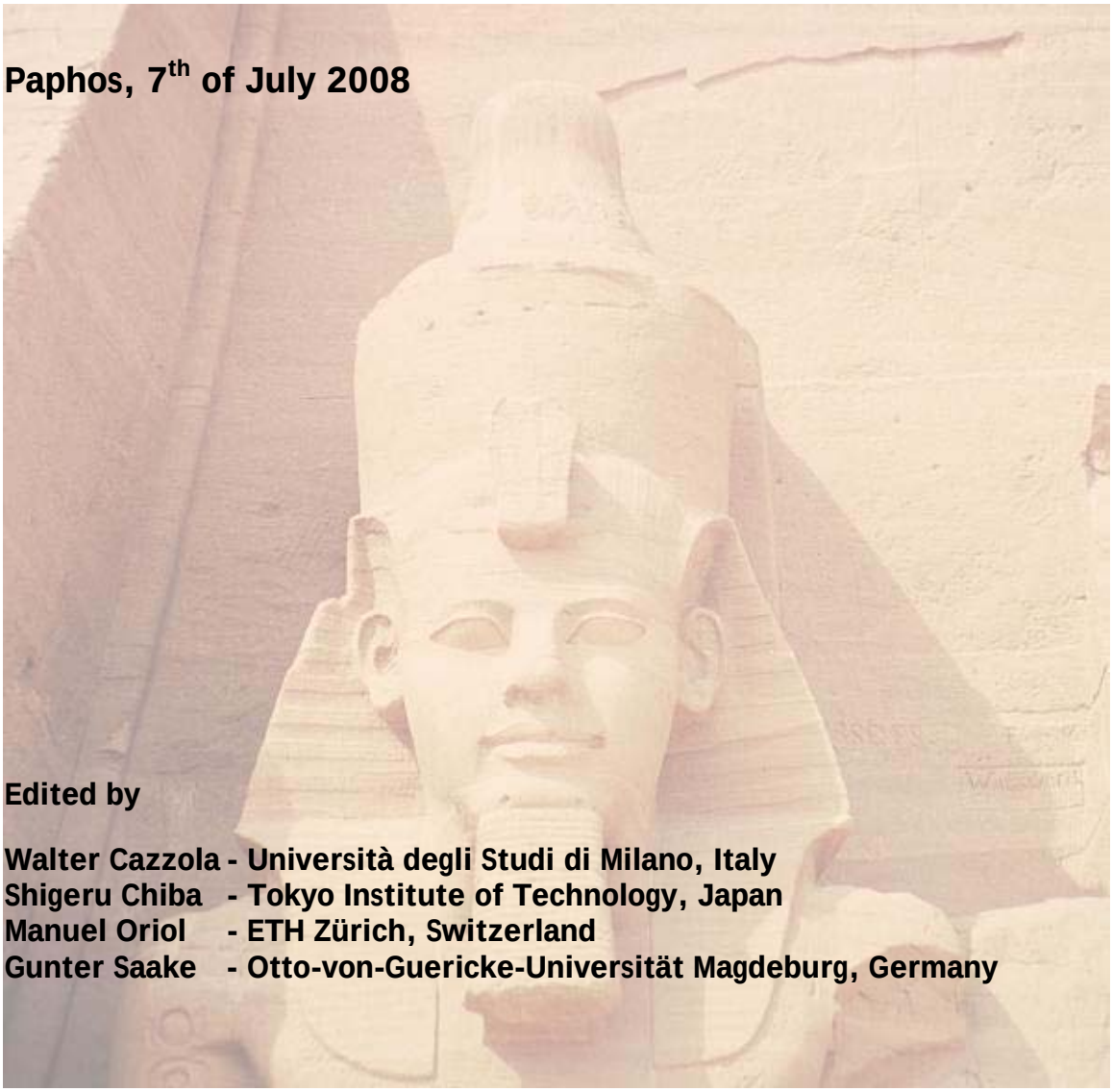
RAM-SE'08 – ECOOP'08 Workshop on Reflection, AOP, and Meta-Data for Software Evolution

(Proceedings)



Paphos, 7th of July 2008

Edited by

Walter Cazzola - Università degli Studi di Milano, Italy
Shigeru Chiba - Tokyo Institute of Technology, Japan
Manuel Oriol - ETH Zürich, Switzerland
Gunter Saake - Otto-von-Guericke-Universität Magdeburg, Germany

Preprint no. xx of University of Magdeburg.

Foreword

Software evolution and adaptation is a research area, as the name states, in continuous evolution, that offers stimulating challenges for both academic and industrial researchers. The evolution of software systems, to face unexpected situations or just for improving their features, relies on software engineering techniques and methodologies. Nowadays a similar approach is not applicable in all situations e.g., for evolving nonstopping systems or systems whose code is not available.

Reflection and aspect-oriented programming are young disciplines that are steadily attracting attention within the community of object-oriented researchers and practitioners. The properties of transparency, separation of concerns, and extensibility supported by reflection and aspect-oriented programming have largely been accepted as useful for software development and design. Reflective features have been included in successful software development technologies such as the Java language and the .NET framework. Reflection has proved to be useful in some of the most challenging areas of software engineering, including Component-Based Software Development (CBSD), as demonstrated by extensive use of the reflective concept of introspection in the Enterprise JavaBeans component technology.

Features of reflection such as transparency, separation of concerns, and extensibility seem to be perfect tools to aid the dynamic evolution of running systems. They provide the basic mechanisms for adapting (i.e., evolving) a system without directly altering the existing system. Aspect-oriented programming can simplify code instrumentation providing a few mechanisms, such as the join point model, that allow for the exposure of some points (*join points*) in the code or in the computation that can be modified by weaving new functionality (aspects) at those points either at compile-time, load-time, or ruin-time. Meta-data represent the glue between the system to be adapted and how it has to be adapted; the techniques that rely on meta-data can be used to inspect the system and to dig out the necessary data for designing the heuristic that the reflective and aspect-oriented mechanisms use for managing the evolution.

It is our belief that current trends in ongoing research in reflection, aspect-oriented programming and software evolution clearly indicate that an interdisciplinary approach would be of utmost relevance for both. Therefore, we felt the necessity of investigating the benefits that the use of these techniques on the evolution of object-oriented software systems could bring. In particular we were and we continue to be interested in determining how these techniques can be integrated together with more traditional approaches to evolve a system and in discovering the benefits we get from their use.

Software evolution may benefit from a cross-fertilization with reflection and aspect-oriented programming in several ways. Reflective features such as transparency, separation of concerns, and extensibility are likely to be of increasing relevance in the modern software evolution scenario, where the trend is towards systems that exhibit sophisticated functional and non-functional requirements. For example, systems that are built from independently developed and evolved COTS (commercial off-the-shelf) components; that support plug-and-play and end-user directed reconfigurability; that make extensive use of networking and internetworking; that can be automatically upgraded through the Internet; that are open; and so on. Several of these issues bring forth the need for a system to manage itself to some extent, to inspect components' interfaces dynamically, to augment its application-specific functionality with additional properties, and so on. From a pragmatic point of view, several reflective and aspect-oriented techniques and technologies lend themselves to be employed in addressing these issues. On a more conceptual level, several key reflective and aspect-oriented principles could play an interesting role as general software design and evolution principles. Even more fundamentally, reflection and aspect-oriented programming may provide a cleaner conceptual framework than that underlying the rather 'ad-hoc' solutions embedded in most commercial platforms and technologies, including CBSD technologies, system management technologies, and so on. The transparent nature of reflection makes it well suited to address problems such as evolution of legacy systems, customizable software, product families, and more. The scope of application of reflective and aspect-oriented concepts in software evolution conceptually spans activities related to all the phases of software life-cycle, from analysis and architectural design to development, reuse, maintenance, and, therefore also evolution.

The overall goal of this workshop – as well as of its previous editions – was that of supporting circulation of ideas between these disciplines. Several interactions were expected to take place between reflection, aspect-oriented programming and meta-data for the software evolution, some of which we cannot even foresee. Both the application of reflective or aspect-oriented techniques and concepts to software evolution are likely to support improvement and deeper understanding of these areas. This workshop has represented a good meeting-point for people working in the software evolution area, and an occasion to present reflective, aspect-oriented, and meta-data based solutions to evolutionary problems, and new ideas straddling these areas, to provide a discussion forum, and to allow new collaboration projects to be established. The workshop is a full day meeting. One part of the workshop will be devoted to presentation of papers, and another to panels and to the exchange of ideas among participants.

In this fifth edition of the workshop, we had an interesting keynote by Hidehiko Masuhara that has investigated why the abstraction mechanisms as AOP in programming languages are crucial for modular software development. This keynote was particularly interesting and raised several issues and lively discussion among the workshop attendees.

This volume gathers together all the position papers accepted for presentation at the fifth edition of the Workshop on Reflection, AOP and Meta-Data for Software Evolution (RAM-SE'08), held in Paphos on the 7th of July, during the ECOOP'08 conference. We received many interesting submissions and due to time restrictions and to guarantee the event quality we had to select only a few of them, the papers that, in our opinion, are more or less evidently interrelated to fuel a more lively discussion during the workshop. Now, a few months after the workshop, we can state that we achieved our goal. The presentations were interesting and the subsequent panels stimulated a lively and rich set of ideas and proposals. We are sure that in the next months we will see many papers by the workshop attendees and the fruit of such lively discussions.

The success of the workshop is mainly due to the people that have attended it and to their effort to participate to the discussions. The following is the list of the attendees in alphabetical order.

Arcelli, Francesca	Figueiredo, Eduardo	Masuhara, Hidehiko
Bierman, Gavin	Götz, Sebastian	Oriol, Manuel
Cazzola, Walter	Guerra, Eduardo	Ostrowski, Krzystof
Cech Previtali, Susanne	Havinga, Wilke	Pukall, Mario
Chiba, Shigeru	Herrmann, Stephan	Sanen, Frans
de Roo, Arjan	Kakousis, Constantinos	Vandemonde, Yves

We have also to thank the Department of Informatics and Communication of the University of Milan, the Department of Mathematical and Computing Sciences of the Tokyo institute of Technology and the Institute für Technische und Betriebliche Informationssysteme, Otto-von-Guericke-Universität Magdeburg for their various supports.

January 2009

W. Cazzola, S. Chiba, M. Oriol and G. Saake
RAM-SE'08 Organizers

Contents

Classic Software Evolution

A Case Study for Aspect Based Updating. 1
Susanne Cech Previtali and *Thomas R. Gross*
(ETH Zürich, Switzerland).

Runtime Adaptations within the QuaD^2-Framework. 7
Steffen Mencke, *Martin Kunz* and *Mario Pukall*
(Otto von Guericke University Magdeburg, Germany).

Modeling Context-Dependent Aspect Interference Using Default Logics. 15
Frans Sanen, *Eddie Truyen* and *Wouter Joosen*
(Katholieke Universiteit Leuven, Belgium).

Exploring Role Based Adaptation. 21
Sebastian Götz and *Ilie Şavga*
(Dresden University of Technology, Germany).

Annotations for Seamless Aspect Based SW Evolution. 27
Susanne Cech Previtali and *Thomas R. Gross*
(ETH Zürich, Switzerland).

Object Roles and Runtime Adaptation in Java. 33
Mario Pukall
(Otto von Guericke University Magdeburg, Germany).

A Case Study for Aspect-Based Updating

Susanne Cech Previtali and Thomas R. Gross

Department of Computer Science, ETH Zurich, Switzerland

Abstract. Rather than upgrading a software system to the next version by installing a new binary, software systems could be updated "on-the-fly" during their execution. We are developing a software evolution system that leverages aspect technology. As changes typically spread across several classes, we can handle updates like other crosscutting concerns: we encapsulate all changes, constituting a logical update, in one aspect. In this paper, we evaluate our approach. We report on a case study about the evolution of a Java application. The analysis provides details about how classes change between versions, and how these changes would be expressed in terms of updating aspects. Unfortunately, not all kinds of changes can be expressed using the aspect model. The results of our study, however, reveal that many changes fit our aspect-based approach.

1 Introduction

Dynamic software evolution represents an interesting technique to update software systems at run-time and is particularly helpful for systems that must be continuously available and up-to-date.

Our approach to the dynamic evolution of object-oriented software systems [1, 3] treats updates in a manner similar to crosscutting concerns in aspect-oriented programming: all changes that belong to a logical update are encapsulated in one aspect. We are developing a software evolution system that implements this idea. To compute the required updates, the system compares *statically* two complete versions of a Java program and deduces their structural differences. The structural differences constitute the individual changes. The system identifies the dependences between the changes and encapsulates these changes in an aspect. The dynamic aspect system PROSE [6–8] achieves dynamic software evolution by *dynamically* integrating the aspects.

In this paper, we report on a case study of an open-source Java program to determine if the evolution steps can be expressed as a sequence of updating aspects. The result of this case study reveals that—although not all evolution steps can be handled this way—most changes are limited to the implementation of methods and thus do not change the specification of classes. We show that many evolution steps can be decomposed and thus modularized, as indeed the actual changes concern "clusters" of few interacting classes.

The remainder of the paper is organized as follows: Sect. 2 explains the methodology of the case study. Sect. 3 presents the results of the evaluation and discusses the applicability of the updating model based on the results of the study. Sect. 4 concludes the paper.

2 Case Study

We have implemented a system to analyze compiled Java programs. All classes constituting the old and the new version of a program are compared to deduce the structural differences. First, classes, and recursively fields and methods, are matched as pairs to compute the sets of enduring, added, and removed entities. Second, the enduring entities are compared to compute the sets of unchanged and modified entities and the kinds of modifications. Based on the structural differences, the tool creates a method call-graph taking into account the static and dynamic target types and deduces the dependences between the changes. We refer to earlier work for a detailed description of the system architecture [1] and the corresponding algorithms [3]. Note that our current implementation matches two versions based on only the name and thus handles a rename as an removal and addition.

For the case study, we describe the evolution of a program from different points of view. First, we present the number of unchanged, modified, added, and removed classes. Then, we detail specific changes of the modified classes based on the actual modifications between two versions of an analyzed application. This data is based on the information available in the class file [4]: A **class** header stores the direct super-class and the interfaces a class implements, as well as the type parameters of generic classes. The header includes the access modifiers that determine whether a class is e.g., abstract, final, or synthetic. Furthermore, the header records the Java version. A **field** is characterized by its type, the access modifiers, the initial value, and generic parameters. A **method** is described by its body, the access modifiers, the return and argument types, exceptions, and generic parameters. Last, we discuss the updating aspects necessary to evolve the different versions.

3 Results

We have chosen Apache Tomcat 5.5, which implements version 2.4 of the Servlet and version 2.0 of the JSP specification, because it provides more than 20 releases. We downloaded the compiled releases of the "deployer" distribution and included all available Jar-files. **tomcat-5.5** initially consists of 399 classes, 1678 fields, and 3706 methods. In its latest release, **tomcat-5.5** includes 461 classes, 1902 fields, and 4348 methods.

The first part of Table 1 shows the coarse-grained evolution of **tomcat-5.5**. Mostly, classes are not changed between two versions except for release 5.5.1 when 60% of the existing 399 classes were modified. Classes were added only in six versions. With three exceptions, classes are never removed. Fields are mostly stable, on average 99% are not changed. Only in nine versions, fields are modified; and only in eleven versions, fields are removed. In eleven versions, fields are added; in particular, in version 5.5.3, 177 fields are added. Similar to fields, methods are very stable. On average, only 1% of the methods changes, less than 1% are added or removed.

Table 1. Evolution of **tomcat-5.5**.

	Evolution									Modification															
	Classes			Fields			Methods			Classes					Fields			Methods							
	Unchanged	Modified	Added	Removed	Unchanged	Modified	Added	Removed	Unchanged	Methods	Version	Fields	Super-class	Interfaces	Access	Type	Value	Body	Return type	Argument types	Access	Exceptions
5.5.0																							
→5.5.1	164	235	0	0	1676	1	3	1	3601	104	6	1	77	229	5	0	0	1	0	0	104	0	0	0	0
→5.5.2	384	15	0	0	1676	0	0	4	3668	41	1	2	15	0	3	0	0	0	0	0	41	0	0	0	0
→5.5.3	366	31	40	2	1640	0	177	36	3613	56	487	41	31	0	7	0	0	0	1	1	56	1	0	0	0
→5.5.4	422	15	0	0	1816	0	4	1	4114	38	5	4	15	0	2	0	0	0	0	0	38	0	0	0	0
→5.5.5	425	12	1	0	1819	1	1	0	4108	49	5	0	12	0	1	0	0	0	1	0	49	1	0	0	0
→5.5.6	430	8	0	0	1821	0	0	0	4133	29	0	0	8	0	0	0	0	0	0	0	29	0	0	0	0
→5.5.7	423	15	1	0	1820	0	10	1	4122	38	22	2	15	0	1	2	0	0	0	0	38	0	0	0	0
→5.5.8	420	19	0	0	1828	2	2	0	4138	44	2	0	18	0	3	0	0	0	2	0	44	1	0	0	0
→5.5.9	403	36	1	0	1816	13	19	3	4078	94	31	12	36	0	18	0	0	0	13	0	94	1	1	0	0
→5.5.10	413	25	1	2	1841	0	11	7	4136	61	15	6	25	0	5	0	0	0	0	0	60	1	0	0	0
→5.5.11	430	9	0	0	1852	0	0	0	4182	30	0	0	9	0	0	0	0	0	0	0	30	0	0	0	0
→5.5.12	417	17	17	5	1848	0	21	4	4159	41	61	12	17	0	2	0	0	0	0	0	41	0	0	1	0
→5.5.13	435	16	0	0	1868	1	0	0	4218	43	3	0	16	0	1	0	0	0	0	0	43	0	0	0	0
→5.5.14	442	9	0	0	1868	1	0	0	4230	34	0	0	9	0	1	0	0	0	0	0	34	0	0	0	0
→5.5.15	440	11	0	0	1868	1	1	0	4226	38	1	0	11	0	2	0	0	0	0	0	38	0	0	0	0
→5.5.16	440	11	0	0	1870	0	0	0	4232	33	0	0	11	0	0	0	0	0	0	0	33	0	0	0	0
→5.5.17	441	10	0	0	1870	0	0	0	4233	32	0	0	10	0	0	0	0	0	0	0	32	0	0	0	0
→5.5.20	439	12	0	0	1865	0	0	5	4227	38	0	0	12	0	1	0	0	0	0	0	38	0	0	0	0
→5.5.23	423	28	0	0	1857	4	3	4	4197	65	4	3	27	0	6	0	0	0	4	0	65	0	0	0	0
→5.5.25	431	20	0	0	1857	1	0	6	4214	50	0	2	20	0	4	0	0	0	1	0	50	0	1	0	0
→5.5.26	425	24	12	2	1846	0	56	12	4134	95	119	35	24	0	9	0	0	0	0	0	95	0	2	5	1

3.1 Modifications

In the following, we discuss the specific modifications shown in the second part of Table 1. We first present modified classes, then fields and methods.

Class modifications. The most frequent changes are method modifications. Only in version 5.5.1, the most frequent change regards the change of the Java version, when 60% of the classes were compiled from Java 1.2 to Java 1.4. The Java version (consisting of a major and minor version number in the class file header) defines the version of the class file format and consequently the minimal required Java virtual machine. A change in that version number may either reflect the migration of the Java development tools or the conscious usage of a new language feature. As we use the compiled application in bytecode form, we do not distinguish between the two cases. The inheritance structure of **tomcat-5.5** is very stable. Only in version 5.5.7, two classes extended different super-classes; and in version 5.5.26, one class removed an interface (i.e., java/lang/Serializable). Access modifiers and generic parameters are never changed (and are consequently omitted in the table).

Field modifications. In **tomcat-5.5**, fields are rarely changed. The access modifiers account for the most frequent change. This change consists of removing the modifier static (12 private fields and one protected in 5.5.9, two private fields in 5.5.23, one private field in 5.5.25). There is one change of the initial value in release 5.5.15. The few type changes refer to changes of a container type (i.e., java/util/Vector to java/util/List) or different representation (i.e., using a java/lang/ThreadLocal rather than a java/util/Hashtable keyed by thread-identifier for storing thread-local data).

Method modifications. The most prominent method change regards the change of the method body; in 99% only the body is changed. There are only a few changes of the return or argument types or access modifiers. These changes always imply an adaptation of the method body. Note that our analysis is conservative because it considers modifications to debugging information in the classfiles as method body changes. Debugging information is irrelevant for the update because it is ignored by the virtual machine. Consequently, the reported number of method body changes are an upper limit.

3.2 Updates

The updating approach cannot handle all kinds of changes. In such a case, the application cannot be updated at run-time and, as a consequence, must be restarted with the new version. For example, the updating model cannot update the superclass (one release in **tomcat-5.5**), as the aspect model does not provide a means to define such modifications. Type changes of fields require the adaptation of existing objects (four releases including a total of five changes). To enable such modifications, we can use an indirection mechanism that keeps a hidden extension field in the first version [5]). Using such a mechanism, the programmer needs to annotate field changes with according transformation functions. The updating system then installs aspects that wrap field accesses with the provided transformation functions [2]. As an alternative, we have described an extension

to the aspect system using a copying garbage collector that could iterate over the object graph thereby transforming the objects [1].

Table 2 shows the aspects necessary to update the application. Column *Aspects* shows the number of aspects that contain the number of methods given in the first column. Column *Advised classes* shows the number of aspects that advise the number of classes given in the header, in relation to the total number of methods contained in the aspect. Column *Virtual methods* indicates the number of aspects that redefine the number of virtual methods given in the header, in relation to the total number of methods contained in the aspect. Overall, the table shows that most aspects encompass only a small number of classes and methods. Additionally, the number of virtual methods updated is small and explicit dispatching is therefore rarely required.

Table 2. Necessary updating aspects.

Methods	Aspects	Advised classes			Virtual methods			
		1	2	3	0	1	2	12
1	976	976	0	0	976	0	0	0
2	36	13	23	0	19	17	0	0
3	10	8	2	0	2	0	8	0
4	4	1	2	1	3	0	1	0
5	5	1	3	1	2	3	0	0
6	1	0	0	1	1	0	0	0
7	1	0	1	0	1	0	0	0
10	2	1	1	0	1	1	0	0
13	1	0	1	0	1	0	0	0
23	1	1	0	0	0	0	0	1
24	1	0	0	1	0	1	0	0

4 Concluding remarks

We analyzed more than 20 releases of **tomcat-5.5** that capture four years of its evolution. The results of this case study confirm our expectations: **tomcat-5.5** exposes fairly localized changes and thus allow the modular decomposition of an update. There are various evolution steps the updating model can handle, and software developers may consider dynamic aspect-based updating as an alternative approach to achieve dynamic software evolution.

Acknowledgments. The work presented in this paper was partially supported by the National Competence Center in Research on Mobile Information and Communication Systems (NCCR-MICS), a center supported by the Swiss National Science Foundation under grant number 5005-67322.

References

1. S. Cech Previtali and T. R. Gross. Dynamic Updating of Software Systems Based on Aspects. In *22nd IEEE International Conference on Software Maintenance (ICSM'06)*, pages 83–92, 2006.
2. S. Cech Previtali and T. R. Gross. A Case Study for Aspect-based Updating. In *5th ECOOP Workshop on Reflection, AOP and Meta-Data for Software Evolution (RAM-SE'08), Paphos (Cyprus)*, July 2008.
3. S. Cech Previtali and T. R. Gross. Extracting Updating Aspects from Version Differences. In *LATE '08: Proceedings of the 2008 AOSD Workshop on Linking Aspect Technology and Evolution*, pages 1–5, New York, NY, USA, 2008. ACM.
4. T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley Publishing Company, Inc., Reading, Massachusetts, 2nd edition, 1999.
5. I. Neamtiu, M. Hicks, G. Stoyle, and M. Oriol. Practical Dynamic Software Updating for C. In *ACM SIGPLAN 2006 Conference on Programming Language Design and Implementation (PLDI'06)*, pages 72–83, 2006.
6. A. Nicoară, G. Alonso, and T. Roscoe. Controlled, Systematic, and Efficient Code Replacement for Running Java Programs. In *ACM SIGOPS/EuroSys European Conference on Computer Systems 2008 (EuroSys'08)*, 2008.
7. A. Popovici, G. Alonso, and T. Gross. Just-in-time Aspects: Efficient Dynamic Weaving for Java. In *2nd International Conference on Aspect-Oriented Software Development (AOSD'03)*, pages 100–109, 2003.
8. A. Popovici, T. R. Gross, and G. Alonso. Dynamic Weaving for Aspect-Oriented Programming. In *1st International Conference on Aspect-Oriented Software Development (AOSD'02)*, pages 141–147, 2002.

Runtime Adaptations within the QuaD^2-Framework

Steffen Mencke, Martin Kunz and Mario Pukall

Otto-von-Guericke University, P.O. Box 4120, 39016 Magdeburg, Germany

Abstract The importance of providing integration architectures in every field of application is beyond controversy these days. Unfortunately, existing solutions are focusing mainly on functionality. But for the success of Systems Integration in the long run, the quality of developed architectures is of substantial interest. Therefore, a framework for quality-driven creation of architectures is proposed in [1]. The idea fundamentally bases on functional and non-functional runtime adaptations.

1 Introduction

Due to manifold advantages of high-flexible infrastructures compared to monolithic products a lot of initiatives propose approaches for the integration of single components (e.g. services, content). Semantic metadata provides the basis for the automation of this process. But those approaches lack from a throughout consideration of empirical data. Either only functional requirements or single quality attributes are taken into consideration.

The presented general QuaD^2-Framework (Quality Driven Design) is intentionally described in an abstract way to enable an applicability to different fields, e.g. e-learning content provision, service oriented architectures and enterprise application integration. For this reason a general terminology is used and special domain-specific instantiations are described elsewhere (e.g. in [2] or [3]).

In contrast to existing approaches the QuaD^2-Framework reveals a holistic orientation on quality aspects. It combines semantic web technologies for the fast and correct assembly of elements and quality attribute evaluations for the best possible assembly decisions.

Several points of runtime adaptations reveal the advantages of the framework in order to enable an up-to-date entity assembly and presentation. In fact that targets the quality-driven selection of appropriate entities as well as the experience-based selection of process and quality models.

2 QuaD^2-Framework

The major goal of the described core process is the assembly of an infrastructure consisting of single entities. Such an entity is metadata-annotated functionality and may be depicted by e.g. services, agents or content fragments in concrete applications.

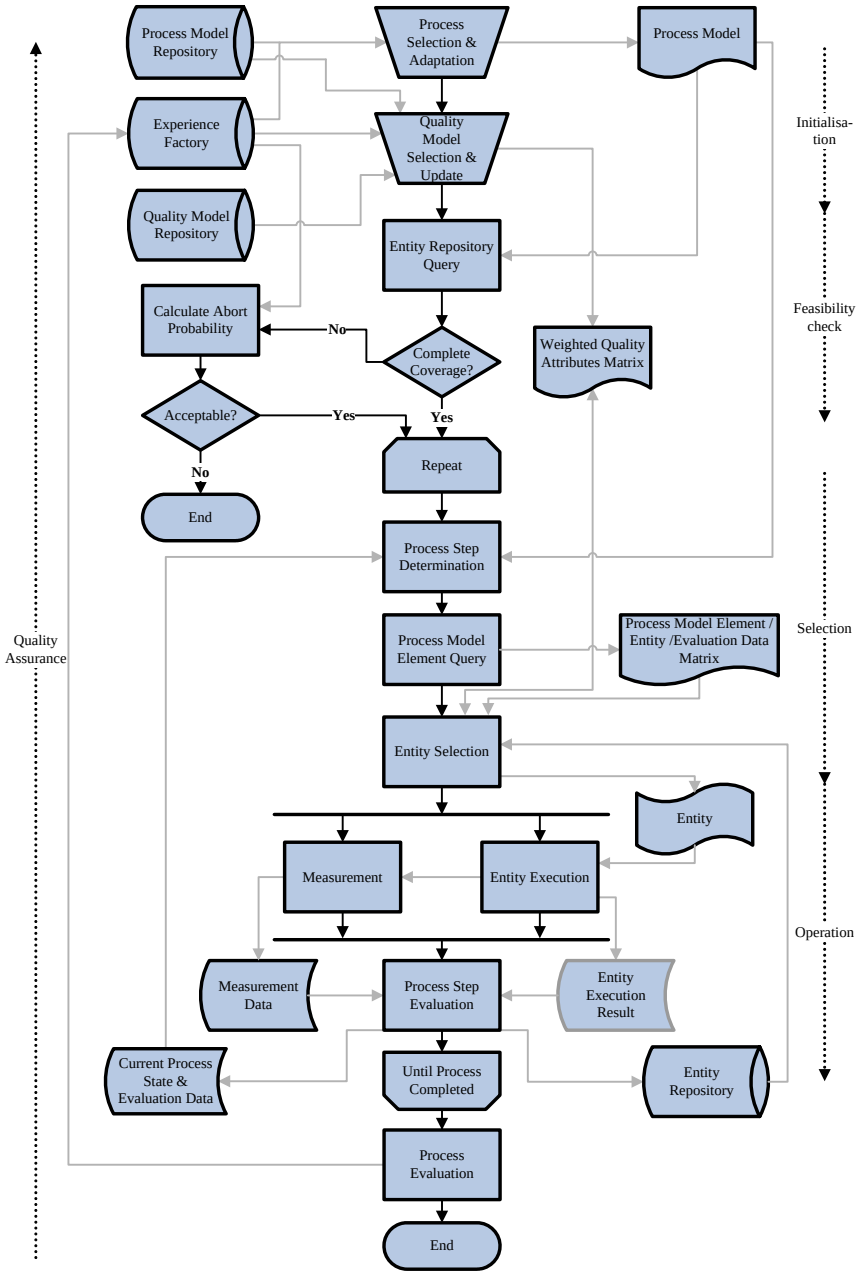


Figure 1. QuaD^2-Framework [1].

The QuaD^2-Framework is shown in Figure 1.

In general the subprocesses of this empirical-based assembly process are the initialization, the feasibility check (checking the functional coverage), the selection process based on empiricism as well as the operation of the established application. The basis of the approach is a collection of semantically-annotated sources: the process model repository, the entity repository, a quality model repository and furthermore an experience factory.

The process model repository is the source for process models that serve as descriptions for the functionality of the aspired distributed system. Example for such processes can be e.g. didactical approaches descriptions [4].

An important source for empirical quality evaluations are quality models being provided by a quality model repository. The specification of a certain quality model is realized by selecting and weighting appropriate attributes.

The entity repository contains entities, their semantic description and their evaluation data regarding all defined quality attributes.

The selection and adoption of process models and quality models are difficult tasks which constitutes the need for guidance and support. Based on the work of Basili and Rombach the usage of an Experience Factory is proposed, that contains among others an Experience Base and Lessons Learned [5].

3 Runtime Adaptation

Runtime adaptation is performed at several points within the framework. In fact, that targets the experience-supported selection of an adequate process model, the experience-supported selection of an appropriate quality model as well as the functional entity selection.

3.1 Process Model Selection

The selection of an appropriate process model that defines the functional requirements for the parts of the later distributed system is the first step. Due to the fact, that such a choice can be a manual process, it should be supported by an experience factory providing knowledge and experiences - lesson learned - for the decision for or against a specific process model for the current need. The process model essentially base on semantic metadata to allow the later automatic mapping of semantically described entity functionalities to the functional requirements specified by the process model. According to [6] only formal descriptions of those models are applicable.

With the chosen process model a set of concrete distributed systems within the specified functional range is possible.

3.2 Quality Model Selection

The second step of the presented approach is a selection of a quality model from a quality model repository. This is intended to be done automatically. For certain domains manual adaptations can be more efficient. A manual individualization of this predefined set of quality attributes as well as of their importance weighting is also possible. For these purposes an experience factory can be helpful again.

As a result of this step a process model and importance-ranked quality attributes are defined. Thereby the quality-related aspects of the framework are adapted to the specific needs of the particular user.

3.3 Quality-Driven Entity Selection

With these process model and quality model information, process step three is able to determine whether enough available entities exist to provide an acceptable amount of functionality demanded by the process model. If there is no acceptable coverage after the negotiation subprocesses, then an abort probability based on already collected data can be computed. The user needs to decide whether he accepts the probability or not. If not the distributed system provision process will be aborted.

In the case of an acceptable coverage the runtime subprocesses can start. The first determines the next process step to be executed following the process model. Therefore information about the last process steps can be taken into consideration to optimize the next process step execution. Now, up-to-date entity information, their evaluation values as well as the data of the quality model are available to identify the best entity possible.

Following the defined necessities and given data the entity selection is formally described below. For the following formulas let PM be the chosen process model. Function $f^{funct}(PM)$ specified in Formula 1 is used to determine the set of entities E from the entity repository. Each of them can deliver the functionalities specified within the chosen process model (cp. Formula 2).

$$f^{funct} : \text{Process model} \mapsto \{\text{Entity}, \dots\}. \tag{1}$$

$$E = f^{funct}(PM). \tag{2}$$

Using the classic normalization approach presented in Formula 3 (normalizing to the interval from 0 to 1), the evaluation values $v_{i,j}$ of quality requirements j defined in the quality model must be normalised for each entity i. These $v_{i,j}$ are the measurement/simulation values to anticipate the optimal decision for the next process step.

$$v_{i,j}^{norm} = \frac{v_{i,j} - min(v)}{max(v) - min(v)}. \tag{3}$$

With the help of the weighted requirements matrix from the (maybe adjusted) quality model the last step - the identification of the optimal entity according to the empirical data and the quality model QM - can be performed (see Formulas 4 to 8). Formula 4 adjusts the normalized evaluation values to ensure proper calculation. If $v = 1$ describes the best quality level then no adjustments are necessary, otherwise a minimum extremum is desired and $1 - v$ must be calculated.

$$f^{mm}(v) = \begin{cases} v & \text{if a maximal } v \text{ is the best,} \\ 1 - v & \text{if a minimal } v \text{ is the best.} \end{cases} \tag{4}$$

$$f^{eval}(e_i) = \{\sum_{j=0}^{n-1} f^{mm}(v_{norm}^{i,j}) | e_i \in E \wedge n = |QM|\}. \tag{5}$$

$$V = \{f^{eval}(e_i) | \forall e_i \in E\}. \tag{6}$$

$$e^{worst} = e_{index}, index = min(\{x | v_x = min(V)\}) \wedge e_{index} \in E. \tag{7}$$

$$E' = E \backslash e^{worst}. \tag{8}$$

To determine the best evaluated entity, Formulas 5 to 8 are repeated until E' contains only 1 element. It provides the needed functionality and is the most appropriate one according to the specified quality model.

3.4 Process Model Types for Adaptation

The process models may vary in their basic structure according to the special, application-dependent requirements. According to this, their processing and thereby the automated adaption can be classified [7] Amongst others, the following types can be identified.

Sequential: Sequential process models are used for the modeling of sequential assembly and execution processes. Conditions are used to define functional decisions and to thereby create the adapted target system: maybe an adapted infrastructure, an e-Learning course or a measurement infrastructure.

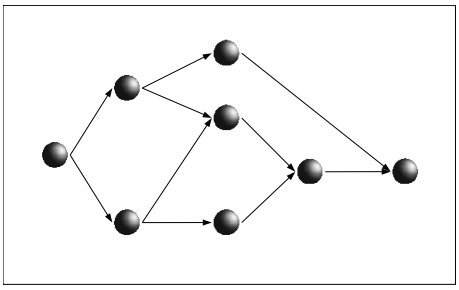


Figure 2. Sequential Process Models

Sequential with separated supervision: Sequential process models with separated supervision are used for the modeling of sequential assembly and execution processes, too. They additionally include downstream supervision process steps.

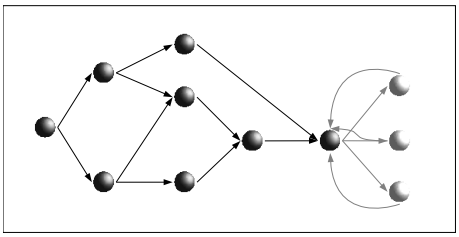


Figure 3. Sequential Process Models with Separated Supervision

Sequential with integrated supervision: Sequential process models with integrated supervision are similar to the one described above. In contrast, the supervision points back to the creation process steps.

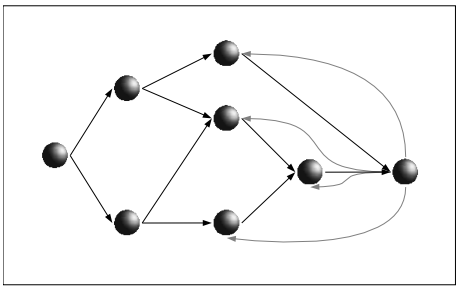


Figure 4. Sequential Process Models with Integrated Supervision

Supervision: Supervision process models only target the supervision of an existing system. Several conditions point away from a central event handling process step.

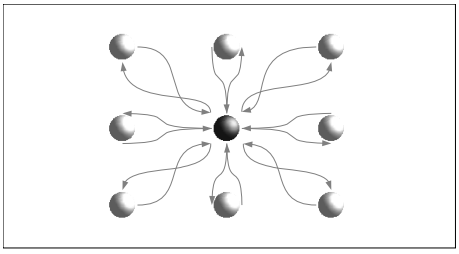


Figure 5. Supervision Process Models

Externally influenced: All types of process models being described above can be externally influenced by events outside the currently defined model. Thereby, meta-dependencies can be modeled.

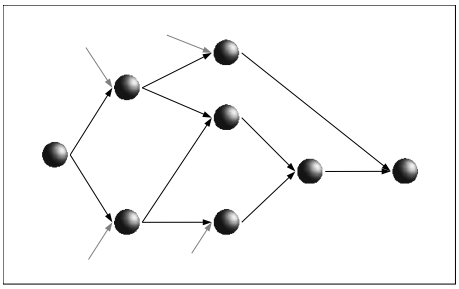


Figure 6. Externally Influenced Process Models

4 Conclusion and Further Work

The QuaD^2-Framework can be implemented using various technologies as e.g. ontologies, web services and agents. The presented quality-driven approach uses semantic descriptions for processes automation and supports different quality models and quality attribute evaluations.

Automatic quality measurement, evaluation and quality-driven entity selection within the general QuaD^2-Process are major building blocks for an high quality automatic runtime adaptation.

An implementation of this approach for specific systems is currently being performed. For the areas of e-Learning systems [2] and software measurement infrastructures [3] first components are realized.

References

1. Kunz, M., Mencke, S., Rud, D., Dumke, R.: Empirical-Based Design – Quality-Driven Assembly of Components. In: Proceedings of the IEEE International Conference on Information Reuse and Integration (IRI 2008), Las Vegas, Nevada, USA (2008)
2. Mencke, S., Dumke, R.R.: A Hierarchy of Ontologies for Didactics-Enhanced E-learning. In Auer, M.E., ed.: Proceedings of the International Conference on Interactive Computer aided Learning (ICL2007), Villach, Austria (2007)
3. Kunz, M., Schmietendorf, A., Dumke, R., Wille, C.: Towards a Service-Oriented Measurement Infrastructure. In: Proceedings of the 3rd Software Measurement European Forum (SMEF 2006), Rome, Italy (2006) 197–207
4. Mencke, S., Dumke, R.: Didactical Ontologies. Emerging Technologies in e-Learning (iJET) **3**(1) (2008) 65–73
5. Basili, V.R., Caldiera, G., Rombach, H.D.: The Experience Factory. In Marciniak, J.J., ed.: Encyclopedia of SE. Volume 1. John Wiley & Sons (1994) 511–519
6. Mencke, S., Zbrog, F., Dumke, R.: Useful e-Learning Process Descriptions. In: Proceedings of the 4th International Conference on Web Information Systems and Technologies (WEBIST 2008). Volume 1., Funchal, Madeira, Portugal, INSTICC Press (2008) 460–463
7. Mencke, S.: Proactive Ontology-Based Content Provision in the Context of e-Learning. PhD thesis, Otto-von-Guericke University of Magdeburg (2008)

Modeling context-dependent aspect interference using default logics

Frans Sanen, Eddy Truyen, and Wouter Joosen

Distrinet Research Group, Department of Computer Science, K. U. Leuven,
Celestijnenlaan 200A, B-3001 Heverlee, Belgium
{frans.sanen, eddy.truyen, wouter.joosen}@cs.kuleuven.be

Abstract. Explicitly representing aspect interactions is vital so that they can be shared and used in the course of system evolution. As a consequence, guidance can be given to the software developer and automated support for handling interactions becomes possible. In this paper, we propose to use default logics for modeling context-dependent aspect interference. We motivate and illustrate our work by an example interference from the domotica world.

Keywords: Aspect interactions, knowledge, interference, default logics.

1 Introduction

Aspect interference is a well-known challenging problem with current aspect-oriented programming technology. As it has already been motivated in [10], explicitly representing aspect interactions results in an important form of knowledge that can be shared and used in the course of system evolution. If specified formally enough, software systems can exploit this knowledge to autonomously reconfigure themselves to detect and resolve undesired aspect interferences, by using existing safe dynamic reconfiguration support similar to the one in [13].

In this paper, we want to make the case for modeling support for context-dependent interferences. We define aspect interference as a conflicting situation where one aspect that works correctly in isolation does not work correctly anymore when it is composed with other aspects. A context-dependent interference is an interaction that might or might not occur if certain aspects are composed depending on the runtime context at hand. Or more formally: "Given an aspect A that is woven into a system S, there exists a set of contextual conditions C_A associated with aspect A such that, when at least one element of C_A evaluates to true, the execution of the aspect A will cause an error in the execution of system S. A contextual condition is defined as a boolean expression that evaluates over properties of the context in which the aspect is deployed – contextual properties." Obviously, the context of aspect A does not only consist of the system S but also involves all the other aspects that are simultaneously woven into S. As a consequence, context information entails key information pieces that we need to express.

We consider this particular problem of context-dependent aspect interferences in the case of aspect-oriented middleware [14, 7, 13] which uses AOP for implementing middleware services. Subtle aspect interferences exist in a middleware environment. Consider the example of a power saving aspect and an integrity aspect using symmetric encryption [11]. A symmetric encryption key has a limited lifetime and therefore should be regenerated upon expiration, which is very computationally

intensive. Only when the power of the device being used is low and the key is about to expire, interference arises between both the power saving and integrity aspect.

A prerequisite for the scenario of systems capable of autonomously reconfiguring themselves to resolve context-dependent interferences is that interaction knowledge has to be specified in an unambiguous way. We have found no satisfactory solutions in current work on interaction modeling. We will elaborate on this later in the paper.

The rest of this paper is structured as follows. Section 2 elaborates on the need for modeling context-dependent interactions. It also shortly indicates that current approaches lack sufficient support in this regard. We propose to use default logics for modeling context-dependent interactions in Section 3 before concluding in Section 4.

2 Modeling aspect interactions

To be able to share and use aspect interactions in the course of system evolution, we need a means for modeling them. Some work already exists where interactions are modeled separately, but to the best of our knowledge, these suffer from several shortcomings, especially in the context of context-dependent interactions. In the NFR framework [2], Chung et al. introduce the concept of correlating (i.e. interacting) non-functional requirements. It for instance can be expressed that using a compressed format to store information deteriorates (*hurts*) its response time. However, such a representation cannot take into account the concrete context in which the interaction arises, e.g. when the CPU load is above a certain threshold. Similarly, interaction modeling in feature models [4, 6] allows you to express that feature *A* requires or excludes feature *B*, but this is not flexible enough to provide any means to model the context on which an interaction depends. Classen et al. [3] consider feature interactions as the simultaneous presence of several features causing malfunctions, hence ignoring the potential context dependence of an interaction. Finally, Pawlak et al. [8] propose a language to abstractly define an execution domain, advice codes and their often implicit execution constraints. Especially the latter are relevant because exactly these represent the context in which undesired effects occur, e.g. a network overload situation. These conditions are key information pieces we need to express.

The pedagogical example interaction we will use throughout the rest of this paper is situated in a home integration system product line context and borrowed from [5]. Home integration systems are a new and emerging set of systems combining features in the area of home control, home security, communications, personal information, health, etc. Each feature easily can be mapped to one or more aspects implementing it. Imagine a domotica product that helps to protect the housing environment. On the one hand, your personal product entails a flood control feature which shuts off the water main to the home during a flood. On the other hand, it also contains a fire control feature that turns on some sprinklers during a fire. Turning the sprinklers on during a fire and flooding the basement before the fire is under control results in a really undesirable interaction since the flood control feature will shut off the home's water main, rendering the sprinklers useless. As a result, your house further will burn down.

In order to have a correct representation for our example interaction, three scenarios have to be considered: (1) the basement is flooded, (2) a fire in the house is detected and (3) the basement is flooded as a result of the sprinklers trying to extinguish the fire.

Traditional methods and technologies often offer support to prioritize features in relationship with one another. However, we are convinced that such a prioritization not always is feasible to overcome context-dependent interactions. One of the main reasons is because priorities are far less flexible. First of all, an interaction between two features having the same priority cannot be resolved. Secondly, the priority of two features related to one another can be different in varying circumstances. For instance, suppose there are two additional features included in your domotica product: a presence simulation feature that turns lights on and off to simulate the presence of the house occupants and a doorkeeper feature which controls the access to the house and allows occupants to talk to the visitor [12]. Obviously, we would like the doorkeeper not to give away the fact that nobody is at home if there is an unidentified person in front of the door to prevent the owners from a burglary.

3 Using default logics

Default logics haven been originally proposed by Reiter [9] as a non-monotonic logic to formalize reasoning with default assumptions. It allows us to make plausible conjectures when faced with incomplete information and draw conclusions based upon assumptions. [1] As an intuitive example of what can be expressed, consider the well-known principle of justice in our Western culture: "In the absence of evidence to the contrary, assume that the accused is innocent." In this section, we shortly will overview both the syntactic sugar and semantics (informally) of default logics by applying it to our example interaction from above. Next, we discuss the relevance of using default logics in our example.

3.1 Syntax and semantics

A default theory T is a pair (W, D) consisting of a set W of predicate logic formulas (background theory or *facts* of T) and a set D of defaults. The default explicitly representing our example interaction is presented below (1) and should be thought of being used together with the classical rule that is also shown (2).

$$\frac{waterInBasement: \neg active(fireControl)}{active(floodControl)} \quad (1)$$

$$fireDetected \rightarrow active(fireControl) \quad (2)$$

According to default (1), if we know that $waterInBasement$ is true and $\neg active(fireControl)$ can be assumed, we can conclude $active(floodControl)$. Because of rule (2), $active(fireControl)$ will be concluded upon fire detection.

The three parts of a default rule are called the *prerequisite* φ, *justifications* ψ_i and *conclusion* χ respectively. Hence, the general explanation of any default rule is given by "if we believe that *prerequisite* is true, and the *justification* is consistent with our current beliefs, we also believe the *conclusion*". In other words, given a default $\varphi: \psi_1, \psi_2, \ldots / \chi$, its informal meaning is: if φ is known, and if it is consistent to assume ψ_1,

ψ_2, … then conclude χ. It is consistent to assume ψ_i iff the negation of ψ_i is not part of the background theory *W*.

At this point, it is important to realize that classical logic is not appropriate to model this situation. Imagine the following rule as an alternative for (1).

$$waterInBasement \wedge \neg active(fireControl) \rightarrow active(floodControl) \qquad (3)$$

The problem with this rule is that we have to definitely establish (basically because of the closed world assumption) that the fire control feature is not active before applying this rule. As a consequence, the flood control service never would be able to become active.

The semantics of default logic typically is defined in terms of extensions. Intuitively, an extension seeks to extend the set of known facts (i.e. background theory) with "reasonable" conjectures based on the applicable defaults. More formally, a default φ: ψ_1, ψ_2, … / χ, is applicable to a deductively closed set of formulas *E* iff $\varphi \in E$ and $\neg\psi_1 \notin E$, $\neg\psi_2 \notin E$, ... You can think of *E* as the context in which φ should be known and with which ψ_i should be consistent.

3.2 Discussion

We will now revisit our default (1) together with its semantics. Intuitively, this rule states that the flood control service will be activated upon detection of water in the basement, unless the fire control feature is active. It is easy to see that with this representation all possible scenarios are represented correctly. In each of these scenarios, the set *D* of defaults contains default (1). The only two facts that are relevant when searching extensions are $waterInBasement$ and $fireDetected$.

If, on the one hand, a sensor detects water in the basement, then the background theory *W* will include $waterInBasement$. Because of default (1), the only valid extension is the one where flood control service will become active (we conclude $active(floodControl)$ because $waterInBasement$ (the prerequisite) is true and the justification $\neg active(fireControl)$ is not inconsistent with what is currently known.

On the other hand, if a fire is detected by the system, *W* will include $fireDetected$ and classical rule (2) fires so that $active(fireControl)$ also becomes true in the extension. If later (the third scenario), as a consequence, the basement will be flooded, default (1) can no longer be applied. Note that this is exactly what we wanted.

In our approach, the context in which an interaction occurs is made explicit via one or more justifications in a default rule. By taking certain conditions into account, the solution of the interaction lies in the fact that the justifications need to be invalidated in order to have a correct functioning system. Because of this, an interaction is prevented from occurring while normal execution behavior also easily can be captured and isn't influenced.

4 Conclusion

To conclude, we started from the observation that modeling aspect interactions results in an important form of knowledge that can be shared and used in the course of

system evolution. We propose to use default logics for representing aspect interactions. The main advantage of this approach is that the interaction becomes explicit in the justification part of a default rule. Therefore, undesired interactions can be prevented from happening by invalidating one of the justifications of the default rule representing the interaction.

Acknowledgments. This work is partially funded by the Interuniversity Attraction Poles Programme Belgian State, Belgian Science Policy, Research Fund K. U. Leuven and European Commission grant IST-2-004349: European Network of Excellence on Aspect-Oriented Software Development (AOSD-Europe), 2004-2008.

References

1. Antoniou, G.: A tutorial on default logics. ACM Computing Surveys 31 (4), pp. 337-359, 1999.
2. Chung, L., Nixon, B. A., Yu, E., Mylopoulos, J.: Non-functional requirements in software engineering. Kluwer academic publishing, Norwell, 2000.
3. Classen, A., Heymans, P., Schobbens, P.: What's in a Feature: A Requirements Engineering Perspective. Proceedings of the 11th International Conference on Fundamental Approaches to Software Engineering (FASE'08), pp. 16-30, 2008.
4. Czarnecki, K., Eisenecker, U. W.: Generative Programming. Addison Wesley, London, 2000.
5. Kang, K.C., Lee, J., Donohoe, P.: Feature-oriented product line engineering. IEEE Software, vol. 19, no. 4, pp. 58-65, 2002.
6. Kang, K., Cohen, S., Hess, J., Novak, W., Peterson, A.: Feature-oriented domain analysis (FODA) feasibility study. Technical report CMU/SEI-90-TR-021.
7. Lagaisse, B., Joosen, W.: True and transparent distributed composition of aspect components. 7th International Middleware Conference, pp. 41-62, 2006.
8. Pawlak, R., Duchien, L., Seinturier, L.: CompAr: Ensuring safe around advice composition. 7th Internation Conference on Formal Methods for Open Object-Based Distributed Systems, 2008.
9. Reiter, R.: A logic for default reasoning. Artificial Intelligence 13 (1-2), pp. 81-132, 1980.
10. Sanen, F., Truyen, E., Joosen, W.: Managing concern interactions in middleware. 7th International Conference on Distributed Applications and Interoperable Systems, 2007.
11. Sanen, F., Truyen, E., Joosen, W., Jackson, A., Nedos, A., Clarke, S., Loughran, N., Rashid, A.: Classifying and documenting aspect interactions. Proceedings of the 5th AOSD Workshop on Aspect, Components, and Patterns for Infrastructure Software, pp. 23-26, 2006.
12. Schwanninger, C. et al.: Confidential list of requirements on a Totally Integrated Home platform. Siemens internal document, 2006.
13. Truyen, E., Janssens, N., Sanen, F., Joosen, W.: Support for Distributed Adaptations in Aspect-Oriented Middleware, 7th International Conference on Aspect-Oriented Software Development, 2008.
14. Truyen, E., Vanhaute, B., Joosen, W., Verbaeten, P., Jorgensen, B.: Dynamic and Selective Combination of Extensions in Component-based Applications. 23rd International Conference on Software Engineering, pp. 233-242, 2001.

Exploring Role-Based Adaptation

Sebastian Götz and Ilie Şavga

Department of Computer Science, Dresden University of Technology, Germany,
{sebastian.goetz|is13}@mail.inf.tu-dresden.de

Abstract. The adapter design pattern [1], commonly used for integration and evolution in component-based systems, is originally described by *roles*. In class-based systems, the conventional realization of the pattern spuriously maps these roles to classes. The recent appearance of mature languages supporting roles as first order programming constructs poses the question whether realizing this pattern directly in roles offers benefits comparing to class-based realization. This paper explores the feasibility of role-based adaptation and discusses its benefits and challenges.

1 Introduction

When assembling independently developed components, it is often the case that their public interfaces do not fit to each other. If components cannot be adjusted directly (e.g., when assembling third-party components), an adapter needs to be placed between them to bridge interface incompatibilities. Gamma et al. [1, p. 139] describes the adapter design pattern by 4 collaborating roles (*Client*, *Target*, *Adapter* and *Adaptee*) and shows a possible pattern implementation as a mapping of these roles to classes.

For our running example, assume a university management system (UMS), in which the concept of student is modeled by interface `Student` and implemented by class `StudentImpl`. Among other interface methods, the class implements the `getGrades` method that retrieves subjects and grades of the student from a file used for serialization. This method is used also in the implementation of `printGrades` that prints out subjects and grades of a student.

Later, due to new system requirements, it is decided to buy a sophisticated reporting component that replaces the simple functionality previously realized directly by `StudentImpl`. Moreover, UMS is integrated with a persistence component that is now responsible for saving and retrieving data. To retrieve student grades, `StudentImpl` must now call the persistence component to get data. To print this information, `StudentImpl` must wrap it before sending to the reporting component, because the signature of printing method in `Student` (expected by existing clients) differs from the one of the reporting component (`Report.printReport` expecting report component's specific `DataRow` as its parameter). So, `StudentImpl` must translate between the two interfaces and becomes effectively a class-based adapter (Figure 1).

Figure 2 shows internals of the `printGrades` method of `StudentImpl` that performs the actual translation. Using student identity (for simplicity, "this"), the method constructs the corresponding SQL query, retrieves data from the persistence component using the `getGrades` method and fills them into the type required by the reporting

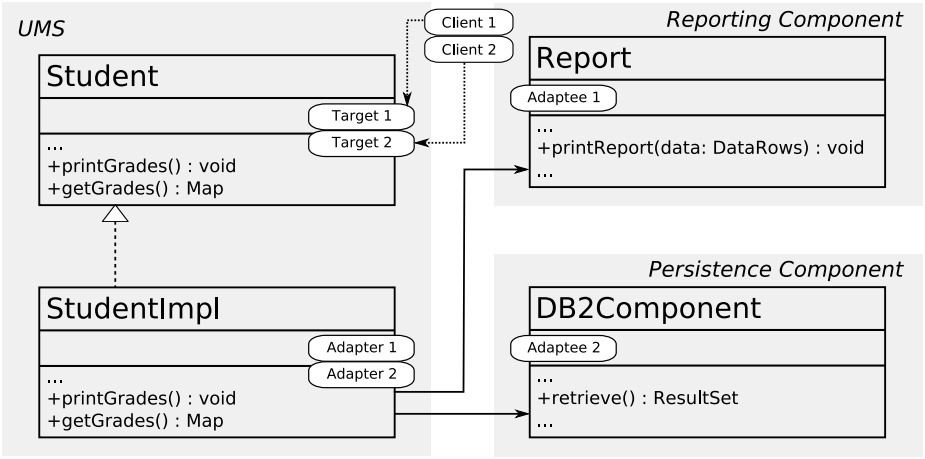


Fig. 1. Class-based adapter. Classes are annotated with roles these classes implement.

component. In addition, now the `getGrades` method (code not shown) itself is an adaptation method calling `retrieve` of the newly introduced persistence component and converting its `ResultSet` to the `Map` of the `Student` interface being adapted.

The main drawback of this class-based adapter realization is that the code responsible for different tasks is highly intertwined. For instance, in lines 11 and 12 the code realizing logic for the data retrieval and for reporting concerns is joined. When realizing this adapter, developers need to consider in fact the static types and semantics of all three domains involved (i.e., of the report and persistence components and of the UMS itself). In real life scenarios with possibly many interrelated components be-

```
1  Report report;
2  DBComponent db;
3  public void printGrades() {
4    //construct an SQL query for this student
5     String query = createSQLQueryByTime(this);
6     //retreive student subject-mark pairs
7     ResultSet srs = this.getGrades(query);
8     //fill in and send the report data
9     DataRow reportData = new DataRow();
10    while (srs.next()) {
11     reportData.add(srs.getString(''Subject''));
12     reportData.add(srs.getString(''Mark''));
13   }
14   report.printReport(reportData);
15 }
```

Fig. 2. Implementation of `StudentImpl.printGrades`

ing integrated, such inability to separately realize each concern increases the time and error-proness of adaptation. Even more important, an adapter is itself a software artifact inevitably requiring maintenance. In case the adapter needs to be modified (for example, to improve its performance), developers need to understand its often extremely complex implementation.

The situation aggravates furthermore when the public interfaces of components, on which the adapter depends, evolve as well. In our running example, an upgraded version of the report component may change the signature of `Report.printReport`. For example, in an older component version, its void method was throwing an exception in case of a printing failure and in the new version the method returns a new type `DocumentPrinting` containing details of method's execution. To accommodate the adapter to these changes, its whole code needs to be thoroughly investigated and understood. Often this needs to be done by developers others then the adapter's initial developers. Because the adaptation decisions are made dependent on each other in the code, a bug made when adjusting one component may propagate to other adapter's parts. For instance, if `getGrades` of the persistence component evolves and a bug is made when adjusting to its changes, this bug will also be reflected in the behavior of the adapter's `printGrades`.

All in all, these maintenance problems stem from the fact that the adaptation concern mentally modeled by four roles of [1] is lost in transition to the class-based adapter implementation. Presumably, preserving these roles explicit in the implementation brings benefits comparing to class-based adaptation. Using a language supporting roles as first-class citizens, we investigate the feasibility, benefits and drawbacks of role-based adaptation.

2 Role-based Adaptation

To implement the role-based adapter of our running example, we use a relatively new yet rich language ObjectTeams/Java—a stable well-tested Java extension supporting roles and collaborations [2]. However, since the language consists of several specific terms that need lengthy explanation, in this paper we refrain from its specific terminology. Instead, taking into consideration the run-time responsibilities, we dissect the concept of the Adapter role into Mediator, *In*- and *Out*-(sub)roles. Similarly to conventional aspects, an In-role is responsible for handling the incoming data into the adapter and an Out-role is in charge of passing data flow further to the adaptee. The Mediator represents the description of the collaboration—it mediates between In- and Out-Roles. This separates adaptation code responsible for communication with the target (In-Role) and adaptees (Out-Roles) from integration logic (Mediator), i.e. the description of how target and adaptees collaborate.

Figure 3 depicts how role-based adaptation can be realized for our running example. Each role is played by (instances of) and mapped to a single class. The key difference to the class-based adapter is that the target and adaptee roles are realized directly as roles. The adapter role is represented by the mediator, which is realized as a class. This is because the mediator describes the collaboration, but is not part of it and thus is not a role. As a consequence, integration logic is decoupled from target and adaptees.

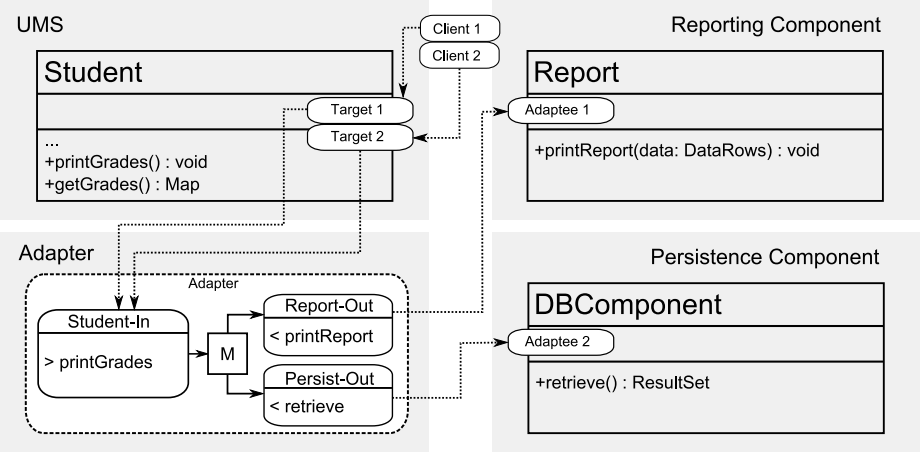


Fig. 3. Role-based adapter. Roles annotate classes playing those roles.

Additionally there is another separate role for each target class. Note, that In- and Out-roles for each adaptation role in concern are realized separately and are encapsulated in the corresponding role realizations, preserving thus adaptation decisions. Regardless of how they are actually mapped to classes and boil down to the execution code, these explicit roles can be maintained separately and do not intertwine. Moreover, adding new adapter's responsibilities (e.g., to adapt yet another commercial component) becomes easier due to the separation of adaptation concerns. If the structure or behavior of target or adaptees change, only the appropriate In- or Out-Roles need to be adjusted. Thus role-based adapters lead to less effort for maintenance. In a class-based realization, such separation is only possible using the complex role object pattern [3], which is in fact a workaround of language limitations to realize roles directly.

3 Challenges and Limitations

An important conceptual issue to be mentioned is that applying role-based adaptation to adapt class-based components reduces potential power of a pure object-based design (as envisioned by Reenskaug [4]). In our case it is not possible to realize the pattern only in roles, because at least some of them need to be bound to actual components' classes. In particular, in a strongly-typed class-based system, at least the target class needs to be specified statically.

A limitation inherent to ObjectTeams/Java is that a role can only have a single base class. As a consequence, the adapter's roles cannot be realized by instances of different, unrelated classes at run-time. This decreases reuse, because the adapter's roles, once defined, can only be used for a single class. If another class needs the same functionality, another role needs to be defined again, possibly duplicating the same implementation.

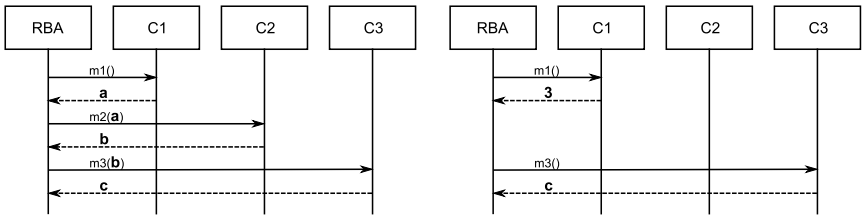
The major practical challenge we stipulate regarding role-based adaptation is that the learning curve implied by the application of a new technique may not be accepted

by developers. Since developers are in general reluctant to learn new programming languages and, even more important, have to admit a certain degree of obsoleteness of their conventional class-based adapter realization, it is not clear, whether such technique can be easily accepted by them.

4 Future Work

The applicability and feasibility of role-based adaptation needs to be supported by empirical data collected in an *experiment*. For the experiment two teams of five students each are needed, where the first team uses conventional class-based adapters and the second team role-based adapters. The experiment consists of two phases. In advance to the experiment both teams develop the initial version (without adapters) of the university management system (UMS) as introduced in Section 1, i.e. data is retrieved from a comma separated file and grades are printed to the console. In the first phase both teams realize data retrieval using a persistence component and printing using a reporting component. In the second and last phase both teams need to adjust their adapters according to a set of changes of the integrated components. Throughout the experiment time measurement and software metrics are used to collect empirical data. Additionally surveys shall be answered by each student initially (expectations) and after each phase. The collected data enables an empirical comparison of how much initial effort is needed to introduce role-based adapters (first phase) and how much less effort is needed for maintenance in case role-based adapters have been used (second phase).

The collaboration of In- and Out-Roles to integrate a set of components is often of the same form. Components are often connected in a sequential order, i.e. the returned values of the first component are passed as arguments to the second component and so forth. Figure 4(a) depicts such collaboration in a sequence diagram like notion. Remind the running example, where the persistence components return value is passed as an argument to the reporting component. The mediator in such scenarios is always the same, except for the In- and Out-Roles it uses. Another often recurring scenario is the usage of the first components return value to decide which of the other integrated components is to be used. The mediator *picks* one of the Out-Roles, based on the first components return value. Figure 4(b) depicts such collaboration. The first components (C1) return value (3) is used by the role-based adapter to decide that the third component is to be



(a) Sequential role-based adapter. (b) Pick role-based adapter.

Fig. 4. Special kinds of role-based adapters.

used. It may thus be beneficial to describe *abstract, generic mediators*, which are parameterizable by In- and Out-Roles. Further special kinds of mediators and mechanisms to describe parameterizable role-based adapters are to be identified.

5 Conclusion

Using conventional class-based realization of adapters leads often to highly complex adaptation code that is hard to understand, maintain and evolve. The role-based realization of adapters leads to more initial coding effort and requires the developers to learn a new programming language. But long term benefits are worth these initial efforts.

A role-based realization of adapters in a language supporting roles as first-order programming constructs avoids the spuriously mapping of roles to classes and may considerably *reduce code complexity* due to the separation of adaptation concerns in the resulting implementation. Even more important, such realization *preserves initial adaptation decisions* made and contributes furthermore to the maintainability of adapters. Moreover developers are able to focus on one adaptation concern at a time, which leads to *easier development of adapters*. Highly skilled developers will not sense simplification, because they are used to write complex adapters. But for example new members of a developer team are able to understand existing adapters in much shorter time.

We will further investigate the frontiers of role-base adaptation, its practical realization, advantages and limitations in the Bachelor thesis of one of the paper's authors [5].

References

1. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, Reading, Massachusetts (1995)
2. Herrmann, S., Hundt, C., Mosconi, M.: ObjectTeams/Java Language Definition - version 1.0. Technical Report 2007/03, Technical University Berlin (2007)
3. Bäumer, D., Riehle, D., Siberski, W., Wulf, M.: The role object pattern. In: PLoP'97: Proceedings of the 4th Pattern Language of Programming Conference. (1997)
4. Reenskaug, T.: Working with Objects: The OOram Software Engineering Method. Manning Publications (1996)
5. Götz, S.: Role-based adaptation (2008) http://www1.inf.tu-dresden.de/~s9288421/papers/goetz-gb-thesis.pdf.

Annotations for Seamless Aspect-Based Software Evolution

Susanne Cech Previtali and Thomas R. Gross

Department of Computer Science, ETH Zurich, Switzerland

Abstract. We are developing a dynamic software evolution system that leverages aspect technology to encapsulate software updates. Ideally, an evolution system provides as much automation as possible. Certain changes, however, defeat automation. For instance, field additions cannot be concisely captured without the feedback of the programmer. Rather than reconstructing the missing information in retrospect, we propose to gather the necessary meta-data along with the development process. We use Java annotations for that purpose: for instance, programmers may annotate added fields with their corresponding initialization. In some cases, the software development environment may even infer annotations from the actions taken by the programmer, and therefore, annotations enable a seamless software evolution cycle.

1 Introduction

Our approach to the dynamic evolution of object-oriented software systems [3, 4] treats updates in a manner similar to crosscutting concerns in aspect-oriented programming: all changes that belong to a logical update are encapsulated in one aspect. We are developing a software evolution system that implements this idea. To compute the required updates, the system compares *statically* two versions of a Java program in bytecode form and deduces their structural differences. The structural differences constitute the individual changes. The system identifies the dependences between the changes and encapsulates these changes as aspects. For deploying the aspects, the system relies on a dynamic aspect sub-system [7–10].

The software evolution system automates the version comparison, the deduction of update dependences, and the generation of aspects. The calculation of dependences and aspects depends on the extracted differences. Unfortunately, not all differences can be easily detected by a tool; e.g., name changes require special heuristics [5] or human interaction [2]. Furthermore, adding or changing the type of fields requires the programmer to provide code for transforming existing objects.

In this paper, we discuss how the programmer may convey meta-data in the form of Java annotations during software development and maintenance to automate the program comparison for update generation. The remainder of the paper is organized as follows: Sect. 2 discusses related work. Sect. 3 introduces annotations for our software evolution system. Sect. 4 concludes the paper.

2 Related work

The dynamic software updating (DSU) [5] system detects name changes of functions by partially matching the abstract syntax tree of successive versions of C programs. JDiff [1, 2] is a tool to compare object-oriented programs. The tool analyzes the bytecode from class files and compares programs at a statement-level granularity. To detect name changes, the authors propose human interaction during the process of comparison. Currently, our implementation does not detect name changes and consequently handles a rename as an addition and delete.

The DSU system [6, 12] generates type transformer functions, which must be completed by the developer to provide explicit conversions. Type evolution is handled by wrapping the original type and adding a version number and fixed-size extra space to prepare for eventual growth of the type over time. For each field access, the compiler inserts code to return the underlying representation. As type evolution is limited by the fixed amount of extra space reserved in the initial version, the authors mention the use of indirection in type definitions at the cost of an extra dereference per access. We suggest also to use indirection to accommodate object evolution, but, rather than providing type transformations when the release is ready, programmers encode the transformations along with the development, once they are aware of the details.

Robbes et al. [11] propose to record semantic changes from the refactoring information provided by an software development environment such as Eclipse to understand software evolution. Instead, we use refactoring information to generate Java annotations to achieve dynamic software evolution.

3 Annotations

In the following, we present how Java annotations may be used to tag changes and how the software evolution system can exploit this information.

3.1 Name changes

We propose a simple meta-data based approach to handle name changes. The approach relies on annotations that are generated by the software development environment whenever the programmer uses the renaming refactoring facility. The annotation must indicate the previous name of the renamed entity. Figure 1 shows two versions of a class. The new version (on the right) renames the method `start()` to `run()`; the annotation above the method declaration denotes the old name.

Currently, our implementation handles a rename as an addition and delete. With the information from the annotations, we can link these pairs and leave it up to the system to take proper action. The system may replace the new names with the original ones in the compiled class files or may update the symbol table.

```
/* Version 1 */
class S {

   void start() { ... }
}
```

```
/* Version 2 */
class S {
   @Rename (previous="start")
   void run() { ... }
}
```

Fig. 1. Annotations for renaming.

3.2 Field changes

Annotations may also facilitate field modifications that require object evolution, i.e., the adaptation of existing objects at run-time. Such changes modify the old object layout, by e.g., adding another field or by changing the type of an existing field.

Field additions. Adding a new field to a type requires all existing objects to be modified to support reading and writing this new field. Following Neamtiu et al. [6], we propose an indirection system in which a special field, ext, is added to all classes to support future growth. When a class must be updated with a new field, a special "delta class" is generated containing any added fields. Existing objects have their ext field set to an instance of this delta class, and field access in the original code is rewritten to accomodate the extra layer of indirection. Although the constructor in a new version initializes new objects appropriately (i.e., including the added field), special care must be taken to "initialize" the added field in existing objects. Using annotations, a programmer can indicate a method to initialize the added field.

Figure 2 provides an example. The new class declaration in the middle shows the added field field3 annotated with the name of the initialization method. This initialization method must be declared in the new class itself. The class declaration on the left contains a field ext of type Object to accommodate future field additions. The class declaration on the right lists the delta class; its special constructor links the existing object to this extension.

```
/* Version 1 */
class S {
   A field1;
   B field2;
   Object ext;
}
```

```
/* Version 2 */
class S {
   A field1;
   B field2;
   @Initialize(name="init")
   C field3;

   void init() {
      field3 = field1.addTo(field2);
   }
}
```

```
/* Update from version 1 to version 2 */
class DeltaS {
   C field3;
   Object ext;

   /* Special constructor */
   DeltaS(S old) {
      field3 = old.field1.addTo(old.field2);
      old.ext = this;
   }
}
```

Fig. 2. Annotations for field additions.

Type transformations. Another update concerns the modification of the type of an existing field. We propose two kinds of strategies to accommodate these changes. One strategy encompasses bidirectional type transformations. A bidirectional type transformation allows converting back and forth between two versions of a type. Examples are String changed to StringBuffer and int to Integer. The other strategy encompasses unidirectional type transformation. In contrast to bidirectional type transformation, unidirectional type transformations are not reversible. Examples are LinkedList changed to HashMap and int to float.

Both bidirectional and unidirectional type transformations may be expressed as annotations in the new version of the class. The annotation is attached to the corresponding field and indicates the necessary type transformation method. The example in Fig. 3 shows the new version of class S (on the left), which changed the type of its field from String to StringBuffer. The corresponding updating aspect (on the right) redefines a caller method C.run() to redirect the access using these transformation methods oldToNew() and newToOld() defined in the delta class DeltaS (not shown).

```
/* Version 2 */
class S {
  @BidirectionalTransformation
  (toNew="oldToNew", toOld="newToOld")
  StringBuffer msg;

  StringBuffer oldToNew(String prev) {
    return new StringBuffer(prev);
  }
  String newToOld(StringBuffer next) {
    return next.toString();
  }
}
```

```
/* Update from version 1 to version 2 */
class Update extends Aspect {
  /* Setters/getters of S, version 2 */
  void setMessage(S s, StringBuffer msg) {
    s.msg = DeltaS.newToOld(msg);
  }
  StringBuffer getMessage(S s) {
    return DeltaS.oldToNew(s.msg);
  }
  /* Redefine client using class S */
  redefine C.run() {
    S s = new S();
    setMessage(s, new StringBuffer("hallo"));
    StringBuffer msg = getMessage(s);
  }
}
```

Fig. 3. Annotations for type transformations.

Unidirectional type transformations may be handled similar to field additions. The field of the new type is added to the existing object and the type transformation function is used to initialize the added field. Consider a LinkedList that is replaced by an array; the elements of the array can be filled with the elements of the LinkedList. Field access must be reflected in the methods that use the fields. These methods will be redefined by updating aspects.

4 Concluding remarks

We have shown how meta-data in the form of Java annotations may facilitate software evolution. Such annotations are created along with the development

process and thus allow the programmer to indicate transformations while writing the new version. Annotations can either be directly provided by the programmer or inferred by the software development system. In the latter case, the software development system considers actions taken by the programmer and formulates an appropriate annotation. By including annotations and type transformations, the software evolution system may automate program comparison and update generation, and thus achieves a seamless evolution step.

Acknowledgments. The work presented in this paper was partially supported by the National Competence Center in Research on Mobile Information and Communication Systems (NCCR-MICS), a center supported by the Swiss National Science Foundation under grant number 5005-67322.

References

1. T. Apiwattanapong, A. Orso, and M. J. Harrold. A Differencing Algorithm for Object-Oriented Programs. In *19th IEEE International Conference on Automated Software Engineering (ASE'04)*, pages 2–13, 2004.
2. T. Apiwattanapong, A. Orso, and M. J. Harrold. JDiff: A Differencing Technique and Tool for Object-Oriented Programs. *Journal of Automated Software Engineering*, 14(1):3–36, 2007.
3. S. Cech Previtali and T. R. Gross. Dynamic Updating of Software Systems Based on Aspects. In *22nd IEEE International Conference on Software Maintenance (ICSM'06)*, pages 83–92, 2006.
4. S. Cech Previtali and T. R. Gross. Extracting Updating Aspects from Version Differences. In *LATE '08: Proceedings of the 2008 AOSD Workshop on Linking Aspect Technology and Evolution*, pages 1–5, New York, NY, USA, 2008. ACM.
5. I. Neamtiu, J. S. Foster, and M. Hicks. Understanding Source Code Evolution Using Abstract Syntax Tree Matching. *SIGSOFT Software Engineering Notes*, 30(4):1–5, 2005.
6. I. Neamtiu, M. Hicks, G. Stoyle, and M. Oriol. Practical Dynamic Software Updating for C. In *ACM SIGPLAN 2006 Conference on Programming Language Design and Implementation (PLDI'06)*, pages 72–83, 2006.
7. A. Nicoară and G. Alonso. Dynamic AOP with PROSE. In *International Workshop on Adaptive and Self-Managing Enterprise Applications (ASMEA'05)*, pages 125–138, 2005.
8. A. Nicoară, G. Alonso, and T. Roscoe. Controlled, Systematic, and Efficient Code Replacement for Running Java Programs. In *ACM SIGOPS/EuroSys European Conference on Computer Systems 2008 (EuroSys'08)*, 2008.
9. A. Popovici, G. Alonso, and T. Gross. Just-in-time Aspects: Efficient Dynamic Weaving for Java. In *2nd International Conference on Aspect-Oriented Software Development (AOSD'03)*, pages 100–109, 2003.
10. A. Popovici, T. R. Gross, and G. Alonso. Dynamic Weaving for Aspect-Oriented Programming. In *1st International Conference on Aspect-Oriented Software Development (AOSD'02)*, pages 141–147, 2002.
11. R. Robbes and M. Lanza. A Change-based Approach to Software Evolution. *Electronic Notes in Theoretical Computer Science (ENTCS)*, 166:93–109, 2007.
12. G. Stoyle, M. Hicks, G. Bierman, P. Sewell, and I. Neamtiu. *Mutatis Mutandis*: Safe and Flexible Dynamic Software Updating. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 29(4), 2007.

Object Roles and Runtime Adaptation in Java

Mario Pukall

Otto-von-Guericke University, P.O. Box 4120, 39016 Magdeburg, Germany
pukall@iti.cs.uni-magdeburg.de

Abstract. Program maintenance usually decreases the programs availability. This is not acceptable for highly available applications. Thus, such applications have to be changed at runtime. Furthermore, since it is not predictable what changes become necessary and when they have to be applied, highly available applications have to be enabled for unanticipated runtime adaptation at deploy-time [1]. We developed an object role-based approach which deals with these requirements.

1 Introduction

Maintenance of highly available applications, such as banking systems or security applications, is a cost-intensive task. This is due to the fact that maintenance usually causes time periods of unavailability. Unfortunately, such programs can not be prepared statically (i.e., at compile or load-time) for all changes that may become necessary at runtime [1]. For that reason highly available applications must be enabled for unanticipated changes at deploy-time, i.e., for unanticipated changes at already loaded program parts.

Recent work suggests different approaches for runtime program adaptation in Java. Approaches like *Javassist* [2, 3] or *AspectWerkz* [4–6] allow unanticipated changes until load-time, but not at deploy-time. Other approaches allow only for anticipated changes, e.g., object wrapping [7–11]. However, approaches such as PROSE [12, 13] and DUSC [14] allow unanticipated changes at deploy-time. Unfortunately, PROSE uses a modified Java virtual machine (JVM). For that reason the utilization of this approach is restricted to environments which support the PROSE virtual machine. DUSC lacks of object state keeping class updates when simultaneously updating the class interface. We conclude that non of these approaches enables stateful Java programs for unanticipated changes at deploy-time while running in a standard JVM.

In this paper we present the basic idea of an object role-based approach which enables stateful Java applications for unanticipated runtime adaptation even at deploy-time. It works with the Java HotSpot virtual machine[1] and combines object wrapping and Java HotSwap[2].

[1] The Java HotSpot virtual machine is the standard virtual machine of Sun's current Java 2 platforms.

[2] Java HotSwap is supported by the Java HotSpot virtual machine.

2 Motivating Example

Similar to static program changes, runtime program changes usually effect different parts of a program. Figure 1 depicts a simple program which manages and displays sorted lists. At the moment of program start it offers the *bubble sort* algorithm in order to sort a list. The length of the lists which have to be sorted grows while the program is running. At some point of execution time it is noticed that the bubble sort algorithm is to slow to sort the lists in reasonable time. For that reason the bubble sort algorithm has to be replaced by a faster sorting algorithm, e.g., the *quick sort* algorithm. In order to apply the required changes class *SortedList* as well as class *DisplayList* must be modified (Figure 1).

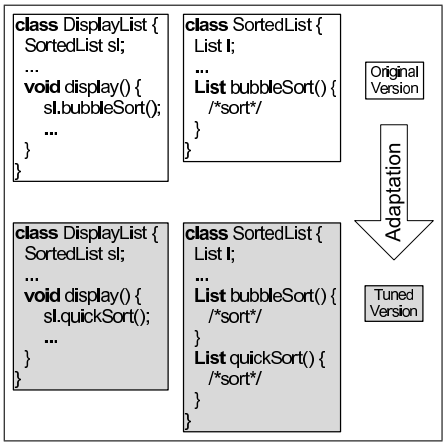


Fig. 1. Unanticipated adaptation.

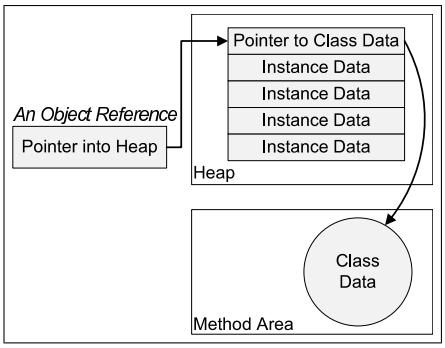


Fig. 2. Programs in the HotSpot VM [15].

3 Runtime Changes and the Java Virtual Machine

To understand the restrictions and possibilities for runtime program adaptation in the HotSpot VM it is necessary to know how a program is represented in the virtual machine. As depicted in Figure 2 the *heap* of the HotSpot VM stores the runtime data of all class instances, whereas the *method area* is the memory area of the HotSpot VM which stores all class (type) specific data.

The most adequate approach to alter a running program in Java is to replace a class in the JVM and update its objects according to the changes. However, this is difficult to realize in the HotSpot VM, since object references, object data, and class data are directly wired (see Figure 2). In order to replace a class in this virtual machine all instances of the class have to be destroyed and recreated.

Beside these restrictions the HotSpot VM enables method implementation swapping at runtime. This mechanism is called Java HotSwap and is provided

by the Java Virtual Machine Tool Interface [16]. Unfortunately, Java HotSwap does not allow to remove or add methods.

4 Runtime Changes and Object Roles

To systematically adapt a running program it is necessary to identify what objects have to be changed and what changes have to be applied to each object. We observed that the degree of changes depends on the role an object plays in the adaptation context, whether it acts as a *caller* or a *callee*. For instance in Figure 1 an object of class *DisplayList* acts as a caller (i.e., it uses functions of class *SortedList*), whereas an object of class *SortedList* acts as a callee.

4.1 Kinds of Callee Changes.

A callees job is to offer its functions to other objects (callers). To satisfy the requirements of its callers it may have to provide new or changed functions. For example callee *SortedList* must be extended by method *quickSort()* in order to speed up the display function of caller *DisplayList*. Due to the variety of callee changes we believe that, in order to offer new or changed functions, nearly each part of a callee has to be changeable.

4.2 Kinds of Caller Changes.

The reason for changing an object in its role as a caller is to call new, changed, or alternative functions provided by the callees it owns. These calls are largely implemented within the callers methods. For that reason changing a caller only requires modifications of the caller method implementation that contains the function call chosen for adaptation. For instance in our scenario method *display()* of class *DisplayList* has to be changed in order to call method *quickSort()* instead of method *bubbleSort()* of class *SortedList*.

5 Object Wrapping and Java HotSwap

In the following we present the basic idea of a runtime program adaptation approach which serves the required changes at objects playing role callee and objects playing role caller.

5.1 Callee Adaptation using Object Wrapping

An appropriate strategy for runtime callee adaptation is object wrapping. It means to embed the callee within another object denoted as *wrapper*. Within the wrapping the callee still provides its functions as usual, whereas the wrapper adds the necessary changes. Compared to the strategy of class replacement (as suggested in Section 3) object wrapping induces two major advantages. First,

the callee's class must not be unloaded, redefined and reloaded, i.e., the class instances must not be destroyed and recreated. Second, the callee keeps its state.

Remembering our motivating example from Section 2 a callee of type *SortedList* can be extended via a wrapping such as shown in Figure 3. Here wrapper *SortedListWrap* adds the required quick sort algorithm (method *quickSort()*), while it forwards calls to method *bubbleSort()* of callee *SortedList*. The handover of the callee reference happens in the constructor of the wrapper.

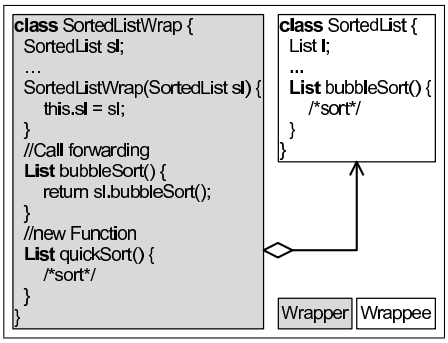


Fig. 3. Callee adaptation via wrapping.

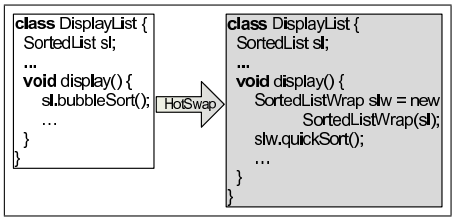


Fig. 4. Caller adaptation via Java HotSwap.

5.2 Caller Adaptation using Java HotSwap

While callee adaptation can be achieved using object wrapping, two open issues exist. First, the wrapping must be deployed. Second, the function calls of the caller have to be changed. Both tasks can be performed using Java HotSwap. Figure 4 illustrates the procedure according our motivating example. In order to apply the quick sort algorithm to *DisplayList* the implementation of method *display()* is swapped. The new method implementation wraps callee *SortedList* by wrapper *SortedListWrap* and calls the *quickSort()* method.

6 Conclusion and Future Work

In this paper we proposed unanticipated runtime program adaptation at deploy-time as an issue of changing objects. We suggested that the necessary degree of object changes depends on the role an object plays, i.e., whether it acts as caller or callee. Unfortunately, standard Java virtual machines, such as the Java HotSpot virtual machine, do not natively offer functions for all required object changes. For that reason we developed an approach which serves the whole bandwidth of required object changes. It combines object wrapping and Java HotSwap in order to enable unanticipated runtime adaptation at deploy-time.

Even though the basic approach presented in this paper is suitable for many use cases, a lot of open questions exist. In current work we look into how to

achieve consistency and how to apply persistent wrappings. We also evaluate the execution speed of programs which are modified using our runtime adaptation approach.

References

1. Pukall, M., Kuhlemann, M.: Characteristics of runtime program evolution. In Cazzola, W., Chiba, S., Coady, Y., Ducasse, S., Kniesel, G., Oriol, M., Saake, G., eds.: Proceedings of ECOOP'2007 Workshop on Reflection, AOP and Meta-Data for Software Evolution (RAM-SE'07), Berlin, Germany (2007) 51–57
2. Chiba, S., Nishizawa, M.: An easy-to-use toolkit for efficient java bytecode translators. In: Proceedings of the second International Conference on Generative Programming and Component Engineering (GPCE'03). (2003)
3. Chiba, S.: Load-time structural reflection in java. Lecture Notes in Computer Science (2000)
4. Vasseur, A.: Dynamic aop and runtime weaving for java – how does aspectwerkz address it? In: DAW: Dynamic Aspects Workshop. (2004)
5. Bonér, J.: Aspectwerkz – dynamic aop for java. Invited talk at 3rd International Conference on Aspect-Oriented Software Development (AOSD). (2004)
6. Bonér, J.: What are the key issues for commercial aop use: how does aspectwerkz address them? In: Proceedings of the 3rd International Conference on Aspect-Oriented Software Development (AOSD). (2004)
7. Hunt, J., Sitaraman, M.: Enhancements: Enabling flexible feature and implementation selection. In: Proceedings of the 8th International Conference on Software Reuse (ICSR'04). Lecture Notes in Computer Science, Springer (2004) 86–100
8. Kniesel, G.: Type-safe delegation for run-time component adaptation. In: Proceedings of the 13th European Conference on Object-Oriented Programming (ECOOP'99), London, UK, Springer-Verlag (1999) 351–366
9. Bettini, L., Capecchi, S., Venneri, B.: Extending java to dynamic object behaviors. In: Proceedings of the ETAPS'2003 Workshop on Object-Oriented Developments (WOOD'03). Volume 82 of ENTCS. (2003)
10. Büchi, M., Weck, W.: Generic wrappers. In Bertino, E., ed.: Proceedings of the 14th European Conference on Object-Oriented Programming (ECOOP'00). Volume 1850 of LNCS. (2000) 201–225
11. Bettini, L., Capecchi, S., Giachino, E.: Featherweight wrap java. In: Proceedings of the 2007 ACM symposium on Applied computing (SAC'07), New York, NY, USA, ACM (2007) 1094–1100
12. Nicoara, A., Alonso, G., Roscoe, T.: Controlled, systematic, and efficient code replacement for running java programs. In Sventek, J., Hand, S., eds.: Proceedings of the 2008 EuroSys Conference, ACM (2008) 233–246
13. Nicoara, A., Alonso, G.: Dynamic aop with prose. In Castro, J., Teniente, E., eds.: Proceedings of the CAiSE'2005 Workshop on Adaptive and Self-Managing Enterprise Applications (ASMEA'05), FEUP Edições, Porto (2005) 125–138
14. Orso, A., Rao, A., Harrold, M.: A technique for dynamic updating of java software. In: Proceedings of the International Conference on Software Maintenance (ICSM'02), Washington, DC, USA, IEEE Computer Society (2002) 649–658
15. Venners, B.: Inside the Java 2 Virtual Machine. Computing McGraw-Hill. (2000)
16. Sun: Java virtual machine tool interface version 1.1. Technical report, Sun Microsystems (2006) http://java.sun.com/javase/6/docs/platform/jvmti/jvmti.html.