an indirect coupling.

Using indirect coupling was already suggested in the literature[2], however it has not been exploited yet. We

[1] For our meta-model we use MeMoJ: a metrics oriented meta-model for structural analysis of Java code developed at "Politehnica" University of Timişoara by Radu Marinescu, Daniel Raţiu and Mircea Trifu.

[2] A similar idea was proposed by Koschke in [2]

have found that indirect coupling is especially effective in grouping library classes together.

In the *Compaction* phase, the multigraph structure is reduced to a simple undirected graph. First we compute a weighted sum of the above mentioned couplings which we call *directed similarity* and then assign the maximum of the two directed similarities to the resulting undirected edge of the graph.

In the last phase, the undirected graph is clustered using a two-pass MST-like clustering algorithm to produce the final subsystem decomposition.

Further details can be found in [1].

16.3 Evaluation

We have developed an evaluation environment called ACT (Adaptive Clustering Testbed). ACT was written in Java, on top of MeMoJ and using RECODER[3] as fact extractor. Using ACT, we have made a comparative study of both the architecture-aware adaptive clustering technique and a conventional non-adaptive clustering technique.

The comparative study is based on two criteria: *Accuracy* and *Optimality*.

A recovered subsystem decomposition is considered accurate if it is "meaningful" to a software engineer. This means that the resulting subsystems should contain only semantically related architectural components and that all the semantically related architectural components should be in a single subsystem. We assess both techniques by comparing their resulting decompositions to specific reference decompositions (the original package structure and the ideal CRP structure) using the MoJo metric (see [5]). Further details and the argumentation for choosing these particular reference decompositions can be found in [1].

Optimality is measured using two metrics we have defined: *average cohesion* of the subsystems and *average coupling* between the subsystems of a given decomposition.

We have applied the above mentioned evaluation procedure on two case studies: *the Java AWT library* and *the SSHTools project* for three different parameters of the clustering algorithm.

For the *Java AWT library*, measurements show an average increase in accuracy (decrease of the MoJo value) of 19% for the architecture-aware adaptive clustering technique when comparing the decompositions to the original package structure and an average increase of 57% when comparing them to the ideal CRP structure. The average cohesion of the subsystems increased by 12% in the case of our approach. As for the average coupling values, they were slightly higher for our approach in 2 out of 3 experiments, but this is due to the fact that the non-adaptive approach created a much smaller number of large clusters thus turning many of the inter-cluster dependencies into intra-cluster dependencies.

In the case of the *SSHTools project*, the measurements revealed exactly the same thing as the ones made on the AWT library. The MoJo values show an average increase in accuracy for our approach of 23% when comparing the decompositions with the original package structure and an increase of 64% when comparing the decompositions with the ideal CRP structure. In the case of architecture-aware adaptive clustering, the optimality measurements show an average increase of 3% of the average cohesion metric and an average decrease of 10% of the average coupling metric.

The results presented in this section clearly show that architecture-aware clustering provides significantly better results than non-adaptive techniques both in terms of optimality and especially accuracy.

16.4 Conclusion

Our paper contributes to the software architecture recovery research by combining the strengths of clustering-based and pattern-based techniques. It proposes an approach which benefits from architectural clues that may be seen as traces of the high-level design of a system, the original software developers had in mind in early days of the system's life span. These clues are used to guide an adaptive clustering process to recover that architecture.

Additionally, we have introduced a new indirect coupling metric for measuring the strength of coupling given by common usage and, to our knowledge, we are the first to use it to effectively cluster together library code, thus providing an elegant solution to the problem of omnipresent entities encountered in other clustering approaches.

We feel that our results of using architecture-aware adaptive clustering are very encouraging and we believe that further research in that direction is fully justified.

Bibliography

[1] M. Bauer and M. Trifu. Architecture-aware adaptive clustering of OO systems. In *Proceedings of the Eighth CSMR*, pages 3–14. IEEE, 2004.

[2] R. Koschke. *Atomic Architectural Component Recovery for Program Understanding and Evolution*. PhD thesis, Institute of Informatics, University of Stuttgart, Oct 1999.

[3] S. Mancoridis, B. S. Mitchell, C. Rorres, Y. Chen, and E. R. Gansner. Using automatic clustering to produce high-level system organizations of source code. In *Proceedings of the Sixth IWPC*, pages 45–52. IEEE, 1998.

[4] K. Sartipi and K. Kontogiannis. A graph pattern matching approach to software architecture recovery. In *Proceedings of the ICSM*, pages 408–419. IEEE, 2001.

[5] V. Tzerpos and R. C. Holt. Mojo: A distance metric for software clustering. In *Proceedings of the Sixth WCRE*, pages 187–193. IEEE, 1999.

[3] RECODER is an open source Java framework for source code meta-programming jointly developed at FZI Forschungszentrum Informatik Karlsruhe and the University of Karlsruhe.

Erfahrungsberichte

17 Wartung von Standard-Software-Systemen am Beispiel von myToys.de

Michael Müller-Wünsch
myToys.de GmbH
muewue@myToys.de

17.1 Einführung

myToys.de wurde 1999 als Händler für Produkte (Bekleidung, Spielwaren, Babyartikel, usw) für Kinder und Jugendliche gegründet und ist heute einer der bedeutendsten Anbieter im Vertrieb dieser Artikel über das Internet.

Anfang 2000 wurde die Entscheidung getroffen, die IT-Anwendungssystem-Architektur auf Standard-Software-Systemen basieren zu lassen. Das Internet-Portal, das die gesamte Produktdarstellung der mehr als 45.000 Artikel und die Einkaufsfunktionalität zum Endkunden abbildet, wurde auf der Entwicklungsplattform Enfinity (http://www.enfinity.de/) von der Fa. Intershop realisiert. Für alle anderen betrieblichen Funktionen, wie bspw. Auftragsmanagement, Einkauf, Buchhaltung wurde eine Entscheidung zugunsten der ERP-Software Oracle Applications (http://www.oracle.com/lang/de/applications/) getroffen.

Auch wenn beide Systeme zum Zeitpunkt der Kaufentscheidung schon eine gewisse Marktpräsenz besaßen, mußten doch einige Konfigurations- und Anpassungsarbeiten zur Erfüllung der Erfordernisse des Geschäftssystems von myToys.de gemacht werden. So konnte nach ca. 5-monatiger Entwicklungszeit im September 2000 die gestaltete Anwendungssystemarchitektur in Betrieb genommen werden. Vergleicht man diesen Prozeß mit einem Hausbau und dem Einzug, dann begann sofort mit dem Einzug die Pflege und Wartung des Gesamtsystems.

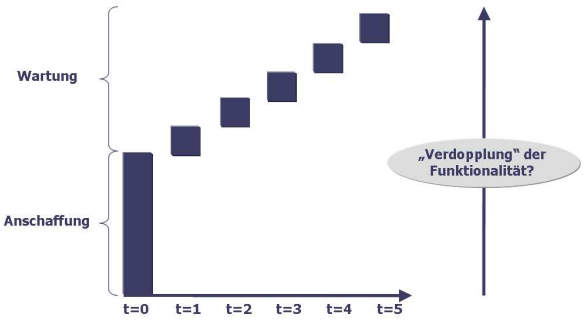
Der vorliegende Beitrag diskutiert anhand des Beispiels von myToys.de die Herausforderungen an das IT-Management bei der Einführung und beim Betrieb von Software-Systemen zur Erfüllung eines Geschäftszwecks. Dabei steht vor allen Dingen die wirtschaftliche Unterstützung von Ge- schäftsprozessen durch Software-Anwendungssysteme im Vordergrund, was bei einem internet-basierten Unternehmen wie myToys.de von herausragender Bedeutung ist.

17.2 Management der Software-Wartung

Mit dem Kauf von Standard-Software zur Unterstützung von Geschäftsprozessen als Alternative zur Individualprogrammierung trifft ein Unternehmen eine bedeutende Entscheidung über den zukünftigen Ressourceneinsatz.[1]

Neben den reinen Anschaffungskosten, die meistens über eine 3 bis 5-jährige Afa verteilt werden, fallen zusätzlich bis zur Inbetriebnahme die internen und externen Aufwendungen für die Software-Anpassungsarbeiten an. Nach Inbetriebnahme stellen die Hersteller von Standard-Software-Produkten ihren Kunden jährlich typischerweise durchschnittlich 20% des Anschaffungslistenpreises als Software-Wartungsgebühren in Rechnung. Mit diesem „Wartungsvertrag“ ist der Kunde normalerweise berechtigt, „kostenlos“ Weiterentwicklungen am Standard-Produkt einzusetzen (=Update-Berechtigung).

Allerdings bezahlt er dafür innerhalb von fünf Jahren nochmals den kom- pletten und nicht durch Einkaufsverhandlungen ggf. reduzierten Preis für die Software.



Nicht zwingend ist in diese Wartungsverträge die Möglichkeit eingeschlos- sen, auch „bedeutende“ Funktionserweiterungen zu nutzen (=Upgrade-Berechtigung). Hier läuft zur Zeit bspw. eine sehr intensive Diskussion im Umfeld der neuen Release-Politik SAPs (http://www.computerwoche.de/index.cfm?pageid=267&type=ArtikelDetail&id=80115594).

Inwieweit die neue, zusätzliche Funktionalität aus den Wartungsarbeiten des Software-Anbieters genutzt werden kann, hängt aber auch von den Anpassungen und Erweiterungen des Unternehmens während der Einführung des Standard-Software-Systems als auch später in der Betriebsphase ab. Häufig erweisen sich diese Arbeiten als Fallstricke zur problemlosen Adaption von funktionalen Neuerungen beim Anwenderunternehmen.

Für die Anwenderunternehmen bedeutet dies, daß sie sich ein gewisses Maß an Release-Fähigkeit erhalten müs-

[1] In der wirtschaftlichen Bewertung bleibt hier unbetrachtet die Diskussion zur Aktivierung von Standard-Software.

sen. Das führt oberflächlich dazu, die Software nur so einzusetzen, daß die Geschäftsprozesse standardisiert unterstützt werden. Hieraus kann aber unter Umständen folgen, daß es nur wenig Differenzierungspotential gegenüber anderen Wettbewerbern bei der Gestaltung der Geschäftsprozessen gibt.

Entscheidet man sich somit für den Einsatz von Standard-Software-Systemen, um (a) eine höhere Geschwindigkeit bei Time-To-Market zu erzielen, (b) ein gewisses Maß an Mindestqualität an funktionsfähiger Software voraussetzen zu können und (c) die Prozeßkosten des Geschäftsvorgangs zu reduzieren, dann sind diese Vorteile nicht zwingend nachhaltig.

Da im Nachgang zu einer solchen Investitionsentscheidung doch erhebliche Folgekosten in der Wartung anfallen, müssen neue Ansätze entwickelt werden, damit die IT-Kosten für Anwendungssysteme auch für die Zukunft beherrschbar bleiben.

17.3 Ausblick

Vielversprechend scheinen hier Entwicklungen im Kontext der OpenSource-Community zu sein. Auch wenn nach wie vor viele Unternehmen hier eine berechtigte Skepsis auf Praxistauglichkeit an den Tag legen, so sind doch die Erfolge insbesondere bei den Infrastruktursystemen (z.B. Fileserver, eMail-Server) beachtlich.

Im Bereich der betrieblichen Anwendungssysteme besteht zur Zeit die lei- der noch nicht objektivierbare Hoffnung, daß diese Software-Systeme eine höhere Qualität als bisherige proprietäre Systeme mit sich bringen und die Anschaffungs- und Wartungskosten deutlich unter denen vergleichbarer Software-Anbieter liegen könnten.

---

18 Heidelberg Eye Explorer - Die technologische Neuausrichtung Erfahrungsbericht aus einem Reengineering-Projekt

Martin Moro
Lehrstuhl für Wirtschaftsinformatik III, Universität Regensburg
Martin.Moro@wiwi.uni-regensburg.de

Abstract

In diesem Papier wird sowohl die gewählte Vorgehensweise als auch die erlebten Schwierigkeiten in einem Reengineering-Projekt vorgestellt. Der Schwerpunkt liegt dabei auf der Vermittlung von Erfahrungen und der Motivation für eine Standardisierung der Entwicklungsprozesse. Abgerundet wird mit Vorschlägen für einen Bewertungsprozess zur Unterstützung bei der Entscheidung zwischen divergierenden Ansätzen.

Schlüsselworte: Software Engineering, Prozess, nichtfunktionale Anforderungen, Qualität, Bewertung, Designalternativen

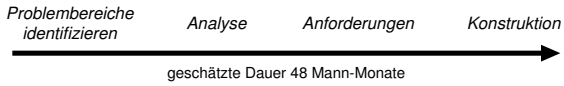
18.1 Einleitung und Problembereich

Die Firma Heidelberg Engineering entwickelt seit 10 Jahren Technologien zur Bildaufnahme im Bereich des Auges. Es entstanden Lasererkennungssysteme zur Abtastung von Auge, Netzhaut, Sehnerv und Blutgefäßen. Zur Visualisierung der Messdaten wurde die Software Eye Explorer [1] entwickelt.

Kunden von Heidelberg Engineering sind Augenarztpraxen und Kliniken. Während Augenarztpraxen nur ein Aufnahmegerät und einen PC benutzen, unterhalten Kliniken mehrere Aufnahmegeräte sowie eine netzwerkweite Software-Installation. Der historisch gewachsene Eye Explorer ist technologisch nicht für den Einsatz in Kliniken vorbereitet. Die wachsende Nachfrage gab aber den Impuls für ein Reengineering. Für die Bewältigung des Reengineerings wurde eine Kooperation aus der Firma Heidelberg Engineering, der Uniklinik Regensburg und dem Lehrstuhl für Wirtschaftsinformatik III geschlossen.

18.2 Der Weg zur neuen Softwarearchitektur

Schnell wurde klar, dass das Vorhaben weite Teile des Eye Explorers betreffen wird. Zu Beginn wurde daher folgende grundsätzliche Vorgehensweise überlegt:



18.2.1 Problembereiche identifizieren

Der erste Arbeitsschritt definierte anzustrebende Verbesserungen aus der Sicht des Softwareanwenders. Wichtig war die intensive Kommunikation mit allen Beteiligten, insbesondere mit dem Uniklinikum Regensburg. Ergebnis war eine Liste an konkreten Schwierigkeiten in der Anwendung des Eye Explorers. Unterstützung lieferte hierzu eine bereits bei der Firma Heidelberg Engineering geführte Liste an Supportanfragen. Viele Informationen stammen aus Interviews mit den Systembetreuern und Ärzten der Klinik.

Die Punkte auf der so erhaltenen Liste wurden priorisiert und nach ihrer Machbarkeit abgeschätzt. Einge-

brachte Wünsche zur Erweiterung der Funktionalität wurden aussortiert aber für eine spätere Berücksichtigung vorgemerkt. Relevante Anforderungen finden sich demnach vielmehr im nicht-funktionalen Sektor, bspw. eine Verbesserung der Flexibilität im Bezug auf unterstützte DB-Schnittstellen oder auch der Anbindung von Clients über das Internet.

18.2.2 Analyse

Ziel der Analyse war es, die Software zu verstehen und Probleme auf der Ebene des Sourcecodes zu erkennen. Als Struktur der bisherigen Softwarelösung wurde dabei extrahiert:

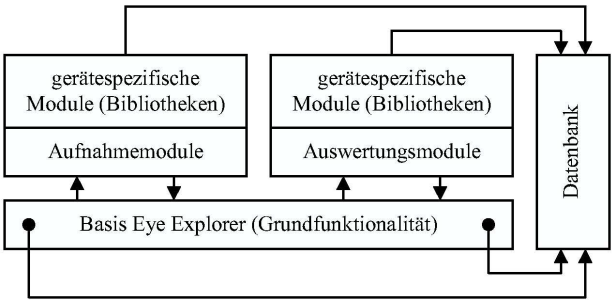


Abbildung 18.1: Architektur der bisherigen Software

Die wichtigsten Probleme ließen sich wiederkehrend in der Software nachweisen. Hauptproblempunkte waren die hochgradigen Verflechtungen innerhalb der Module (in der Abbildung 18.1 als Pfeile angedeutet), insbesondere der Zugriff auf die DB war verstreut. Viele Module benutzten eigene Datenstrukturen und speicherten Ergebnisse zwischen, so dass die DB nicht immer aktuell anzutreffen war. Die Probleme gaben einen Hinweis auf die betroffenen Softwaremodule (448.000 von 563.000 LOC, Anteil von ca. 80 %).

Im Projekt basierte die Analyse auf dem Sourcecode und geringer technischer Dokumentation. Dies erwies sich bei der Analyse durch Externe (Lehrstuhl) als Defizit, ein Softwaremodell wäre zum besseren Verständnis hilfreich gewesen. Kompensiert wurde dies durch eine Vielzahl an Gesprächen und Reviews.

18.2.3 Anforderungen

Neben den funktionalen Anforderungen waren die Qualitätsanforderungen ausschlaggebend:

1. Sowohl für Einzelplätze als auch für Netzwerke geeignet (ca. 80 % sind aber Einzelplätze).
2. Bisheriger Eye Explorer leicht migrierbar.
3. Datenbank flexibel austauschbar.
4. Auf wachsende Anforderungen skalierbar.
5. Offen für die Ankopplung angrenzender Systeme.

Zusätzliche Anforderungen des Reengineerings:

1. Die bereits entwickelten Module sollen beibehalten werden (Schutz des geleisteten Aufwandes).
2. Als Entwicklungssprache gilt weiterhin C++.

Erst die iterative Überarbeitung in einer Reihe von ausführlichen Gesprächen führte zu einer ausreichend vollständigen Liste. Für die Entscheidungsfindung einer optimalen Architektur war diese aber notwendig.

18.2.4 Konstruktion

Begonnen wurde bei der Konstruktion mit einer informellen Sammlung von Lösungsideen. Die potentiellen Ansätze wurden als Story-Cards notiert, um eine Diskussion am runden Tisch zu ermöglichen. Die Story-Cards enthielten dabei eine Bezeichnung der Lösungsidee, eine Beschreibung, eine schematische Zeichnung und eine Auflistung aller bis dahin bekannten Vor- und Nachteile im Projekt. Für einen schnellen Überblick bewährte sich ein Umfang von zwei Seiten. Im Projektverlauf wurden die Story-Cards (vgl. auch [2]) mit neuen Erkenntnissen bestückt, so dass mitunter auch ein Umfang von bis zu 50 Seiten zu finden war (vgl. Abbildung 18.2 für ein Beispiel einer Story-Card).

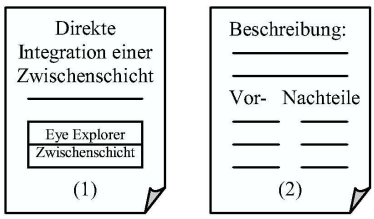


Abbildung 18.2: Beispiel einer Story-Card

Der Vergleich der verschiedenen Lösungsansätze stellte sich insbesondere bei der Prognose des Erfüllungsgrades an nicht-funktionalen Anforderungen als schwierig heraus. Wie sollte z.B. die geforderte Flexibilität sichergestellt werden ? Und: Wie sollte dies den Lösungsansätzen in dieser frühen Phase angesehen werden ? Gelöst wurde dies durch Rücksprache mit Experten, Recherchen in Erfahrungsberichten zu den Technologien lieferten wertvolle Informationen aus der Praxis. Verfahrensweisen zu denen keine Materialen gefunden werden konnten, wurden durch Prototypen verifiziert.

Als nützlich erwiesen sich Szenarios zur Quantifizierung nicht-funktionaler Anforderungen (vgl. [3, S.267], [4, S.33]). durch die jeweils interessierten Personen. Beispielsweise wurde die nicht-funktionale Anforderung der Flexibilität der DB-Schnittstellen auf Oracle, MySQL und MS SQL Server beschränkt. Die nicht-funktionalen „weichen“ Anforderungen wurden damit erst konkret greifbar. Ergebnis war eine Kriterienmatrix der Lösungsansätze mit einer Gegenüberstellung ihrer Eigenschaften. Der Vergleich führte zu der Erkenntnis, dass nur eine Mischung aus den Ideen zum Erfolg führen kann.

Als für dieses Projekt optimale Lösung etablierte sich eine Form einer nachrichtenbasierten Middleware. Diese ist auf die Arbeitsumgebung anpassbar. Für Einzelplätze ist sie direkt in die Software linkbar, Netzwerke lässt sie über Protokolle kommunizieren. Durch die Nutzung von C++ kann bereits entwickelter Sourcecode wiederverwendet werden. Die Verwendung des Protokolls XML ermöglicht eine Öffnung des Eye Explorers für angrenzende Systeme. Die Integration über den Linker hingegen sichert den Einzelplätzen die geforderte Leichtgewichtigkeit.

18.3 Die neue Softwarearchitektur

Für die neue Softwarearchitektur wurden die „Rosinen aus den zuvor gefundenen Lösungsideen gepickt“. Eingezogen wurde eine neue Plattform, die sowohl die fachlichen Klassen als auch die Kommunikation mit der DB kapselt (siehe Abbildung 18.3).

Fertiggestellt ist die neue Plattform sowie der Applikationsserver in seiner Grundfunktionalität, sowie der Test auf Funktionalität. Ausstehend ist der Test auf die Erfüllung nicht-funktionaler Anforderungen. Der Aufwand bis heute liegt bei ca. 14 Mann-Monaten. Noch durchzuführen ist die Integration der Plattform in den Eye Explorer. Bei einem geschätzten noch zu erbringenden Aufwand von 12 Mann-Monaten bewegt sich das Projekt über der Hälfte der Fertigstellung.

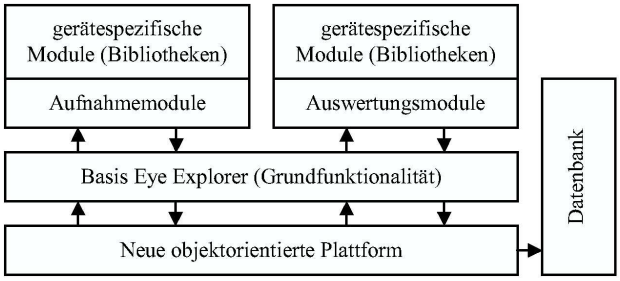


Abbildung 18.3: Struktur der neuen Softwarearchitektur

18.4 Standardisierungsmöglichkeiten im Prozess

Vorgehensmodelle wie der Rational Unified Process [5] helfen bei den ersten drei Tätigkeiten, aber wenig bei der Konstruktion. Insbesondere im vorliegenden Projekt waren Schwierigkeiten bei der Suche nach der optimalen Lösung zu verzeichnen.

Ein standardisierter Bewertungsprozess kann bei der Bewertung von Lösungsansätzen auf die Erfüllung nichtfunktionaler Anforderungen bereits während dem Softwareentwurf helfen. Er wendet visuelle Bewertungsverfahren und Metriken an, um Problembereiche aufzuspüren und Lösungsalternativen zu vergleichen. Eine zur Zeit entstehende Dissertation beschäftigt sich ausführlich mit diesem Thema.

18.5 Zusammenfassung und Ausblick

Das vorgestellte Reengineering-Projekt ist typisch für derartige Entwicklungsvorhaben. Die Balance aus der Ausrichtung auf neue Technologien und dem Schutz von entwickelten Komponenten zu finden war eines der Kernpunkte im Entwicklungsprozess. Schwierigkeiten bereitete weiterhin der Vergleich alternativer Lösungsideen. Die Prognose des tatsächlichen Verhaltens von Softwareentwürfen in der späteren Software ist ein schwer mit Werten zu stützendes Unterfangen.

Die Kernidee für eine optimale Lösung beruht auf dem Gedanken, während des Entwurfs in der Softwarearchitektur „Stellschrauben“ vorzusehen, an denen während der Implementierung bestehende Designdefizite ausgeglichen werden können. Dieser Gedanke scheint durchaus auf verwandte Projekte übertragbar. Die gewonnenen Erfahrungen fließen aktuell in eine Dissertation ein, die sich gezielt mit der Problematik der Bewertung von Softwareentwürfen beschäftigt.

Literatur

[1] Eye Explorer. `http://www.hdeng.de/h2e`, 2004

[2] Bennicke, Marcel; Rust, Heinrich: Messen im Software-Engineering und metrikbasierte Qualitätsanalyse. `http://www.visek.de`, 2004

[3] Rupp, Chris: Requirements-Engineering und -Management. Professionelle, iterative Anforderungsanalyse für die Praxis. München, Wien: Carl Hanser Verlag, 2002

[4] Clements, Paul; Bachmann, Felix; Bass, Len; Garlan, David; Ivers, James; Little, Reed; Nord, Robert; Stafford, Judith: Documenting Software Architectures - Views and Beyond. Addison-Wesley, 2002

[5] IBM Rational: Rational Unified Process Whitepapers. `http://www.ibm.com/software/rational`, 2004

---

19 C^{4}D oder Wie ich lernte, mit Code Clones zu leben

Udo Borkowski
IT Consultant, Noppiusstr. 3, 52062 Aachen
`udo.borkowski@gmx.de`

19.1 Einleitung

Das Auftreten von Code-Duplikaten (auch „Code Clones“ oder „Copy/Paste Code“ genannt) in Softwaresystemen wird allgemein als ein Qualitätsproblem betrachtet [1][3], das insbesondere auch bei der Wartung von großen Systemen an Bedeutung gewinnt. Das übliche Vorgehen zum Lösen diese Problems ist das (automatische) Finden der Code-Duplikate („Clone Detection“) und das anschließende „Herausfaktorieren“ des gemeinsamen Codes („Clone Removal“).

Verschiedene Techniken und Werkzeugen sind bekannt, die der Clone Detection dienen (eine Übersicht gibt z.B. [2]) und dabei auf unterschiedlichsten Konzepten beruhen. Zum Clone Removal werden im wesentlichen Makros [1] sowie Refactoring [5] eingesetzt [6].

Im folgenden werden einige Probleme aufgezeigt, die sich aus dem oben skizzierten Vorgehen, insbesondere durch den Clone Removal Schritt, ergeben. Es wird ein neues Verfahren „Code Clone Change Conflict Detection" (CCCCD oder C^4D) beschrieben, das diese Probleme vermeidet und weitere Vorteile aufweist.

19.2 Probleme bei Clone Detection/Removal

Ein wesentliches Problem von Code-Duplikaten liegt darin, dass i.A. nicht sichergestellt werden kann, dass während der Wartung eines Codefragmentes die entsprechenden Änderungen in allen eventuell vorhandenen Duplikaten durchgeführt werden. Hauptgrund hierfür ist, dass im Normalfall der zu ändernde Code keinen Hinweis auf Duplikate enthält. So hängt es von dem Hintergrundwissen des Entwicklers bzw. seiner „Weitsicht" ab, ob auch Duplikate geändert werden.

Durch den Einsatz von Makros bzw. durch Extrahieren von gemeinsamen Code in Funktionen (etwa durch das Refactoring „Extract Method" [4]) während des Code Removals wird diese Gefahr gebannt, da der Entwickler jetzt nur eine Codestelle ändern muss.

Allerdings hat dieses Code Removal Vorgehen auch Nachteile, von denen einige kurz aufgezeigt werden sollen:

Falsche Duplikate: die durch Clone Detection gefundenen und automatisch entfernten Duplikate können falsch in dem Sinne sein, dass sie nur zufällig aus gleichem Code bestehen. Dies kann zwei Folgen haben:

- wird die Inkonsistenz erkannt, so muss vor der Änderung des falschen Duplikates die Herausfaktorisierung rückgängig gemacht werden (z.B. durch „Inline Method" [4]).
- wird die Inkonsistenz nicht erkannt, führt der Entwickler wahrscheinlich einen Bug ein, da die Änderungen auch für die „Aufrufe" der falschen Duplikate durchgeführt werden.

Code-Duplikate-Auflösung eingeschränkt: nicht alle Code-Duplikate können mit diesem Vorgehen aufgelöst werden. Beispielsweise gibt es Schwierigkeiten bei überlappenden Codefragmenten.

Generierter Code: Das Auflösen von Codeklonen, die durch Generatoren entstanden sind, kann zu Schwierigkeiten beim späteren, inkrementellen Einsatz der Generatoren führen.

Hoher Aufwand beim Clone Removal: eine systematische, manuelle Entfernung von allen erkannten Code-Duplikaten ist in einem typischen Legacy-System in der Regel zu aufwendig, abgesehen von möglichen Fehlern, die ein manuelles Verfahren mit sich bringen würde.

Hoher Aufwand bei Wartung: Eine automatische Entfernung von Code-Duplikaten führt u.a. zu dem bereits oben beschriebenen Problem der falschen Klone. Um dies zu lösen, müssten die Entwickler bei Änderungen in Funktionen/Makros stets alle Aufrufkontexte überprüfen, da sie nicht davon ausgehen dürften, dass Funktionen entsprechend ihrer Semantik aufgerufen werden (sondern eventuell nur aufgrund ihres gemeinsamen Codes).

Verständlichkeit: die Verständlichkeit/Lesbarkeit von Code mit Makro- bzw. Funktionsaufrufen, insbesondere von automatisch generierten, kann sich gegenüber dem Original verschlechtern.

19.3 Code Clone Change Conflict Detection (C^4D)

Das Verfahren „Code Clone Change Conflict Detection" (CCCCD oder C^4D) vermeidet die oben beschriebenen Schwierigkeiten, indem es die während der Clone Detection gewonnenen Informationen nicht benutzt, um die Duplikate zu entfernen, sondern um zukünftige Änderungen potentieller Klone zu erkennen. Für diese Änderungen wird dann geprüft, ob sie konsistent in allen zugehörigen Duplikaten durchgeführt sind. Im Einzelnen wird dabei wie folgt vorgegangen:

1. Mit Hilfe von Clone Detection Werkzeugen wird eine Liste von potentiellen Klonpaaren erzeugt. Ein Klonpaar enthält dabei zwei Listen von Codefragmenten, die (exakte oder ähnliche) Kopien voneinander sind, sowie ggf. Parameterisierungsinformationen.
2. Bei nachfolgenden Änderungen am Code wird geprüft, ob sich diese Änderungen auf Codestellen beziehen, die in einem Codefragment eines potentiellen Klonpaares enthalten sind.
3. Wird eine solche Änderung entdeckt, folgt nun die Überprüfung, ob die Änderung auch entsprechend im Gegenstück-Codefragment des Klonpaares durchgeführt wurde. Sollte dies nicht der Fall sein („Change Conflict"), wird dem Entwickler die Situation mitgeteilt und er kann folgendermaßen darauf reagieren:
   (a) Der Entwickler entscheidet, dass die Änderung beide Codestellen betrifft und zieht die Änderung im Gegenstück ebenfalls nach. Hierbei wird er sich ggf. entscheiden, Makros bzw. Refactorings wie „Extract Method" einzusetzen, um so im folgenden die parallele Wartung mehrerer Codestellen zu vermeiden. An dieser Stelle liegt es in der Hand des Entwicklers, andere Nachteile von Code-Duplikaten (wie etwa Codegröße, mangelnde Abstraktion,...) zu beheben. Die Change Conflict Erkennung dient hier sozusagen als Auslöser.
   (b) Der Entwickler entscheidet, dass die Änderung nur die von ihm geänderte Codestelle betrifft.

Dies kann prinzipiell zwei Gründe haben:

i. Es handelt sich um einen falschen Klon (s.o). In diesem Fall wird das potentielle Klonpaar als „falsch“ markiert und bei den weiteren Prüfungen nicht mehr beachtet.
ii. Es handelt sich um eine Modifikation eines echten Klons, die aber nur für einen Teil des Klonpaares gilt. In diesem Fall wird die Modifikation als weiterer Parameter dem Klonpaar hinzugefügt, wobei der Parameterwert des Gegenstück-Codefragmentes leer ist.

Anmerkungen

Das Verfahren ist weitgehend unabhängig von den in 1. erwähnten Clone Detection Werkzeugen. Es ist darüberhinaus der parallele Einsatz mehrerer Werkzeuge möglich, so dass auf der Vereinigungsmenge der erkannten potentiellen Klonpaare gearbeitet werden kann. Überlappende Codefragmente sind dabei kein Problem. Auch ist das Verfahren weitgehend von der Programmiersprache unabhängig, soweit für diese ein passendes Clone Detection Werkzeug existiert.

Zu 2. ist anzumerken, dass eine Änderung einer Codestelle ggf. mehrere Klonpaare betreffen kann. Dies kann beispielsweise dann der Fall sein, wenn ein Clone Detection Werkzeug die transitive Hülle einer Klonklasse als Klonpaare ausgibt [2] oder wenn überlappende Clonefragmente vorliegen.

Bei 3. ist zu beachten, dass die Art der Prüfung von dem Klontyp [2] abhängt. Besonderes Augenmerk ist dabei auf Typ 2 und Typ 3 Klons und „ähnlichen“ Klons zu richten. Hierzu wird zuerst versucht, durch Vergleich der ursprünglichen Klonpaar-Codefragmente eine geeignete Parameterisierung des Klonpaares zu ermitteln. Die Parameterisierung geht dabei über eine reine Umbenennung von Namen hinaus, sondern deckt auch „Modifikationen“ ab. Dann wird bei den geänderten Codefragmenten geprüft, ob die Änderungen durch entsprechende Anpassungen von Code oder Parametern abdeckbar sind. Im Zweifelsfall wird eine „Ungleichheit“ gemeldet, die der Entwickler dann ggf. manuell auflösen kann.

Bei 3.a. ist erwähnenswert, dass eine Änderung im Gegenstück-Codefragment ggf. eine Änderung in einem weiteren Klonpaar darstellt, so dass sich hier der beschriebene Prozess ab 2. wiederholt. Dies ist insbesondere dann der Fall, wenn ein Klonpaar zu einer Klonklasse von mehreren Klonpaaren gehört.

Im Fall 3.b. sind zusätzlich Vorkehrungen zu treffen, dass durch die ausgewählten Maßnahmen andere Klonpaare, die eventuell mit dem betroffenen Klonpaar eine Klonklasse bilden, beachtet werden. Auch können dem Entwickler hier weitere Eingriffsmöglichkeiten angeboten werden, etwa das „Splitten“ in zwei Klonepaare u.ä.

19.4 Zusammenfassung

Es wurde eine neue Herangehensweise an das Code-Duplikat-Problem vorgestellt, die wesentliche Schwierigkeiten der klassischen Verfahren vermeidet. Darüberhinaus bietet sie weitere Vorteile, wie weitgehende Unabhängigkeit von Programmiersprachen und den verwendeten Clone Detection Werkzeugen, Behandeln von „nichtwohlgeformten“ Duplikaten, Einsetzbarkeit in großen Softwaresystemen usw.

Das neue Verfahren stellt darüber hinaus einen relativ einfachen Weg dar, den Umgang mit Code-Duplikaten in den Entwicklungsprozess zu integrieren und so „mit Code Clones zu leben“.

19.5 Ausblick

Das beschriebene Verfahren wird zur Zeit in einem Werkzeug umgesetzt. Von dessen Einsatz in produktiven Systemen werden weitere Einsichten in die Möglichkeiten und Grenzen von C^4D erwartet.

19.6 Literaturverzeichnis

[1] Baxter, Ira D., Andrew Yahin, Leonardo Moura, Marchelo Sant'Anna und Lorraine Bier: Clone Detection Using Abstract Syntax Trees. in Proc. ICSM, 1998.

[2] Bellon, Stefan: Vergleich von Techniken zur Erkennung duplizierten Quellcodes. Diplomarbeit, Universität Stuttgart, 2002.

[3] Ducasse, Stéfane, Matthias Rieger und Serge Demeyer: A Language Independent Approach for Detecting Duplicated Code. in Proc. ICSM, 1999.

[4] Fowler, Martin: Refactoring: Improving the Design of Existing Code. Addison-Wesley, 1999

[5] Opdyke, W.: Refactoring Object-Oriented Frameworks. PhD thesis, University of Illinois at Urbana-Champaign, 1992.

[6] Rysselberghe, F. Van and S. Demeyer: Evaluating Clone Detection Techniques. In proceedings of the International Workshop on Evolution of Large Scale Industrial Applications (ELISA). pages 25-36, 2003

20 Experimental Program Analysis

Holger Cleve
Andreas Zeller
Lehrstuhl für Softwaretechnik, Universität des Saarlandes, Saarbrücken, Germany
{cleve,zeller}@cs.uni-sb.de

Abstract

Program analysis long has been understood as the analysis of source code alone. In the last years, researchers have also begun to exploit tests and test results. But what happens if the analysis process itself drives the test, running actual *experiments* with the analyzed program? This position paper explores some of the possibilities that arise. As a first proof of concept, our *AskIgor* web service automatically isolates cause-effect chains for given failures—without any source code: "Initially, GCC was invoked with a C program to be compiled. This program contained an addition of 1.0; this caused an addition operator in the intermediate RTL representation; this caused a cycle in the RTL tree—and this caused GCC to crash."

20.1 Reasoning about Programs

When programmers attempt to understand a program, they use a wide variety of techniques to gather knowledge:

- At the core of things lies *deduction*—reasoning from the abstract program code to what can happen in a concrete program run. Typical deduction techniques include static analysis and verification.
- Deduction, by nature, does not take concrete facts into account—that is, facts from concrete program runs. These are extracted by *observation,* as exemplified in classical debugging tools.
- If a program is executed multiple times—in a test suite, for instance—one can attempt to *induce* abstractions from the concrete runs. Techniques that exploit induction are dynamic invariants (= summarizing multiple runs), or coverage metrics (= finding code that is executed in failing runs only)
- As part of the understanding process, programmers also *experiment* with the program in question—by generating and controlling multiple runs designed to support or refute hypotheses about the program. Experimentation is hardly ever found in tools.

As sketched in Figure 20.1, these reasoning techniques form a hierarchy—for instance, induction is not possible without observation, and each technique can put to use any static knowledge as deduced from the code.

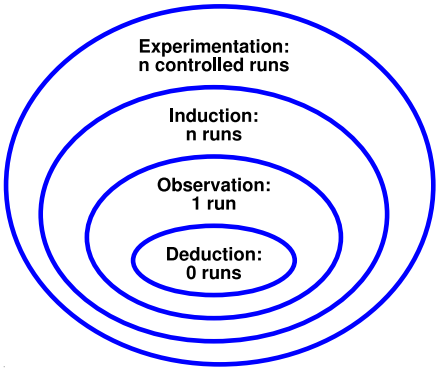


Figure 20.1: Program analysis: A hierarchy

20.2 Finding Causes

The one reasoning technique that makes use of all others is *experimentation*—the only one that can find out the *cause* of some effect. In fact, to prove causality, one needs two experiments: One where both cause and effect occur, and one where neither occur (with everything else unchanged, the cause preceding the effect, and both cause and effect being minimal). This is the way that experimental science finds the causes for natural phenomena.

Experimentation is the classical technique for *debugging:* If we want to know whether some variable x is the cause for some failure, we need to find an alternate value for x where the cause does not occur. To find such a value, and to know that changing x, instead of, say, y, is the great challenge in debugging a program—or, more generally, in understanding the cause of some effect.

One possible source for alternate values are alternate runs—that is, runs where the failure does not occur. Let us assume we have two runs: one run A where some effect occurs, and one run B where it does not. Let us further assume we interrupt program execution at some common point in the source code. If we now transfer the entire program state from A to B and resume execution, A's effect should occur in B, too. In fact, assuming deterministic execution and perfect transfer, B should behave exactly as A.

But what happens if we transfer only *part of the state* from A to B? This is an experiment whose outcome can hardly be predicted. For one thing, we would probably end up in an inconsistent state; resuming execution would lead

nowhere. However, if B now fails, one might argue that the part of the state just transferred was indeed the cause for the given effect. Using a simple strategy, this process can be repeated until the cause is narrowed down to some minimal difference between the two program states (Figure 20.2)—for instance, one pointer to the wrong element. All one needs is an automated test that checks whether the behavior of the modified run B is now A's (= the part transferred caused the effect) , still B's (= the part transferred does not cause the effect), or something different.

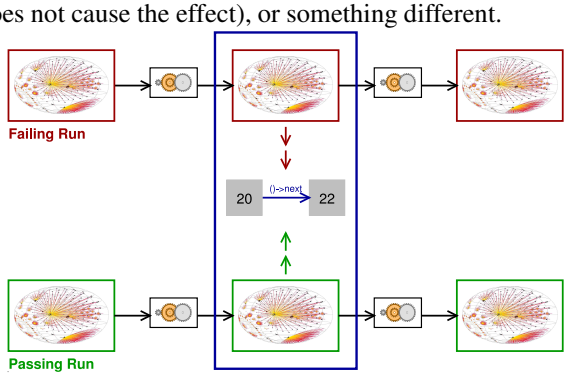


Figure 20.2: The process in a nutshell

Applying this technique at various points during the program execution eventually reveals the *cause-effect chain* from input to outcome—in the case of program failures, a short and concise diagnosis about how the failure came to be [1].

20.3 A Debugging Server

As a proof-of concept. we have built a *debugging server* called *AskIgor* ("Ask Igor")—a service that tells you why your program fails:

Submit a Program. You have a program that shows some repeatable, non-intended behaviour—for instance, the GNU compiler (GCC) crashing on some input.
You call up the *AskIgor* Web site and submit the *cc1* executable—the program that crashes. You also specify two invocations: one where the program fails, and one where it passes.

Read the Diagnosis. *AskIgor* presents the diagnosis on its Web page (Figure 20.3). The diagnosis takes the form of a *cause-effect chain*: First, this variable had this value, therefore, that variable got that value, and so on—until the program state causes the behaviour in question.
In our GCC example, the two inputs differ by the string " + 1.0" in the code (Step 1); this causes a *PLUS* operator in the intermediate RTL representation (Step 2: a new RTL node); this causes a cycle in the RTL tree (Step 3: *link* points back to itself)—and this cycle causes the compiler to crash (Step 4).

Fix the Bug. In order to fix the program, one must *break* the cause-effect chain—that is, ensure that at least one of the failure-inducing variable values no longer occurs. This is done by distinguishing *intended* from *non-intended* states—a decision left to you.
In our case, the non-intended program state is the cycle in the RTL tree. To find out how this state came to be, you can have *AskIgor* compute the cause-effect chain for the respective subsequence of the execution ("How did this happen?").

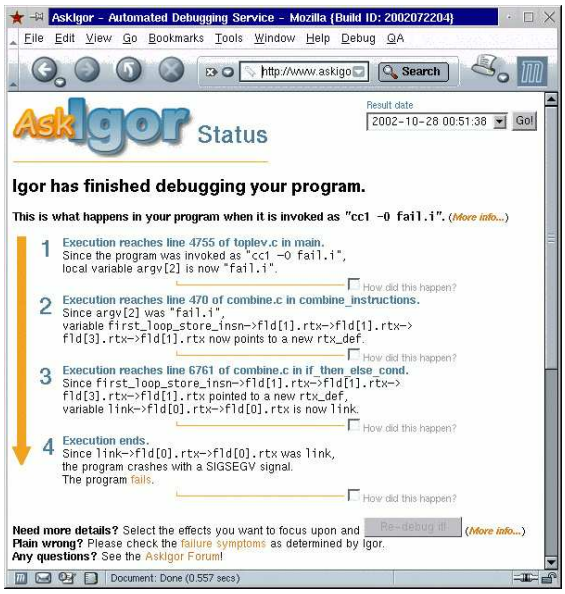


Figure 20.3: The *AskIgor* debugging server

The entire diagnosis is obtained by experimentation alone—no source code is required, and no abstraction from source code takes place.

20.4 Perspectives

Automated program analysis is much more than just analyzing code. In this paper, we show that automated experimentation opens several new perspectives for program analysis: Armed with just an automated test, one can automatically narrow down the causes of specific effects. All this is enabled by the wealth of computing power given to us; yet, we have only begun to combine the different reasoning techniques. This is a great time for investigation and cooperation.

More information about *AskIgor* and related work can be found on our web site

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

Bibliography

[1] Andreas Zeller. Isolating cause-effect chains from computer programs. In *Proc. ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE)*, pages 1–10, Charleston, South Carolina, November 2002.

21 Supporting Reverse Engineering Tasks with a Fuzzy Repository Framework

René Witte
Concordia University, Department of Computer Science
Montréal, Québec, Canada
me@rene-witte.net

Ulrike Kölsch
T-Systems International GmbH, Fasanenweg 5, Leinfelden, Germany
koelsch@acm.org

Abstract

Software reverse engineering (RE) is often hindered not by the lack of available data, but by an overabundance of it: the (semi-)automatic analysis of static and dynamic code information, data, and documentation results in a huge heap of often incomparable data. Additionally, the gathered information is typically fraught with various kinds of imperfections, for example conflicting information found in software documentation vs. program code.

Our approach to this problem is twofold: for the management of the diverse RE results we propose the use of a repository, which supports an iterative and incremental discovery process under the aid of a reverse engineer. To deal with imperfections, we propose to enhance the repository model with additional representation and processing capabilities based on fuzzy set theory and fuzzy belief revision.

Keywords: fuzzy reverse engineering, meta model, extension framework, iterative process, knowledge evolution

21.1 Knowledge Acquisition Process in RE

Reverse engineering can be described as a process of knowledge acquisition by proposing, validating, or falsifying hypotheses in order to form an abstract model. Deriving a conceptual model from a given implementation is a task fraught with ambiguity and vagueness because of the impedance mismatch problem.

Due to the large amount of information and the number of colliding hypotheses an automated support for the reverse engineers is highly needed. This automation must be able to keep track of the numerous analysis results, the proposed hypotheses as well as the interdependencies between all gained reverse engineering artifacts. Additionally, an automated support has to be able to detect and propagate information supporting or contradicting specific hypotheses or reverse model entities.

To our understanding the reverse engineering repository is the ideal point to offer such services. The meta model of a repository defines the representation abilities as well as the data manipulation capability of the reverse engineering process using it [1]. In order to improve and automate the RE process we are currently developing a fuzzy extension framework that can deal with imperfection and supports the continuing evolution of a knowledge basis with hypotheses based on formal concepts of fuzzy set theory and belief revision.

21.2 Knowledge Capturing and Representation

The RE repository and its meta model play a central role in establishing the technology that is most needed for dealing with imperfect information and hypotheses during reverse engineering. We developed a fuzzy extension framework that enhances the modeling capacity of RE meta models in such a way that imperfect data can now be handled.

The central requirement for such a model is the ability to handle several kinds of imperfections: *uncertain* and *vague* information, as well as the ability to handle *inconsistencies*. These requirements have also been identified by other research groups, e.g. in [2].

In this section we briefly outline the concept of our fuzzy extension framework for meta models of reverse engineering repositories. For a detailed description please refer to [3].

The formal basis for dealing with the mentioned kinds of imperfections is fuzzy set theory. For our fuzzy RE repository we deploy a fuzzy-set based model that has been developed especially for the use within information systems [4]. It provides the necessary features for supporting the requirements of the RE process, especially the ability to backtrack information and modify a knowledge base through non-monotonic belief revision operators.

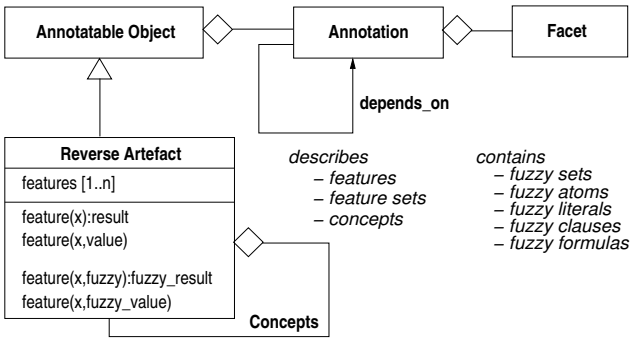


Figure 21.1: The generic fuzzy model

The key idea is to add imperfect information as an orthogonal extension through a frame-like concept, which we call *annotations*. Each object (or part of an object,

like an attribute) holding RE information can thus be *annotated* with container objects referring to one or multiple meta-information objects, called *facets*. Within our model, facets can store imperfect RE information in form of *fuzzy components*, like fuzzy sets, fuzzy literals, fuzzy clauses, or fuzzy formulas.

The annotation model offers the possibility to fuzzify individually all artifacts of a meta model, e.g. objects, relationships, features, and feature sets. As shown in Figure 21.1, every artifact inherits from the generic meta-object ANNOTATABLE OBJECT. Through inheritance all properties and operations of the fuzzy model now become available within the extended meta model.

21.3 Knowledge Consolidation and Hypothesis Support

The fuzzy enhanced repository model already allows us to store more detailed information for each RE artifact. Information from different sources (e.g., different analysis tools) can be stored together and, provided the tools have been adapted to the fuzzy format, automatically compared.

But this is only the first step in providing a semantically richer repository framework: we also need to support different computations based on the fuzzy data structures. For example, although the repository can store conflicting information about an artifact, such conflicts are not automatically recognized and resolved. But the sheer amount of data makes it impractical to simply rely on the reverse engineer to detect and resolve such conflicts. Instead, we want to support the RE process by defining higher-level methods that allow to specify relationships and interdependencies between artifacts. Gathered data will then automatically be compared with already existing information.

Finding and automatically resolving conflicts is achieved by executing *fuzzy belief revision* operators [5] on the gathered RE information. The idea of belief revision is to maintain a consistent knowledge base by removing (revising) existing information that cause a conflict with any added new information. By using fuzzified versions of these operators, we can allow for a varying *degree* of inconsistency within a repository: most often the information found by various tools in different sources will not match exactly due to inherent uncertainties in the used data or applied methods. We can even adjust the allowed degree of inconsistency over time: at the beginning of a RE process, where there is not much known about a system, it is permissible to start with a low degree of internal consistency, then gradually increasing it over time and finally converging to a single consistent view of the examined system.

To also allow such belief revision operators across *different* concepts and artifacts, we enhanced the structural model outlined in the last section with the concept of *fuzzy dependency graphs*. Dependency graphs show how the RE artifacts within a repository are connected to each other, they externalize knowledge about a system from a RE engineer by storing the dependencies between the artifacts in the repository. Annotations attached to RE artifacts become nodes in this graph, and directed edges represent dependencies between artifacts. An example for a dependency would be the relationship between the *type* of a program variable and its *usage* within the application. An additional data dictionary holds transformation functions that show precisely how a change to the information at one node affects the dependent nodes.

By using the dependency graph, information from one RE step is not simply stored in the repository, it can now trigger a graph revision operation that propagates changes throughout the repository. The performing RE engineer can adjust the required degree of consistency as outlined above and also select preferences on the available data, which will influence which information are removed in case of a conflict.

More about the theoretical foundations of this work can be found in [3, 4].

21.4 Conclusions and Future Work

Our approach deals with two important problems within the reverse engineering domain: the integration of heterogeneous results through a RE repository and the (semi-) automatic support of result consolidation and hypothesis support through fuzzy belief networks.

The core idea of our work is the acknowledgment that imprecision, inconsistency, and vagueness are unavoidable elements in reverse engineering. Instead of ignoring such imperfections, we provide tailored support for them based on the established theoretical foundation of fuzzy set theory. This allows us to deal with imperfect information explicitly, adding new capabilities to the RE domain.

With the theoretical and technical foundations in place, we are currently preparing experiments on several real-world examples that will combine information obtained through automated analysis of source code with a domain model gathered by text mining the program's documentation and specification through natural language analysis.

Bibliography

[1] Jean-Marie Favre, Mike Godfrey, and Andreas Winter, editors. *Preproceedings of the 1st International Workshop on Meta-Models and Schemas for Reverse Engineering*, Victoria, British Columbia, Canada, November 2003.

[2] Jens H. Jahnke and Andrew Walenstein. Reverse Engineering Tools as Media for Imperfect Knowledge. In *Proc. of the 7th WCRE*. IEEE Computer Society Press, 2000.

[3] Ulrike Kölsch and René Witte. Fuzzy Extensions for Reverse Engineering Repository Models. In *Proc. of the 10th WCRE*. IEEE Computer Society Press, 2003.

[4] René Witte. *Architektur von Fuzzy-Informationssystemen*. BoD, Norderstedt, Germany, 2002. ISBN 3-8311-4149-5.

[5] René Witte. Fuzzy Belief Revision. In *9th Intl. Workshop on Non-Monotonic Reasoning*, pages 311–320, Toulouse, France, April 19–21 2002. `http://rene-witte.net`.

22 Reverse-Engineering durch Identifikation von Eingabedaten-Äquivalenzklassen aus Programmabläufen

Rainer Schmidberger
Abteilung Software-Engineering, Institut für Softwaretechnologie, Universität Stuttgart
www.iste.uni-stuttgart.de/se

Kurzfassung

Die Programmlogik betriebswirtschaftlicher Software-Systeme implementiert häufig Entscheidungstabellen, die durch eine sehr direkte Verwendung der Eingabedaten als Bedingungskriterien geprägt sind. Speziell mit COBOL implementierte Systeme, die häufig seit vielen Jahren im Einsatz sind, zählen zu dieser Gruppe. Die ursprünglich zugrunde liegenden Entscheidungstabellen sind aber oft weder erhalten noch gepflegt worden. Zur wirtschaftlichen Wartung oder Neu-Implementierung sind diese Tabellen jedoch eine wichtige Voraussetzung.

Dieser Artikel beschreibt ein Verfahren, wie bei solchen Systemen, für die Programmcode sowie große Mengen von Eingabedaten verfügbar sind, die Entscheidungstabellen über Eingabedaten-Äquivalenz-klassen ermittelt werden können.

22.1 Einführung

Die funktionale Beschreibung eines Programms ist die Zuordnung von Ausgabedaten zu allen möglichen Eingabedaten. Sie kann daher prinzipiell aus einer Implementierung ermittelt werden, indem alle möglichen Eingaben bearbeitet und die Ausgaben gesammelt werden. Die Ausgabedaten definieren Äquivalenzklassen der Eingabedaten; diese können in Form einer Tabelle – der Entscheidungstabelle – dargestellt werden.

Praktisch ist dieser Weg in der Regel nicht gangbar, weil die Menge der Eingabedaten weit größer ist, als über beherrschbare Tabellen darzustellen wäre. Selbst bei einem kleinen Programm mit nur wenigen Integer-Eingaben gibt es 10^{20} Einträge und mehr.

Bei dem in der Folge beschriebenen Verfahren zur Ableitung der Entscheidungstabelle für Systeme mit vielen Eingabedaten werden anstelle aller möglichen Eingabedaten die folgenden zwei Informationsquellen verwendet:

1. der Programmcode der untersuchten Software und hier speziell die Verzweigungen und deren Bedingungsterme
2. die existierenden Eingabedaten des regulären Betriebs

Offensichtlich stellen die existierenden Eingabedaten des regulären Betriebs nur einen Bruchteil der theoretisch vollständigen Eingabedatenmenge dar; innerhalb der Menge der fachlich sinnvollen Eingabedaten ist ihr Anteil aber typischerweise relativ hoch.

22.2 Definitionen

Sei d_i ein vollständiger Eingabedatensatz für das Programm P. Mit dem Datensatz d_i durchläuft P deterministisch einen ganz bestimmten Programmpfad, den Pfad p_i. Der Pfad ist die vollständige Sequenz der ausgeführten Anweisungen inklusive Wiederholungen usw. Wir bilden nun Äquivalenzklassen von Eingabedatensätzen. Zwei Eingabedatensätze d_i und d_j sind schwach äquivalent, wenn die beiden zugeordneten Pfade p_i und p_j gleich sind.

Jeder Pfad ist bestimmt durch den Eintrittspunkt in das Programm und durch die Sequenz von Entscheidungen, die an den Verzweigungen getroffen werden. Eine Entscheidung an einer Programmverzweigung besteht, wie von Frühauf, Ludewig und Sandmayr beschrieben, aus einem oder mehreren Termen [1].

Wir gehen hier davon aus, dass jedes Programm genau einen Eintrittspunkt hat und dass Verzweigungen die Form von if- oder case-Anweisungen haben. Schleifen werden noch gesondert betrachtet.

Sei beispielsweise ein kurzes Programm (COBOL-Syntax)

```
0010 IF A >= 5 AND <=9
0020   STATEMENT1
0030 ELSE
0040   STATEMENT2
0050 END-IF
```

Wenn A den Wert 2 hat, liefert uns die Ausführung des Programms einen Pfad (`0010`, `0040`), die Sequenz von Termen, die den Pfad bestimmen und außerdem den Hinweis, dass es zu diesem Term weitere Pfade, mindestens einen weiteren Pfad, für den Fall gibt, dass die auf A angewendeten Terme ein anderes Resultat liefern. Da wir zunächst nicht wissen, wie viele Pfade sich hier verbergen, sprechen wir unbestimmt von einem „dunklen Pfad“.

Zwei schwach äquivalente Eingabedatensätze d_i und d_j sind stark äquivalent, wenn alle angewendeten Terme das gleiche Resultat ergeben, wenn also die annotierten, d.h. um die ausgewerteten Terme erweiterten Pfade, gleich sind. Im Beispiel oben wäre ein Datensatz dem ersten stark äquivalent, wenn $A = 4$ enthielte, denn der annotierte Pfad ist in beiden Fällen `0010` ($A \geq 5 \rightarrow$ `FALSE`, $A \leq 9 \rightarrow$ `TRUE`), `0040`. Dagegen wäre $A = 10$ nur schwach äquivalent, der annotierte Pfad ist `0010` ($A \geq 5 \rightarrow$ `TRUE`, $A \leq 9 \rightarrow$ `FALSE`), `0040`.

22.3 Verfahren zur Ermittlung von Eingabedaten-Äquivalenzklassen

Die These dieser Arbeit ist, dass stark äquivalente Eingabeklassen jeweils einem Fall in einer Entscheidungstabelle entsprechen. Darum werden viele, womöglich alle verfügbaren Eingabedatensätze verarbeitet und die annotierten Pfade verglichen. Wenn ein neuer annotierter Pfad entdeckt wird, kann er mit den Variablenwerten, die in den Termen verwendet wurden, als neuer Fall gespeichert werden.

Wenn noch kein einziger Datensatz angewendet wurde, stellt das gesamte Programm einen dunklen Pfad dar. Jede Ausführung (d.h. jeder Programmlauf mit einem weiteren Datensatz) erweist sich entweder als stark äquivalent zu einer früheren Ausführung oder beseitigt einen dunklen Pfad, indem sie einen neuen annotierten Pfad und im der Regel auch neue dunkle Pfade liefert.

In der Praxis ist natürlich nicht damit zu rechnen, dass am Ende alle dunklen Pfade beseitigt sind. Für die verbliebenen dunklen Pfade können unterschiedliche Ursachen vorliegen:

1. Der gefundene dunkle Pfad ist fachlich durchaus sinnvoll und gewollt. Es war nur kein entsprechender Eingabedatensatz angewendet worden.
2. Der gefundene dunkle Pfad ist bedeutungslos geworden, weil er eine Ausnahmebehandlung für Eingabedatensätze darstellt, die so nicht mehr existieren (weil es diese Eingabedatensätze z.B. nur zeitlich befristet gab)
3. Es handelt sich um eine Fehlerbehandlung für einen nicht aufgetretenen Fehler
4. Es handelt sich beim dunklen Pfad um Programmcode, der für sinnvolle Eingabedaten nicht erreichbar ist und damit unsinnig ist.

Welche dieser Ursachen vorliegt, kann automatisch nicht erkannt werden. Hierzu sind Programmdurchsichten erforderlich.

Schleifen werden derzeit noch nicht automatisch erkannt und behandelt. In der aktuellen Umsetzung wird speziell eine äußere Programmschleife, die über die Eingabedaten iteriert, so behandelt, dass die Programmzeilen für Schleifenein- und austritt von Hand festgelegt werden. Mit dem Eintritt in diese Schleife beginnt die Betrachtung des Pfades für den jeweiligen Datensatz, und mit dem Austritt endet sie wieder.

22.4 Technische Umsetzung

Die technische Umsetzung wurde für die Programmiersprache VS COBOL II vorgenommen. Der gesamte Ablauf gliedert sich in drei Abschnitte:

1. Parsen, Analysieren und Instrumentieren des COBOL-Programmcodes. Der Parser wurde über eine verfügbare COBOL-Grammatik und den Java-Compiler-Compiler erstellt.
2. Durchführen der Programmabläufe im instrumentierten Programmcode mit den Eingabedaten. Protokollieren der Pfade und Bedingungsterme in einer relationalen Datenbank.
3. Ermitteln der Entscheidungstabellen durch Datenbank-Auswertung: Stark äquivalente Eingabedaten werden in der Entscheidungstabelle zu einer Zeile zusammengefasst. Alle auf die Eingabedaten angewendeten Bedingungsterme bilden den Eintrag der linken Spalte. Der Pfad die rechte Spalte.

22.5 Zusammenfassung der Ergebnisse

Eingabedaten-Äquivalenzklassen können über das beschriebene Verfahren aus einem bestehenden Programmcode mit verfügbarer Grammatik sowie regulären Produktionsdaten gewonnen werden. Für COBOL-Programme wurde eine exemplarische Umsetzung erfolgreich implementiert. Der Einsatz in einem konkreten Projekt mit etwa 40 KLOC läuft derzeit.

Die automatische Behandlung von Schleifen, „goto“-Verzweigungen und Exceptions hinsichtlich der Auswirkung auf die Entscheidungstabelle wird noch weiter untersucht.

Literatur

[1] Frühauf K., Ludewig J., Sandmayr H.,: Software-Prüfung. vdf Hochschulverlag AG, Zürich, 1997

23 Adaptive Erkennung von Entwurfsmängeln in objektorientierter Software

Jochen Kreimer
Universität Paderborn, Institut für Informatik, Fürstenallee 11, 33102 Paderborn
jotte@uni-paderborn.de

23.1 Einleitung

Qualität von Software. Die Qualität von Software kann je nach Anwendungsgebiet an unterschiedlichen Kriterien gemessen werden. Für große Software-Systeme spielen u. a. Kriterien wie Wartbarkeit, Verständlichkeit und Erweiterbarkeit eine wichtige Rolle [1].

Entwurfsmängel erkennen. Unser Ziel ist es, Fehler im Entwurf von Software-Systemen zu erkennen und somit „schlechte“ — unverständliche, schwer erweiter- und änderbare — Programmstrukturen zu vermeiden.

Diese Fehler nennen wir Entwurfsmängel. Sie stehen eng mit Eigenschaften der Programmstruktur in Beziehung und sind daher unterhalb der System-Architektur anzuordnen. Eine Vielzahl von Entwurfsmängeln finden sich in der Literatur als *„Design Heuristics“* [7], *„Design Characteristics“* [10] oder *„Bad Smells“* [3]. Die Autoren bezeichnen Entwurfsmängel i. d. R. durch sprechende Begriffe und erklären dem Software-Entwickler wie solche Entwurfsmängel erkannt und behoben werden können.

Prominente Entwurfsmängel sind z. B. „Große Klasse“ oder „Lange Methode“.

Da die Suche zeitaufwändig und vom persönlichen Empfinden und Erfahrungsschatz des Suchenden abhängig ist, bietet sich eine Werkzeugunterstützung an. Werkzeuge wie *„CodeCrawler“* [5] und *„Crocodile/CrocoCosmos“* [8] visualisieren quantitative und strukturelle Eigenschaften eines analysierten Programms und bieten dem Suchenden unterschiedliche Sichtweisen zur Analyse großer Programme an.

Unser Ziel ist es jedoch, dem Benutzer die Interpretation und Analyse der Programmeigenschaften abzunehmen und automatisiert Hinweise auf Entwurfsmängel zu liefern. Dabei soll ein flexibler Interpretationsmechanismus eingesetzt werden, der sich den speziellen Anforderungen und Konventionen eines Projektes, einer Entwicklergruppe oder eines Anwendungsgebietes anpasst.

23.2 Adaptive Erkennung

In diesem Ansatz kombinieren wir Metrik-basierte Ansätze und Maschinelle Lernverfahren zu einem adaptiven Verfahren.

Metriken. Wir folgen dem Ansatz aus [6] und ordnen jedem Entwurfsmangel eine Menge von Programmeigenschaften zu, die wir durch objektorientierte Entwurfsmetriken ausdrücken. Häufig werden Größen-, Komplexitäts-, Kopplungs- oder Kohäsionsmetriken verwendet.

Wir setzen statische Programmanalyse mit klassischen Methoden der Kontroll- und Datenflussanalyse sowie der abstrakten Interpretation ein. Die Analyseergebnisse führen u. a. zu einem Programmabhängigkeitsgraphen der als abstraktes Modell des Analysegegenstandes dient und zur Metrikberechnung genutzt wird.

Maschinelle Lernverfahren. Bei Lernverfahren [11, 4] werden statistische Methoden eingesetzt, um aus einer gegebenen Menge von Beispiel-Instanzen zu „lernen“. Man spricht hier von einer Trainingsphase.

Eine Instanz besteht aus einer Menge von Attributen, denen Werte zugeordnet sind. Ein spezielles Attribut, das Zielattribut, beschreibt das Lernziel.

Das Lernverfahren erstellt aus dem Trainingsinstanzen einen Klassifizierer, z. B. in Form eines Entscheidungsbaumes, der bereits erlernten und zukünftigen Instanzen ein Zielattribut zuordnet.

Adaption. Wir verwenden Lernverfahren, um für jeden Entwurfsmangel einen speziellen Klassifizierer zu erstellen. Die Menge von Metriken, die dem Entwurfsmangel zugeordnet sind, bilden die Instanzen für das Lernverfahren. Als Zielattribut wird die Information verwandt, ob ein Entwurfsmangel vorliegt oder nicht.

So lassen sich zunächst Ergebnisse der Metrikberechnung von Programmobjekten, die bekanntermaßen Entwurfsmängel enthalten, als Trainigsbeispiele verwenden.

Später können ganze Systemteile analysiert und nach Entwurfsmängeln durchsucht werden. Hierzu werden die Ergebnisse der Metrikberechnungen aller enthaltenen Programmobjekte den entsprechenden Klassifizierern übergeben, die dann im Zielattribut einen Entwurfsmangel anzeigen.

23.3 Werkzeug für *Eclipse*

Wir haben das prototypische Werkzeug *„IYC — It's Your Code“* als *Plugin* für die freie Entwicklungsumgebung *Eclipse* [2] entwickelt.

Das Werkzeug analysiert Java-Bytecode und erstellt, neben einigen spezialisierten Analysen, einen Programmabhängigkeitsgraphen.

Als maschinelles Lernverfahren setzen wir den „J48“-Klassifizierer aus der *WEKA*-Bibliothek [9] ein.

Der Benutzer kann in Konfigurationsdialogen Entwurfsmängel definieren, indem er vorgegebene Metriken zuordnet. Die Benutzeroberfläche von *Eclipse* erlaubt Programmobjekte in unterschiedlichen Sichten auszuwählen. In Kontextmenüs können Trainings- und Analysevorgänge angestoßen werden.

Gefundene Entwurfsmängel werden dem Benutzer präsentiert. Dieser kann einzelne Vorschläge abweisen oder akzeptieren und damit das Lernverfahren weiter trainieren.

23.4 Beobachtung und Ausblick

Diese Entwurfsmangelerkennung wird auf drei Ebenen beeinflusst:

1. Die Möglichkeiten einen Entwurfsmangel zu modellieren wird durch die Menge der Analysen bzw. Metriken, die zur Modellierung von Entwurfsmängeln bereitgestellt werden, beeinflusst.
2. Die Modellierung der einzelnen Entwurfsmängel durch Auswahl der zu benutzenden Metriken liegt in der Hand des Benutzers. Hier wird die grobe Charakteristik eines Mangels festgelegt.
3. Der Trainings- und Lernprozess erlaubt die feine Einstellung ohne dass der Benutzer eigene Grenzen für Metriken und deren Verknüpfung spezifizieren müsste.

Die Wirksamkeit des vorgestellten Verfahrens muss durch empirische Untersuchungen gezeigt werden. Hierzu suchen wir noch Projektpartner.

Literaturverzeichnis

[1] F. Abreu and R. Brito. Objectoriented Software Engineering: Measuring and Controlling the Development Process, 1994.

[2] Eclipse.org Consortium, `http://www.eclipse.org`. *Eclipse.org Main Page*, 2003.

[3] Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts. *Refactoring: Improving the Design of Existing Code*. Addison Wesley, 1999.

[4] T. Hastie, R. Tibshirani, and J. Friedman. *The Elements of Statistical Learning.* Stats. Springer, 2001.

[5] Michele Lanza. *Object-Oriented Reverse Engineering — Coarse-grained, Fine-grained, and Evolutionary Software Visualization*. PhD thesis, University of Berne, May 2003.

[6] Radu Marinescu. *Measurement and Quality in Object-Oriented Design*. PhD thesis, University of Timisoara, 2002.

[7] Arthur J. Riel. *Object-Oriented Design Heuristics*. Addison Wesley, 1996.

[8] Frank Simon, Frank Steinbruckner, and Claus Lewerentz. Metrics Based Refactoring. In *CSMR*, pages 30–38, 2001.

[9] Weka 3 — Data Mining with Open Source Machine Learning Software in Java, `http://www.cs.waikato.ac.nz/~ml/weka/`.

[10] Scott A. Whitmire. *Object Oriented Design Measurement*. John Wiley & Sons, Inc., 1997.

[11] Ian H. Witten and Eibe Frank. *Data Mining*. Morgan Kaufmann, Los Altos, US, 2000.

---

24 Automatisierte Behebung von Strukturproblemen in objektorientierten Systemen

Adrian Trifu
Olaf Seng
Thomas Genssler
FZI Forschungszentrum Informatik, Karlsruhe, Germany
{trifu,seng,genssler}@fzi.de

24.1 Einführung

In diesem Papier stellen wir einen integrierten, qualitätsgetriebenen und werkzeugunterstützten Ansatz zur Evolution objektorientierter Systeme vor, der dazu beiträgt die existierende Lücke zwischen der Suche von Strukturproblemen und ihrer Behebung zu schließen. Unser Ansatz basiert auf dem neuen Konzept der “Correction Strategies” [TD03], und existierenden Arbeiten zur Problemerkennung und zur Codetransformation [Ciu99], [GK03].

24.2 Ansatz

Unser Ansatz besteht aus drei Phasen. In der ersten Phase - der Problemerkennung und -analyse - wird zuerst ein Qualitätsziel angegeben, und Strukturprobleme, die dieses Ziel negativ beeinflussen können werden vorgeschlagen. Um die Strukturprobleme im Quelltext zu suchen und zu finden, benutzen wir bereits existierende Methoden und Werkzeuge. Sobald ein oder mehrere Strukturprobleme gefunden und bestätigt wurden, wird in der Lösungsanalyse-

phase mit Hilfe der “Correction Strategies” ein Restrukturierungsplan erstellt, der in der abschließenden Restrukturierungsphase umgesetzt wird. Im Folgenden werden wir auf diese drei Phasen näher eingehen.

24.2.1 Problemerkennung und -analyse

Die Auswahl der Typen von Strukturproblemen, nach denen gesucht werden soll, ist von einem vom Benutzer vorgegebenen Qualitätsziel abhängig. Ein Qualitätsziel besteht dabei aus einer Gewichtung vorgegebener Qualitätsfaktoren wie z.B. Erweiterbarkeit, Flexibilität, Verständlichkeit ...

Für jeden Strukturproblemtyp, den wir erkennen können, wurde von uns vorgegeben, welchen Einfluss ein Auftreten eines solchen Problems auf die Qualitätsfaktoren hat. Qualitätsfaktoren und Einfluss der Strukturprobleme auf diese wurden vorerst einmal ad hoc bestimmt. Genauere Werte könnte man durch die Durchführung von Fallstudien und Experimenten und Recherchen in der Literatur gewinnen, was momentan im Rahmen des Forschungsprojektes QBench[1] geschieht.

Oft ist es nicht möglich, alle gefundenen Strukturprobleme zu entfernen, da die Behebung jedes einzelnen zwar werkzeuggestützt aber nicht vollautomatisch erfolgen kann, und somit relativ viel Aufwand benötigen kann. Die Gewichtung der Strukturprobleme macht es möglich, zuerst diejenigen zu beheben, die den größten Einfluss auf das Qualitätsziel haben.

Abhängig vom Kontext der Software (z.B. Echtzeitsystem, Prototyp, ...) kann dem Benutzer ein vordefiniertes Qualitätsziel vorgeschlagen werden, das im allgemeinen für Anwendungen des gewählten Kontexts geeignet ist. Der Kontext hat zudem Einfluss auf die Auswirkungen der Problemstrukturen auf die Qualitätsfaktoren und die Schwellwerte, die zur automatischen Erkennung der Strukturprobleme verwendet werden. Die automatisch gefundenen Strukturprobleme müssen manuell als solche bestätigt werden, damit im nächsten Schritt eine Lösung für jedes einzelne gefunden werden kann.

24.2.2 Lösungsanalyse

Ziel der Lösungsanalyse ist es, eine Menge von Restrukturierungen abzuleiten, die das vorhandene Strukturproblem entfernen und die gewählten Qualitätsfaktoren verbessern. Hierzu werden sogenannte “Correction Strategies” eingesetzt. Eine “Correction Strategy” ist eine strukturierte Beschreibung, die einem gegebenen Strukturproblem eine Menge von möglichen Lösungen zuordnet. Beispielsweise kann eine *Gottklasse* sowohl durch ein Aufteilen der Klasse in eine oder mehrere Klassen als auch durch reines Verschieben von Methoden und Attributen entfernt werden. Jede dieser potentiellen Lösungen kann dabei aus mehreren Teillösungen und Entscheidungen zwischen diesen aufgebaut sein.

Da für jede der möglichen (Teil)Lösungen die Auswirkungen auf die Qualitätsfaktoren bekannt ist, kann Schritt für Schritt die Lösung berechnet werden, die einen möglichsten guten Einfluss auf die Qualitätsfaktoren hat. “Correction Strategies können mit einem Expertensystem verglichen werden.

24.2.3 Reorganisation

Zur Reorganisation wird ausgehend vom Quelltext ein zweischichtiges Modell aufgebaut. Die obere Schicht bietet eine konzeptuelle Sicht auf objektorientierte Sprachen. Diese Sicht enthält typische Entwurfseinheiten wie Klassen und Methoden und Beziehungen zwischen diesen wie z.B. Methodenaufrufe und Attributzugriffe. Auf dieser Ebene sind eine Menge von Transformations- und Analyseoperationen definiert. Mit Hilfe der Analyseoperationen können Modellelemente mit bestimmten Eigenschaften ausgewählt und Vor- und Nachbedingungen von Transformationen spezifiziert und überprüft werden.

Konzeptuelle Transformationen sind Grundoperatoren zur Modifikation eines Modells. Sie reichen von einfachen Veränderungen (z.B. ein Attribut zu einer Klasse hinzufügen) bis zu komplexeren Refactorings (z.B. eine Methode von einer Klasse in eine andere zu verschieben). Diese konzeptuellen Transformationen sind atomar und garantieren Korrektheit in Bezug auf Syntax und statische Semantik der zugrunde liegenden Programmiersprache. Es können weitere Vor- und Nachbedingungen angegeben werden, mit denen man zusätzliche Nebenbedingungen formulieren, die z.B. Verhaltensbewahrung garantieren.

Die untere Schicht stellt die Verbindung zur Semantik der tatsächlichen Sprache her. Die konzeptuellen Einheiten werden auf die entsprechenden Einheiten der Sprache abgebildet, und es werden sprachabhängige Vor- und Nachbedingungen bereitgestellt. Momentan ist die untere Schicht für JAVA spezifiziert und implementiert, eine Unterstützung von C#[2] ist in Arbeit.

Zur Spezifikation komplexer Refactorings kann eine Skriptsprache namens Inject/J benutzt werden. Diese Skriptsprache erlaubt die einfache Navigation in einem Modell, die Auswahl bestimmter Modellelemente anhand ihrer Eigenschaften, und die Ausführung von Transformationen.

24.3 Werkzeugunterstützung

Jede der Phasen unseres Ansatzes wird durch Werkzeuge unterstützt. Der Advanced Refactoring Wizard (ARW) dient als Integrationsplattform für unsere Problemerkennungs- und Refactoring-Technologie und führt den Nutzer durch den gesamten Prozess.

Zur Problemsuche benutzen wir ein Opensource-Werkzeug namens jGoose[3], mit dem momentan nach 18 Strukturproblemen wie z.B. Gottklassen oder Flaschenhalsklassen gesucht werden kann. Die Lösungsanalyse er-

[1] `http://www.qbench.de`
[2] `http://recoder-cs.sf.net`
[3] `http://jgoose.sf.net`

folgt mit einem prototypischen Werkzeug, das momentan "Corrections Strategies" für vier Strukturprobleme unterstützt. Zur Behebung der Strukturprobleme verwenden wir ein Werkzeug namens Inject/J[4]. Alle drei Werkzeuge wurden am FZI entwickelt. Zur Zeit können wir mit unserer Werkzeugkette den kompletten Prozess für Java abdecken. Die Problemsuche und die Lösungsanalyse sind sprachunabhängig für objektorientierte Programmiersprachen gehalten, vorausgesetzt ein entsprechender Faktenextraktor existiert. Momentan unterstützen wir hier JAVA und C++. Um die Reorganisation für mehrere Sprachen zu unterstützen ist natürlich viel mehr Aufwand erforderlich. Zur Zeit arbeiten wir an einer Unterstützung von C#.

24.4 Fallstudie

Unser Werkzeug konnte auf eine in JAVA geschriebene Fallstudie kleiner bis mittlerer Größe (63 Klassen, 34000 LOC) erfolgreich angewandt werden. Eine der in diesem System gefundenen Gottklassen erwies sich als besonders problematisch. Sie enthielt 71 Methoden und ca. 1400 LOC, und stand in direkter zyklischer Abhängigkeit zu einer anderen Klasse. Bei der Lösungsanalyse musste zwischen den Möglichkeiten Klasse aufteilen und Methoden/Attribute verschieben entschieden werden. Da zusätzlich Methoden der Gottklasse als deplatzierte Methoden identifiziert werden konnten, und das Anlegen einer neuen Klasse die Komplexität erhöht hätte, wurde entschieden einen Teil der Methoden zu verschieben. In diesem Fall konnte das Verschieben mit Inject/J vollautomatisch durchgeführt werden, da es sich bei den zu verschiebenden Methoden um statische Methoden handelte. Die direkte zyklische Abhängigkeit konnte damit ebenfalls beseitigt werden.

24.5 Zusammenfassung und Ausblick

In diesem Papier haben wir eine integrierte, qualitätsgetriebene Methode zur Evolution objektorientierter Softwaresysteme vorgestellt. Ein wichtiger Bestandteil unserer Methode sind die sogenannten "Correction Strategies", mit denen die sichere Behebung von Strukturproblemen unter Berücksichtigung eines vorgegebenen Qualitätsziels geplant werden kann.

Einer unserer nächsten Schritte besteht in der Ausarbeitung von "Correction Strategies" für weitere Strukturprobleme, und eine Evaluation anhand weiterer realer Fallstudien. Zudem arbeiten wir an einer verbesserten, feingranulareren Version unseres Qualitätsmodells. Ein weiterer Punkt ist die Frage nach einer globalen Optimierung. Momentan betrachten wir jedes Strukturproblem getrennt. Es stellt sich jedoch die Frage, ob mit isolierter Behebung eines oder mehrerer Strukturprobleme eine globale Verbesserung eines Softwaresystems erreicht werden kann.

Literaturverzeichnis

[Ciu99] CIUPKE, OLIVER: *Automatic Detection of Design Problems in Object-Oriented Reengineering*. in *Proceedings of Technology of Object-Oriented Languages and Systems - TOOLS 30*, pp. 18–32, 1999.

[GK03] GENSSLER, THOMAS and VOLKER KUTTRUFF: *Source-to-Source Transformation In The Large*. in *Proceedings of the Joint Modular Language Conference*. Springer LNCS, August 2003.

[TD03] TRIFU, ADRIAN and IULIAN DRAGOŞ: *Strategy Based Elimination of Design Flaws in Object-Oriented Systems*. in DEMEYER, SERGE, STÉPHANE DUCASSE and KIM MENS eds.: *Proceedings of the Fourth Workshop on Object–Oriented Reengineering, in Conjunction with ECOOP 2003*, pp. 55–61, 2003.

---

25 Recovering Design Elements in Large Software Systems

Jörg Niere
University of Siegen, Software Engineering Group
joerg.niereuni-siegen.de

Introduction. Reverse engineering large systems means to be able to analyse programs with more than 100K lines of code. Especially, recovering design elements means, e.g. detecting classes, relations or design-patterns, or recovering the architecture of a program. All activities need large analysis parts, because they nearly span over the whole system and manifest themselves not only in small parts.

In addition to the large analysis parts, design elements are usually described on a high abstract level. For example, design-pattern descriptions introduced by Gamma et al. contain more parts written in prose than parts described with diagrams. This makes them flexible for the application in an actual system design, but hard to detect according to of the large variety of implementation variants.

[4] http://injectj.sf.net

Current approaches of recovering design elements use automatic analyses. On the one hand, approaches performing fine-grained analyses fail, because of the large system size. On the other hand, approaches performing coarse-grained analyses produce too many false-positives. In general we have a trade-off between the granularity of the analysis and the size of the system to be analysed.

Calibrating rule catalogs. Our approach to recover design elements is to use a rule catalog, which describes the design elements in terms of graph rewrite rules [4]. The analysed source code is presented as an abstract syntax graph or may be represented as any other kind, e.g. control- or data-flow graphs.

The idea of our approach is that the rule catalog to analyse the system contains only those rules that fit to the system. In means of design patterns the rules describe only the implementation variants included in the system and no others. Therefore, our approach highly involves the reverse engineer in an anlysis process to calibrate the rule catalog to his/her needs, cf. Figure 25.1. This allows for performing fine-grained and coarse-grained analysis and for letting the catalog as small as possible enabling analysis of large systems.

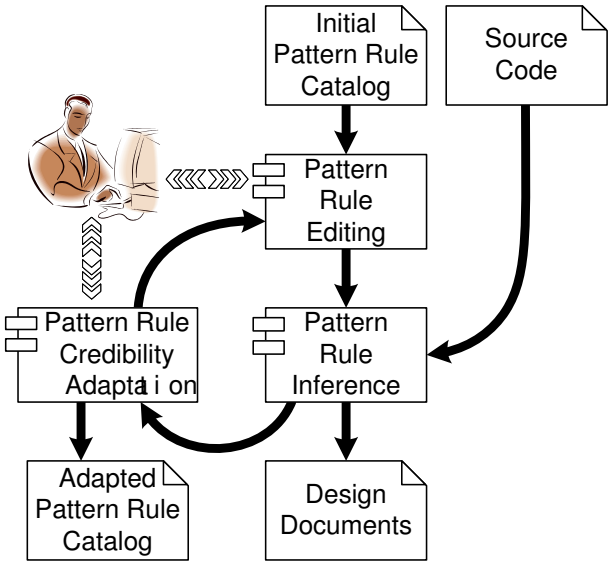


Figure 25.1: Interactive Rule Catalog Calibration

The process starts with an initial catalog of rules, which the reverse engineer may edit (*Editing*). In the *Inference* step, the rules are automatically applied by an inference algorithm, which produces meaningful results early and not only after a complete analysis of the system. The reverse engineer can interrupt the inference to inspect the results produced so far.

Our approach supports the inspection of the reverse engineer with accuracy values assigned to each result [5]. The accuracy values result from credibility values which are attached to the rules. So far, the choice of the credibility values highly depends on the personal experience of the reverse engineer. To overcome this limitation the process includes the *Credibility Adaption* step, where the credibility values are automatically adapted based on changes of accuracy values made by the reverse engineer. In addition to changing accuracy values the reverse engineer may also add hypotheses and let them validate in the inference step. For more details see [1]

Practical experiences. The approach has been implemented as plugins for the Fujaba Tool Suite and is available under www.fujaba.de. For each step in the process exists a separate plugin in order to test different inference or adaptation strategies.

The prototype has been used to analyse different systems. For example it has been used to detect design-patterns in Java's Abstract Windowing Toolkit (AWT) consisting of about 114 KLOC. The screenshot in Figure 25.2 shows a cut-out of the results of this analysis as UML class diagrams. The detected design-patterns are represented by dashed ovals consisting of the pattern's name and the accuracy value. We took this analysis as proof-of-concept, because the design-patterns included in the library are known.

In order to show the scalability of the approach, we analysed the Fujaba Tool Suite itself with about 750 KLOC. The goal of the analysis was also the detection of design-patterns.

We have also used the prototype to analyse the JigSaw webserver and have recovered the hot spots of the system detecting architecture patterns. The results were comparable to others produced by different reverse engineering tools. During the analysis process early produced meaningful results have been taken to proof hot spot hypotheses at certain parts of the system. Compared to other tool's results, ours have not only pointed out the hot spots but also additional information such as bridges between parts or central data structures.

The prototype was also successfully used in an industrial context on an application with more than 300 KLOC. The result of the analysis was that design decisions have been neglected by the developers implementing the system.

Further research. Our approach is not limited to detect design-patterns. Currently the prototype is used to detect Anti-Patterns and to rewrite 'bad' patterns by 'good' ones. Therefore the rules get property values describing positive and negative aspects.

Another research direction is to combine the presented approach with other ones. As example, the approach has been combined with a dynamic analysis. The static analysis indicates possible design-pattern instances, which will be verified by a dynamic analysis. Technically, an (intermediate) analysis result is a graph and therefore exchangeable with other tools using common exchange formats such as GXL.

A significant aspect concerning the overall duration of an analysis is the choice of the initial rule catalog. Using a catalog with coarse-grained rules means much iteration

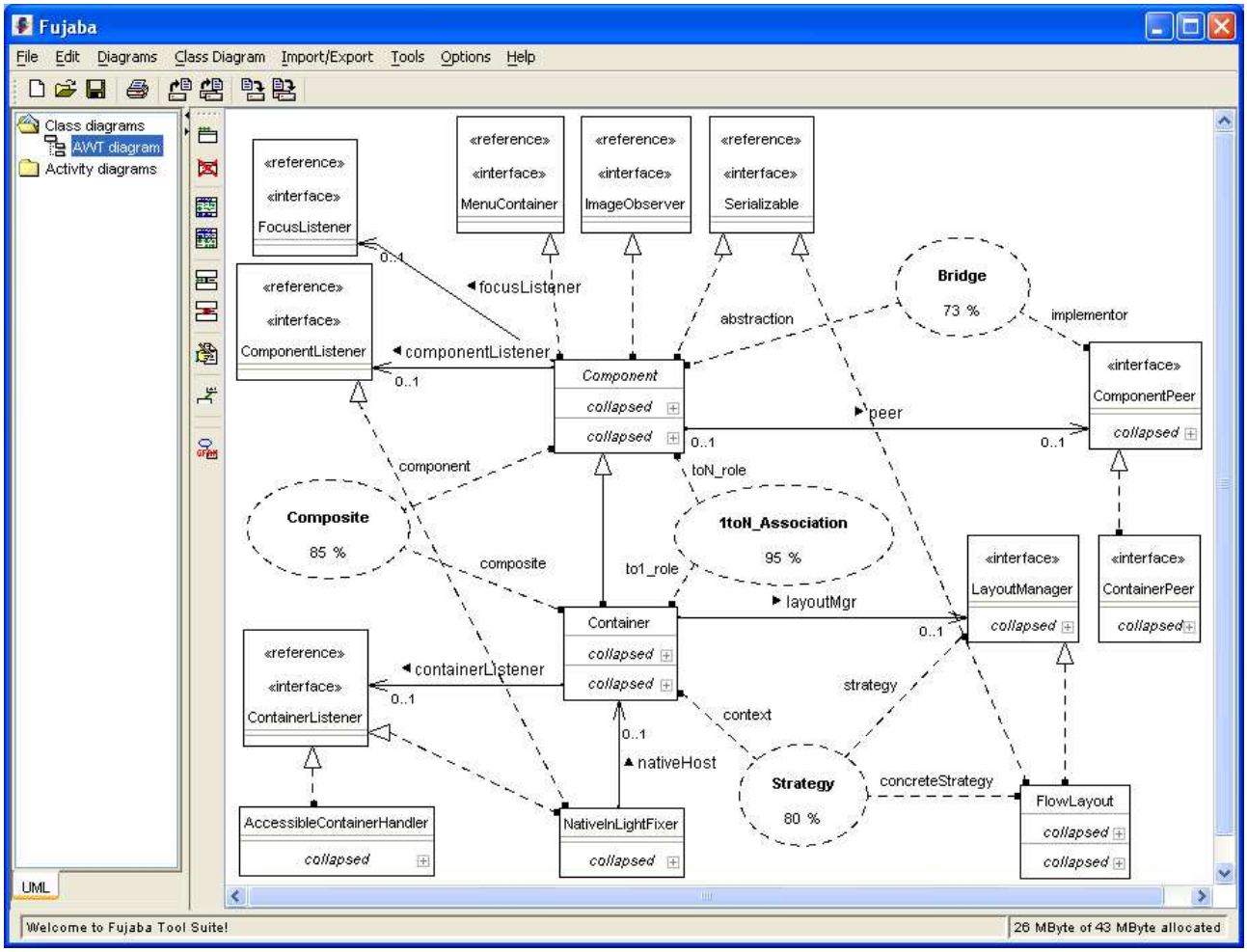


Figure 25.2: Annotated UML Class Diagram

to get a detailed result. Starting with a catalog with fine-grained rules may have the effect that the rules are not applicable, which means to get no results. Therefore we currently investigate the opportunity to use problem management systems for catalogs and their relations to analysed systems.

Bibliography

[1] J. Niere. *Incremental Design-Pattern Recognition*. PhD thesis, University of Paderborn, Paderborn, Germany, 2004. in german (to appear).

[2] J. Niere, W. Schäfer, J. Wadsack, L. Wendehals, and J. Welsh. Towards pattern-based design recovery. In *Proc. of the* 24^{th} *International Conference on Software Engineering (ICSE), Orlando, Florida, USA*, pages 338–348. ACM Press, May 2002.

[3] J. Niere, J. Wadsack, and L. Wendehals. Handling large search space in pattern-based reverse engineering. In *Proc. of the* 11^{th} *International Workshop on Program Comprehension (IWPC), Portland, USA*, pages 274–279. IEEE Computer Society Press, May 2003.

26 Specifying Patterns for Dynamic Pattern Instance Recognition with UML 2.0 Sequence Diagrams[1]

Lothar Wendehals
Software Engineering Group, Department of Computer Science University of Paderborn, Germany
lowende@upb.de

26.1 Motivation

Design recovery, which means extracting design documents from source code, assists a reverse engineer in understanding a software system. For design documentation design patterns [1] are suitable. By recognizing instances of design patterns in the system's source code the implicit design can be documented.

Most approaches use static analysis techniques e.g. [3]. In object-oriented languages static analysis only is not sufficient, since structurally similar patterns are not distinguishable from each other. Design patterns often differ only in their behavior. Polymorphism and dynamic method binding prevent a correct static analysis of method invocations that are essential to analyze behavior. Thus, for a precise recognition of design pattern instances a combination of static and dynamic analyses is reasonable, e.g. [2].

For tool supported recognition patterns have to be formally specified. Prolog is a common specification language, e.g. [2]. Others use script languages like Perl with regular expressions [3]. We developed a more intuitive graphical specification language for static analysis based on graph grammars [4].

We extend our static analysis process by dynamic analysis of method traces [6]. The process starts with static analysis of the source code resulting in a set of pattern instance candidates. The set is then verified by dynamic analysis. In the following the main focus is on a pattern specification language for behavioral patterns based on sequence diagrams.

26.2 Example

There are lots of design patterns that are structurally equal or at least similar such as *Decorator* and *Chain of Responsibility*, *Strategy* and *Bridge* or *Strategy* and *State*, which are depicted in Figures 26.1 and 26.2.

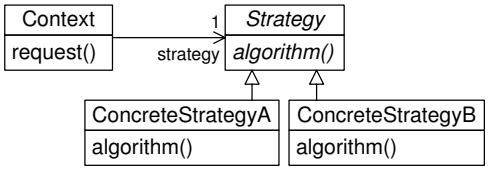


Figure 26.1: The *Strategy* Design Pattern

A *Strategy* design pattern lets an algorithm vary independently from the client that uses it. An abstract class Strategy defines the algorithm interface, which is implemented by different ConcreteStrategy classes.

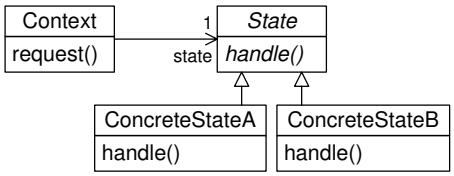


Figure 26.2: The *State* Design Pattern

The *State* design pattern allows an object to alter its behavior depending on its internal state. An interface for handling the behavior is defined by the abstract class State and implemented by different ConcreteStates. Implementations of these two design patterns are not distinguishable by static analysis. The next section shows how their behavior can be specified by sequence diagrams for a precise recognition.

26.3 Pattern Specification

In [1] there are some hints how the two design patterns *Strategy* and *State* differ in their behavior. It is said for a *Strategy* that *"A context forwards requests from its clients to its strategy. Clients usually create and pass a ConcreteStrategy object to the context..."*. So changing the concrete strategy is done by the client.

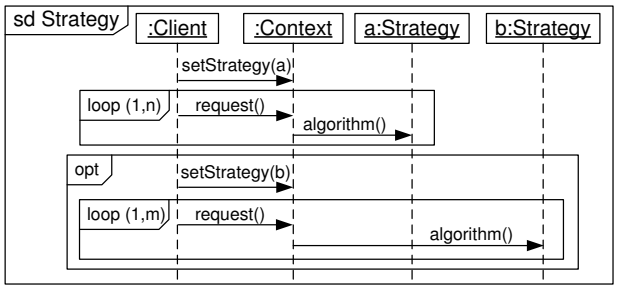


Figure 26.3: Sequence Pattern for *Strategy*

Figure 26.3 depicts a UML 2.0 sequence diagram for a *Strategy* that formalizes the description above. It is called a sequence pattern and can be read as follows: first of all a strategy object must be set on the context object by the client. An arbitrary number of calls from the client must be delegated to the strategy. Then the strategy may be changed by the client and another arbitrary number of delegations may be processed.

In UML 2.0 a new syntax element *Combined Fragment* is introduced to sequence diagrams. It is visualized as a rectangle with an operator within the upper left corner. The operator *loop (1,n)* e.g. defines a sequence repeated at least

[1]This work is part of the FINITE project funded by the German Research Foundation (DFG), project-no. SCHA 745/2-1.

once or up to n times. The operator *opt* defines an optional sequence. Other operators are *alternatives*, *negative*, *consider*, etc.

The operator *consider* has some methods as parameter. Only calls of those methods are displayed within the sequence, but they may be interleaved by other calls that are ignored. In sequence patterns the *consider* operator is implicitly used. Only calls of methods used within the sequence are considered in the pattern matching, other interleaving calls are ignored. In the given example there could be numerous other calls between the first setStrategy call and the loop, but no second setStrategy call. Thus, a sequence pattern does not describe a single execution sequence but a set of sequences with a common subsequence. This subsequence can be compared to some kind of slice of the program's method call trace where only a few method calls are considered.

For the *State* pattern it is said in [1]: *"... Clients can configure a context with State objects. Once a context is configured, its clients don't have to deal with the State objects directly. Either Context or the ConcreteState subclasses can decide which state succeeds another and under what circumstances."*. So the states are changed by the context or the states.

Figure 26.4 depicts the sequence pattern for a *State*. The state must be changed during runtime, otherwise different states make no sense. This is specified as an alternative, where the state can be switched either by the current state or by the context.

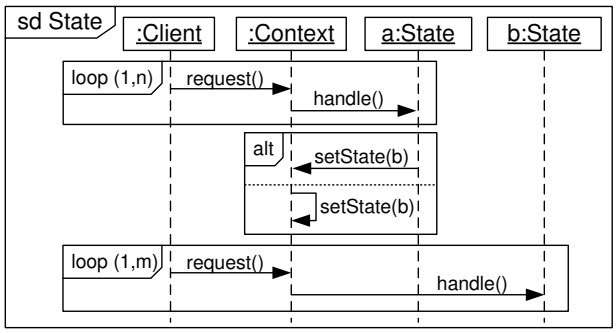


Figure 26.4: Sequence Pattern for *State*

The dynamic analysis uses these sequence patterns to verify pattern instance candidates from static analysis. A candidate both for *Strategy* and *State* can now be classified as one of the two patterns or as a false positive. Note, the class and method names are not considered in the analysis. They are just for a better readability of the sequence patterns.

26.4 Current and Future Work

In the static analysis we allow for the composition of structural patterns to new patterns to reduce the effort of specification. It has to be researched if reusing patterns is reasonable for sequence patterns, too.

The composition of patterns is especially used for higher level design patterns composed of lower level clichés. We are currently working on the specification of sequence patterns for those clichés and of course for all design patterns.

In [5] we describe how pattern instances can be rated by fuzzy values to express the accuracy of the match and to help the reverse engineer assessing the results. For sequence patterns the number of ignored calls within the matching method trace can be used to rate the match. If only a few ignored method calls interleave the given pattern sequence, the match has a high accuracy, otherwise, it has a low accuracy.

Bibliography

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

[2] D. Heuzeroth, S. Mandel, and W. Löwe. Generating design pattern detectors from pattern specifications. In *Proc. of the* 18^{th} *International Workshop on Automated Software Engineering (ASE), Montreal, Canada*, October 2003.

[3] R. Keller, R. Schauer, S. Robitaille, and P. Page. Pattern-based reverse-engineering of design components. In *Proc. of the* 21^{st} *International Conference on Software Engineering, Los Angeles, USA*, pages 226–235. IEEE Computer Society Press, May 1999.

[4] J. Niere, W. Schäfer, J. Wadsack, L. Wendehals, and J. Welsh. Towards pattern-based design recovery. In *Proc. of the* 24^{th} *International Conference on Software Engineering (ICSE), Orlando, Florida, USA*, pages 338–348. ACM Press, May 2002.

[5] J. Niere, J. Wadsack, and L. Wendehals. Handling large search space in pattern-based reverse engineering. In *Proc. of the* 11^{th} *International Workshop on Program Comprehension (IWPC), Portland, USA*, pages 274–279. IEEE Computer Society Press, May 2003.

[6] L. Wendehals. Improving design pattern instance recognition by dynamic analysis. In *Proc. of the ICSE 2003 Workshop on Dynamic Analysis (WODA), Portland, USA*, May 2003.

27 *JTransform* A Tool for Source Code Analysis

Holger Eichelberger
Jürgen Wolff von Gudenberg
Institut für Informatik, Am Hubland, 97074 Würzburg, Germany
{eichelberger,wolff}@informatik.uni-wuerzburg.de

Abstract

Code transformation and analysis tools provide support for software engineering tasks such as style checking, testing, calculating software metrics as well as reverse- and re-engineering. In a reverse step information is picked from a source text in order to facilitate its comprehension, its visualization or the check of agreed standards concerning coding quality or layout.
In this short paper (see [3] for details) we describe *JTransform*, a general Java source code processing and transformation framework. It consists out of a Java parser generating a configurable parse tree and various visitors (transformers, tree evaluators) which produce different kind of outputs.

27.1 Architecture

The main design goals for *JTransform* have been extendibility by subclassing and plugins, easy configuration of applications and ease of use. Therefore we extensively used design patterns like visitor, factory methods, abstract factory, composite or strategy.

The tool is split into two configurable phases, the front end and the back end.

27.1.1 Front End

A JavaCUP generated Java parser builds the parse tree. The parser is configurable in order to generate parse tree nodes containing different amounts of information. Therefore the parse tree is generated by a factory which consists of a set of factory methods. As default the node's syntactic structure and its source code position is registered, type information can be added. During the static type resolution information from imported classes, not part of the source code files may be collected.

27.1.2 Back End

The parse tree is then traversed by at least one visitor to transform the tree itself performing syntactical checks or static type resolution, or to produce the application-specific output. Generally, different back ends need different information stored in the parse tree nodes. The collection of method calls and the detection of general dependencies between classes , such as a cast expression can optionally be performed while calculating the static type references.

27.2 Configuration

In order to infer additional information, usually the nodes are subclassed and a specialized visitor, which controls the information inference mechanism is implemented. In order to tell *JTransform* to use the new nodes, a specialized nodes factory has to be implemented. Interpretation and implementation of options is possible.

All back ends should extend one class that provides configuration methods for redirection of input and output. Multiple back ends may be chained.

As an example of dynamic configuration by components the source code checking application implements different source code checks, each as a single component.

Furthermore, an XML configuration or project file contains a section of global features which are applied to all files, and at least one section which describes local features applied to a set of files specified in that section.

27.3 Applications

27.3.1 Visualisation by UML Diagrams

A visitor transforms the program into UMLscript that is input for the diagram drawing framework Sugibib [4]

27.3.2 Programming Conventions

Source code formatting and naming conventions are checked whether they conform to commonly agreed standards [2, 7].

Various visitors performing static source code analysis are applied in order to highlight dangerous code that may be the source for tricky errors. Take the facts that the statement following an if or for is not a block, or that an iterator changes accidentally the underlying collection, as examples.

27.3.3 Forbidden Classes and Packages

In an educational environment or for security reasons some classes or packages may be forbidden. The Java access controller restricts the usage of library classes by defining

a policy file.this mechanism is extended with *JTransform*. As an example, dynamic loading of classes and reflection can be forbidden, if user code is not allowed to make references to the `ClassLoader` and its subclasses, the class `Class` and all classes in `java.lang.reflect`. This rule is simple but very restrictive, often it is sufficient to declare certain methods of `ClassLoader` as forbidden in the user code.

27.3.4 Refactoring

A simple application of *JTransform* is to change class names, method names, parameter names, package names, move classes between packages and move entire packages and, of course, to correct all references to these classes and packages in the rest of the source code. As a further application, the changes and incompatibilities which result from deleting an attribute or a method or when changing the type of an attribute or parts of the signature of a method can be displayed; of course the changes can be executed but this usually requires further work by the programmer.

27.3.5 Clone Detection

Based on the information provided by the parse tree we implemented an algorithm that detects clones in the source code [8]. It is an adaption of the algorithm for finding frequent itemsets.

27.3.6 Metrics

Different kinds of primitive source code metrics like the number of packages, the number of inner classes or the amount of comment in the source can be calculated. Additionally metrics like reuse ratio or sophisticated metrics like those proposed in [1, 5] can be calculated as well as experimental metrics.

27.4 Conclusion

We have described the architecture and the applications of *JTransform*, a general-purpose Java source code transformation framework. *JTransform* has been designed according to usual object-oriented design principles, well known design patterns have been used. It consists of a front end producing an annotated parse tree and various back ends.

Custom back end applications can simply be plugged in by implementing a tree visitor. An alternative architecture of the tool can transform the parse tree to an XML document [3]. Thence, user applications may use an appropriate XML format such as XSLT/XPath as a general purpose query and transformation language. The stability of *JTransform* itself was validated by a test suite based on [6], the source code of the Java library and different Java online courses.

Bibliography

[1] Chidamber S, Kemerer C. A Metrics Suite for Object Oriented Design *IEEE Transactions on Software Engineering* **20**(6):476–493.

[2] Code Conventions for the Java^{TM} Programming Language `http://java.sun.com/docs/codeconv/html/CodeConvTOC.doc.html`

[3] Eichelberger H, Wolff von Gudenberg J. Object-oriented Processing of Java Source Code Software Practice and Experience, 2004, to appear

[4] Eichelberger H, Wolff von Gudenberg J. On the Visualization of Java Programs, In *LNCS 2269: Software Visualization* S. Diehl (ed.) Springer: Berlin, 2002; 295–306

[5] Genero M, Piattini M, Calero C. Early measures for UML class Diagrams *L'Objet* 2000 **6**(4):29–35.

[6] Jacks (Jacks is an Automated Compiler Killing Suite) `http://www-124.ibm.com/developerworks/oss/cvs/jikes/~checkout~/jacks/jacks.html`

[7] D. Lea Draft Java Coding Standard `g.oswego.edu/dl/html/javaCodingStd.html`

[8] V. Wahler *Erkennung von Klonen in Java-Programmen mit Data-Mining-Techniken*, Institut f. Informatik, Universität Würzburg, Diplomarbeit, 2003

28 Textual vs. Graphical Visualization of Fine-Grained Dependences Extended Abstract

Jens Krinke
FernUniversität in Hagen
Jens.Krinke@FernUni-Hagen.de

28.1 Introduction

A slice extracts those statements from a program that potentially have an influence onto a specific statement of interest that is the slicing criterion. Slicing has found its way into various applications. It is mostly used in the area of software maintenance and reengineering, e.g. in testing, impact analysis, and cohesion measurement.

One of the main slicing approaches uses reachability analysis in program dependence graphs (PDGs). Program dependence graphs mainly consist of nodes representing the statements of a program, and control and data dependence edges:

- Control dependence between two statement nodes exists if one statement controls the execution of the other (e.g. through if- or while-statements).
- Data dependence between two statement nodes exists if a definition of a variable at one statement might reach the usage of the same variable at another statement.

For the interprocedural variants IPDG and SDG the graphs are extended with additional interprocedural edges (which are not discussed here). The (*backward*) *slice* $S(n)$ of an IPDG at node n consists of all nodes on which n (transitively) depends via an interprocedurally realizable path.

The program dependence graph itself and the computed slices within the program dependence graph are results that should be presented to the user if not used in following analyses. As graphical presentations are often more intuitive than textual ones, a graphical visualization of PDGs is desirable.

28.2 Visualization of PDGs

Layout of graphs is a widely explored research field with many general solutions available in graph drawing tools. We evaluated some of these tools (*daVinci*, *VCG* and *dot*) to lay out PDGs. Our experience with these tools to layout PDGs has been disappointing. The resulting layouts were visually appealing but unusable, as it was not possible to comprehend the graph. The reason is that the viewer has no cognitive mapping back to the source code, which is the representation he is used to. The user expects a representation that is either similar to the abstract syntax tree (as a presentation of the syntactical structure), or a control-flow-graph like presentation.

Because this general approach to layout PDGs had failed, a declarative approach has been implemented. It is based on the following observations:

1. The control-dependence subgraph is similar to the structure of the abstract syntax tree.
2. Most edges in a PDG are data dependence edges. Usually, a node with a variable definition has more than one outgoing data dependence edge.

The first observation leads to the requirement to have a tree-like layout of the control dependence subgraph with the additional requirement that the order of the nodes in a hierarchy level should be the same as the order of the corresponding statements in the source code. The second observation leads to an approach where the data dependence edges should be added to the resulting layout without modifying it. As most data dependence edges would now cross large parts of the graph, a Manhattan layout is adequate. This enables an orthogonal layout of edges with fixed start and end points. This approach has been implemented in a tool that visualizes system dependence graphs. Starting from a graphical representation of the call graph, the user can select procedures and visualize their PDGs. Through selection of nodes, slices can be calculated and are visualized through inverted nodes in the PDGs laid out.

Experience with the presented tool shows that the layout is very comprehensible for medium sized procedures and the user easily keeps a cognitive map from the structure of the graph to the source code and vice versa. This mapping is supported by the possibility to switch between a textual visualization of the source code and the graphical layout of the current procedure. Sets of nodes marked in the graph can be highlighted in the source code and marked regions in the source code can be highlighted in the graph. Together with additional navigational aids, it is easy to see what statements influence which other statements and how.

However, experience has shown that the graphical visualization is still too complex. For larger procedures the number of nodes and edges is too high, and it takes very long to follow edges across multiple pages by scrolling.

The presented graphical visualization has been found to be far too complex for large programs and non-intuitive for visualization of slices. Therefore the graphical visualization has been extended with a visualization in source code. This causes a non-trivial projection of nodes onto source code, because of the fine-grained structure of the dependences between statements.

28.3 Visualization of Locality

Independent of visualization, one of the problems in understanding a slice is to decide why a specific statement is included in that slice and how strong the influence of that statement is onto the slicing criterion. A slice cannot answer these questions as it does not contain any qualitative information. Probably the most important attribute is *locality*. Users are more interested in facts that are near the current point of interest than on those far away. A simple but very useful aid is to provide the user with navigation along the dependences: For a selected statement, show all statements that are directly dependent (or vice versa).

A more general approach to accomplish locality in slicing is to limit the length of a path between the criterion and the reached statement. Using paths in program dependence graphs has instead of paths in control flow graphs has an advantage. A statement that has a direct influence on the criterion will be reached by a path with length one, independent of textual or control flow distance.

Distance-limited slices cannot simply be visualized with the techniques presented in the previous section without any modification. Another possibility is to indicate the distances from the (slicing) criterion for any node in the (possibly distance-limited) slice. The textual visualization from the previous section is therefore modified not only to highlight the nodes in the textual representation, but also to give any source code fragment a color that represents the distance of the equivalent nodes to the criterion. The slicing algorithm needs not to be changed in order to accommodate the distance computation—it is sufficient to remember the distance of a node during breadth-first search.

28.4 Abstract Visualization

For large-scale program understanding the presented visualization techniques are not very helpful. If an unknown program is analyzed, the very detailed information of program dependences and slices is overwhelming, and a much less detailed information is needed. The user who tries to understand the program will start with variables and procedures and not with statements. To understand a previously unknown program, it is helpful to identify the 'hot' procedures and global variables—the procedures and variables with the highest impact on the system.

This section will show how slicing and chopping can help to visualize programs in a more abstract way, illustrating relations between variables or procedures. *Chopping* reveals the statements involved in a transitive dependence from one specific statement (the source criterion) to another (the target criterion). A chop for a chopping criterion (s, t) is the set of nodes that are part of an influence of the (source) node s onto the (target) node t.

It is possible to define slices for variables or procedures informally:

1. A (backward) slice for a criterion variable v is the set of statements (or nodes in the PDG) which may influence variable v at some point in the program.
2. A (backward) slice for a criterion procedure P is the set of statements (or nodes in the PDG) which may influence a statement of P.

These definitions can be adapted to the other slicing and chopping variants, including the adaption of the needed algorithms. It will not be presented here, as it is straightforward.

As previously noted, it is helpful to identify the 'hot' procedures and global variables. However, to identify them, we have to measure the procedures' and variables' impact on the system. A simple measurement is to compute slices for every procedure or global variable and record the size of the computed slices. However, this might be too simple. A slightly better approach is to compute chops between the procedures or variables. A visualization tool has been implemented that computes a $n \times n$ matrix for n procedures or variables, where every element $n_{i,j}$ of the matrix is the size of a chop from the procedure or variable n_j to n_i. The matrix is painted using a color for every entry, corresponding to the size—the bigger, the darker. With this tool, it is easy to get an overall impression of the software to analyze. Important procedures or global variables can be identified on first sight and their relationship can be studied. Doing this as a preparing stage aids in later, more thorough investigations with traditional slicing visualizations like the ones presented in the previous sections.

28.5 Conclusions

Despite the widespread use of graphical visualization in software maintenance and reverse engineering, our and other's experiences for graphical visualization of program dependence and program slices are different. For tasks related to large-scale understanding graphical visualization has proven to be successful. The main reason is that the number of nodes (or objects) to be visualized is kept very low by clustering techniques. Tasks related to understanding dependences in detail (like program dependences and slices) suffer from the sheer amount of data to be visualized. The various experiences show that graphical visualization has more disadvantages than advantages in this area.

The visualization of slices in textual form has shown to be much more effective, because the programmer is accustomed to representations similar to source code. However, slices are still hard to understand due to the loss of locality. Distance-limited slicing and its visualization can help, because it limits the distance of the influence to the current point of interest. The visualization of the distance shows immediately how important a statement is for the current influence.

For large-scale program understanding none of the detailed slicing visualizations are helpful. The presented approach to visualize the influence range of variables and procedures by visualizing the size of chops can help the user to identify "hot spots" of the program very fast.

29 Rechnergestützte Diagnose in Software-Entwicklung und -Test

Dierk Ehmke
Hochstr. 59, D-64285 Darmstadt
mail@d-ehmke.de

Zusammenfassung

PeriPlus wurde für die rechnergestützte Diagnose in Software-Entwicklung und -Test komplexer Systeme entwickelt. Einsatzgebiete sind etwa Daily-Build und Test. PeriPlus erkennt und analysiert Probleme, benachrichtigt gezielt Administratoren oder Problemverursacher, bereitet Ausgaben auf und stellt Metriken bereit. Ein Prototyp wird erfolgreich für große Softwaresysteme eingesetzt.

29.1 Beobachtungen

Im Folgenden konzentrieren wir uns auf die Routinetätigkeiten im Daily-Build. Routine schließt hier unter anderem ein:

- Administratoren schauen häufig nach, ob der Daily-Build noch läuft.
- Im Fehlerfall werden große Log-Dateien (mehrere MB groß, verteilt auf viele Dateien) analysiert, um das Problem zu finden. Hinzu kommt, dass das Herausfiltern oft vorkommender doppelter oder vernachlässigbarer Fehlermeldungen ebenfalls Zeit kostet. Weit verstreute Einträge in verschiedenen Logdateien führen dazu, dass Fehlerhinweise übersehen werden. Diese Arbeiten erfordern viel Disziplin, wirken demotivierend und ihre häufige Wiederholung ist bekannte Ursache für weitere Fehlleistungen.
- Ermitteln derjenigen Person, die für dieses Problem verantwortlich ist, sowie deren Benachrichtigung.
- Nach dem Lauf automatisierter Smoke-Tests entscheiden, ob Testeingangskriterien erfüllt wurden.
- Überwachung der Code-Qualität etwa mit Lint, Purify und Style-Guide-Checkern. Anhand der Logfiles werden die Entwickler ermittelt, die ihren Code überarbeiten müssen. Das wird in vielen Organisationen sporadisch gemacht und gerät leicht in Vergessenheit. Andere ‘schleichende’ Trends: Entwickler stellen Code nicht zurück, die Testabdeckung sinkt, Entwickler kommen mit ihren Aufgaben nicht voran.
- Viele Entwickler schauen sporadisch nach dem Status des Daily-Build und unterbrechen zu diesem Zweck ihre Arbeit.
- Da der Projektfortschritt wesentlich vom Daily-Build abhängt und jederzeit Probleme auftreten können, werden besonders kompetente Mitarbeiter für die Administration benötigt.

Wichtig hier: Es gibt kaum Werkzeugunterstützung für die aufgezählten Tätigkeiten.

Teilweise liegt die Ausführungsdauer dieser Tätigkeiten im Bereich weniger Minuten. [DeMarco, Timothy Lister] beschreibt, dass es nach kleinen Unterbrechungen fünfzehn Minuten dauert, bis die ursprüngliche Aufgabe weiterbearbeitet wird.

29.2 Zielsetzung

Manche der beschriebenen Routinetätigkeiten sind komplex und bedürfen intensiver Expertenarbeit, viele sind trivial. Wir schätzen, dass die Paretoregel (80-20-Regel) zutrifft, das heißt 80% der Probleme sind eher trivial und 20% sind Expertenprobleme.

Komplexe Probleme lassen sich schwerlich automatisch diagnostizieren, spätestens wenn sie semantischen Ursprungs sind. Triviale Probleme dagegen scheinen geeignete Kandidaten für eine rechnergestützte Lösung zu sein.

Ziel war es, zunächst diese trivialen Probleme in den Griff zu bekommen und Administratoren davon zu entlasten. Dabei sollte möglichst nicht in die bestehenden Umgebungen eingegriffen werden und möglichst viele elektronisch vorliegende Informationen/Daten genutzt werden. Dieses minimal invasive Vorgehen ist wichtig, da Daily-Build-Umgebungen komplex sind, Durchläufe viel Zeit benötigen (häufig mehr als zwölf Stunden, [McConnell] liefert ein Beispiel mit 36 Stunden) und Abbrüche sehr teuer sind.

29.3 Grobstruktur des Diagnosesystems PeriPlus

PeriPlus gliedert sich in die drei Komponenten Fakten-Scanner, Diagnose und Aktoren, die im Folgenden beschrieben werden.

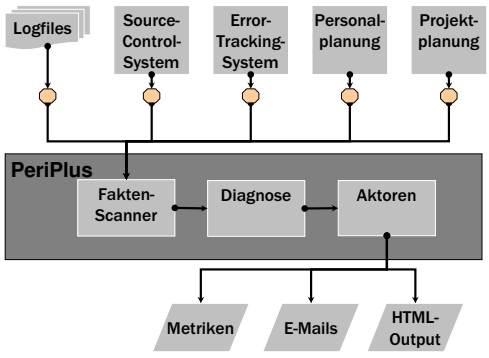


Abbildung 29.1: Grobstruktur des Diagnosesystems PeriPlus

29.3.1 Fakten-Scanner

Fakten-Scanner lesen und filtern die Daten aus der Systemumgebung und konvertieren sie in eine interne Repräsentation. Beispiele für Fakten-Scanner:

- Log-Dateien von Compilern, Lint, Purify, Style-Guide-Checkern und aus automatischen oder manuellen Tests
- Source-Control- bzw. Versions-Management- Systeme
- Daten aus dem Fehlerverfolgungssystem und Werkzeugen für die Personal- und Projektplanung

29.3.2 Diagnose

Die Diagnosekomponente erstellt aufgrund der Fakten Diagnosen. Beispiele dafür sind:

- Zuordnung: Bereits beim Quellcode ist die Zuordnung nicht eindeutig. Quellcode-Owner kann diejenige Person sein, die die Datei im Source-Control-System angelegt hat, oder diejenige, die sie zuletzt geändert hat. Danach kann aber auch die Zuständigkeit gewechselt bzw. die Person die Organisation verlassen haben. Schwieriger wird es bei Testfällen, die in der Regel unabhängig von der Quellcodestruktur sind und in gröbere funktionale Bereiche gegliedert sind. Schließlich muss für temporäre Abwesenheiten (Krankheit, Urlaub etc.) eine Stellvertreterregelung beachtet werden.
- Fehlerdiagnose: neue (das heißt, beim letzten Daily Build trat kein Fehler auf) Fehler im Quellcode, Netzprobleme, Linkfehler, Installationsfehler, Fehler in den Skripten und Fehler in Testfällen.
- Qualitätsdiagnose: Anhand der Daily-Build- Ergebnisse inklusive Smoke-Tests wird entschieden, ob Testeingangskriterien für aufwändigere Tests erfüllt sind oder das Produkt releasefähig ist.

29.3.3 Aktoren

Aktoren machen die Diagnose-Ergebnisse in der Umgebung wirksam. Beispiele dafür sind:

- Nachrichten versenden an Verursacher, Prozessverantwortliche, Projektleiter und Management.
- Erzeugung übersichtlicher Ausgaben
- Erklärungskomponente: Wir streben nicht an, dass das System hundertprozentig zutreffende Diagnosen erstellt. Das macht eine Erklärungskomponente erforderlich, mit deren Hilfe Entwickler die Diagnosen nachvollziehen und kontrollieren können.
- Fehlerverfolgungssystem: Einträge generieren für Probleme. So münden Diagnoseergebnisse direkt in den etablierten Workflow der Entwicklungsorganisation.
- Reparieren von Problemen
- Metriken-Erstellung
- Projektstatus, Tracking: Bei testgetriebener Entwicklung kann automatisch aus der Testabdeckung der Projektstatus ermittelt werden.

29.4 Ergebnisse

Der Prototyp wird in der Entwicklung zweier komplexer, strategischer Produkte seit vier Jahren eingesetzt. Das eine Produkt besteht aus 25.000 Quelldateien und ist überwiegend in Java geschrieben. Hier werden Diagnosen für den Daily-Build inklusive automatisierter Smoke- Tests erstellt, per E-Mail versand und HTML-Ausgaben erzeugt. Das andere hat einen Umfang von 3,4 Millionen LOC und ist überwiegend in C und C++ entwickelt. Hier werden basierend auf Lint- und Purify-Ausgaben kritische Code-Segmente ermittelt und die zuständigen Entwickler darüber informiert.

Es hat sich gezeigt, dass der beschriebene Ansatz die Erwartungen erfüllt. Triviale Probleme werden vom System diagnostiziert. Problemverursacher können oft zutreffend ermittelt werden.

Anstatt den Prozess ständig zu überwachen und aus umfangreichen Logfiles Probleme zu diagnostizieren, warten Administratoren den Empfang einer E-Mail ab und überprüfen das vom Diagnosesystem ausgegebene Resultat.

Für das Nadelöhr Kommunikation haben schon kleine Verbesserungen große Wirkung. So entfällt das sporadische Durchsuchen einer Unmenge von Log-Files, weil alle sich darauf verlassen können, per E-Mail relevante Informationen zu erhalten. Zudem bleiben nicht betroffene Entwickler gänzlich unbehelligt, weil gezielt nur die Verursacher der Probleme benachrichtigt werden.

29.5 Diskussion

Entscheidend für den Erfolg dürfte die Beschränkung auf die trivialen Probleme gewesen sein. Bereits hierfür kann ein signifikanter Nutzen verzeichnet werden. Wir erwarten, mehr verallgemeinerbare Regeln zu finden, die immer komplexere Sachverhalte diagnostizieren können. Schlussfolgerungen

Bedenkt man die Bedeutung zunehmender Komplexität und Kritikalität sowie der Faktoren Ressourcenknappheit und Kommunikationsaufwand in Softwareprojekten und berücksichtigt die Tatsache, dass viele Projekte scheitern, legen die gezeigten Ergebnisse nahe, den beschriebenen Ansatz weiterzuentwickeln.

Unter dem Namen PeriPlus wird aktuell ein Nachfolgesystem entwickelt, das voraussichtlich im 3. Quartal 2004 in einer ersten Version vorliegen wird.

29.6 Literaturverzeichnis

[DeMarco, Timothy Lister] DeMarco, Timothy Lister. Wien wartet auf Dich! Carl Hanser Verlag, München, Wien, 1991.

[Gyhra] N. Gyhra. Einsatz eines Expertensystems im Qualitätssicherungsprozess eines komplexen Software- Produktes. Fachhochschule Darmstadt, Fachbereich Informatik, Darmstadt, 2001.

[McConnell] Steve McConnell, `http://www.stevemcconnell.com/ieeesoftware/bp04.htm`, Stand 25. März 2004.

[Puppe] F. Puppe. Einführung in Expertensysteme. Springer Verlag, Berlin, 2. Auflage, 1991.

30 Die Rolle der Architektur im Kontext der a-posteriori Integration

Thomas Haase
RWTH Aachen, Lehrstuhl für Informatik III, Ahornstr. 55, D-52074 Aachen
thaase@i3.informatik.rwth-aachen.de

30.1 Einleitung

Der SFB 476 IMPROVE [7] beschäftigt sich mit der informatischen Unterstützung von Entwicklungsprozessen in der Verfahrenstechnik. Die Zielsetzung des SFB auf der softwaretechnischen Ebene beinhaltet hierbei die Konzeption und Realisierung einer integrierten Werkzeugumgebung, welche eine durchgängige Unterstützung der einzelnen Entwicklungsschritte ermöglicht. In dieser Arbeitsumgebung werden existierende Werkzeuge zur Durchführung der verschiedenen Entwicklungsaktivitäten (z.B. Grobentwurf, Simulation, ...) und Werkzeuge zur Realisierung erweiterter Unterstützungsfunktionalitäten (z.B. Prozessplanung, -steuerung und -kontrolle, inkrementelle Konsistenzsicherung zwischen den unterschiedlichen Entwicklungsdokumenten oder multimediale Kommunikation) [6] a-posteriori miteinander integriert. Hierbei erfolgt die Integration sowohl zwischen den existierenden und neuen Werkzeugen als auch zwischen den neuen Werkzeugen untereinander (siehe Abbildung 30.1).

30.2 Architekturbasierte Integration

Motivation. Bei der Integration einer Menge von Werkzeugen zu einem, wie im vorherigen Abschnitt skizzierten, Gesamtverbund repräsentieren Architekturen das Bindeglied zwischen den zu integrierenden Werkzeugen. Entsprechend der in der Literatur üblichen Definition von (Software-)Architekturen als *Beschreibung der Struktur der Komponenten eines (Software-)Systems und ihrer Beziehungen zueinander* (vgl. z.B. [1, 5]) beschreiben sie im Kontext eines integrierten Systems die für die Integration relevanten Aspekte. Dies sind z.B. die zur Verfügung stehenden Schnittstellen der einzelnen Werkzeuge, die Struktur und die Schnittstellen zusätzlicher Integrationskomponenten, z.B. Wrapper zur Homogenisierung der existierenden Schnittstellen und/oder Datenmodelle [2, 3], die Aufrufbeziehungen zwischen den verschiedenen Werkzeugen zur Realisierung der intendierten Integrationsfunktionalität, die Verteilung der Werkzeuge, die Realisierung von Aufrufbeziehungen mittels verschiedener Middleware-Techniken (z.B. COM, CORBA, ...) etc. In diesem Sinne beschreibt die Architektur eines integrierten Systems die verbindenden Teile dieses Systems, nicht aber die Interna der einzelnen Werkzeuge.

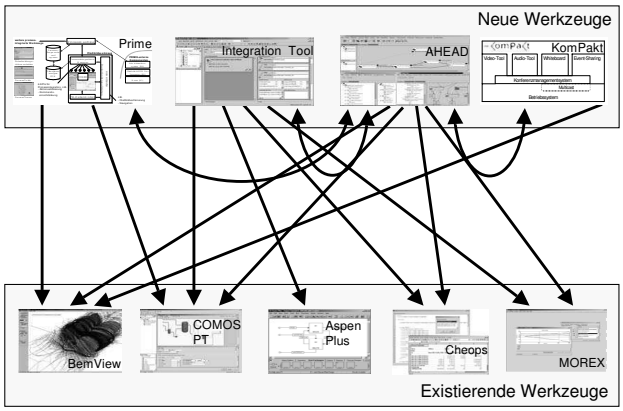


Abbildung 30.1: Integrierte Werkzeugumgebung

Werkzeugunterstützung. In bezug auf die genannten Anforderungen an Architekturen für integrierte Systeme wird eine grobgranulare Beschreibung des Systems, wie sie in Abbildung 30.1 gezeigt wird, diesen sicherlich nicht gerecht. Für eine im obigen Sinne adäquate Architekturbeschreibung ist vielmehr eine Verfeinerung derselbigen hinsichtlich der relevanten Integrationsaspekte notwendig. Dies war die Motivation für die Entwicklung eines Architekturentwurfwerkzeuges, genannt FIRE3 (*Friendly Integration Refinement Environment*) [4], welches eine bzgl. des Integrationssachverhaltes spezifische Architekturverfeinerung unterstützt.

Im Gegensatz zu allgemeinen Modellierungswerkzeugen, die für den Architekturentwurf in einem beliebigen Anwendungskontext einsetzbar sein sollen, fokussiert FIRE3 auf den Entwurf integrierter Systeme. Hierzu wird eine initiale, grobgranulare Architektur (wie z.B. in Abbildung 30.1 gezeigt) schrittweise in eine konkrete Architektur transformiert, welche die obigen Aspekte berücksichtigt. Dabei sind die möglichen Architekturverfeinerungen nicht beliebig, sondern sie basieren auf spezifischen Transformationsmustern, die das Wissen über typische Integrationssituationen widerspiegeln, z.B. die verschiedenen Zugriffsmöglichkeiten auf die Daten eines Werkzeuges, die Adaption von Schnittstellen durch Wrapper oder die Realisierung verteilter Kommunikationsbeziehungen mittels diverser Middleware-Techniken. Abbildung 30.2 zeigt exemplarisch zwei Screenshots des Werkzeuges: ausgehend von einem Entwurf der logischen Architektur des inte-

grierten Systems, welche z.B. die Struktur der Wrapper-Komponenten beschreibt (oberer Screenschot in Abbildung 30.2), generiert das Werkzeug eine isomorphe Verteilungsarchitektur (unterer Screenschot in Abbildung 30.2), welche anschließend weiter verfeinert werden kann resp. muß. Diese Verfeinerungen betreffen z.B. die Auswahl einer konkreten Middleware-Technik zur Realisierung eines entfernten Methodenaufrufs.

Des weiteren erlauben es Analysen, die Konsistenz und die Vollständigkeit der Architektur zu überprüfen. Animierte Kollaborationsdiagramme ermöglichen zudem die Visualisierung typischer Interaktionsszenarien der integrierten Werkzeuge.

30.3 Zusammenfassung

In diesem Papier wird die These vertreten, daß Architekturen eine zentrale Rolle bei der Integration von Werkzeugen spielen. Insbesondere lassen sich auf Architekturebene charakteristische, immer wiederkehrende Integrationssituationen identifizieren. Die werkzeuggestützte Anwendung solcher Muster, wie es in dem Werkzeug FIRE3 prototypisch realisiert ist, vereinfacht den Entwurf integrierter Systeme und gestaltet ihn effizienter.

Danksagung

Der Autor bedankt sich für die finanzielle Unterstützung durch die Deutsche Forschungsgemeinschaft (DFG) im Rahmen des Sonderforschungsbereiches 476 “Informatische Unterstützung übergreifender Entwicklungsprozesse in der Verfahrenstechnik”.

Literaturverzeichnis

[1] GARLAN, D. und D. E. PERRY: *Introduction to the Special Issue on Software Architecture*. IEEE Transactions on Software Engineering, 21(4):269–274, 1995.

[2] HAASE, T.: *A-posteriori Integration verfahrenstechnischer Entwicklungswerkzeuge*. Softwaretechnik-Trends, 23(2):32–34, 2003.

[3] HAASE, T.: *Semi-automatic Wrapper Generation for a-posteriori Integration*. In: *Proc. of the Workshop on Tool Integration in System Development (TIS 2003)*, Seiten 84–88, Helsinki, Finland, 2003.

[4] HAASE, T., O. MEYER, B. BÖHLEN und F. GATZEMEIER: *Fire3: Architecture Refinement for a-posteriori Integration*. In: PFALTZ, JOHN L., M. NAGL und B. BÖHLEN (Herausgeber): *Applications of Graph Transformations with Industrial Relevance: 2nd Intl. Workshop, AGTIVE 2003*, LNCS 3062, Seiten 461–467. Springer, 2004.

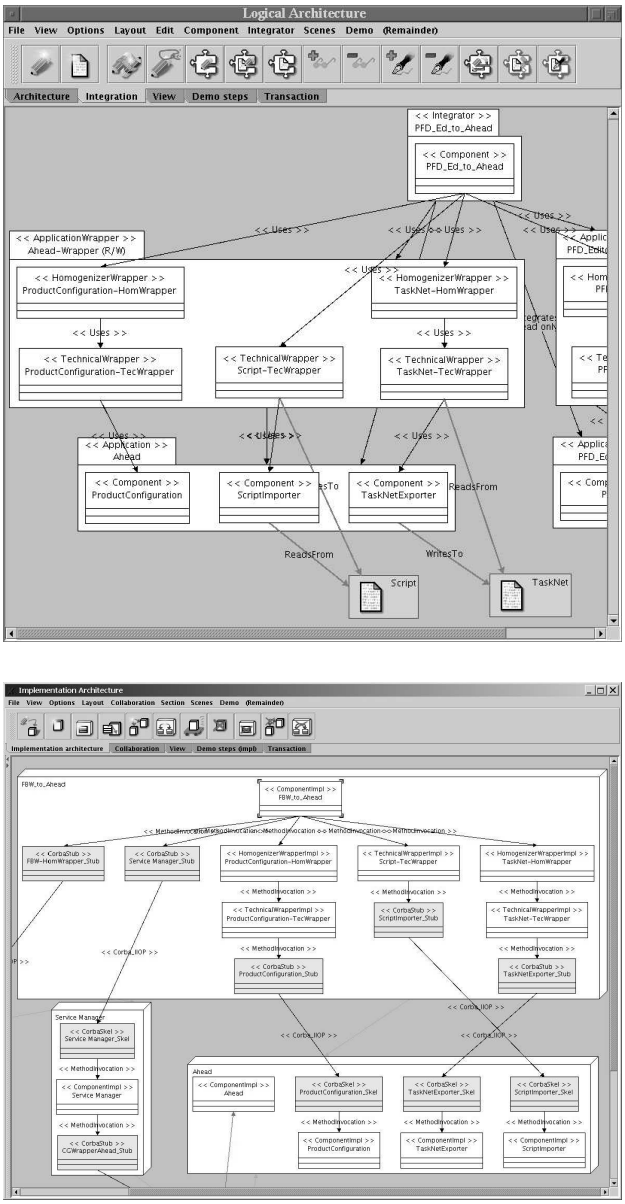


Abbildung 30.2: FIRE3 – Screenshots

[5] NAGL, M.: *Softwaretechnik: Methodisches Programmieren im Großen*. Springer, 1990.

[6] NAGL, M., R. SCHNEIDER und B. WESTFECHTEL: *Synergetische Verschränkung bei der a-posteriori Integration von Werkzeugen*. In: NAGL, M. und B. WESTFECHTEL (Herausgeber): *Modelle, Werkzeuge und Infrastrukturen zur Unterstützung von Entwicklungsprozessen*, Seiten 137–154. Wiley-VCH, 2003.

[7] NAGL, M. und B. WESTFECHTEL (Herausgeber): *Integration von Entwicklungssystemen in Ingenieuranwendungen: Substantielle Verbesserungen der Entwicklungsprozesse*. Springer, 1999.

31 Komponierung, Konfiguration und Adaptierung von heterogenen Software-Komponenten

Uwe Zdun
Wirtschaftsuniversität Wien, Abteilung für Wirtschaftsinformatik, Wien, Österreich
zdun@acm.org

Zusammenfassung

In diesem Papier erläutern wir einige Probleme der Komponierung, Konfiguration und Adaptierung von heterogenen Software-Komponenten. Einige bekannte Pattern können verwendet werden, um diese Problem zu lösen. Aus diesen Lösungen entstand die Idee, die Pattern als systematisches Konzept für den Sprachentwurf zu verwenden – welches dann in der Sprache Frag umgesetzt wurde.

31.1 Einleitung

Komponenten sollen wiederverwendbare, in sich geschlossene Bausteine eines Software-Systems sein. Verschiedene Komponentenansätze haben auch einen starken Fokus auf Aspekten der Komponierung, Konfiguration und Adaptierung von Software-Komponenten, aber diese Aspekte sind noch nicht wohldefiniert. In diesem Papier soll eine Sprache vorgestellt werden, die sich in ihrem Sprachentwurf auf diese Aspekte konzentriert und somit einen einheitlichen Ansatz der Komponierung, Konfiguration und Adaptierung von Software-Komponenten darstellt.

Es gibt viele Komponentenbegriffe. Die Definition von Szyperski [5] soll hier als ein Startpunkt dienen: "*A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties.*" Diese Definition umfasst Server-Komponenten-Ansätze (wie EJB, CCM und COM+), Java Beans, Komponenten-Frameworks in Skriptsprachen (wie Tcl, Python, Perl und Visual Basic), Active X controls, C Bibliotheken mit klar abgegrenzten Schnittstellen und viele andere Arten von Komponenten. Die Definition kann somit recht breit interpretiert werden, was insbesondere aus einer (praktischen) Reengineering-Perspektive wichtig ist, denn in der Praxis sind Komponenten oft nicht idealtypische Komponenten, sondern sind groß, haben keine erzwungenen Komponentengrenzen und haben keine definierten Schnittstellen.

Alle genannten Komponentenmodelle verfügen nichtsdestotrotz über Ansätze oder Best Practices zur Komponierung, Konfiguration und Adaptierung von Software-Komponenten. Hier stellen sich – insbesondere aus einer Maintenance und Reengineering Perspektive – eine Reihe von offenen Problemen:

- Viele Komponentenansätze sind nur für eine Sprache oder Plattform entworfen worden und nicht für heterogene Situationen (wie in vielen Alt-Systemen). Insbesondere sind Ansätze zur tieferen Sprachintegration, die bspw. Destruction Order, Inheritance Hierarchies, Garbage Collection und ähnliche Spracheigenschaften automatisiert beachten, kaum umgesetzt.
- Einige Komponentenansätze erlauben nicht das dynamische Deployen und Un-Deployen von Komponenten. Ferner fehlen gerade in Alt-Systemen oft Komponenten-Deployment-Abstraktionen, wie Package Konzepte. Diese nachträglich einzubringen bedeutet einen erheblichen Aufwand.
- Komponenten, insbesondere von Dritten, müssen oft angepasst werden, bspw. bei Schnittstellenänderungen und Versionsänderungen. Viele Komponentenansätze unterstützten die Adaptierung von Komponenten gar nicht oder nicht zur Laufzeit.
- Um auf Komponenten zugreifen zu können, ist es notwendig ihre Schnittstelle zu kennen; zum Teil werden diese zur Laufzeit benötigt. Es ist auch oft hilfreich, wenn man diese Schnittstellen erzwingen kann, wie im Facade Pattern [2].
- Um Re-Kompilierungen zu vermeiden, sollten Komponenten nicht nur durch Übergabeparameter konfiguriert werden, sondern es sollte auch weitere deklarative und verhaltensbeschreibende Konfigurationsmöglichkeiten geben.

Für jedes dieser Probleme existieren geeignete Lösungen. Eine Vielzahl dieser Lösungen ist bereits in Form von Software-Patterns dokumentiert. Eine EuroPLoP 2003 Focus Group hat die wichtigsten Pattern in diesem Bereich untersucht und kategorisiert [10]. Die Idee hinter der Sprache Frag ist, dem Entwickler eine Sprache an die Hand zu geben, welche die Lösungen in diesen Pattern in einer wiederverwendbaren Art und Weise als Sprachelemente implementiert. Die Sprache kann ferner in einer einfachen Art und Weise mit existierenden Umgebungen (bspw. anderen Sprachen, Plattformen, Komponentenansätzen, Rahmenwerken, Alt-Systemen, etc.) kombiniert werden.

31.2 Sprachunterstütze Patterns für Komponierung, Konfiguration und Adaptierung von Software-Komponenten in Frag

Frag ist eine objekt-orientierte Erweiterung der Sprache Tcl, die speziell für Einbettung in andere Systeme und deren Integration entworfen wurde. In diesem Abschnitt soll erklärt werden, wie Frag Patterns für die Komponierung, Konfiguration und Adaptierung von Software-Komponenten unterstützt und wie die obengenannten Probleme aufgelöst werden. Für eine detaillierte Sprachbeschreibung sei auf `frag.sourceforge.net` verwiesen.

Sprachintegration Viele Komponentenansätze sind nur für eine Sprache ausgelegt. Frag implementiert das Split Object Pattern [9] und ein extrem flexibles Objekt-Konzept. Dadurch kann Frag automatisiert an andere Sprachen, wie C, C++, Java und Tcl, angepasst und mit ihnen weitgehend integriert werden. Wie in [9] beschrieben, umfasst diese Integration beispielsweise Objekt-Identitäten, Klassenhierarchien und andere Objekt-Beziehungen. Ein wichtiger Aspekt für die Wartung und Konfiguration von Systemen ist: Frag kann vollständig ist diese anderen Host-Sprachen als Glue-Sprache eingebettet werden. Insbesondere läuft Frag auch in einer Java Virtual Machine über den Jacl [1] Interpreter.

Deployment von Komponenten Einige Server Component Patterns [6] beschreiben Aspekte des Deployments, insbes. Component Installation, Component Package und Assembly Package. Diese Pattern werden durch den Tcl `package` Mechanismus unterstützt. Somit steht eine geeignete Deployment-Abstraktion für Host-Sprache und Glue-Sprache Komponenten zur Verfügung. Das Pattern Component Configurator [4] erlaubt zur Laufzeit Komponenten zu deployen und wieder aus dem System zu entfernen. Frag unterstützt dieses Pattern durch ein dynamisches Objekt-System und Introspection Options [7]. Dies bedeutet, dass Komponenten dynamisch zur Laufzeit als Packages geladen werden können und die Konfiguration dieser Komponenten nicht vor der Laufzeit bekannt sein muß.

Komponenten-Adaptierung Frag unterstützt die Pattern Interceptor [4] und Message Interceptor [7] durch ein Mixin-Konzept in der Sprache. Jede Komponente lässt sich dynamisch mit Vorher- und Nachher-Verhalten erweitern. Mit Interzeptoren kann man flexibel auf Schnittstellenänderungen, Versionsänderungen, und ähnliche Situationen reagieren. Frag kann als Indirection Layer [7] für Komponenten genutzt werden. D.h. Nachrichten im System können abfangen und manipuliert werden. Beispielsweise zeigen wir in [9], wie man Komponentenaufrufe für ein dynamisches Feature-Tracing umleiten kann. Diese Sprachmittel können auch zur Adaptierung beim dynamischen Komponenten-Deployment eingesetzt werden.

Schnittstellenbeschreibungen Frag unterstützt die Pattern Component Wrapper [3] und Wrapper Facade [4], welche das Einbinden von externen Komponenten und Libraries beschreiben. Über Component Wrappers und Schnittstellen-Klassen (wie Facades [2]) kann das Pattern Explicit Export/Import [3] unterstützt werden. Durch das Pattern Introspection Options [7], das in Frag für jedes Sprachelement umgesetzt ist, können diese Information zur Laufzeit abgefragt werden.

Komponentenkonfiguration Frag ist als Tcl-Erweiterung automatisch eine Command Language [8]. Command Languages werden oft für dynamische Konfigurierungen mit Verhalten benutzt. D.h. Frag ist entworfen, um dynamisch neues Verhalten in Komponenten einzubringen. Überdies können in Frag aber auch andere Metadaten als Annotations [6] und Metadata Tags [7] verwendet werden, wie bspw. XML-Formate. Diese dienen hauptsächlich der Konfiguration der Komponenten mit deklarativen Parametern und der Angabe von Metadaten über die Komponenten zur Selbstdokumentation.

31.3 Zusammenfassung

In diesem Papier haben wir den Pattern-basierten Entwurf der Sprache Frag beschrieben. Es wurden nicht alle der Pattern beschrieben, die in Frag zur Komponierung, Konfiguration und Adaptierung von Software-Komponenten eingesetzt werden, sondern nur die wesentlichen. Aber man sieht, dass auf diese Weise erhebliche Probleme in diesem Bereich in einer systematischen Art und Weise angegangen und sprachlich gelöst werden können. Anwendungsfälle der Sprache werden bspw. in [9] beschrieben.

Literaturverzeichnis

[1] M. DeJong and S. Redman. Tcl Java Integration. http://www.tcl.tk/software/java/, 2003.

[2] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[3] M. Goedicke and U. Zdun. Piecemeal legacy migrating with an architectural pattern language: A case study. *Journal of Software Maintenance and Evolution: Research and Practice*, 14(1):1–30, 2002.

[4] D. C. Schmidt, M. Stal, H. Rohnert, and F. Buschmann. *Patterns for Concurrent and Distributed Objects*. Pattern-Oriented Software Architecture. J. Wiley and Sons Ltd., 2000.

[5] C. Szyperski. *Component Software – Beyond Object-Oriented Programming*. ACM Press Books. Addison-Wesley, 1997.

[6] M. Völter, A. Schmid, and E. Wolff. *Server Component Patterns – Component Infrastructures illustrated with EJB*. J. Wiley and Sons Ltd., 2002.

[7] U. Zdun. Patterns of tracing software structures and dependencies. In *Proceedings of EuroPlop 2003*, Irsee, Germany, June 2003.

[8] U. Zdun. Some patterns of language and component integration. draft; submitted to EuroPLoP 2004, 2004.

[9] U. Zdun. Using split objects for maintenance and reengineering tasks. In *8th European Conference on Software Maintenance and Reengineering (CSMR'01)*, Tampere, Finland, Mar 2004.

[10] U. Zdun and M. Voelter. Europlop 2003 focus group on component composition. In *Proceedings of EuroPlop 2003*, Irsee, Germany, June 2003.

32 On Analyzing the Interfaces of Components

Jens Knodel
Fraunhofer Institute for Experimental Software Engineering (IESE), Sauerwiesen 6,
D-67661 Kaiserslautern, Germany
knodel@iese.fraunhofer.de

Abstract

Reusing existing software components can significantly reduce the effort needed for the development of new products. In order to enable reuse, existing conceptual components have to be identified, well documented and in some cases migrated into physical component. This paper presents an approach that helps when migrating parts of existing software systems into components to be reused in other projects.

Keywords: component, interfaces, request-driven reverse architecting, reverse engineering, software architecture

32.1 Introduction

Reuse is a promising solution for challenges for software-developing organizations and their need for reducing cost, effort and time-to-market, the increasing complexity and size of the software systems, and increasing requests for high-quality software and individually customized products for each customer.

Documented interfaces are one of the prerequisites for effective reuse of components. Reuse works when the developers know which functionality is provided by a component and how to access the functionality implemented in such a component. Components in the context of interface analysis are collections of source code entities (e.g., files, groups of logically related routines, single or groups of classes or packages, or even whole subsystems). The interface analysis technique can be applied for one of the following purposes:

- Reduction of the complexity of given components with respect to the number of offered routines by minimizing the provided interfaces to only the actually used interfaces when to facilitate reuse
- Documentation of source code spots in usage lists where to change accesses to a component when migrating the software system towards component-based development.
- Extension of architectural descriptions (e.g., the module and/or the code view, see [1]) by explicit notation of the provided functionality of a component.
- Migration of a group of entities towards an encapsulated component with explicit boundaries.

Our interface analysis technique reveals the connections of the subject component to the rest of the software system, or if it should be migrated into a separate component in future, it documents the spots to be changed and how the future component is embodied in the system. To achieve these goals we apply reverse engineering techniques in form of the interface analysis as described in the next sections.

32.2 Interfaces Analysis

The interface analysis technique is part of an analysis catalogue developed at Fraunhofer IESE. All analyses of this catalogue can be considered as request-driven reverse architecting analyses. Each analysis in this catalogue operates on a fact base produced by fact extraction from existing artifacts (e.g., architecture descriptions, the source code). An analysis is always initiated by a concrete request that delivers the results on demand. The results of an analysis are (architectural) views or subset of views. Such a view contains only selected aspects of the systems that are of interest in the context of the given request.

A prerequisite for the interface analysis is that the fact base contains interface related information (e.g., calls or method invocation of class methods, access to variables or class members). Figure 32.1 shows the inputs to the interface analysis, a component or a collection of entities that should be migrated towards a component.

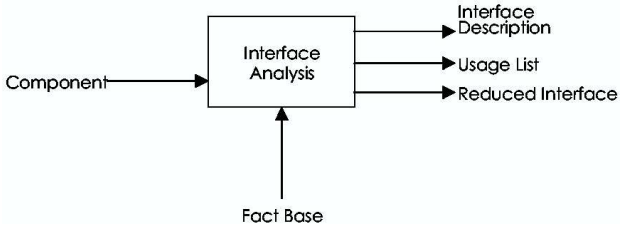


Abbildung 32.1: Interface Analysis

The inputs are then analyzed with respect to dependencies of the component (or the collection of entities) to the rest of software system whereby dependencies means for instance data dependencies, import relations, inheritance, or caller-callee dependencies. The resulting interface descriptions will be twofold:

- Required dependencies: the required dependencies are entities that the component needs in order to work properly. Usually a component communicates with other components to accomplish its tasks. In the migration context, it has to be decided whether those required source code entities become part of the to-be build component or they constitute a separate component. For reusing of a component it is important to know about such dependencies beforehand because they may influence the decision whether to reuse the component or not.
- Provided dependencies: the provided dependencies (or usage list) are the part of the given component

where the system accesses the component that is where it needs some functionality implemented in the component in order to work properly. There are two options, on the hand we can only document the currently used entities from the component (i.e., when changing the signature of a routine, it is beneficial to locate each place where a call has to be adjusted) and, on the other hand, we are able to document the complete interface offered to the outside (i.e., when reusing the component in another context, it is beneficial to know about the complete interface). To document both is required when planning to achieve both goals.

An interface description usually contains a list of routines, global variables, and class members that can be accessed from the outside. Variables and class members may lead to additional implementation effort for get and set access that should be scheduled in the migration. The interface description we produce includes the signature and the return parameter of the specific routines as well as the locations of different usages and in which file they are implemented. The description basically resolves the kind of dependencies which is present for each instance (e.g., calls, inheritance, data dependencies, imports, etc.), but it is possible to focus only on specific kinds of relations. Figure 32.2 shows an example for such an interface description.

```
Required Routines of class1.method1:
   Class2.method2
     Located in: c:/code/file2.java
        Type: method
        Signature: void method2
            ( value: int )
        Returns: void
```

Abbildung 32.2: Interface Description Example

Tools automate the task of analyzing the interfaces by querying the fact base. The queries can be parameterized by the list of components (or the collections of entities) for which the interfaces should be described. Single entities of a component will usually overlap in their interface description or their usage list. For this reason, we can create a single description for the whole component that contains no redundancies. These descriptions can be sorted by different criteria (e.g., places of usages, name of routines, classes, etc). The results of such an interface analysis can now be utilized to achieve the different purposes mentioned in the beginning.

32.3 Conclusion

The interface analysis documents the provided interfaces and the actually used interfaces of source code entities. This information can be used for the following purposes:

- Migrating collections of entities into an encapsulated component.
- Reducing the broadness of provided interfaces to only the actual needed and therefore reducing the complexity in terms of the size.
- Documenting the description and usage of entities in an automated way.

The presented interface analysis technique helps to counteract the degeneration of the interfaces. A more detailed description of the approach and how it is embedded in architectural design in a product family context can be found at [2]. We applied the interface analysis successfully in the context of two industrial case studies. Future work will involve the application in further case studies and the integration of the interface analysis together with other analysis we provide in our catalogue of request-driven reverse architecting analyses.

32.4 Acknowledgement

This work is partially funded by the European Commission under EUREKA 2023/ITEA-ip00009 „FAct based Maturity through Institutionalization Lessons-learned an Involved Exploitation of System-family engineering“ (FAMILIES).

32.5 References

[1] C. Hofmeister, R. Nord, D.: Applied Software Architecture, Addison-Wesley, 2000.

[2] J. Bayer, T. Forster, D. Ganesan, J.-F. Girard, I. John, J. Knodel, R. Kolb, D. Muthig: Definition of Reference Architectures based on Existing Systems, Technical Report, Fraunhofer IESE, March 2004

33 Evolution of Language Interpreters

Ralf Lämmel
Vrije Universiteit, De Boelelaan 1081a, NL-1081 HV Amsterdam
CWI, Kruislaan 413, NL-1098 SJ Amsterdam
ralf@cwi.nl

Keywords Operational Semantics, Evolution, Re-Engineering, Program Transformation, Language Interpreters, Language-Based Tools

Language-based tools involve some (ad-hoc) elements of a language's intended (say, formal) semantics. These elements relate to the specific service provided by a tool. For instance, a program analyser might implement a so-called abstract interpretation; a transformation tool might employ algebraic laws as well as typing and scoping rules; a translator implements the syntax-to-syntax edge in the commuting diagram with nodes for the two syntaxes and the two semantics of the involved languages.

Type checkers and language interpreters can be seen as prototypical examples of language-based tools. Such tools implement the language semantics rather directly. The extensibility of such semantics-oriented programs (and the underlying formal semantics descriptions) has received ample attention in the programming-language community. There exist highly advanced approaches to the reuse of language descriptions or components thereof, e.g., monadic-style semantics, action semantics, abstract state machines, strategic programming, modular SOS, and modular attribute grammars. This suggests that the domain of language descriptions is suitable for studying evolution of language-based tools in general.

We aim at a pragmatic, re-engineering-like approach to the evolution of language-based tools. That is, we provide a simple, transformation-based approach to the evolution of language-based tools that are encoded in simple rule-based notations. Our focus is on interpreters. We assume that language interpreters (or other language-based tools) are programmed in rule-based notations such as structural operational semantics, definite clause grammars, attribute grammars, conditional rewrite systems, or constructive algebraic specifications. The evolution of an interpreter is then represented by meta-programs on such rule-based object programs, while relying on a suitably designed operator suite for evolutionary transformations.

We present a series of examples for interpreter evolution. Starting from a simple expression language, we go through a development that touches upon pure and impure extensions, higher-order functional, object-oriented languages, and aspect-oriented languages, as well as semantics of different styles, i.e., small-step vs. big-step SOS.

Pointers

This approach is presented at some length in [1]. (An early instance of evolutionary transformations for rule-based programs is developed in the author's PhD thesis.) One specific proof-of-concept implementation of the approach is available as the Rule Evolution Kit [2].

Bibliography

[1] R. Lämmel. Evolution of Rule-Based Programs. *Journal of Logic and Algebraic Programming*, 73, 2004. 52 pages; Special Issue on Structural Operational Semantics; To appear.

[2] The Rule Evolution Kit, version 0.77, 11 Feb. 2004. http://www.cs.vu.nl/rek/.

34 Two co-transformations of grammars and related transformation rules

Wolfgang Lohmann
University of Rostock, Albert-Einstein-Str. 21, D-18051 Rostock
wlohmann@informatik.uni-rostock.de

34.1 Introduction

Despite of their basic role in software development, grammars are still often considered as static artifacts, used for language definition or once prepared for tool construction. Hence, there is insufficient support for working with grammars as software artifacts themselves. However, grammars are permanently changed, e.g. to debug them, to improve languages, to create grammars for tools. In [1] the authors demonstrate, how transformation tasks can be simplified, when underlying grammars are tailored especially to subtasks. The need for a discipline of grammar engineering has been emphasized by [2].

In the following, we will describe two examples of co-transformations. A co-transformation transforms mutually dependent software artifacts of different kinds simultaneously, while the transformation is centred around a grammar (or schema, API, or a similar structure) that is shared among the artifacts [3]. We are interested in consequences of grammar adaptations to transformations based on these grammars, and investigate, if the transformation rules can be migrated to work with the modified grammar.

34.2 Extending grammar rules

A slight change to a grammar by the insertion of a single non-terminal can lead to corrupt transformation rules, as these are based on the original grammar. While adapting the grammar information about the grammar changes can be collected. It can then be used to derive a migration of transformation rules by adapting the patterns which are based on the abstract syntax of the grammar. Since patterns are extended, it is necessary to define default values for introduced positions. Additionally, changed semantics has to be expressed with additional rules. The method is described more detailed in [4].

The approach has been used to tackle the problem of layout preservation in source-to-source transformations like refactoring. Non-terminals for layout information were introduced in the grammar at each terminal. The rules, for example, those for specifying a refactoring, had to be adapted to work with the extended abstract syntax. For the transport of layout information, a heuristics has been used.

Note, that the approach can also be used to simplify patterns for rewriting rules. Several transformation tasks need parts of the patterns only. The original grammar can be considered to be the extended subgrammar for that pattern. Thus, rules over the subgrammar can be migrated to work on abstract syntax trees according to the original grammar.

34.3 Semantics-preserving left recursion removal

Several tools for source-to-source transformation are based on top-down parsers. Top-down parsers are simple. Their structure is similar to the grammar, and it is easy to add semantics. However, they restrict the user to use grammars without left recursion. Removing left recursion of a given grammar often makes it unreadable, and prevents a rewriter to concentrate on the original grammar. Additionally, the question arises, whether the tool implements the semantics of the original language, if it is implemented based on a different grammar than in the original language definition. Moreover, existing implementations of semantics for the original grammar cannot be reused directly. A co-transformation on attribute grammars can help here.

A grammar and transformation rules on its abstract syntax can be considered as attribute grammar. It is possible to remove left recursion in the grammar and at the same time migrate the semantic rules. In the case of S-attribute grammars, each newly introduced non-terminal gets the synthesized attributes of the original non-terminal as well as inherited attributes of the same type. The computation is redirected to the inherited attribute of the non-terminal following the recursive one in the grammar rule. The ϵ-derivation causes a copy of the result computed so far to the corresponding synthesized attribute, which are then just copied upwards. Similarly, other types of attribute grammars can be adapted, like I-attributed and multi-pass attribute grammars. The approach is explained and justified in detail in [5].

Using the approach, it is possible to use small and easy top-down based tools for simple maintenance task with left recursion containing grammars recovered from YACC specifications, as long as they do not contain ϵ-productions. The approach also contributes to simplifying rewriting rules, since a programmer can continue to use semantic rules on a better readable left recursive grammar.

34.4 Final remarks

The two given examples are grammar adaptations, where much of necessary migrations of the related transformation rules can be derived automatically. The general concept is pictured in Fig. 34.1.

For a real support of grammar changes and related rules there is still much work to be done.

The given grammar adaptations can be applied to make writing transformations easier. The first helps in abstracting away unnecessary complexity when rewriting. The user can concentrate to specify the tasks while the automatic grammar adaptation and migration of rules takes care of the ‘uninteresting‘ parts.

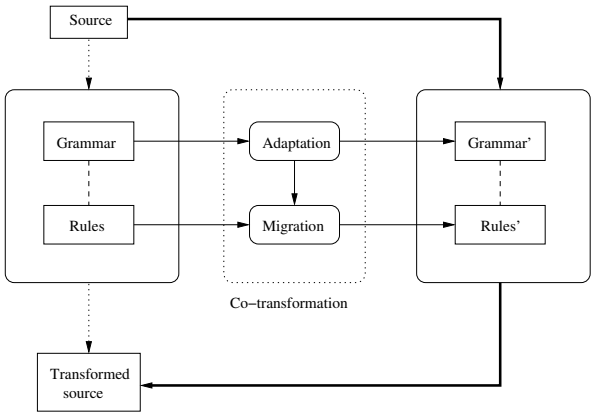


Figure 34.1: Co-transformation: grammars and rules

The second example enables the user to specify program transformations based on a grammar containing left recursion even if he uses top-down parsing technology. Hence, he is freed from using more complex grammar resulting from algorithm of left recursion removal. Moreover, it provides a method to reduce the distance of an original grammar in a specification and some 'tool'-tailored grammar necessary for technical reasons.

We are looking for more patterns, where a modification of the grammar induces a change to transformation rules over programs according to the grammar. A known example is the transformation of attribute grammars to an S-attribute grammar.

Acknowledgement

The work on automatic removal of left recursion while preserving semantics profited from collaboration with Markus Stoy during his diploma thesis.

Bibliography

[1] T.R. Dean, J.R. Cordy, A.J. Malton, and K.A. Schneider. Agile Parsing in TXL. *Journal of Automated Software Engineering*, 10(4):311–336, October 2003.

[2] P. Klint, R. Lämmel, and C. Verhoef. Towards an engineering discipline for grammarware. 32 pages, submitted for journal publication, August17 2003.

[3] Ralf Lämmel. Transformations everywhere. *Science of Computer Programming*, 2004. To appear; The guest editor's introduction to the SCP special issue on program transformation.

[4] Wolfgang Lohmann and Günter Riedewald. Towards automatical migration of transformation rules after grammar extension. In *Proc. of 7th European Conference on Software Maintenance and Reengineering (CSMR'03)*, pages 30–39. IEEE Computer Society Press, March 2003.

[5] Wolfgang Lohmann, Günter Riedewald, and Markus Stoy. Semantics-preserving migration of semantic rules after left recursion removal in attribute grammars. In *Proc. of 4th Workshop on Language Descriptions, Tools and Applications (LDTA 2004)*, 2004.

---

35 Experiences with lightweight checks for mass-maintenance transformations

Niels Veerman
Vrije Universiteit, De Boelelaan 1081a, 1081 HV Amsterdam, The Netherlands
nveerman@cs.vu.nl

Keywords automated modifications, lightweight approach, mass-maintenance

The use of automated modifications has gradually increased in the field of software maintenance. These maintenance transformations can be found where software needs to be maintained. Tools for code analysis have become more and more indispensable to maintain large systems, and large-scale modifications, such as Euro conversions, Y2K repairs or database migrations, have called for automated maintenance transformations, as argued for in [2].

The increasing interest in automated maintenance has heightened the need for control over automatic transformations. For instance, automated changes on business-critical systems should not jeopardize the operations of a company. Such changes usually consist of several complex transformations, making verification of an entire transformation on a large system difficult. Millions of lines of code can be affected by a mass update, and it is not very practical to inspect the resulting programs by hand. An alternative, extensive testing, can be expensive and not always feasible in practice. Automated massive changes are often carried out by software renovation companies, and they usually do not possess the required hard- and/or software to, for instance, compile and test a mainframe system. Although the owner of the system does have this option, compiling and testing can be an expensive exercise so possible errors should be detected in an early stage. Therefore, a different way is needed to control such maintenance transformations.

Another approach is to prove the correctness of a transformation in advance. However, to prove the correctness of mass-maintenance transformations, one would need the semantics of the programming language. Such an approach has several drawbacks. Besides the fact that there can be several different semantics for a language because of different dialects, compilers, compiler flags and operating systems (see [1]), it can be an expensive process and also prone to errors, and as soon as something changes (e.g. compiler version or operating system version) a different semantics needs to be developed. Moreover, one would also have to prove the used transformations system correct, for instance the preprocessor, parser and prettyprinter and so on. For these reasons, a correctness proof for mass-maintenance transformations on legacy systems is not practical, and we need other ways to make sure the updated systems are behaving as expected.

We have experimented with a lightweight approach to check large-scale maintenance transformations. Our starting point is two large-scale transformation projects that we carried out. Our approach concentrates on transformations for software maintenance, but can also be used in other areas. We present a range of checks to identify errors in transformations and discuss their application to real life cases.

Bibliography

[1] R. Lämmel and C. Verhoef. Cracking the 500-Language Problem. *IEEE Software*, pages 78–88, November/December 2001.

[2] C. Verhoef. Towards Automated Modification of Legacy Assets. *Annals of Software Engineering*, 9:315–336, 2000.