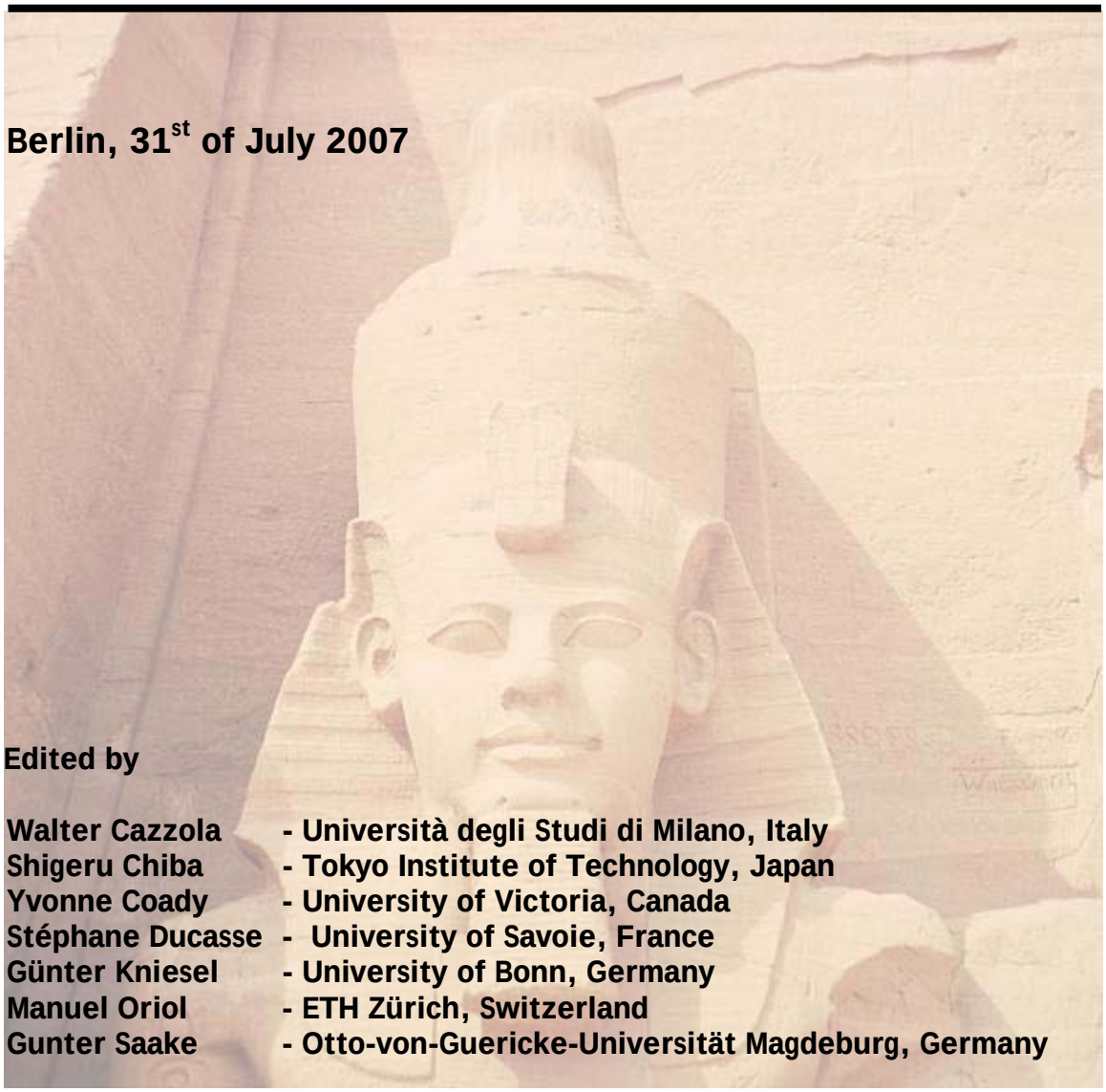
# RAM-SE'07 – ECOOP'07 Workshop on Reflection, AOP, and Meta-Data for Software Evolution

(Proceedings)



**Berlin, 31st of July 2007**


**Edited by**

**Walter Cazzola** - Università degli Studi di Milano, Italy
**Shigeru Chiba** - Tokyo Institute of Technology, Japan
**Yvonne Coady** - University of Victoria, Canada
**Stéphane Ducasse** - University of Savoie, France
**Günter Kniesel** - University of Bonn, Germany
**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 forth edition of the workshop, we had an interesting keynote by Shigeru Chiba that has investigated the real benefits provided by aspects to evolve the software and which properties are desirable to support software evolution. This keynote was particularly provocative and raised several issues and lively discussion among the workshop attendees. To the interested reader an extended abstract can be found in the first part of these proceedings.

This volume gathers together all the position papers accepted for presentation at the forth edition of the Workshop on Reflection, AOP and Meta-Data for Software Evolution (RAM-SE'07), held in Berlin on the 31st of July, during the ECOOP'07 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.

| | | |
|---|---|---|
| Blair, Gordon | Masuhara, Hidehiko | Südholt, Mario |
| Cazzola, Walter | Mens, Kim | Sørensen, Fredrik |
| Chiba, Shigeru | Mosser, Sebastian | Saake, Gunter |
| Greenwood, Phil | Oriol, Manuel | Smaragdakis, Yannis |
| Hoffman, Kevin | Piccioni, Marco | Tanter, Éric |
| Huang, Shan Shan | Pini, Sonia | Tao, Yonglei |
| Irmert, Florian | Pukall, Mario | Ueyama, Jó |
| Kienle, Holger | Rashid, Awais | Yonezawa Akinori |

A special thank is for Mario Südholt that has chaired the invited speaker and fed the lively discussion that followed the talk and the whole bunch of people that participated to and animated the panel at the end of the day.

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.

November 2007

W. Cazzola, S. Chiba, Y. Coady, S. Ducasse,
G. Kniesel, M. Oriol and G. Saake
RAM-SE'07 Organizers

# Contents

## Keynote on How We Should Use Aspects



## Classic Software Evolution



## Aspect-Oriented and Reflection for Software Evolution

## Aspects and Evolution: How to Use Aspects

Keynote speaker:
Shigeru Chiba, Tokyo Institute of Technology, Japan

Chairman: Mario Südholt, École des Mines de Nantes, France

# How we should use aspects

Shigeru Chiba
Tokyo Institute of Technology

**Abstract**

Besides classic logging and the observer pattern, several applications of aspect-oriented programming (AOP) have been proposed so far. This talk reviewed those applications and discussed what properties of AOP are significant and promising for software evolution. It also discussed what are unique features of AOP against related technology such as reflection and mixin layers.

## 1 Logging

Logging is a classic application of AOP (Aspect-Oriented Programming). It is a simple but real application that we can see in several commercial products. I heard that logging by AOP is used in WebSphere and MySQL. However, this application example might have been giving some developers an impression that AOP is a program transformation technique. For example, at the AOAsia 2007 workshop, a few researchers presented techniques for detecting some problems in a given program, such as memory leaks, by analyzing a trace log, which is generated by an AspectJ aspect [2]. Since this use of AspectJ is under the hood, the users of such a detecting tool do not see or write an aspect at all; AspectJ is totally a program transformation toolkit for inserting a logging code in the program. It is used only by the tool implementer just because it provides an easy-to-use programming front-end or a simple language for describing program transformation.

Although AspectJ accidentally can be used as a program translator, it is not the tool designed primarily for program transformation. Their capability is limited and other tools such as Javassist [3] allow more complex transformation. Furthermore, AOP is never a technique for program transformation. As we know well, it is a programming paradigm for *human* developers, who want to describe a crosscutting concern. We should be careful when we discuss applications in which AOP is used only for an implementation-level concern and it is never seen or written by the end users. In the case of the above detection tool of memory leaks, we should understand that an AOP language simplified the implementation of logging because the tool has only to generate an extra code and it does not have to modify an existing code. Without AOP, the tool implementation would be more complicated but this fact is irrelevant to end-users'

concerns.

## 2 Code reuse/share

Another well-known application of AOP is code reusing and sharing. AOP languages allow us to merge similar code fragments found in several different classes and methods. These fragments can be replaced with a single advice body if the join points of the fragments are enumerated in a pointcut definition. We do not have to repeatedly describe similar code by hand at every join point.

To merge similar code fragments, an advice body must be described as generically as possible. We should notice that generic description has been enabled by language mechanisms other than AOP's ones. In AspectJ, genericness is achieved by reflection such as thisJoinPoint. Although the proceed operator available in an around advice is a special form for AOP, it can be regarded as syntactical support for reflection. Although AOP is a good mechanism for specifying where a generically described code fragment (*i.e.,* an advice body) is instantiated, it is never a mechanism for enabling generic description itself. In fact, some researchers have proposed to introduce an advice body parametrized by meta variables [7]. For better genericness, we have to combine such parametrization and reflection with an AOP mechanism like pointcut-advice. Thus, exploring code reusing/sharing by AOP too much might mislead us into a study of non-AOP mechanisms.

## 3 Composability

Implementing a functional concern as a pluggable module is also an application of AOP. For example, the observer pattern described in AspectJ [5] is such a concern. Santos *et al.* reported that an aspect is useful for implementing a new functional feature of the JHotDraw drawing editor [9]. Because of its (syntactic-level) obliviousness property [4], a module written in an AOP language can be connected and disconnected to other modules without modifying the source code.

This type of applications should be more explored because it is strongly related to the essence of AOP. An initial challenge of AOP was to enable software to be composed of several modules independent of each other. Here, the independence means that the code of each module is clearly separated from the others and thus the module can be connected and disconnected to/from others without source-code changes. Since some modules are inherently crosscutting, enabling this independence is not straightforward. As Kiczales suggested in his keynote talk at AOAsia 2006 [6], it is a wrong assumption that a program can be divided into *statically* localized units with a well-defined *static* interface. For software evolution, or even for maintenance, we often have to change (or refactor) the "interfaces" of modules. We must sometime add new extension points to the interfaces or make callback methods available. We may have to expose

some internal data structures that have been hidden behind the interfaces so that those data structures can be used for debugging, performance tuning, or implementing a new functional feature. These changes of the interfaces involve source-code modification.

AOP languages should support this changeability of interfaces in a flexible but controlled manner. This is still an open problem today. Although one idea is statically declaring possible changes as open modules [1, 8] enables, it damages the flexibility; at design time, we must expect all changes needed for future evolution. On the other hand, keeping the maximum flexibility will break interfaces, which are inherently static contracts. An interface that we can change at any time at any degree is never the same concept that we know today as an interface.

# 4 Summary

A number of applications of AOP have been proposed so far. In this talk, I picked three kinds of applications: logging, code reusing/sharing, and composability. Although they are interesting applications, logging might have been giving a wrong impression, which is that AOP is program transformation. Code reusing/sharing might have been confusing developers; genericness is not an inherent essence of AOP. I pointed out that exploring composability applications could help us reveal an essence of AOP. AOP languages should enable a module "interface" to be altered without source-code changes for future software evolution. I mentioned that this ability of AOP is essential at least from the viewpoint of software evolution.

# References


[1] Aldrich, J., "Open Modules: Modular Reasoning About Advice," in *ECOOP 2005*, LNCS 3586, pp. 144–168, Springer-Verlag, 2005.

[2] Chen, K. and J. Chen, "Aspect-Based Instrumentation for Locating Memory Leaks in Java Programs," the AOAsia 3 workshop, Beijing, July 23, 2007.

[3] Chiba, S., "Load-time structural reflection in Java," in *ECOOP 2000*, LNCS 1850, pp. 313–336, Springer-Verlag, 2000.

[4] Filman, R. E. and D. P. Friedman, "Aspect-Oriented Programming is Quantification and Obliviousness," *Aspect-Oriented Software Development*, Addison-Wesley, pp. 21–35, 2005.

[5] Kiczales, G., E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold, "An Overview of AspectJ," in *ECOOP 2001 – Object-Oriented Programming*, LNCS 2072, pp. 327–353, Springer, 2001.

[6] Kiczales, G., "A Call to Arms: What should Modularity Mean?," keynote speach, AOAsia 2006 workshop, Tokyo, September19, 2006.

[7] Kniesel, G., T. Rho, and S. Hanenberg, "Evolvable Pattern Implementations Need Generic Aspects," ECOOP 2004 Workshop on Reflection, AOP and Meta-Data for Software Evolution, Oslo, 2004.

[8] Ongkingco, N., P. Avgustinov, J. Tibble, L. Hendren, O. de Moor, and G. Sittampalam, "Adding open modules to AspectJ," in *Int'l Conf. on Aspect Oriented Software Development (AOSD'06)*, pp. 39–50, ACM Press, 2006.

[9] Santos, A. L., A. Lopes, and K. Koskimies, "Framework specialization aspects," in *AOSD '07: Proceedings of the 6th international conference on Aspect-oriented software development*, pp. 14–24, ACM Press, 2007.

## Classic Software Evolution

Chairman: Walter Cazzola, Università di Milano, Italy

# Toward Computer-Aided Usability Evaluation for Evolving Interactive Software

Yonglei Tao

School of Computing and Information Systems
Grand Valley State University
Allendale, Michigan, US

**Abstract.** Recurrent redesign on an application's user interface is driven by changing requirements, user profiles and experiences, as well as technologies. User interface evolution also has impact to the application itself, which imposes a great challenge on providing tool support to ensure a smooth transformation in this process. We in this paper explore the suitability of using an aspect-oriented approach to computer-aided usability evaluation. Using aspects, a support tool is not only flexible for collecting data to address diverse usability considerations in the evolutionary process but also adaptable to continuous changes in the application. We also discuss our future research on other relevant issues about such a tool.

**Keywords:** Interactive Software Design, Usability Evaluation, Tool Support, Aspect-Oriented Programming.

## 1 Introduction

Interactive software evolves along one or more dimensions during its lifetime, including functionality, architecture, code, and user interface. Changes in one dimension often affect, interact, and impact others [1]. As such, the evolution of an interactive application imposes a great challenge not only on developing the application itself but also on providing tool support to ensure a smooth transformation in this process [2].

Usability is a key quality attribute for the success of interactive applications. A practical solution to building a usable product is early and ongoing usability evaluation [3]. In usability evaluation, users use an application to complete a pre-determined set of tasks. Information on user behavior with respect to the user interface is captured and analyzed to determine how well the user interface supports users' task completion. Since evaluation activities such as data collection and analysis are very time-consuming, tool support is indispensable [5, 6].

Recurrent redesign on an application's user interface is driven by changing requirements, user profiles and experiences, as well as technologies. Usability considerations vary in the process of user interface evolution [4]. As a new feature is introduced, for example, the attention is focused on the flow of the basic user-system

interaction, including the coordination of data exchange between the user and the system as well as the navigational structure. While a new interaction style is made available to support a particular user group for effective use of the application, however, usability considerations largely reflect on physical, spatial, and visual characteristics of screen elements. A support tool must be flexible for collecting data at a level of abstraction that is appropriate to address specific needs in different stages of the evolutionary process.

As the user interface evolves, new windows may be introduced; existing windows may be combined, split, or removed; and screen elements may be added, removed, or replaced. Changes in the user interface inevitably affect various components in the application [7]. In Java, for examples, even re-layout of screen elements requires to alter a few lines of code. Hence, a support tool must also be adaptable to continuous changes in the application.

In this paper, we explore the suitability of using an aspect-oriented approach to computer-aided usability evaluation. We describe how to use aspects to capture user interface events that occur when the user interacts with an application's user interface. Using aspects for data collection paves the way for analyzing the acquired data and identifying potential usability problems. We also discuss our future research on other relevant issues about such a tool.

## 2 Related Work

AOP (Aspect-Oriented Programming) is known as an effective way of modularizing crosscutting concerns such as monitoring, tracing, and logging [8, 10]. As far as we know, using an aspect-oriented approach to computer-aided usability evaluation, however, does not seem to have received as much attention as it should have been, in part due to the gap between the communities of SE (Software Engineering) and HCI (Human-Computer Interaction).

Proposals about automatic techniques for capturing user interface events can be found in the literature [5]. Some of those techniques capture events at the keystroke or system level regardless of the usability issues under consideration. Recording data at that level produces voluminous log files and makes it difficult to map recorded usage into high-level tasks [6]. Usability-related information can also be obtained by instrumenting the target program or its platform. Because such information does not appear at one particular place, instrumentation in a traditional way tends to be distributed throughout the target code [9]. Obviously, techniques as such are inappropriate when changes in an application occur quite often. Adaptive techniques, such as AOP, are more promising for our purposes [10, 11].

Java-style interfaces enhance, facilitate, and even make possible the flexibility, modifiability, and extensibility that are highly desirable in object-oriented design [12]. Interfaces can also improve the quality of aspect-oriented design [13]. We use interfaces to expose crosscutting behavior against which aspects are defined. Using

interfaces and aspects jointly provides the benefit of adaptability for automatic support for usability evaluation.

## 3 The MVC Architecture for Interactive Applications

The Model-View-Controller architecture (MVC) was originally designed for applications that provide multiple views for the same data [15]. It has gradually become the central feature of modern interactive applications. Based on the object-oriented principles, MVC describes an application in terms of three fundamental abstractions: models, views, and controllers. Roughly, the model manages application data, the view is responsible for visual presentation, and the controller handles input events for views. By encapsulating the three abstractions into separate components, MVC minimizes the impact of user interface changes and increases the reusability of domain objects.

User interface events are generated as natural products of the normal operation of an interactive application, including input events (such as the user clicking on a command button) and output events (such as the application bringing up a message box). Sequences of events result from steps taken by the user in completing tasks. In MVC, the view and controller take appropriate actions when they are notified of corresponding events. Separating the three abstractions also exposes user interface events within the application.

We in this paper use a GUI (Graphical User Interface) application AccountManager, adopted from [14] with modifications, as an example. Briefly, the application is a Java program intended to manage several bank accounts for a customer. A text view and a bar graph view are provided for each account to display the account information. A pie chart view is provided to display the customer's total assets held in all the accounts. Also a text field and two buttons are provided for each account, where the former allows the user to enter an amount and the latter to withdraw and deposit the input amount, respectively.

### 3.1 Model and View

Figure 1 is a UML class diagram that illustrates key classes in the model and view of the application's MVC architecture. As shown in Figure 1, class Account is a model and classes PieChartView, TextView, and BarGraphView represent three views for an Account.

When an account changes state, all of its views are notified and updated to reflect the change. A well-known design pattern, Observer, describes an effective way to establish such a one-to-many dependency [16]. We use Java's Observable class and Observer interface to implement the Observer pattern. As shown in Figure 1, class Account extends class Observable and the three view classes (i.e., PieChartView, TextView, and BarGraphView) implement interface Observer.

Note that each view defines its own update() method to refresh its display and the account notifies its views by invoking their update() methods. As a result, a call to the update() method for an object of any class in the Observer-based class hierarchy indicates the occurrence of an output event.

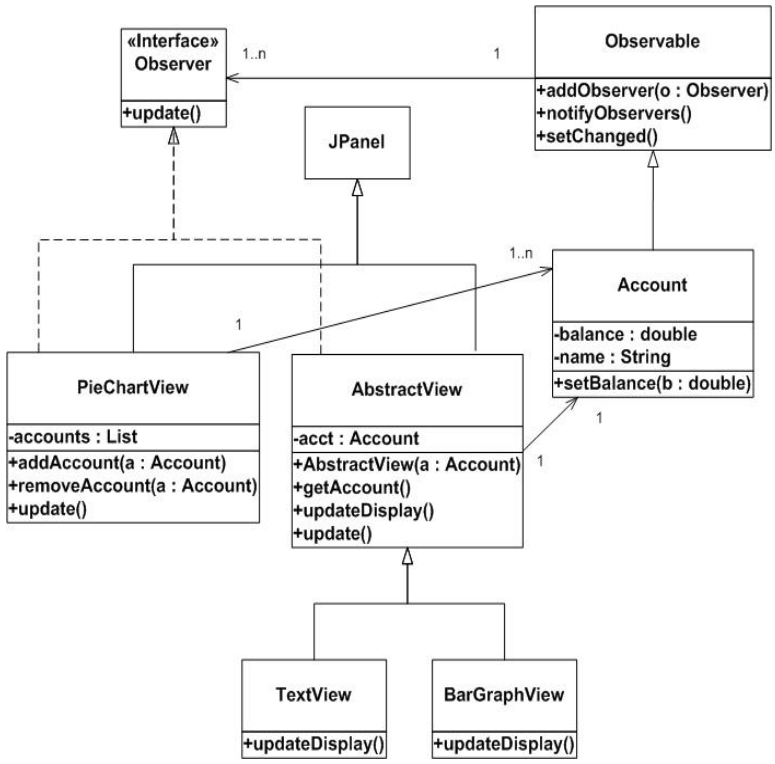



**Fig. 1.** Model and View

### 3.2 Model and Controller

Due to limited space, we omit the UML class diagram for the model and controller. In Java, a listener class is responsible for handling an input event. Application AccountManager defines two listener classes, one for executing a transaction for an account when a button is clicked and the other for validating the user input entered in a text field. Both listener classes handle the same type of input events, that is, action events in Java. Action events originate from the user's actions with respect to screen elements such as buttons, menu items, and text fields. Java provides an ActionListener interface with method actionPerformed() for handling action events. Hence, the two listener classes must implement the ActionListener interface.

Note that each listener class has to define its own actionPerformed() method for handling an action event. Application AccountManager notifies a listener of the user's action by invoking its actionPerformed() method. As a result, a call to the actionPerformed() method for an object of any class that implements the ActionListener interface indicates the occurrence of an input event in this application.

## 4 Data Collection with Aspects

AspectJ is an extension to the Java programming language [10]. It provides constructs to modularize crosscutting concerns that would otherwise result in code scattered over multiple modules. We use the aspect construct to capture user interface events.

Java-style interfaces are essential to adaptability. An interface is a collection of method signatures. It defines a standard protocol to interact with an object without knowing or caring about what class that object belongs to. In application AccountManager, interface Observer specifies a standard way for a model to notify its views and interface ActionListener for the application to notify an event listener. We expose crosscutting concerns of interest through interfaces against which aspects are defined. Using interfaces allows us to specify crosscutting behavior without being committed to a particular class hierarchy. As a result, the dependency of the aspects code on specific features of the user interface is loosened.

### 4.1 Capturing Output Events

In application AccountManager, each view is updated when being notified of state change in its model. Such a notification is made through a call to the update() method for each view. Aspect UpdateView, as declared below, is intended to capture output events that occur when views are notified.

```
import java.awt.*;
import java.util.*;

public aspect UpdateView {
    // Pointcut Declaration
    pointcut traceUpdate (Object obj)
          : cflow (execution (
          void Observer+.update (
                      Observable, Object)))
          && args (Observable, obj);
```

```
    // Advice Definition
    after(Object obj): traceUpdate(obj) {
          System.out.println (obj +
                      " view updated");
    }
}
```

Aspect UpdateView defines a pointcut traceUpdate() to capture joint points that make a call to the update() method for a view. It also defines a piece of advice to identify the account whose state change causes the output event. Here, pointcupt traceUpdate() takes an event argument from the advised joint point and passes it to the advice, which gives the advice the information it needs.

In the declaration of pointcut traceUpdate(), Observer+ means any class that implements the Observer interface, including both the current and potential ones. As such, the introduction of a new view or removal of an existing view has little impact to aspect UpdateView.

### 4.2 Capturing Input Events

When the user clicks a button or enters data in a text field, a listener is notified of the action event. We define an aspect to capture action events. Basically, this aspect contains a pointcut to capture joint points that make a call to method actionPerformed() for a listener. It also contains a piece of advice that receives an event object from the advised joint point and uses it to identify the event source.

We can use such an aspect to capture action events without having to worry about from which screen elements they originate or by which handlers they are handled. Changes in the user interface with respect to the originating screen elements of the action event, such as adding or removing a button, will affect the related handler classes. But they won't have much impact to that aspect and it will continue to function as it is specified.

Note that other types of input events also occur when the user interacts with the application's user interface, such as mouse and key events. Java provides a listener interface for each type of input event. Similarly, handler classes and aspects can be specifically defined to capture other types of input events. Such an addition does not affect the existing ones in any way. While action events contribute to usability information at the application level, mouse and key events contribute primarily to usability information at a lower level of abstraction. Use of those aspects selectively would allow us to address different usability considerations.

When application AccountManager runs with the aspects described above, a list of input and output events will display on the screen, showing which button is clicked, which view is updated, and so forth. Such a list of events provides the basis for the follow-up activities in usability evaluation.

## 5 Summary and Future Research

As demonstrated by the above example, the aspect-oriented approach is suitable for building a support tool for usability evaluation. Using aspects not only provides the advantage of flexibility but also offers the benefit of adaptability. In addition, using aspects makes possible not only to collect usability-related information from the captured events but also to obtain relevant information available elsewhere in the application, which is helpful for a meaningful interpretation of certain data. Compared with some of the existing techniques that require an additional step to extract appropriate information from the raw data, the aspect-oriented approach is more effective.

It is worth noting that although we use Java in the example application, our approach, which is based on the notions of MVC, interfaces/abstract classes, and aspects, is language independent.

In addition to data collection, it is equally important to provide tool support for data analysis. Analyzing the acquired data manually would be difficult and tedious without tool support. In addition to data collection, relevant issues as listed below require further research:

(1) Identifying tasks and sequences of tasks that the user is intended to accomplish from the acquired data. User interface design is centered on tasks (or use cases) [3]. As a consequence, tasks are a natural unit of data for analysis purposes. Well-defined tasks in the requirements specification provide a basis for identifying tasks. Here, it is important to separate application-specific knowledge from general processing logic for the sake of adaptability.

(2) Analyzing data obtained from multiple users to measure usability attributes of a user interface and to identify potential issues affecting them. Examples of quantitative measures include time to complete a task, task frequencies, range of functions used, and number of errors or repeated errors. More importantly, analyzing the acquired data enables us to find indicators for potential usability problems, for example, areas in which mistakes were made, unnecessary or undesirable steps were taken, and extra assistances (such as undo and on-line help) were requested. Failure for a (group of) user to follow a navigational path as expected may indicate the lack of adequate visual clues for what the user needs to know. Often, whether or not visual guidance is adequate depends on the user who uses the application. Here, a challenge is to find out to which user group it is adequate and to which one it is not. We will investigate use of data mining techniques in this regard.

In addition, AspectJ is classified as a static AOP system. Static AOP systems allow weaving in aspects at compile or load-time. On the other hand, the dynamic AOP systems allow weaving aspects in at run-time. As a result, programmers can dynamically plug and unplug an aspect in/from running software [17]. Obviously, a dynamic AOP system seems to be more appropriate for our purposes. It is also an interesting issue that deserves future attention.

## References


[1] Lehman, M., Ramil, J.: Evolution in Software and Related Areas. Proceedings of the International Workshop on Principles of Software Evolution (IWPSE 2001), Vienna, Austria.

[2] Gall, H., Lanza, M.: Software Evolution: Analysis and Visualization. Proceedings of the International Conference on Software Engineering (ICSE 2006), Shanghai, China.

[3] Torres, R.: Practitioner's Handbook for User Interface Design and Development. Prentice-Hall (2002).

[4] Ferre, X., et al.: Usability Basics for Software Developers. IEEE Software, January/February (2001) 22-29.

[5] Ivory, M., Hearst, M.: The State of the Art in Automating Usability Evaluation of User Interfaces. Computing Survey, Vol. 33, No. 4, ACM (2001) 470-516.

[6] Hilbert, D., Redmiles, D.: Extracting Usability Information from User Interface Events. Computing Surveys, Vol. 32, No. 4, ACM (2000) 384-421.

[7] Costabile, M., et al.: Supporting Interaction and Co-evolution of Users and Systems. Proceedings of the International Conference on Advanced Visual Interfaces (AVI 2006), Venezia, Italy.

[8] Low, T.: Designing, Modeling and Implementing a Toolkit for Aspect-oriented Tracing (TAST). Proceedings of the Workshop on Aspect-Oriented Modeling with UML (AOSD 2002), Univ. of Twente, The Netherlands.

[9] Deters, M., Cytron, R.: Introduction to Program Instrumentation using Aspects. Proceedings of the ACM OOPSLA Workshop on Advanced Separation of Concerns in Object-Oriented Systems (OOPSLA 2001), Tampa Bay, Florida, USA.

[10] Laddad, R.: AspectJ in Action: Practical Aspect-Oriented Programming. Manning Publications Co. (2003).

[11] Tao, Y.: Capturing User Interface Events with Aspects. Proceedings of the 12th International Conference on Human-Computer Interaction, LNCS 4553, Springer-Verlag Berlin Heidelberg (2007) 1171-1180.

[12] Coad, P., Mayfield, M.: Java Design: Building Better Apps & Applets. Prentice Hall (1999) 223-288.

[13] Griswold, W., et al.: Modular Software Design with Crosscutting Interfaces. IEEE Software, January/February, IEEE (2006) 51 – 60.

[14] Deitel, H., et al.: Advanced Java 2 Platform: How to Program. Prentice-Hall (2002) 85-134.

[15] Krasner, G., Pope, S.: A Cookbook for Using the Model-View-Controller User Interface Paradigm in Smalltalk-80. JOOP (1988), Aug. / Sept.

[16] Gamma, E., et al.: Design Patterns: elements of Reusable Software Architecture. Addison-Wesley (1995).

[17] Sato, Y., et. al.: A Selective, Just-In-Time Aspect Weaver. Proceedings of the 2nd International Conference on Generative Programming and Component Engineering (GPCE), LNCS 2830, Springer-Verlag (2003).

# Towards Runtime Adaptation in a SOA Environment

Florian Irmert, Marcus Meyerhöfer, Markus Weiten

Friedrich-Alexander University of Erlangen and Nuremberg
{Florian.Irmert,Marcus.Meyerhoefer}@cs.fau.de,markus@weiten.de

**Abstract.** Service Oriented Architecture (SOA) promotes the utilization of available services to develop completely new applications in a context which has not been foreseen as these services were implemented. Unfortunately the interfaces respectively the behaviour of a service often do not fit exactly to the new domain. Slight changes would be necessary to reuse them in the new environment. This paper presents an approach to integrate dynamic AOP into a SOA platform to adapt existing services at runtime to new requirements. Services can then be reused without the need of stopping and redeployment.

## 1 Introduction

Service Oriented Architecture (SOA) has recently gained widespread attraction because of its promise to allow for easier and more flexible adaptations of the software infrastructure of a company to fast changing business requirements. The core idea of SOA is to decouple functional units and expose them as independent services to other programs and thereby foster reuse. Moreover, this allows to create new applications or services by merely combining already available ones in new ways.

However, in a business environment there are of course some services which have been build specifically for concrete applications (e.g. client management). Such services are usually designed without reuse in mind but are determinated by the requirements of the service users; often service user and service provider design the services in a collaborative process. In order to be usable by new services or applications, those company internal services would have to be adapted. In modern environments with high demands on application availability this leads to the necessity to modify running services. It is not viable to take a service offline and deploy an updated version as possibly many other services depend on such a service. Furthermore, it might also happen that different services have different or even conflicting requirements on a service they use. In such cases specific modifications should be applied for each consumer individually.

In this paper we present an approach to adapt services at runtime. We examine this problem from a technical point of view[1]. Runtime adaptation [1,2,3]

[1] Semantical issues—e.g. which modifications should be disallowed in order to ensure correct application behaviour—are beyond the scope of this paper and area of future work.

can be achieved with dynamic Aspect Oriented Programming (d-AOP) [4] and therefore we have integrated a d-AOP framework into a praxis proven SOA environment, the OSGi Service Platform.

## 2 Technical Background

In the next Section we give a short introduction to the OSGi Service Platform and d-AOP.

**OSGi Service Platform** The OSGI Service Platform is described as a "Java based application server for networked devices, however small or large they are" [5]. Originally designed for embedded devices and home service gateways, it has become prominent for building SOA applications. Several applications (deployed in a special bundle format) can coexist inside the OSGi Service Platform, while each is loaded by a different class loader. Consequently only one Java Virtual Machine is needed. Also lifecycle management is supported for all hosted applications; there is an API to install, start, stop and de-install the application bundles without restarting the framework. This "Hot-Deployment" feature is very interesting in the context of SOA, because adding new services to the platform does not affect running services. Bundles can provide their functionality as a service by publishing their interfaces in the OSGi Service Registry. Other bundles employ this registry to discover and bind services (fig. 1). While there are a number of implementations of the OSGi Specification [6,7] we decided to use Equinox [8], published by the Eclipse Foundation, because it is a well proven implementation which offers all necessary features for our approach (Section 4).

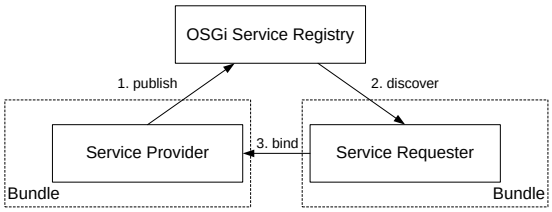



**Fig. 1.** OSGi Service Registry

**Dynamic AOP** The term "dynamic aspect-oriented programming" is most commonly used if aspects can be deployed and activated at runtime. Dynamic AOP can be realized e.g. with a modified JVM [9] or bytecode modification [10]. We decided to use JBoss AOP in our approach, because its successful application is exemplified by its usage in the JBoss Application Server. JBoss AOP inserts

hooks at potential joinpoints. Each time such a hook is called in the program flow, a central Aspect Manager is called (fig. 2). This instance manages the aspects and decides whether to apply them depending on the joinpoint. Obviously new aspects can be added to the Aspect Manager at runtime.

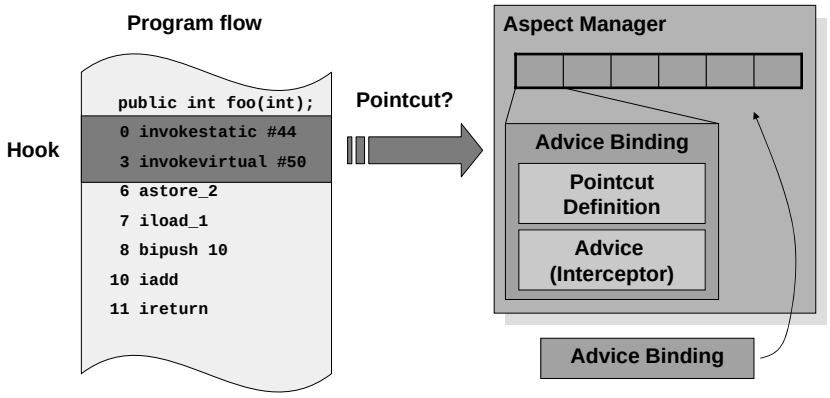



**Fig. 2.** Central Aspect Manager

## 3 Problem Domain

Figure 3 illustrates a typical scenario where available services are used to build new services respectively applications. The new service **E** utilizes available services **A** and **C**. Often services cannot be used exactly as they are. A common approach is to implement wrappers to adapt their behaviour. As a simple example service **E** in figure 4 uses service **A** but has to transform the result from miles to kilometers. A wrapper class performs the necessary transformation. If there are many services which want to use service **A**, but also require kilometers instead of miles, it would be desirable to modify **A**. In the example of figure 3 this would be no problem, because service **A** is used only by service **E**. Service **A** could therefore be taken offline and replaced by a modified version without affecting other services. Unlike service **A**, the adaptation of service **C** would be much more difficult, because service **D** depends on service **C**, too. If service **C** is stopped to apply code changes for adaptation, service **D** would be perturbed as well. A runtime mechanism is needed to enable modifications of the behaviour of a service "online". Runtime enhancement of services allows to adopt existing services to unforeseen requirements. Therefore it facilitates modifications of services just-in-time and without interference of other services which depend on services under modification.

Previtali [11] exploits aspects for dynamic updates and employs AOP's features like method or field interception. With d-AOP these modifications can be

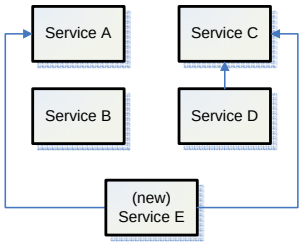



**Fig. 3.** SOA example

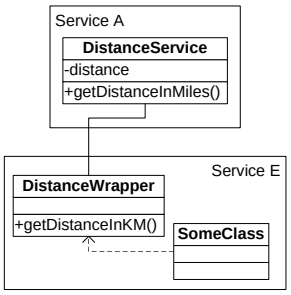



**Fig. 4.** Usage of a wrapper class

applied at runtime. The integration of d-AOP into a SOA framework constitutes the technical basis for adaptations of running services to new system conditions as well as changing business requirements. The main challenge addressed in this paper is to integrate d-AOP seamlessly to ease the development process of service oriented applications.

## 4 Integration of d-AOP into the Equinox OSGi Framework

In our approach we combine the OSGi framework with dynamic aspect-oriented programming to realize dynamic adaptation at runtime. We focus on existing, established technologies represented by OSGi and dynamic aspect-oriented programming frameworks. In this chapter we present common technical details of the aspect-oriented framework we use, as well as we managed to integrate it into OSGi and how we solved the problems that arose from this approach.

### 4.1 Aspects as OSGi Bundles

To provide an integrated environment it is necessary that aspects are deployable as OSGi bundles at runtime. An OSGi bundle consists of Java classes and a bundle manifest containing meta information. Obviously aspects written in Java can be packaged in an OSGi bundle. The integration is realized by mapping the aspect deployment to OSGi bundle lifecycle operations.

The OSGi core specification defines a bundle activator class for each bundle which is invoked when the bundle is installed and started. The bundle activator class must implement a specific interface defining two callback methods, one that is called when the bundle is started and one that is called when the bundle is stopped. Normally these methods are used to register a bundle itself as listener, start necessary threads and to clean up after execution. We utilize these two methods for deploying and undeploying the aspects by calling the corresponding methods of the Aspect Manager. Deploying/undeploying aspects is mapped

to installing/starting and stopping/uninstalling the corresponding bundle that encapsulates the aspect.

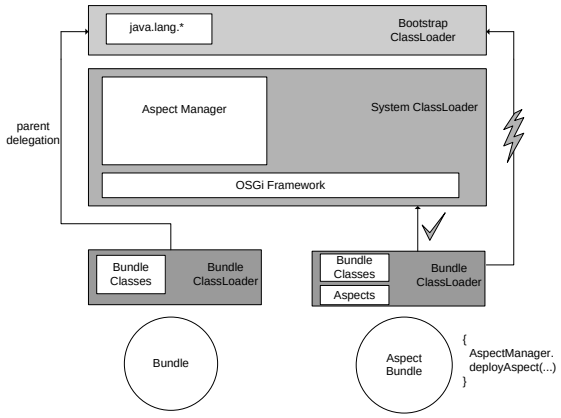



**Fig. 5.** By default, the parent class loader for an OSGi bundle is the bootstrap class loader containing all java.lang.* classes. This is sufficient for all ordinary OSGi bundles (left side). But an aspect bundle must have access to the methods of the global Aspect Manager. The delegation to the bootstrap class loader would prohibit that access, because the Aspect Manager is loaded and defined by the system class loader. That is why parent delegation has to be reconfigured to the system class loader as parent for the OSGi bundle class loader (right side).

### 4.2 Class Loading Issues

OSGi makes use of different class loaders to realize the complete separation of bundle classpaths and namespaces. Thereby the way in which bundles can access classes of other bundles can be controlled. By using different class loaders, bundles can encapsulate classes with identical names and naming conflicts are avoided. In Java a class is not identified solely by its name, but by the combination of its name and its defining class loader. Two classes with the same name can exist in one virtual machine as long as they are defined by different class loaders.

With this concept, in the OSGi framework different bundles can coexist, but it complicates the integration of existing aspect frameworks. Following the Java virtual machine specification [12] a symbolic reference that has not been resolved is loaded with the same class loader as the defining class. A `ClassNotFoundException` would be thrown if the bundle activator would try to instantiate the Aspect Manager (which is loaded by the system class loader), because the bundle activator is always defined by the bundle class loader. This problem is

visualized in figure 5. Including the aspect framework in the bundle classpath does not solve this problem. The Aspect Manager would be loaded and defined by the bundle class loader resulting in an own instance for every aspect bundle, instead of a global one for all bundles.

The Java platform uses a delegation model for loading and resolving classes: Every class loader has a "parent" class loader. Everytime a class is going to be loaded, the class loader initially "delegates" the search for the class to its parent class loader before attempting to find the class itself. Constructors in `java.lang.ClassLoader` and its subclasses allow to specify a parent class loader when a new class loader is instantiated. If a parent class loader is not explicitly specified, the virtual machine's system class loader will be assigned as the default parent.

### 4.3 Delegation

The OSGI specification defines a pre-defined order searching for classes, e.g. if a bundle imports packages from another bundle, the class loader of the exporting bundle is included in the searching and loading process. The complete search order can be seen in figure 6. As described before, the class loading is firstly delegated to a parent class loader. This parent class loader for a bundle is normally the bootstrap class loader, which is responsible for loading the classes for the Java virtual machine and its extensions. In order to enable aspect bundles to access the Aspect Manager, the parent class loader of a bundle has to be the system class loader, because the system class loader is responsible to load and define the Aspect Manager.

The Equinox OSGi framework allows to define the parent class loader for bundles to be defined via a configuration parameter. With the system class loader defined as parent class loader the deployment of aspects as bundles works seamlessly. This solution enables the integration of every "hook-based" d-AOP framework (not only JBoss AOP).

## 5 Related Work

The next section presents other approaches which combine the benefits of OSGi and aspect-oriented programming.

### 5.1 AJEER

Martin Lippert presented his "AspectJ Enabled Eclipse Runtime" (AJEER) [14] a few years ago. AJEER was originally designed to integrate AspectJ [15] into the Eclipse framework. As AJEER was implemented, Eclipse did not make use of the OSGi framework. The weaving part of the AspectJ 1.2 compiler implementation was added to the Eclipse runtime. Although Lippert discusses the options of AJEER being extended to support runtime weaving, it is currently limited to load-time weaving. Furthermore he presents the idea of "runtime-like" weaving:

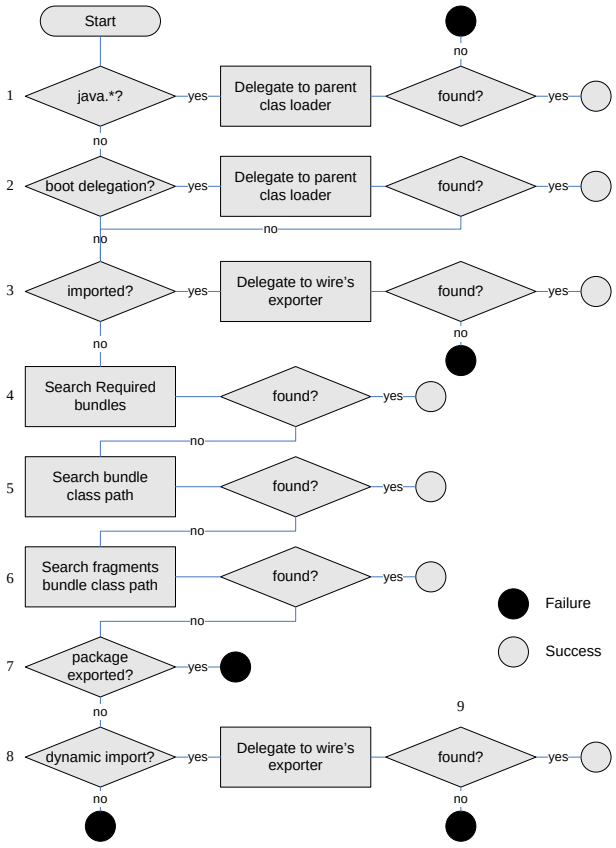



**Fig. 6.** Flowchart for class or resource search order. When a bundles class loader is requested to load a class or to find a resource, the search must be performed in the order described by the flowchart above [13].

Aspects can be added to bundles at runtime and will be activated when the bundle is restarted. Currently also this option is not implemented. Since Eclipse 3 is based upon the OSGi framework, AJEER had been reengineered to support the integration of AspectJ into the new Eclipse kernel. Therefore AJEER can be used to write aspects in AspectJ as deployable OSGi bundles, but the aspects can not be activated at runtime.

### 5.2 AOSGI

AOSGi [16] is a project that emanated from the eclipse equinox incubator. Like AJEER, it is also intended to support load-time weaving in the OSGi environment.

Bytecode weaving is achieved by replacing the default framework adaptor. The AOSGi adaptor intercepts the class loading for each bundle and invokes the AspectJ weaver. This is possible because of the hookable adaptor architecture introduced by Equinox 3.2. Standard AspectJ 5 "aop.xml" files are used by the weaver, each packaged in the bundle containing the concrete aspects it declares. The set of configuration files, and hence aspects, visible to a particular bundle are determined using the normal class loader search order for resources: `ClassLoader.getResources("aop.xml")`. Hence only aspects are woven in a bundle which are defined in bundles it depends on. Two models of aspect application are possible: "opt-in" and "co-opt". Using the "opt-in" model an application developer writes his own aspect or customizes one aspect provided in a library. The "co-opt" model allows someone different than the application developer to write an aspect and package it in a bundle to extend other bundles. AOSGi introduces new OSGi bundle manifest headers to specify which bundles should be affected by an aspect-promoting bundle. One example is the Supplement-Bundle header.

```
Supplement-Bundle: BundleNamePattern
[, BundleNamePattern]*
```

Wildcards are used to specify a set of bundles. This offers high degrees of freedom in respect to the definition how bundles are affected by the aspects. To support additional manifest headers, the adaptor hook mechanism of the Equinox OSGi framework is exploited, which is the common way to add new functionality to the Equinox OSGi framework. Note that this solution is specific to the Equinox OSGi implementation and not part of the OSGi core specification.
The presented approaches integrate aspect-oriented programming seamlessly into the OSGi Service Platform. The main difference to our approach is the lack of dynamicity.

### 5.3 Jadabs

Frei and Alonso present in [17] an approach to integrate a d-AOP framework, which uses dynamic proxies to implement the aspects, into the OSGI Service Platform. They modified the OSGi API and the class loading of the used d-AOP framework (to solve the class loader problem). In contrast to their approach, we do not change the API and we are able to deploy the aspects as bundles, which we consider of utmost importance for a seamless integration.

There are also approaches for integrating AOP into other middleware systems [18,19]. Nevertheless all of them have in common that they (i) do not support dynamic AOP and that they (ii) rely on own implementations. In contrast, our approach uses a well proven middleware (SOA) framework which can be combined with popular DAOP implementations.

## 6 Conclusion

This paper presented an approach to integrate JBoss AOP (which supports d-AOP) into the OSGi Service Platform—an open, modular and scalable SOA environment—represented by Equinox. Deploying and undeploying aspects is mapped to OSGi bundle installation and de-installation. With our integration of the d-AOP framework, adaptation with respect to a changing environment can be achieved at runtime without stopping or redeployment of active services. In this paper the technical aspects of a seamless integration have been presented. Currently, we are working on semantic problems arising when d-AOP is applied and plan to evaluate our framework with a case study.

## References


1. Cámara, J., Canal, C., Cubo, J., Rodriguez, J.M.M.: An Aspect-Oriented Adaptation Framework for Dynamic Component Evolution. In Cazzola, W., Chiba, S., Coady, Y., Saake, G., eds.: RAM-SE, Fakultät für Informatik, Universität Magdeburg (2006) 59–70
2. Greenwood, P.: Dynamic Framed Aspects for Dynamic Software Evolution. In Cazzola, W., Chiba, S., Saake, G., eds.: RAM-SE, Fakultät für Informatik, Universität Magdeburg (2004) 101–110
3. Liu, R., Gibbs, C., Coady, Y.: MADAPT: Managed Aspects for Dynamic Adaptation based on Profiling Techniques. In: ARM '04: Proceedings of the 3rd Workshop on Adaptive and Reflective Middleware, New York, NY, USA, ACM Press (2004) 214–219
4. Popovici, A., Gross, T., Alonso, G.: Dynamic weaving for aspect-oriented programming. In: AOSD '02: Proceedings of the 1st International Conference on Aspect-Oriented Software Development, New York, NY, USA, ACM Press (2002) 141–147
5. OSGiAlliance: About the OSGi service platform: Technical whitepaper. `http://www.osgi.org/documents/collateral/TechnicalWhitePaper2005osgi-sp-overview.pdf` (November 2005)

6. Knopflerfish: Knopflerfish OSGi homepage. http://www.knopflerfish.org/ (March 2007)
7. Apache.org: Apache Felix homepage. http://cwiki.apache.org/FELIX/index.html (March 2007)
8. Eclipse Foundation: Equinox OSGi Framework Homepage. http://www.eclipse.org/equinox (March 2007)
9. Popovici, A., Alonso, G., Gross, T.: Just-In-Time Aspects: Efficient Dynamic Weaving for Java. In: AOSD '03: Proceedings of the 2nd International Conference on Aspect-Oriented Software Development, New York, NY, USA, ACM Press (2003) 100–109
10. Vasseur, A.: Dynamic AOP and Runtime Weaving for Java - How does AspectWerkz Address It? http://aspectwerkz.codehaus.org/downloads/papers/aosd2004-daw-aspectwerkz.pdf, AOSD 2004 International Conference on Aspect-Oriented Software Development, Invited Industry Talk (March 2004)
11. Previtali, S.C.: Dynamic Updates: Another Middleware Service? In: MAI '07: Proceedings of the 1st Workshop on Middleware-Application Interaction, New York, NY, USA, ACM Press (2007) 49–54
12. Lindholm, T., Yellin, F.: Java Virtual Machine Specification. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (1999)
13. OSGi: OSGi R4 Service Platform Core Specification. http://osgi.org/osgitechnology/downloadspecs.asp (July 2006)
14. Lippert, M.: AJEER: An AspectJ-Enabled Eclipse Runtime. In: OOPSLA '04: Companion to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications, New York, NY, USA, ACM Press (2004) 23–24
15. The AspectJ Team: The AspectJ Development Environment Guide. http://www.eclipse.org/aspectj/doc/released/devguide/ (March 2007)
16. Webster, M.: Equinox Incubator: Aspects and OSGi. http://www.eclipse.org/equinox/incubator/aspects/index.php (February 2007)
17. Frei, A., Alonso, G.: A Dynamic Lightweight Platform for Ad-Hoc Infrastructures. In: PERCOM '05: Proceedings of the Third IEEE International Conference on Pervasive Computing and Communications, Washington, DC, USA, IEEE Computer Society (2005) 373–382
18. Vaysse, G., André, F., Buisson, J.: Using aspects for integrating a middleware for dynamic adaptation. In: AOMD '05: Proceedings of the 1st Workshop on Aspect-Oriented Middleware Development, New York, NY, USA, ACM Press (2005)
19. Eichberg, M., Mezini, M.: Alice: Modularization of middleware using aspect-oriented programming. In Gschwind, T., Mascolo, C., eds.: Software Engineering and Middleware: 4th International Workshop, SEM 2004. Volume 3437., Linz, Austria, Springer-Verlag GmbH (March 2005) 47–63

# IDE-integrated Support for Schema Evolution in Object-Oriented Applications

*Position paper*

Marco Piccioni[1], Manuel Oriol[1], Bertrand Meyer[1]

[1] Chair of Software Engineering, ETH Zurich,
Clausiusstrasse 59,
8092 Zurich, Switzerland
{marco.piccioni, manuel.oriol, bertrand.meyer@inf.ethz.ch}

**Abstract.** When an application retrieves serialized objects of a class that changed, it may have to cope with modifications of the semantics. While there are numerous ways to handle the resulting mismatch at runtime, developers are typically required to provide some code to reestablish the intended semantics of the new class. We show here how to instruct an IDE with class version information, in a way that it can provide help and guidance for a semantically correct schema evolution.

**Key words:** object-oriented, schema evolution, type converter.

## 1 Introduction

In object-oriented applications, serializing objects (encoding them in binary or in some other format) is widely used to store data. Once an object has been serialized, it can be stored on disk for later deserialization or sent remotely to other applications. Serialization is more lightweight then either object-oriented and relational full-fledged database solutions. At the same time, it lacks important services like transaction handling and object querying. With respect to a relational database, both serializing objects and using an object-oriented database imply that programmers can remove the object-relational mapping layer from their application. Eliminating the well known object-relational impedance mismatch [2] has a price: the stored objects are more tightly coupled to the application, because an additional layer of indirection is missing. This can be an issue when the corresponding class structure evolves over time. The system is then typically not able to read the previously stored objects of a class anymore because a new version of the class itself is being used. Developers have therefore to gather information on the stored class version, understand how the values of the stored objects relate to the semantics of the new class and finally provide an appropriate conversion routine.

The proposed approach to the schema evolution problem is two-fold. On the one hand we suggest a robust retrieving algorithm that, after trying different techniques to

convert the old objects into the new ones, forbids the application to access potentially semantically inconsistent objects. On the other hand we propose to help the developer that works with different versions of a class and provide a solution integrated to an existing IDE.

Section 2 analyzes four approaches to object serialization, namely Java serialization, .NET serialization, the db4o object-oriented database system and Eiffel serialization. Section 3 presents the algorithm that performs the updates. Eventually Section 4 describes the proposed IDE integration and a first proof of concept implementation using the Eiffel language.

## 2 State of the art

This section describes four existing different approaches to object serialization: Java standard serialization mechanism, the Version Tolerant Serialization within the .Net framework, the db4o object-oriented database solution and the Eiffel serialization framework.

### 2.1 Java serialization

The Java object serialization API, a framework for serializing and deserializing objects, provides the standard wire-level object representation for remote communication, and the standard persistent data format for the JavaBeans component architecture [5]. A class can enable future serialization of its instances by implementing the `Serializable` interface. As suggested by Bloch [1], it is worth noticing that this deceivingly simple addition brings important constraints. In fact, every `Serializable` class has an associated unique version identification number, known as serial version UID. If one does not specify it by declaring a field named *serialVersionUID* and by explicitly giving it a value, the system automatically generates it using an algorithm that closely couples it with the class structure. The default serialized form of an object is therefore an encoding of the physical representation of the object graph rooted at the object itself. Considering that some class details may even vary depending on compiler implementations, unexpected exceptions during deserialization may happen at runtime.

More generally, by implementing the `Serializable` interface the flexibility to change the class implementation in the future significantly decreases, because all its internal representation becomes part of its exported API, thus invalidating encapsulation, one of the key principles of object-orientation. In addition to this, deserialization can be considered an extra-linguistic mechanism for creating objects, and so it should be responsible for establishing the class invariant and for ensuring that no illegal access to the object is possible.

While using the default serialized form can sometimes be appropriate, the ideal serialized form of an object should contain only the logical data represented by the object, and should be independent of the physical representation. This can be achieved by implementing a custom serialized form via methods `writeObject` and

`readObject`, which are reflectively invoked and typically used to establish the class invariant. A developer can also implement `readResolve`, which creates a new object and then delegates to the constructors the task to reestablish the class invariant.

### 2.2 .NET serialization

The .NET framework, starting from version 2.0, provides a set of features, called Version Tolerant Serialization (VTS), which makes it easier to handle serializable types across different versions [11]. VTS comes in two flavors: binary serialization and XML (Soap) serialization. The binary serialization uses a `BinaryFormatter` to provide a compact and efficient byte stream for usage within the .NET framework. When an object is serialized, the name of the class, the assembly, and all the data members are serialized. A requirement placed on the serialized object and all the referenced objects in the object graph is that the corresponding classes have to be tagged with the `Serializable` attribute. As the `Serializable` attribute is not inherited, the serializable status for a class that inherits from an existing serializable class must be stated explicitly. If a data member should not be serialized, it is also possible to tag it using the `NonSerializedAttribute` attribute. A significant advantage to using attributes for events as opposed to using interfaces is that the event mechanism is decoupled from the class hierarchy.

If portability across different platforms is needed, `XmlSerializer` can be used instead. This only serializes public properties and fields.

The .NET framework handles schema evolution issues as follows: when an object of an old version of a class retrieves an object of a newer version with an added data member, the mechanism ignores the latter. When an object of a new version of a class with an added data member retrieves an object of an older version of the class, it does not throw an exception if the new data member is tagged as “optional” with the `OptionalFieldAttribute` attribute.

It’s worth noticing that the constructors of an object are not automatically called during the deserialization process, so implicit class invariant violations may happen if the developer does not take appropriate actions. In these situations the developer may adopt a custom serialization by implementing reflectively invoked methods that provide hooks into the serialization/deserialization process. For example, to provide an ad hoc initialization to a newly added field, one must create a method that accepts as an argument a `StreamingContext` and apply to it the `OnDeserializingAttribute` attribute.

### 2.3 Using an OODBMS for serialization: db4o

When using an object-oriented database like db4o to serialize objects [3, 10], we neither have to change the class schema by implementing an interface like `Serializable` nor have to tag the class with a `Serializable` attribute. The db4o container, `ObjectContainer`, takes care of providing all the needed persistence services. It receives each object as an argument and stores it as-is. The increased transparency, the possibility to have services like transaction handling, object

browsing, native querying, and a very small memory footprint, suggest that this solution can be considered an overall better alternative to pure object serialization for both Java and .NET.

Regarding schema evolution handling, in case the developer needs a custom behavior to reestablish the invariant with respect to an older stored version of the object, there are two possibilities. He can either choose to use reflectively invoked methods like `objectOnActivate` in the object class or, even better, can register listeners to specific `ObjectContainer` events outside the object class. In the latter case, when the container "activates" an object after retrieval, it invokes the method `onEvent`, passing to it the newborn object as an argument, so that it can be properly initialized.

An alternative and interesting scenario occurs when the developer does not foresee the possible issues and "forgets" to code appropriate methods to handle the conversion. Unfortunately, in this case the newly added attributes are automatically initialized to their default values. This is an excessively confident level of transparency, because it may lead to introduce in the system objects whose class invariants may not be valid anymore.

### 2.4 Eiffel serialization

Eiffel serialization [4, 6, 7] presents a solution based on the identification of three steps:

1. Detection of version mismatches for previously stored objects.
2. Notification to the system of such mismatches.
3. Safe conversion of the needed objects on demand.

The implementation of this solution is similar to the one previously reviewed for Java, except for the fact that Eiffel does not need interfaces, because it supports multiple inheritance. Custom behavior can therefore be provided by inheriting from class `MISMATCH_CORRECTOR` and redefining the reflectively invoked feature `correct_mismatch` to establish the class invariant. It is also worth mentioning that in Eiffel an invariant violation is very easy to detect, because the language provides embedded support for Design by Contract providing an explicit way to declare the class invariant itself in the class text via the `invariant` clause.

The Eiffel serialization mechanism cannot be defined as fully "tolerant" either. In fact an exception is raised as a default if a mismatch is detected at runtime and no redefinition of the feature `correct_mismatch` is found. This implementation choice makes therefore impossible for an inconsistent object to be accepted in the system after deserialization because a developer happened to forget to explicitly take care of the conversion.

## 3 Performing the updates: general approach

Programmers that work on different versions of a class have typically very little help in managing these versions. To be aware of the consequences of deserializing objects of old versions of a class they have to run numerous test cases, proportional to the product of the number of stored classes and the number of releases, which can be quite large. These tests cannot be constructed automatically because, in addition to binary compatibility, one must test for semantic compatibility. To allow a higher degree of control on the schema evolution and to keep compatibility with the already existing solutions, the proposed update algorithm consists in several steps, illustrated in the synthetic flow-chart in Fig. 1.

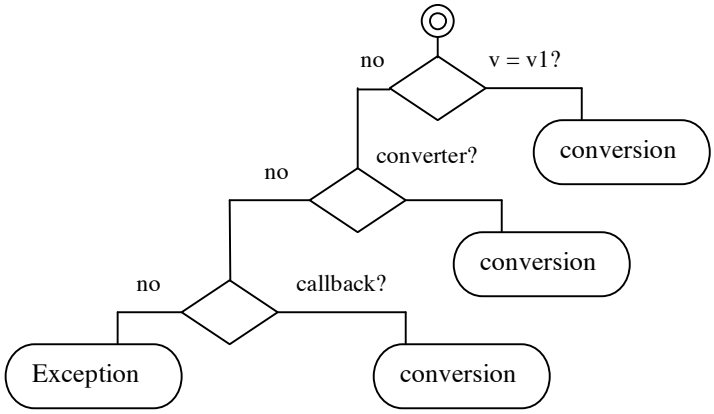



**Fig. 1.** Flow-chart of the proposed update algorithm from an old class version v1 to a new class version v.

The algorithm first checks if the retrieved version is the same as the current version. If it is not, it will check if a *converter* is available in the current class for the stored version (see 4.1). If it is available, it will invoke and pass the retrieved type as an argument. If a *converter* is not available, it will further check if the class inherits from a specific class that can help in handling the mismatch (like `MISMATCH_CORRECTOR` in Eiffel) and invoke a redefined feature (like `correct_mismatch` in Eiffel). If both the last two checks fail, it raises an exception to state clearly that an inconsistency may happen and to stop the application before the inconsistent object can do any damage. Thus the algorithm takes into account several mechanisms for handling schema evolution, and assigns to them different priorities.

## 4 IDE support for handling schema evolution in Eiffel

To ease the task of writing the type converters and the mismatch correctors we propose an integration in the EiffelStudio IDE to make it class-version-aware and therefore capable of providing support to the developer for taking the most appropriate action. This implies augmenting the stored objects with additional meta-information about versions, to be used at retrieval time.

### 4.1 Type converters

Neamtiu et al., in their work on dynamic software evolution [9], proposed to use type converters to update data types at runtime. Their solution relies on heuristics that builds semi-automatically the converters from static analysis of the code and its instrumentation. The Eiffel language has a `convert` clause [4, 8] to specify conversions from a type to another. The mechanism is already used to provide a systematic way to handle conversions between basic (or primitive) types like `INTEGER` or `REAL`, or between `STRING` types as represented in Eiffel and .NET. Similar conversions are supported by most programming languages in an ad hoc fashion. While the basic idea to provide conversion functions that take care of the conversion details is well known, having the converters embedded in the language provides the advantage of a coherent framework that takes care of the conversions at runtime without additional instrumentation.

An important semantic constraint of this approach is that a type is considered to either *conform* (in the sense of inheritance) or *convert* to another. As a type obviously conforms to itself, it is not possible to convert a type to another type as-is. This happens because the two types retain the same name, even if they have a different schema, so the compiler would reject the conversion.

Our assumption is that two versions of the same class are different types. With this in mind, we propose a prototype implementation that is mostly transparent to the developer and generates different names for different versions of the class to use the converter mechanism.

### 4.2 An integrated EiffelStudio GUI

The main intent of our proposal is to guide the developer while delivering a new version of a class as version-aware as possible. This means that the class, once consolidated, knows how to handle all the possible type conversions that may be necessary in the future.

Both type converters and the update algorithm shown in figure 1 are the backbone of a semantically consistent framework for schema evolution.

The GUI is fully integrated in the EiffelStudio class browser. It is the presentation layer of the proposed framework, and fosters interaction between the developer and the underlying mechanism. It performs the following basic tasks:

1. Enables browsing of all the previous class versions.

2. Enables consolidation of the current version so that it is ready to be released (it saves it with the updated version information).
3. Proposes different code templates for the type converters bodies, depending on previously recorded refactorings.
4. Issues an appropriate warning, stating that conversions from older versions to newer ones will not be possible anymore at runtime, if the developer refuses to take appropriate action to handle the conversion.

### 4.3 Implementation details

Before realizing all the steps illustrated above, it is necessary to make a preprocessor in order to tag class names with a version number in a transparent way.

In addition, the GUI backend has to:

1. Record developer's actions, more specifically the different kind of refactorings that may take place.
2. Associate the different refactorings to the different versions.
3. Read the recorded refactorings for the current version in case of consolidation.
4. Generate different code templates for the type converters bodies, depending on the recorded refactorings.
5. Consolidate the converters depending on the developer's choice.

As the overall effort is non-trivial, it is necessary to separate the intended task into different steps, undertaken in an iterative fashion. As a minimum support, the framework does the following:

1. Provides a skeleton implementation of at least a converter body.
2. Performs a test of object creation by checking a possible violation of the current class schema with the invariant of the old version, looking for a possible violation. Issues a warning if a violation occurs.
3. Suggests an initialization that does not invalidate the new invariant for added fields.
4. Issues a warning to the developer, suggesting that it is his responsibility to check and complete the converter implementation, as this operation cannot be fully automated.

In addition to handling the inclusion of a new attribute in the class schema, an extension of the support can include different kinds of refactoring, like:

- Refactorings on data fields:
    - Removing an attribute.
    - Renaming an attribute.
    - Changing an attribute type.
    - Changing an attribute visibility.
- Refactorings on routines:

        - Adding a routine.
        - Removing a routine.
        - Renaming a routine.
        - Changing a routine return type.
        - Changing a routine visibility.
        - Changing the type of arguments of a routine.
        - Changing the order of arguments of a routine.
- Refactorings on classes
        - Renaming a class.
        - Adding an inheritance relationship.
        - Removing an inheritance relationship.
        - Changing the type of a generic parameter.
        - Changing the constraint of a generic parameter.

### 4.4 A first proof of concept

To test the idea of using converters for schema evolution we have tagged class names with version numbers and have showed with a prototype that the approach is feasible. Assignments and argument passing from the old version to the new one are also tested:

http://se.inf.ethz.ch/people/piccioni/software/prototype_code.zip.

Changing explicitly class names is not desirable because the new version would break all the clients that are using the old class version, and in addition a separate concern like serialization should not be so tightly coupled to the class itself via its name. The class name should in fact ideally reflect the underlying abstraction only.

We therefore propose to introduce version numbers to the serialized objects and then use the converters in a way that they accept the same type with a different version number.

To give an idea of the converters syntax, we hereafter show an extract from the prototype implementation:

```
class MY_SAMPLE_CLASS

    create

        make,

        from_my_sample_class_v1

    convert

        from_my_sample_class_v1({MY_SAMPLE_CLASS_V1})

    feature -- Access
```

```
        sample_integer: INTEGER

        sample_string: STRING

        added_attribute:STRING

    feature -- Conversion

        from_my_sample_class_v1(a_v1:MY_SAMPLE_CLASS_V1)

                --the ad hoc converter

        do

            sample_integer := a_v1.sample_integer

            sample_string := a_v1.sample_string

            added_attribute:="This string has been added"

        end

end
```

## 5 Conclusions and Future Work

We have shown how to provide better support for handling schema evolution in object-oriented applications. This can be achieved using a two-sided approach. On the one hand we instruct the system to record specific refactorings across different class versions so that it can propose reasonable templates for the converters. On the other hand we propose to improve the developer's interaction with the system by integrating a module in an existing IDE. To release a fully integrated module we need to:

- Adapt the embedded converter mechanism in the Eiffel language so that it can accept to convert two different versions of the same class.
- Program an extension to the EiffelStudio GUI that enables browsing of all the previous class versions, saves them with the updated version information, proposes different code templates for the type converters bodies (depending on previously recorded refactorings).
- Program an extension to the framework that can include different kinds of refactoring.

This is what we are currently implementing.
In the future, the automatic support can also be further extended including:

- The ability to serialize objects of the current version into objects of previous versions.
- The ability to deserialize objects of more recent versions into objects of previous versions.

## References


1. Bloch, J.: Effective Java. Prentice Hall PTR. (2001)
2. Date C.: Introduction to Database Systems 8th ed. Addison Wesley (2003)
3. db4o object oriented database API documentation, http://www.db4o.org/community/ontheroad/apidocumentation/index.html
4. ECMA committee TC39-TG4, ECMA International standard 367. Eiffel Analysis, Design and Programming Language (2005)
5. Gosling J., Joy B., Steel G. and Bracha G.: The Java Language Specification. 3rd ed. Addison Wesley (2005)
6. Meyer B.: Object Oriented Software Construction. 2nd ed. Prentice Hall PTR (1997)
7. Meyer B.: Eiffel: The Language. Prentice Hall (1992)
8. Meyer B.: Conversions in an Object-Oriented Language with Inheritance, in JOOP (Journal of Object-Oriented Programming), vol. 13, no. 9, January 2001, pages 28-31.
9. Neamtiu I., Hicks M., Stoyle G. and Oriol M.: Practical Dynamic Software Updating for C. ACM Conference on Programming Language Design and Implementation (PLDI). Ottawa, Canada (2006)
10. Paterson J., Edlich S., Hörning H. and Hörning R.: The Definitive Guide to db4o. Apress (2006)
11. Version Tolerant Serialization, http://msdn2.microsoft.com/en-us/library/ms229752.aspx

# Towards correct evolution of components using VPA-based aspects

Dong Ha Nguyen and Mario Südholt

OBASCO project; Ecole des Mines de Nantes - INRIA, LINA; Nantes, France
{Ha.Nguyen, Mario.Sudholt}@emn.fr

**Abstract.** Interaction protocols are a popular means to construct correct component-based systems. Aspects that modify such protocols are interesting in this context because they support the evolution of such components. A major question then is whether aspect-based evolutions preserve fundamental notions of correctness, in particular compatibility and substitutability, of components. In this paper we discuss how such component correctness properties can be proven in the presence of aspect languages of limited expressiveness. Concretely, we show how common evolutions involving VPA-based aspects [12] can be proven correct directly in terms of operators of the aspect language. Furthermore, we present first ideas of how to use existing model checkers for the automatic verification of such properties.

## 1 Motivation

Interaction protocols are a popular means to construct correct component-based systems and document them [19, 6]. Relying on explicit protocols, evolution of component-based systems can be frequently expressed in a concise manner using aspects that modify such protocols [12].

A major question for the evolution of component-based systems is whether evolution preserves compositional properties of these systems, in particular compatibility and substitutability of components, two fundamental notions that are typically defined in terms of subset relationships of trace and failure sets admitted by the original and evolved versions of a system [19, 13]. Currently, almost all component-based systems with interaction protocols have used finite-state protocols; only few work has explored the preservation of compositional properties in the context of aspects modifying such interaction protocols.

In this paper we consider how compositional properties can be defined and verified in the context of the evolution of components that are equipped with a more expressive brand of interaction protocols, protocols defined in terms of Visibly Pushdown Automata (VPA) [2]. VPA allow to define protocols that include well-formed nested contexts, such as correct nesting of recursive calls to and returns from a server. VPAs are strictly more expressive than finite-state automata

* This work has been supported by AOSD-Europe, the European Network of Excellence in AOSD (http://www.aosd-europe.net).

(which generate regular languages) but strictly less so than pushdown automata (which generate context-free languages). In contrast to finite-state based systems, VPA-based protocols allow (some) nested terms to be correctly matched without having to restrict the nesting depth. In contrast to pushdown automata, VP languages are closed under all basic operations, including intersection and complement, and all basic decision problems are decidable. As the main contribution of this paper, we discuss how compatibility and substitutability properties of component-based applications can be proven if interaction protocols are subject to evolution using VPA-based aspects [12].

Concretely, we motivate and sketch three extensions to the VPA-based aspect language that are useful for the evolution of component systems: a more general definition of sequence pointcuts, a new pointcut operator that allows nested contexts to be matched if their depths exceed a threshold and a new advice construct that allows to close an open nested context. Furthermore, we introduce several proof methods that can be used to prove the preservation of compositional properties if the resulting aspect language is used for component evolution. Finally, we present preliminary results how to use existing model checking techniques for the automatic verification of such properties.

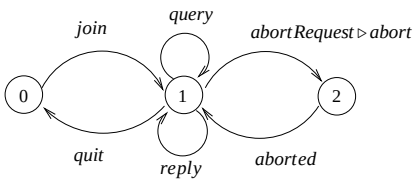
The paper is structured as follows. In Section 2, we motivate and sketch three extensions to our VPA-based aspect language. We introduce corresponding proof methods for software components in Sec. 3. VPA-based aspects and model checking techniques are the subject of Sec. 4. Finally, we give a conclusion and present future work in Sec. 6.

## 2 VPA-based aspects for component evolution

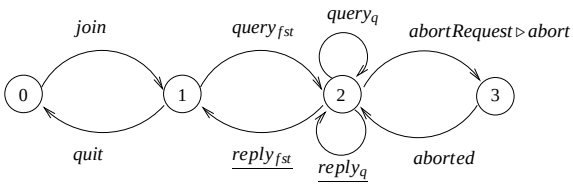
In the following we illustrate the use of expressive, *i.e.,* non-regular aspect languages in the context of (distributed and recursive) P2P algorithms. Fig. 1(a) shows a protocol which allows nodes to join or quit a P2P network, repeatedly execute recursive queries and abort queries if requested (by means of an advice $abortRequest \triangleright abort$).

The regular aspect on the left hand side does not enforce an important restriction: abort requests should only be allowed if there is at least one on-going query (the number of replies occurring at state 1 may equal or even exceed the number of queries in the regular case). The VPA-based aspect shown on the right hand side, however, ensures this property by distinguishing the first query and the matching reply events from the remaining ones by associating stack symbols to transitions (in the figure, stack symbols are set as indexes to execution events and matching replies are underlined). Abort requests therefore may occur only in contexts where at least one query is open. Furthermore, VPAs ensure that there cannot be more *abortRequest* events than query events.

In previous work [12], we have proposed a language for VPA-based aspects and a corresponding execution library. This language allows to define pointcuts in terms of paths in a VPA whose matching during execution of a base application, *e.g.,* an OO program, triggers the execution of advice as illustrated in 1(b). We




(a) Aborting on-going queries (regular)




(b) Aborting on-going queries (VPA)

now illustrate that VPA-based aspects are useful for the evolution of component-based systems by presenting evolutions that can be supported using three new operators that extend our original aspect language.

*Evolution of the recursive structure of P2P algorithms using pointcut operators.* Evolution of distributed algorithms, such as P2P algorithms, often aims at the optimization of the underlying traversal strategy. A simple example of a corresponding heuristic is to perform a more superficial but faster search on nodes whose distance from the root node exceeds a certain threshold. Since VPAs faithfully allow to define the depth of nested terms, such heuristics can be directly expressed using a pointcut operator $D_m^{>k}$ that matches only calls to $m$ that occur at a depth larger than $k$. For example, the following aspect caches queries at depth greater than 5 (where $\mu a. \ldots ; a$ denotes recursion in VPA-based aspects):

$$\mu a.\ D^{>5}_{query_q} \rhd getCacheValue\ ;\ a$$

*Accomodating new execution events through general sequencing of pointcuts.* The evolution of component systems frequently requires to cope with new execution events, either by abstracting from them (*i.e.,* allow interleaving of such events on the protocol level) or, to the contrary, forbid the occurrence of such events. Current aspect languages for protocols typically include a sequence operator ; on the pointcut level, such that terms $a; b$ allow either no interleaving (*i.e.,* the term corresponds to a single-step transition as, *e.g.,* JAsCo's stateful aspects [17]) or interleaving of arbitrary events between occurrences of $a$ and $b$ (as the stateful aspects of [4, 12]).

Using the latter semantics, the aspect

$$\mu a.\ join\ ;\ query_q \rhd createSession\ ;\ a$$

(repeatedly) creates sessions once the current node has joined a P2P network, occurrences of arbitrary events followed by a query. However, the occurrence

of events, *e.g.,* accessing the result of a query before execution of the query, cannot be ruled out with this form of sequencing alone. Relying only on the first semantics, individual transitions yield awkward formulations of non-trivial protocol-modifying aspects because all interleaving of events has to be defined explicitly.

In order to allow the concise definition of protocol evolution by arbitrary interleaving as well as through the (mandatory) absence of interleaving we have introduced a general sequencing operator $;_{\mathcal{I}}$ where $\mathcal{I}$ specifies the set of events, possibly $\emptyset$, that may be interleaved between the argument events.

The evolution consisting in forbidding previous accesses to the query result can then simply be expressed as:

$$\mu a.\ join\ ;_{\neg accessResult}\ query_q \rhd createSession\ ;\ a$$

*Handling of error conditions using advice operators.* The evolution of component-based systems frequently consists in the introduction of behavior to correctly handle error situations. In the case of recursive distributed algorithms error handling may involve the introduction of events that close a number of open recursive calls in order to skip the traversal of part of the underlying distributed network in which an error occured. Using VPA-based aspects such error handling strategies can be expressed using the advice-level operation $closeOpenCall_m$ that closes the open call to $m$: pointcuts matching on nested contexts can then be used to restrict the application of such advice to appropriate parts of the network. The following example illustrates the use of a closing operator to add a number of "fake" replies to queries when the query exceeds a given connection timeout (where $\Box$ denotes the choice between alternatives):

$$\mu a.\ query_f; (\underline{reply_f}\ \Box\ (connectionTimeOut \rhd closeOpenCall_{query_f}))\ ;\ a$$

## 3 Preservation of compositional properties

In this section we address the problem whether compositional systems that are subjected to evolution by VPA-based aspects can be proven to preserve fundamental composition properties. Our main point is that, in contrast to general aspect languages such as AspectJ, VPA-based aspect programs are amenable to formal correctness proofs.

Figure 1 illustrates the underlying model of component evolution and the compositional properties we consider. Starting from two protocols $p_1$, $p_2$ that constrain the interactions of two collaborating components $C_1$, $C_2$ a VPA-based aspect $\mathcal{A}$ is applied to $p_2$ yielding the protocol $p_3$ that defines the interactions of the component $C_3$ after evolution. As indicated in the figure we are interested in two fundamental correctness properties for components, compatibility and substitutability (see, *e.g.,* [13]).

Generally, *e.g.,* if turing-complete pointcut and advice languages are used for component evolution (as in AspectJ where arbitrary Java methods may be called in `if`-pointcuts and advice), such component properties cannot be proven

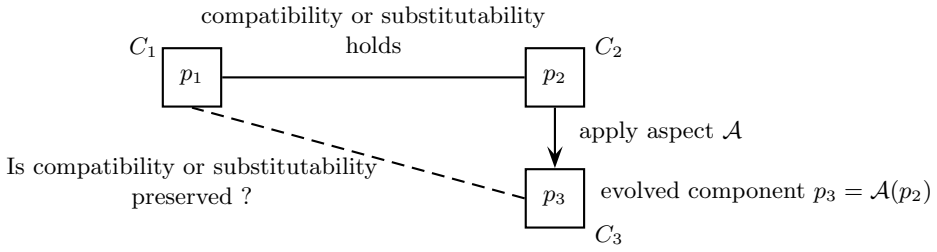



**Fig. 1.** Checking for preservation of compatibility/substitutability

formally. Furthermore, even in specific cases where a proof is possible, it can typically be performed only in terms of the woven program and not simply in terms of the aspects themselves. VPA-based aspects, however, support formal proofs of such properties because of their limited expressiveness and allow some important properties be proven simply by considering properties of the aspect language only. To this end we propose to exploit the "domain specific" characteristics of VPA-based aspects: proofs over nested contexts as well as regular structures can be performed directly in terms of corresponding features of our pointcut (indexed calls) and advice language (*closeOpenCall*).

Concretely, we demonstrate in the following three different types of proofs of property preservation that are supported by VPA-based aspects:

P1) Proofs that depend only on the properties of the aspect language, *i.e.,* that can be performed in terms of the evolution aspect $\mathcal{A}$ only.
P2) Proofs that can be performed in terms of $\mathcal{A}$ and properties of *classes* of protocols to which $p_1$ and $p_2$ belong.
P3) Proofs that require full knowledge of $\mathcal{A}$ and $p_1$–$p_3$.

We use standard trace-based notions of compatibility and substitutability [19] in this paper. Two protocols are compatible if they do not give rise to any conflict during execution, *i.e.,* no unexpected message is received during collaboration of two components according to their respective protocols. Substitutability of components is defined using trace set inclusion: protocol $p_1$ is substitutable for $p_2$ if its trace set is a superset of the trace set generated by protocol $p_2$. Since VPAs are closed under complement ("negation") it is, however, possible to apply the proof methods to more expressive notions of composition properties, for instance substitutability in the presence of failures [13].

In the following we will present three examples that illustrate the different proof types introduced above.

*P1: supporting evolution of error handling.* VPA-based aspects are unique (in particular compared to finite-state based approaches) in being able to handle a large class of traversals of distributed recursive algorithms, such as P2P algorithms. Frequently, error handling in such algorithms consists in terminating the exploration of some part of the network and search elsewhere. The action *closeOpenCall*($m$) that we have introduced in the advice language directly supports such error strategies by allowing to close a nested call of the method $m$.

We can exploit the precisely defined semantics and limited effect of the action *closeOpenCall* to prove some corresponding properties simply in terms of its definition. For example:

> If $p_1$,$p_2$ are protocols that recurse using $m$, $p_2$ is substitutable for $p_1$ and aspect $\mathcal{A}$ employs *closeOpenCall* to add returns of $m$ at the end of the execution of protocol $p_2$, then the adapted protocol $p_3$ is substitutable for $p_1$.

*P2: proving compatibility for depth-cutting heuristics.* Recursive distributed algorithms frequently do not unconditionally stop traversals at the top level, but typically do so only in specific contexts. A common example are heuristics that are formulated in terms of the traversal depth from the node where the search has been initiated. Since VPA-based aspect allow the explicit definition of aspects in terms of the nesting depth using the pointcut operator $D^{>k}_{m_c}$, corresponding compositional properties can be proven in terms of properties of this operator and classes of protocols to which it is applied. For example:

> If
> - $p_1$ belongs to the class of recursive protocols that repeatedly allows recursive remote calls and returns in $m$: $\mu a.m_c \,\square\, \underline{m_c}\ ;\ a$,
> - $p_2$ belongs to the class of protocols that include a remote call to $m$,
> - $p_1$, $p_2$ are protocol compatible and
> - aspect $\mathcal{A}$ employs a depth-defining operator $D^{>k}_{m_c}$ applying over $p_2$
>
> Then $p_1$ and $\mathcal{A}(p_2)$ are also compatible.

This property holds because the aspect may only cut calls to $m$ from traces of $p_2$: the resulting traces remain compatible with those admitted by $p_1$.

*P3: proving substitutability in terms of $p_1$,$p_2$ and $\mathcal{A}$.* Let us reconsider protocols $p_1$,$p_2$ as in the first example *i.e.,* $p_2$ represents a less restrictive recursive protocol than $p_1$ and $p_2$ is substitutable for $p_1$. Assume that now we would like to adapt protocol $p_2$ in order to cut the depth of queries to $k$ using an aspect with a depth-defining operator $D^{>k}_{m_c}$. In this case the resulting protocol $p_3$ is in general not substitutable for $p_1$, since $p_1$ may admit calls of depth deeper than $k$. By an analysis of $p_1$, we may find that the depth limit of $p_1$ is $q$ and $q \leq k$: we can then prove that $p_3$ is actually substitutable for $p_1$.

## 4 Towards model checking of VPA-based properties

We now consider the problem of using model checking techniques as a support for proving the preservation of properties of component systems that are subject to aspect-based evolution. We present first preliminary results on two main issues concerning the application of model checking: (i) adaption of the default model checking procedure to the verification of systems with VPA-based aspects and (ii) selection of an appropriate model checker.

Goldman and Katz [9] have introduced a formal framework for verifying the correctness of an aspect in a modular way. The general ideas of that approach are as follows. Assume that an aspect is defined by its pointcut designator $\rho$ and advice $A$ represented by a single state machine. Furthermore, the assumptions of the aspect about the base programs into which it may be woven are explicitly specified in form of an LTL formula $\psi$ from which a tableau $T_\psi$ can be constructed. Weaving two state machines respectively represented by the tableau $T_\psi$ and aspect advice $A$ according to pointcut $\rho$ results in an augmented state machine for the composed system $\widetilde{T_\psi}$. Then a model checker is employed to verify whether the system $\widetilde{T_\psi}$ satisfies a property $\varphi$ over the complete system. This means that if we could establish that a specific base program satisfies the assumptions $\psi$ we do not have to run the verification process again in case it has already been done for the combination $A$, $\rho$, $\psi$, and $\varphi$ before. Moreover, the real composed system has never to be model checked and thus this verification approach achieves modularity.

We plan to adapt the above framework for model checking to the correctness of VPA-based aspects. The underlying idea of the adaptation is to define an abstraction of VPA-based programs and aspects into regular systems and then apply model checking techniques. Application of existing model checkers requires to fix the depth of recursive contexts matched through VPA-based expressions, which implies to sacrifice some accuracy by using a suitable approximation. Through an extension of the approach of Goldman and Katz we intend to provide a means to efficiently model check abstractions of recursive VPA-based structures in which the depth is fixed but arbitrary.

**Table 1.** Some model checkers and corresponding properties

| Model checker | System model | Property | Remarks |
|---|---|---|---|
| SPIN | Promela (SPIN's language) | LTL | popular |
| NuSMV | Finite state machine | CTL,LTL | popular |
| CBMC | C source code | | |
| Bandera | Java source code | | no longer developed |
| UPPAAL | Real-time automata | CTL | |
| Verus | Real-time automata | CTL | no longer developed |

A second important factor for such an approach is the specific model checker being selected because of their rather different features, such as their system models, that are more or less suitable for our endeavor. Table 1 presents a list of some well-known model checkers and some of their features.

Three properties of model checkers we are particularly interested in are (i) how the input system can be modeled, (ii) what types of properties are supported, and (iii) how large of a system on which it can model check.

*System model.* All model checkers on Table 1 input system models as textual descriptions of state machines in their respective, specific formats. As previously

mentioned, since we are dealing with VPA-based systems, we have to transform them to less expressive automata then encode them in the input formats which are supported by these model checkers. This can be automatically done by a transformation tool, but should be easier for checkers using input languages similar to state machines (such as NuSMV and SPIN) than for checkers using general purpose languages (CBMC and Bandera).

*Property specification.* Most of the model checkers support property specified in some variants of LTL or CTL logics. Among them, model checker NuSMV provides the most flexibility by accepting both LTL and CTL properties. Checkers that are geared towards the handling of specific classes of properties (e.g., real-time properties in the case of UPPAAL and Verus) seem less appropriate for our approach.

*Scalability.* Since the abstraction from VPA-based properties to regular systems generates large finite-state machines (that are of a very specific form), the scalability of model checkers is an important criterion for the feasibility of our approach. However, while the first two features, system and property specification of model checkers, can be evaluated simply, it is more difficult to have a comparative view on the scalability feature since verification tools are typically designed and optimized for different specific domains. Among current model checkers, SPIN and NuSMV are highly recognized for their effectiveness in the presence of large systems.

Taking all the factors into consideration, we have chosen to perform first experiments on the model checking of VPA-based evolution properties using SPIN and NuSMV.

## 5 Related work

There is few related work on aspect-based evolution for component-based systems that considers the preservation of correctness properties for those systems after being changed by aspects. As to the best of our knowledge, this proposal is the first exploiting formal methods to investigate the preservation of compositional properties such as compatibility and substitutability for component-based systems that are subject to evolution by protocol-modifying aspects. However, our work still shares common interests with a large body of work covering aspects, components, and applications of formal methods on analysis and verification.

There are some approaches which consider aspect languages that support protocols, most notably [1, 5, 18]. Approaches [1, 5] feature regular aspect languages and a framework for static analysis of interaction properties. The language introduced by Walker and Viggers[18], one of the very few approaches providing non-regular (but not turing-complete) pointcut languages, proposes tracecuts which provide a context-free pointcut language. However, all of the above approaches do not use the language for an integration of aspects and components

or explore the problem of property-preserving for systems that have protocols being modified by aspects. Farías [7] has proposed a regular aspect language for components that admits advice modifying the static structure of protocols and considered proof techniques for the resulting finite-state based aspects.

There exist a large number of proposals that aim at applying AOP over component-based systems, *e.g.,* [8, 16, 14]. However, the aspect languages in those approaches do not provide explicit support for component protocols. Some of these approaches consider component compatibility, however, in a limited sense to our work: aspects are usually employed in such work to transparently introduce adaptation to components and thus preserve component compatibility. Our approach, in contrast, focuses on preserving protocol compatibility even if aspects have visible effects on interaction protocols.

Few work on evolution of component protocols seems relevant to our work. Braccialia et al. [3] present a formal methodology for automatically adapting components with mismatching interacting behaviors *i.e.,* conflicts at the protocol level. Protocols considered there are expressed by using a subset of $\mu-$calculus. They do not consider how component properties can be proved in terms of proof methods that exploit properties of modification operators. Ryan and Wolf [15] investigate how applications can accommodate protocol evolution. However, this approach concerns mainly syntactic changes on protocols.

Another category of related work is the application of formal methods to analyse aspect systems, such as [10, 11]. Our approach differs from those approaches in that we exploit the protocol-based specificities of our aspect language to prove composition properties of software components.

## 6 Conclusion and future work

In this paper we have investigated the preservation of compositional properties in the context of aspect-based evolutions on components. We have shown that aspect languages of limited expressiveness admit formal proofs of fundamental compositional properties directly in terms of the aspect languages. Concretely, we have shown that VPA-based aspects support formal proofs of component compatibility and substitutability in the presence of aspect-based evolutions of recursive distributed algorithms. As a second contribution, we have introduced three extensions to our VPA-based aspect language that support common evolutions: a more flexible sequencing operator, a depth-defining pointcut operator and a closing operator for recursive calls. Finally, we have presented first ideas of how to use use existing model checkers for the automatic verification of such properties and evaluated the characteristics of some popular model checkers to this end.

In the future we intend to work on extensions of the aspect language for VPA-based properties, extend the set of component evolutions that can be handled with our approach and provide a working method for the automatic verification of such evolutions with existing model checkers.

## References


1. Chris Allan, Pavel Avgustinov, Aske Simon Christensen, et al. Adding trace matching with free variables to AspectJ. In Richard P. Gabriel, editor, *ACM Conference on Object-Oriented Programming, Systems and Languages (OOPSLA)*. ACM Press, 2005.
2. Rajeev Alur and Parthasarathy Madhusudan. Visibly pushdown languages. In *Proceedings of the thirty-sixth annual ACM Symposium on Theory of Computing (STOC-04)*, pages 202–211, New York, June 13–15 2004. ACM Press.
3. Andrea Braccialia, Antonio Brogi, and Carlos Canal. A formal approach to component adaptation. *Journal of Systems and Software*, 2005.
4. R. Douence, P. Fradet, and M. Südholt. A framework for the detection and resolution of aspect interactions. In *Proc. of GPCE'02*, LNCS 2487, pages 173–188. Springer Verlag, October 2002.
5. Rémi Douence, Pascal Fradet, and Mario Südholt. Composition, reuse and interaction analysis of stateful aspects. In *Proc. of 3rd International Conference on Aspect-Oriented Software Development (AOSD'04)*, pages 141–150. ACM Press, March 2004.
6. Andrés Farías and Mario Südholt. On components with explicit protocols satisfying a notion of correctness by construction. In *International Symposium on Distributed Objects and Applications (DOA)*, volume 2519 of *LNCS*, pages 995–1006, 2002.
7. Andrés Farías and Mario Südholt. Integrating protocol aspects with software components to address dependability concerns. Technical Report 04/6/INFO, École des Mines de Nantes, November 2004.
8. Steffen Göbel, Chrstoph Pohl, Simone Röttger, and Steffen Zschaler. The COMQUAD component model — enabling dynamic selection of implementations by weaving non-functional aspects. In *Proceedings of AOSD'04*. ACM Press, 2004.
9. Max Goldman and Shmuel Katz. Maven: Modular aspect verification. In *TACAS*, pages 308–322, 2007.
10. Shmuel Katz and Marcelo Sihman. Aspect validation using model checking. In *Verification: Theory and Practice*, pages 373–394, 2003.
11. Shriram Krishnamurthi and Kathi Fisler. Foundations of incremental aspect model-checking. *ACM Trans. Softw. Eng. Methodol.*, 16(2):7, 2007.
12. Dong Ha Nguyen and Mario Südholt. VPA-based aspects: better support for AOP over protocols. In *4th IEEE International Conference on Software Engineering and Formal Methods (SEFM'06)*. IEEE Press, September 2006.
13. Oscar Nierstrasz. Regular types for active objects. In O. Nierstrasz and D. Tsichritzis, editors, *Object-Oriented Software Composition*, chapter 4, pages 99–121. Prentice Hall, 1995.
14. Nicolas Pessemier, Lionel Seinturier, Thierry Coupaye, and Laurence Duchien. A model for developing component-based and aspect-oriented systems. In *Proceedings of the 5th International Symposium on Software Composition (SC06)*, volume 4089 of *Lecture Notes in Computer Science*, page 259273, Vienna, Austria, mar 2006. Springer-Verlag.
15. Nathan D. Ryan and Alexander L. Wolf. Using event-based translation to support dynamic protocol evolution. In *ICSE '04: Proceedings of the 26th International Conference on Software Engineering*, pages 408–417, Washington, DC, USA, 2004. IEEE Computer Society.
16. Davy Suvée, Wim Vanderperren, and Viviane Jonckers. JasCo; an aspect-oriented approach tailored for component-based software development. In ACM Press,

editor, *Proc. of 2nd International Conference on Aspect-Oriented Software Development (AOSD'03)*, pages 21–29, March 2003.
17. W. Vanderperren, D. Suvee, M. A. Cibran, and B. De Fraine. Stateful aspects in JAsCo. In *Proc. of SC'05*, LNCS 3628. Springer Verlag, April 2005.
18. Robert J. Walker and Kevin Viggers. Implementing protocols via declarative event patterns. In *Proceedings of the ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE-12)*, pages 159 – 169. ACM Press, 2004.
19. Daniel M. Yellin and Robert E. Strom. Protocol specifications and component adaptors. *ACM Transactions of Programming Languages and Systems*, 19(2):292–333, March 1997.

## Aspect-Oriented and Reflection for Software Evolution

Chairman: Awais Rashid, Lancaster University, UK

# Characteristics of Runtime Program Evolution

Mario Pukall and Martin Kuhlemann

School of Computer Science, University of Magdeburg, Germany
{pukall, kuhlemann}@iti.cs.uni-magdeburg.de

**Abstract.** Applying changes to a program typically means to stop the running application. This is not acceptable for applications that need to be highly available. Such applications should be evolved while they are running. Because runtime program evolution is nontrivial we give terms and definitions which characterize this process. We specify two major dimensions of runtime program evolution – time of evolution and types of evolution. To sketch the state of the art we pick out different approaches which deal with runtime program evolution.

## 1 Introduction

Nowadays requirements for complex applications rapidly change due to frequently altering environmental conditions. Erlikh [1] and Moad [2] calculate the costs to maintain and evolve software to be 90 percent of the overall engineering costs. Attending new requirements using well-known software re-engineering techniques results the recurring schedule: stopping the application, applying the changes, testing the application, and restarting the changed application. This is unacceptable for applications that should be highly available, e.g. security applications, web applications or banking systems, because unavailability causes costs. Idioms like design patterns, component-based approaches, and configurable applications help to reduce the costs of maintenance, evolution, and unavailability of applications. Nevertheless, these approaches cannot guarantee availability of applications during evolution.

In this paper we define characteristics of runtime program evolution based on object-oriented programming paradigm. Concerning these characteristics we evaluate different existing approaches of runtime evolution. By this evaluation we try to find out, what the potentials of the approaches in terms of runtime evolution are. For that reason the approach's technical differences remain out of consideration.

## 2 Terms and Definitions

In this section we define terms which characterize runtime program evolution. We define *time* and *types* of evolution as the root categories because these are major features of runtime evolution.

### 2.1 Time of Evolution

We consider 3 different stages of program evolution: *Build time*, *Hire time* and *Deployment time* (see Figure 1). Build time references the process of static software evolution where a software engineer implements the changes in the program's source code. Building the program (i.e., source code compilation) terminates the process. *Hire time* is the time after the compilation of a class but before loading this class, e.g. before creating an instance of a changed class at all. *Deployment time* is the time after loading the code for execution, e.g. after class loading. Hire time and deployment time belong to runtime program evolution, whereas build time belongs to static program evolution.

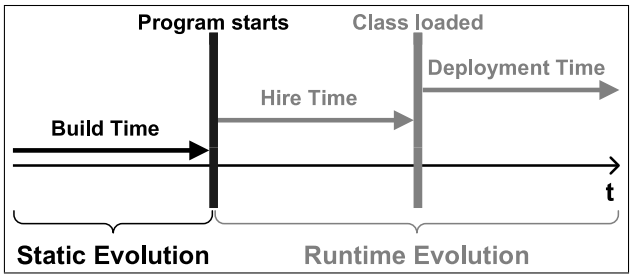



**Fig. 1.** Dimensions of Evolution Time.

### 2.2 Types of Evolution

We identified three fundamental types of runtime evolution: *Predictability*, *Kind of Code* and *Kind of Program Changes*.

**Predictability.** In some cases the application can be prepared for future program evolution – so called *anticipated* program changes. We experienced that the amount of program changes which cannot be foreseen (i.e. unanticipated program changes) is much bigger than the amount of predictable changes.

**Kind of Code.** Kind of Code classifies the code which must be changed to achieve modifications into: *source code*, *byte code* (intermediate code of platform independent languages) and native *binary code* (directly executable by the host system).

**Kind of Program Changes.** Gustavsson and Assmann [3] identified two types of program changes: *source code* and *state* changes. In our work we concentrate on changes of program's source code because source code changes can also effect the program's state. However, program state changes can be prepared using interfaces and introducing the new state through, e.g. Java Remote Method Invocation (RMI) or Java Platform Debugger Architecture (JPDA).

In Table 1 we give our classification of program changes based on source code modifications. The classification respects major attributes of object-oriented

paradigm and contains: changes that result in modified *structure*, *behavior*, *abstraction & encapsulation*, and *inheritance & polymorphism* of program's source code (which is organized in classes).

| Examples | Categories of Source Code Changes | | | |
|---|---|---|---|---|
| | **Structure** | **Behavior** | **Encapsulation & Abstraction** | **Inheritance & Polymorphism** |
| add/remove class | yes | yes | yes | no |
| add/remove base/sub class | yes | yes | yes | yes |
| modify method body | no | yes | no | no |
| change method modifier | no | no | yes | optional |
| add/remove class variable | yes | no | yes | no |
| add/remove base/sub class variable | yes | no | yes | optional |
| add/remove class method | no | yes | yes | no |
| add/remove base/sub class method | no | yes | yes | optional |

**Table 1.** Examples and Classification of Source Code Changes.

Due to space limitations we only depict a subset of possible source code changes. Each kind of source code change influences at least one category of our classification[1]. The benefit of our classification is a better highlighting of the approach's shortcomings concerning the feasibility of source code changes.

## 3 Evaluation

In this section we evaluate different approaches which fit the topic of runtime evolution. Unless there is a huge amount of approaches we limit ourselves to these ones because they can be attached to technical different ideas of approximating runtime program evolution (e.g., aspect-oriented techniques or script-based techniques). Despite the technical differences we only evaluate the approach's spectrum of runtime evolution. We hold that it is less interesting *how* runtime evolution is achieved but rather *what* the potentials of the approach in terms of runtime evolution are.

### 3.1 Javassist

*Javassist* (Java Programming Assistant) enables Java byte code changes [4, 5]. Using Javassist's source level API new or changed byte code can be applied to existing class files without knowledge about the byte code of the classes. The byte code level API allows direct changes of class file content.

**Time of Evolution.** Byte code changes can be applied until *hire time* (Figure 2). Byte code changes after class loading cannot be applied because of limitations of the Java Virtual Machine (JVM).

[1] E.g., "modify method body" influences {behavior}, "add/remove class variable" influences {structure, encapsulation & abstraction}.

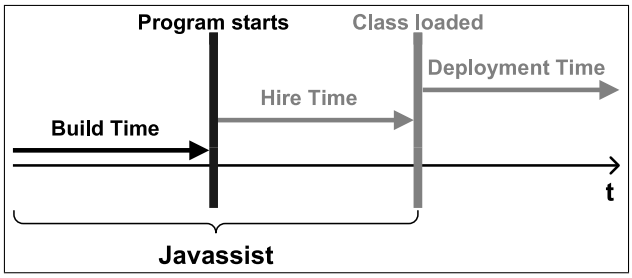



**Fig. 2.** Time of Evolution – Javassist.

**Types of Evolution.** We evaluate Javassist in terms of our classification (Table 1) even if it aims not source code changes. This is due to the fact that Java byte code keeps the object-oriented paradigm.

Javassist enables all kinds of code changes[2], i.e. it covers all categories of our classification. This predestinates the tool for computing *unanticipated* program changes. Unfortunately Javassist lacks the complete bandwidth of runtime evolution.

### 3.2 AspectWerkz

*AspectWerkz* [6–8] is a framework for aspect-oriented programming [9, 10] in Java.

**Time of Evolution.** AspectWerkz allows program changes until *deployment time* (Figure 3). The process of preparing the program for runtime evolution can only be performed until *hire time*.

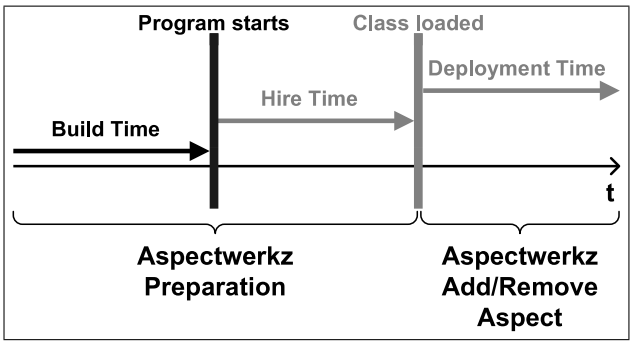



**Fig. 3.** Time of Evolution – Aspectwerkz.

**Types of Evolution.** AspectWerkz offers two different concepts for modifying running applications – aspects and mixins – which enables unanticipated pro-

[2] E.g., "add/remove class", "modify method body", "add/remove class method", etc.

gram changes. Mixins introduce interfaces and their implementations to classes at hire time. To achieve this, methods (interface methods) and a field (instance of the interface implementing class) will be introduced into target class (processed at byte code level). These code changes cover all categories of our classification (Table 1). The preparation for aspect deployment is processed at byte code level and results in modified method bodies. Aspects can be added or removed to (from) prepared program statements at deployment time.

AspectWerkz offers a lot of options to apply program changes at runtime based on aspects and mixins. Unfortunately the aspect deployment must be prepared until hire time. For that reason AspectWerkz does not provide the complete bandwidth of runtime evolution.

### 3.3 The Bean Scripting Framework – Java and JRuby in Concert

The *Bean Scripting Framework* (BSF) enables integration of scripting languages like JRuby[3] into Java programs [12]. The application is made up of the Java part (JP) and the JRuby part (SP). The interaction between the JP and the SP is controlled by the *BSFManager* which handles the scripting engine of JRuby. The BSFManager manages the execution of JRuby scripts and the interchange of object references[4] among the JP and the SP of the application.

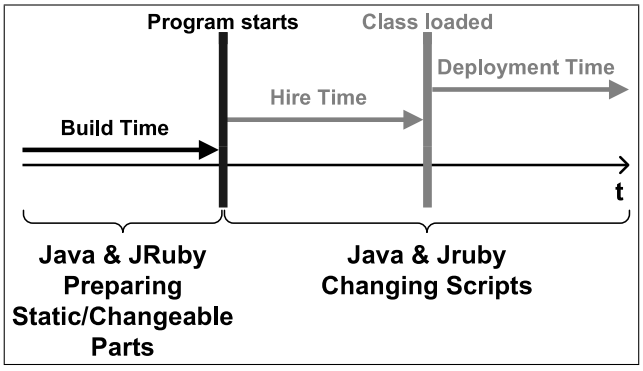



**Fig. 4.** Time of Evolution – Java and JRuby.

**Time of Evolution.** The combination of Java and JRuby enables program changes until deployment time (Figure 4). At build time it is terminated what is static (JP) and what is changeable (SP) in the application.

**Types of Evolution.** Java classes defined within JRuby scripts can be redefined at deployment time. This influences all categories of source code changes (Table 1). Modifications can be applied to existing instances of the redefined class, i.e. unanticipated program changes are possible.

[3] Java implementation of scripting language Ruby[11]
[4] JRuby interpreter can execute scripts which contain Java source code.

Summarized, this approach covers all categories of our classification and enables program evolution until deployment time. This results from the runtime interpreter of JRuby.

## 4 Conclusion

In this paper we characterized runtime evolution of programs. We identified two essential properties of runtime program evolution – time of evolution and types of evolution.

| **Approaches** | **Time of Evolution** | | |
|---|---|---|---|
| | **Build Time** | **Hire Time** | **Deployment Time** |
| Javassist | ✓ | ✓ | – |
| AspectWerkz | ✓ | ✓ | ✓ |
| Java and JRuby in Concert | ✓ | ✓ | ✓ |

**Table 2.** Summary - Time of Evolution.

We reason that the combination of Java and scripting language JRuby using the Bean Scripting Framework offers the most suitable options for enabling runtime program evolution. JRuby scripts are interpreted and enable all kinds of program changes until deployment time (see Table 2). Nevertheless, because of the overhead of script interpretation at program's runtime this approach is not appropriate to performance-critical use cases. Thus we need new approaches which enable runtime program evolution, as offered by interpreted languages, for compiled languages like Java and C++.

## References


1. L. Erlikh: Leveraging Legacy System Dollars for E-Business. IT Professional (2000)
2. J. Moad: Maintaining the competitive edge. DATAMATION (1990)
3. J. Gustavsson and U. Assmann: A Classification of Runtime Software Changes. In: Proceedings of the First International Workshop on Unanticipated Software Evolution (USE). (2002)
4. S. Chiba and M. Nishizawa: An Easy-to-Use Toolkit for Efficient Java Bytecode Translators. In: Proceedings of the second International Conference on Generative Programming and Component Engineering (GPCE). (2003)
5. S. Chiba: Load-Time Structural Reflection in Java. Lecture Notes in Computer Science (2000)
6. A. Vasseur: Dynamic AOP and Runtime Weaving for Java – How does AspectWerkz Address It? In: DAW: Dynamic Aspects Workshop. (2004)
7. J. Bonér: AspectWerkz – dynamic AOP for Java. Invited talk at 3rd International Conference on Aspect-Oriented Software Development (AOSD). (2004)

8. J. Bonér: 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)
9. G. Kiczales and J. Lamping and A. Mendhekar and C. Maeda, C.V. Lopes and J.-M. Loingtier and J. Irwin: Aspect-Oriented Programming. In: Proceedings of the European Conference on Object-Oriented Programming (ECOOP). (1997)
10. K. Czarnecki and U. Eisenecker: Generative Programming: Methods, Tools, and Applications. Addison-Wesley (2000)
11. D. Thomas and C. Fowler and A. Hunt: Programming Ruby: The Pragmatic Programmers' Guide, Second Edition. Pragmatic Bookshelf (2004)
12. V.J. Orlikowski: An Introduction to the Bean Scripting Framework. Presented at Conference ApacheCon US. (2002)

# Aspect-based Introspection and Change Analysis for Evolving Programs

Kevin Hoffman, Murali Krishna Ramanathan,
Patrick Eugster, and Suresh Jagannathan

Purdue University, 305 N. University St., West Lafayette, IN 47907, USA
{kjhoffma,rmk,peugster,suresh}@cs.purdue.edu

**Abstract.** Challenges arise in discovering, managing, and testing the impact of changes made to evolving software. These challenges are magnified in software systems that evolve while running, because the new functionality is piece-wise introduced into a live program with prior state produced by earlier component versions. If new functionality introduced into a live system induces bugs, it can be extremely difficult to analyze exactly which differences led to the incorrect behavior. In order to help programmers plan for evolution, understand the impact of specific evolutionary steps, and to diagnose evolution gone wrong, herein we propose combining the benefits of Aspect-Oriented Programming and reflection with impact analysis techniques from the OO and software engineering disciplines. We contribute a tool that assists with the deployment of new code to evolving software that gives insight as to precisely the behavioral changes between the new code and the code it is replacing within the running system. We conclude by considering the challenges of implementing and deploying such a tool and outline our plans for future research and evaluation.

## 1 Introduction

Evolving software systems are becoming ever more prevalant and important, especially considering the rise of highly-available service-oriented architectures. The rate of change of software systems has been accelerated by deployment of web-based applications, which are expected to run without interruption and with full reliability, notwithstanding their high rate of change and evolution.

The challenges caused by combining high-availability and rapid evolution in the same system are daunting. Techniques and tools to mitigate and overcome such challenges is a topic of intense research.

One such challenge linked to rapidly evolving software is the problem of determining the impact of specific changes to source code – how will these changes propagate and influence the rest of the system? This question is well-known and well-studied in the literature [1–4]. Techniques have emerged for answering this question with meaningful precision for software that does not change at runtime.

One such class of techniques is known as dynamic impact analysis, which strives to improve accuracy over the static analysis of source code by employing

code instrumentation and post-mortem analysis. Code is instrumented to generate trace data at runtime, both versions of the program are then run through one or more test cases, and finally these traces are compared and analyzed in order to infer the impact of the changes with as much precision as possible.

However, in their present state these techniques are difficult to apply to systems that evolve at runtime. While they can be used to understand how two versions differ across some common test cases, they cannot currently be used to discover the influence of new code on the live system where the new code is influenced by the behavior and state of the old code. Another challenge added by live evolution is that there is no beginning or end to the program where traces are typically endpointed, so it is difficult to define endpoints without scattering code or tags throughout the target program.

The contributions of this paper are as follows: First, we present how to combine aspect-oriented programming and reflection to transparently instrument programs while providing flexibility in how this instrumentation is applied and when it is active. Second, we enhance analysis techniques first introduced in [5] that allow the programmer to analyze the traces to achieve precise change analysis. Third, we evaluate the performance overhead of using our tool in long running (evolving) systems by measuring the performance impact both when tracing is and is not active.

## 2 Technique Overview

The general strategy of our approach is to first use aspects to instrument programs so that they generate "interesting" trace data (using the AspectJ 5 load time weaver), and then to analyze these traces (collected from multiple versions (or evolutionary steps) of the program) in order to determine which changes in the source code were responsible for the exhibited changes in program behavior.

We build on the insights and techniques of the Sieve [5] dynamic impact analysis system, but adapt it for object-oriented programming, change the nature of the trace data, and add flexibility and power stemming from implementing the instrumentation via AOP.

### 2.1 Trace Generation

The traces generated by the instrumentation approximate a *complete program trace* – a complete, sequential log of every low-level instruction executed (kept on a per thread basis) – in the same way that static analyses approximate the run-time behavior of a program. Greater accuracy over static analysis can be achieved because the traces have access to actual program state and control flow as the program executes. However, while capturing a complete program trace would be feasible (e.g. by modifying the JVM) and would preserve all information for the analysis, the size of these traces would quickly become challenging for long-running programs and would also cause analysis to be inefficient or even intractable.

To approximate a complete program trace, we use the following strategy: First, the program trace is endpointed so that only pieces of the execution trace are instrumented and recorded. For example, it could use the entrance and exit to a certain high level method as one set of endpoints. Each segment of the complete program trace captured between endpoints is called a trace segment.

For each trace segment several individual traces (representing an approximation of what happened over time) are generated over the lifetime of the trace segment. One individual trace, termed a trace point, is captured for each unique (method, truncated call stack) pair. The same trace point may be active many times over the course of the trace segment (as the same method with the same truncated call stack can be called many times).

As trace events (field accesses and method calls) are captured by the instrumentation, the code first determines the current trace point (creating the individual trace for that trace point if it is the first time entering that trace point) and then adds to the individual trace in the following way. First, information about the event is fed into a hashing function, and this hash value is used as a key into the individual trace (represented using an ordered hash table). If the individual trace does not contain the hash key, then the event is appended to the end of the trace along with a counter initialized to 0. It also includes relevant information about the event such as the source code location corresponding to that event. If the trace already contained the hash key, then the counter associated with that hash key is incremented.

In this way the trace approximates the sequential execution of events during all executions occurring in that trace point during the current trace segment. The use of event hashing and not storing more than one trace entry per hash allows traces to remain small, while the counter helps to model which branches were taken and how many loop iterations were executed.

### 2.2 Trace Analysis

The analysis of the trace data builds on the technique introduced by the Sieve [5] system.

To understand what has changed between different versions of the (running) program, the following procedure is used: First, one or more trace segments are captured on both the old and new versions of the running program. Next, the program identifies for each individual trace in the new version the corresponding individual trace in the old program. Currently the tool uses method name and signature to pair up traces, but more complex heuristics would be needed if method signatures or class names change between versions (as might be the case with an evolving system).

It then computes the minimal differences in execution between each new and old trace point by computing the longest common subsequence algorithm. Each individual trace is represented as a sequence of trace events, so the longest common subsequence between two traces represents the execution events that occurred in both versions of the program. The LCS can then be used to determine

the minimal set of execution events that either did not occur in the new version or were new to the new version.

The result of the analysis can be used by developers to understand the precise effects new component versions have upon program execution. If the input traces were complete program traces, then the results of the analysis would present precisely the points in both time and code at which the behavior of the new version diverged from what the old version would have done. In the trace approximations described above all temporal notions of computation are not recorded (aside from capturing the sequence in which execution events are first encountered within a given trace point), so the results of the analysis represent the precise points in the source code where the execution differed, but do not provide insight into the time of the divergence or how differences observed in different trace points can be ordered with respect to each other.

The traces can be enhanced in several ways to increase the precision of the analysis. First, the definition of a trace point can be extended from (method, truncated call stack) to include additional context, such as actual values of parameters or the number of times the method was called (allowing the capture of different traces for both the first N and last M executions for any given (method, truncated call stack)). Due to our use of aspects and reflection to implement trace generation, all of the above enhancements could be parameterized and adjusted at run-time – an important factor for long running, evolving systems. This parameterization could be used to increase the accuracy of the generated traces where it is anticipated that the increased accuracy is needed, while still avoiding full generation of the complete program trace (unless this is desired as indicated by the parameters).

### 2.3 Example

We present a small example in this section to illustrate how the tracing process works. Consider the two versions of some program shown in Figure 1. The only difference between the two versions is on line 9. For this example a trace point is identified solely by the method name, a location solely identified by a line number, the trace event data includes the values of the parameters, and the hash of a trace event is the hash of the method name and parameter values. We define one trace endpoint using the pointcut `execution(void C.m1(..))`.

Figure 2 shows the individual traces that would be generated for the trace segment that became active as part of the `C.m1` method execution. Each line in a trace consists of the following format: `hash.counter: line: event data`. This example exhibits many illustrative points: First, note that the execution between the two versions would first begin to differ at line 9 – in version A the call to `m3` is made on line 10 and then again on line 12, whereas in version B this call is only made on line 12. This behavior is exhibited by the traces. The trace for `C.m2` for version A shows `m3` was called before `println("p4")` and that it was called one more time sometime after that. The trace for `C.m2` for version B shows `m3` was called after `println("p4")` and that it was not called again after

```
        (Version A)                (Version B)
 1) class C {                1) class C {
 2)  void m1(){              2)  void m1(){
 3)    out.println("p1");    3)    out.println("p1");
 4)    m2(true);             4)    m2(true);
 5)    out.println("p2");    5)    out.println("p2");
 6)  }                       6)  }
 7)  void m2(boolean b){     7)  void m2(boolean b){
 8)    out.println("p3");    8)    out.println("p3");
 9)    if (b)                9)    if (!b)
10)      m3();              10)      m3();
11)    out.println("p4");   11)    out.println("p4");
12)    m3();                12)    m3();
13)    m4();                13)    m4();
14)  }                      14)  }
15)  void m3(){             15)  void m3(){
16)    out.println("p5");   16)    out.println("p5");
17)  }                      17)  }
18)  void m4(){}            18)  void m4(){}
19) }                       19) }
```

**Fig. 1.** Two versions of a program to be traced

```
======== C.m1.txt ======    ======== C.m1.txt ======
81a9db.1:  3: println p1    81a9db.1:  3: println p1
fc3511.1:  4: m2 true       fc3511.1:  4: m2 true
d8247b.1:  5: println p2    d8247b.1:  5: println p2

======== C.m2.txt ======    ======== C.m2.txt ======
56b1b1.1:  8: println p3    56b1b1.1:  8: println p3
5b0e30.2: 10: m3            7f4bdb.1: 11: println p4
7f4bdb.1: 11: println p4    5b0e30.1: 12: m3
42eea0.1: 13: m4            42eea0.1: 13: m4

======== C.m3.txt ======    ======== C.m3.txt ======
5533c9.2: 16: println p5    5533c9.1: 16: println p5
```

**Fig. 2.** Individual traces generated for example program in Figure 1

that. Second, when the LCS is generated between the two versions it would show that the execution diverged after line 8 and then converged back at line 13.

Adjusting the definition of trace point identity and the inputs used in the hash for events, and also allowing timestamps to be used allows programmers to explore changes with varying levels of accuracy and efficiency.

## 3 Implementation

This section describes the role of aspect-oriented programming and reflection in implementing our techniques.

### 3.1 Tool Overview

The tool is composed of a collection of instrumentation aspects that generate trace data and a Java analysis program to extract meaning from the trace data. To run a program using the instrumentation framework a script is used in the place of the `java` command that invokes the AspectJ 5 load-time weaver and configures the instrumentation aspects according to parameters passed on the command line. Parameters can fine tune exactly which parts of the program are instrumented and are also used to adjust the accuracy of trace data (allowing tradeoffs to be made between performance and accuracy).

### 3.2 Aspectized Instrumentation

The generation of trace data is composed of four components: call stack tracking, trace event tracking, trace generation, and trace persistence.

For each thread an object representing the current threads call stack is maintained by an aspect. This aspect intercepts all method calls within the scope of interest using before and after advice and adds to or removes from the thread's call stack object. Each call stack entry records the information required to identify the current trace point (method name, signature, and possibly argument values). An aspect is used instead of a Java exception object to obtain the current call stack both because our technique is faster (the call stack is updated only as it is changed, instead of rebuilding it every time it is needed) and it allows us to gather more information about the call stack (such as the values of parameters to methods).

Events that are of interest to the trace (field accesses and method calls) are modeled by using a pointcut. This pointcut uses an `if` primitive pointcut so that it does not match when there is no active trace segment. A `before` advice uses the above pointcut to execute trace generation logic whenever an event of interest is about to be executed.

The logic to generate the traces works in the following fashion. When an event of interest is about to execute it generates the current trace point using the current call stack for the current thread and the individual trace associated with that trace point is looked up (or created if it does not exist). The data for

the event is collected through reflection, and then a portion of this data is put through a hash function to generate a key. If the key already exists in the hash table that trace entry is 'revisited' to update its counter and timestamps. If the key does not exist, a new trace entry is added to the hash table (the hash table also records the order of the insertions to capture an approximate sequence of execution within the trace point).

When the end of the trace segment is reached the trace persistence component writes all trace data generated during the trace segment into a series of files and then deactivates tracing. One file is generated for each individual trace with the trace entries written in the order in which they were inserted into the hash table. Each trace entry written includes information relevant to the analysis, including the original source location responsible for that execution event. This persistence logic can easily be offloaded onto a background thread so as not to block the progress of the application thread.

### 3.3 Use of AspectJ Reflection

Reflection is used in the implementation primarily in two places. First, the call stack builder uses reflection features of `thisJoinPoint` to extract the source location of method calls. The call stack builder also uses the new reflection features in AspectJ 5 to determine whether or not the method is actually part of an aspect, which can be useful in the analysis stage to analyze the change in how aspects advised the base code between versions of a program.

Second, reflection is used in the advice that advises trace events to capture dynamic information about the event (e.g. values of method parameters) and also for supporting more complex definitions of trace points.

### 3.4 Trace Endpointing

Traditionally, the endpoints of a program trace are defined to be the start and end of a program. However, this is not useful for long running, evolving systems, so a means of defining endpoints with finer granularity is needed. Because aspects are used to perform the instrumentation, the full power of the AspectJ quantization model is available to the programmer in describing where trace segments should begin and end.

When the program is started, abstract pointcuts indicating the (possible) places where trace segments begin and end can be instantiated (by using an XML file which is used by the load-time weaver). The `cflow` and `cflowbelow` primitive pointcuts are then used with the above pointcuts to capture precisely the places where a trace segment begins and ends. The programmer can use dynamic primitive pointcuts in the endpoint pointcuts (e.g. `if`) so that tracing is only activated if certain boolean variables are set. More complex activation logic could be added if required (similar to how logging frameworks allow classes of log messages to be turned on or off) at the cost of higher performance overhead.

Using pointcuts to implement endpointing allows for great flexibility in deciding the scope and precision of tracing operations. The tool could also be

enhanced to allow for more than one trace segment per thread to be active at one time, although this would affect performance (whether or not the impact would be significant is yet to be determined).

### 3.5 Analysis of Trace Data

The analysis is implemented as a Java program that accepts as input the location of two directories, where each directory contains the results from one trace segment (for one particular version of the program). The analysis program also accepts a mapping that tells it how to correlate trace points in one version with the trace points in the other version (by default, this mapping is the identity function).

For each trace point it computes the longest common subsequence of trace events, using the hash value as a sequence element. The output is similar to the common `diff` tool in that it shows which trace events were exhibited in the old version but not the not version and vice versa. The tool assists the user by parsing the rest of the trace data associated with each trace event (source code information) and displays this decoded information to the user. In this way the user can see the source locations where the executions differed. If trace points were defined using a larger context and trace data contained temporal information, as discussed in section 2, then the resulting output would also give insight into *when* the new version first diverged from the old version in addition to information about *where* it did so.

## 4 Discussion

In this section we discuss how the techniques described in this paper relate to software evolution and also discuss the performance impact of using these tools in deployed systems.

### 4.1 Benefits for Software Evolution

The techniques presented in this paper are beneficial for evolving software systems in the following ways:

1. Java software can be transparently instrumented and the behavioral differences between different versions of objects (or similar objects) can be precisely observed.
2. The use of pointcuts to describe trace endpoints gives a great deal of flexibility in dynamically deciding when and what to trace through the use of dynamic pointcuts, while still reaping optimizations made by the AspectJ weaver. For example, before upgrading the evolving system tracing could be turned on for some built in test cases, and then afterwards the traces generated using the new classes could be compared to the old traces to understand exactly what went wrong (or right).

3. AspectJ pointcuts allow developers to selectively specify what should be instrumented (improving efficiency) while still being robust to enhancements and changes to the type system in the future.

We anticipate collaboration with others in this field to further learn how these ideas can be applied.

### 4.2 Performance Implications

In order for these techinques to be most useful for evolving software they must be deployed on production systems. Production systems are much more sensitive to effects on performance, so it is important to understand the performance costs associated with the instrumentation and tracing.

Even when the instrumentation and tracing logic is woven into a program, there are different levels of tracing activity with different corresponding performance costs. The first level is where the code has been instrumented, but there is no active trace segment in *any* thread. In this level the performance cost is due to the dynamic `if` pointcut designator that checks before every method call the value of a boolean flag (in this level, the flag is false, and further logic is not executed). The next level is where there are active trace segments on other threads, but not in the current thread. In this situation the global tracing flag is true (meaning tracing is active somewhere), but the dynamic `if` pointcut designator has to also then check a hash table to see whether or not there is an active trace segment for the current thread (in this case, there is not). Finally, the highest level is where there is an active trace segment for the current thread, so trace data is being generated for every method call (or field access if desired).

We have designed and run two benchmarks to quantitatively understand the performance costs associated with each level of activity. The first one is a microbenchmark that is a program consisting almost exclusively of nested method calls (there are a few loops and addition operations, but the primary computation of the program is calling methods). This microbenchmark was designed to exhibit the actual timing overheads associated with the differing levels of instrumentation. The second benchmark is the Java Linpack benchmark [6], which heavily exercises the floating point processor while also involving a large number of method calls (over 100 million).The goal of this benchmark is to see how the overall performance of a computationally intensive program (involving a large number of method calls) is affected after it has been instrumented. Both of these benchmarks represent worst-case or near worst case scenarios for the instrumentation; applications in which the actual number of method calls is relatively small (e.g. applications built on middleware) will be affected much less.

All benchmark tests were performed on a machine with a 2.33 Ghz Intel core 2 Duo procesor, 2 GB RAM, running OS X. The programs were compiled and run using Sun's Java 1.5.0.07 (Server Hotspot VM) and AspectJ 1.5.3. Before the tests were run all applications were closed and the system left until it entered

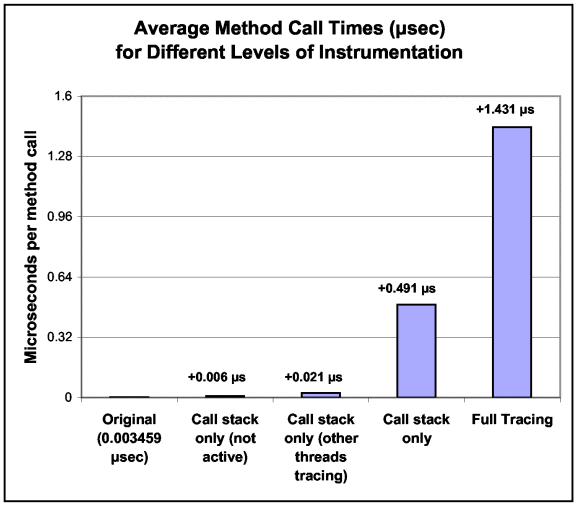



**Fig. 3.** Microbenchmark Results

an idle state. All timings computed do not include the time to load the Java VM or perform weaving. All numbers presented are averages over three runs.

The results of the method calls microbenchmark are presented in Figure 3 showing the average method call times. In this test over seven million method calls were made under each of the different scenarios and then the average time to execute a method call was computed by dividing the wall clock time by the exact number of calls made. (For some of the faster tests hundreds of millions of method calls were made to ensure a stable result.) The results for the Lynpack benchmark as presented in Figure 4 show how much slower instrumented programs ran compared to the original programs.

Applications in production would be in the *call stack only (not active)* level most of the time. At this level the overhead due to code instrumention is 6.1 nanoseconds per method call. For almost all applications this overhead should be negligible. Some computationally intensive programs that make heavy use of method calls in their innermost loops may experience a slowdown similar to Lynpack (9%).

Given that the duration of trace segments should be relatively small (being inactive most of the time – becoming active before and after components change), especially compared to the duration during which there are no active trace segments, the higher overhead present during tracing should not pose a significant obstacle to adoption. Additionally, if performance is an issue, load time parameters can cause the instrumentation to not be applied to certain sections of code such that there is no overhead for these regions of code.

Although our current system has much higher overhead than previously proposed techniques while tracing is active, performance when tracing is not active is much better. For example, path profiling [7] imposes on average an overhead

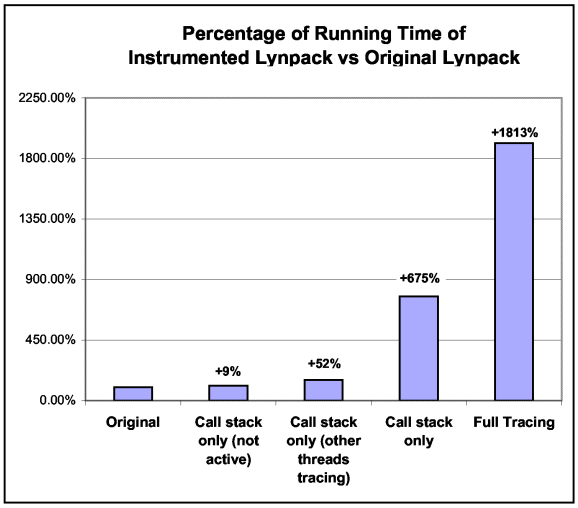



**Fig. 4.** Lynpack Benchmark Results

of 31% to 70% [8], whereas when tracing is not active our system imposes an average overhead of 9%. If tracing is only active for 1% of the time, then the average overhead of our approach becomes 27%. We also note that while path profiling captures execution counts for all acyclic execution paths in the program it does not record the relative order in which these paths were executed. Our tool is able to approximate this ordering and can thereby determine differences between traces with greater accuracy.

## 5 Conclusion

We have presented new techniques for understanding the changes in behavior of programs as they evolve (while running) over time. These techniques show promise to assist in planning, monitoring, and diagnosing evolving systems. There is certainly more to understand relating to the performance of the technique and how to improve it, but our results so far indicate that the overhead is an acceptable burden for most applications. Future work involves further developing and refining the methods for evolving software, applying the technique to one or more case studies, and developing tool and IDE support for easy integration into the development process.

## References

1. J. Law and G. Rothermel, "Whole program path-based dynamic impact analysis," in *ICSE 03*, 2003, pp. 308–318.
2. T. Apiwattanapong, A. Orso, and M. J. Harrold, "Efficient and precise dynamic impact analysis using execute-after sequences," in *ICSE 05*, 2005, pp. 432–441.

3. A. Orso, T. Apiwattanapong, and M. J. Harrold, "Leveraging field data for impact analysis and regression testing," in *ESEC/FSE-11*, 2003, pp. 128–137.
4. X. Ren, F. Shah, F. Tip, B. G. Ryder, and O. Chesley, "Chianti: a tool for change impact analysis of java programs," in *OOPSLA 04*, 2004, pp. 432–448.
5. M. K. Ramanathan, A. Grama, and S. Jagannathan, "Sieve: A tool for automatically detecting variations across program versions," in *ASE 2006*, September 2006.
6. J. Dongarra, R. Wade, and P. McMahan, "Linpack benchmark – Java version," 2007, http://www.netlib.org/benchmark/linpackjava/.
7. T. Ball and J. R. Larus, "Efficient path profiling," in *MICRO*, December 1996, pp. 46–57.
8. G. Ammons, T. Ball, and J. R. Larus, "Exploiting hardware performance counters with flow and context sensitive profiling," *SIGPLAN Notices*, vol. 32, no. 5, pp. 85–96, 1997.

# Morphing Software for Easier Evolution

Shan Shan Huang[1,2], and Yannis Smaragdakis[2]

[1] Georgia Institute of Technology, College of Computing, ssh@cc.gatech.edu
[2] University of Oregon, Department of Computer and Information Sciences
yannis@cs.uoregon.edu

## 1 Introduction

One of the biggest challenges in software evolution is maintaining the relationships between existing program structures. Changing a program component (e.g., a class, interface, or method) typically requires changes in multiple other components whose structure or meaning depend on the changed one. The root cause of the problem is redundancy due to lack of expressiveness in programming languages: Extra dependencies exist only because there is no easy way to model one program component after another, so that changes to the latter are automatically reflected in the former. For example, in the Enterprise Java Bean (EJB) standard, local and remote stub interfaces must mirror the bean class structure exactly. A change in the bean interface must be propagated to the stub interfaces, as well. Tools and methods have been developed to support writing code that is immune to changes in program structure (e.g., [10, 11]). But these tools either require separate declarations of a program's structural properties (e.g., class dictionaries in [11]), or use potentially unsafe runtime reflection [10]. Furthermore, these tools focus on adapting code, and not the static structure of a class or interface, to evolving program structure.[1]

Another obstacle in software evolution is the extensibility of software components, particularly when source code is unavailable. Aspect Oriented Programming (AOP) [9] and its flagship tools, such as AspectJ [8] provide a solution approach. AspectJ allows a programmer to extend a software component by specifying extra code to be executed, or even change the component's original semantics entirely by circumventing the execution of original code, and provide new code to execute in its place. AspectJ is a powerful tool, but often has to sacrifice either discipline or expressiveness. For example, AspectJ aspects are strongly tied to the components they apply to—there is no notion of typechecking an aspect separately from the application where it is used. This means that *generic* AspectJ aspects (i.e., aspects that are specified so that they can be later applied to multiple, but yet unknown, components) are limited in what they can express. For example, AspectJ cannot express intercepting all calls to the methods of one class, and forwarding them to methods of another class, using the intercepted arguments: the aspect to do so needs to be custom-written

[1] Tools have been developed to specifically target generating EJB stubs [5] so that consistency between the bean class and its stubs is managed automatically. But this is a solution to one specific problem, and not generally applicable.

for the specific classes, methods, and arguments it will affect. AspectJ also does not provide explicit means of controlling aspect application. For example, the order of aspect composition may affect behavior in ways unanticipated by the developers.

With these two obstacles in mind, we recently introduced a language feature that we call "morphing" [7]. Morphing supports a powerful technique for software evolution, and it overcomes many of the shortcomings of existing solutions. We discuss morphing through MJ—a reference language that demonstrates what we consider the desired expressiveness and safety features of an advanced morphing language. MJ morphing can express highly general object-oriented components (i.e., generic classes) whose exact members are not known until the component is parameterized with concrete types. For a simple example, consider the following MJ class, implementing a standard "logging" extension:

```
class MethodLogger<class X> extends X {
  <Y*>[meth]for(public int meth (Y) : X.methods)
  int meth (Y a) {
    int i = super.meth(a);
    System.out.println("Returned: " + i);
    return i;
  }
}
```

MJ allows class `MethodLogger` to be declared as a subclass of its type parameter, `X`. The body of `MethodLogger` is defined by static iteration (using the `for` statement—the central morphing keyword) over all methods of `X` that match the pattern `public int meth(Y)`. `Y` and `meth` are pattern variables, matching any type and method name, respectively. Additionally, the `*` symbol following the declaration of `Y` indicates that `Y` matches any number of types (including zero). That is, the pattern matches all `public` methods that return `int`. The pattern variables are used in the declaration of `MethodLogger`'s methods: for each method of the type parameter `X`, `MethodLogger` declares a method with the same name and signature. (This does not have to be the case, as shown later.) Thus, the exact methods of class `MethodLogger` are not determined until it is type-instantiated. For instance, `MethodLogger<java.io.File>` has methods `compareTo` and `hashCode`: the only `int`-returning methods of `java.io.File` and its superclasses.

MJ morphing supports disciplined software evolution in the following ways:

- MJ allows a class's members to mirror those in another class, e.g., one of its type parameters. The structure of an MJ generic class adapts automatically to the evolving interfaces of its type parameters. (MJ's `for` construct can also be used in declaring statements. Thus, MJ can be used to adapt code to changing program structures, as well.)
- MJ generic classes support modular type checking—a generic class is type-checked independently of its type-instantiations, and errors are detected if they can occur with *any* possible type parameter. This is an invaluable property for generic code: it prevents errors that only appear for some type parameters, which the author of the generic class may not have predicted.

– MJ allows programmers to use both a "transformed" version of a class and the original class at will. For example, a programmer may refer to both the original `java.io.File` and its logged version `MethodLogger<java.io.File>` within the same piece of code.
– Order of composition is explicit in MJ: given two MJ classes, adding two pieces of functionality, e.g., logging through `MethodLogger` and synchronization through `MethodSynchronizer`, applying logger before synchronizer is simply, `MethodSynchronizer<MethodLogger<java.io.File>>`.

In addition to the above properties, MJ differs from existing "reflective" program pattern matching and transformation tools [2–4, 12] by making reflective transformation functionality a natural extension of Java generics. For instance, our above example class `MethodLogger` appears to the programmer as a regular class, rather than as a separate kind of entity, such as a "transformation". Using a generic class is a matter of simple type-instantiation, which produces a regular Java class, such as `MethodLogger<java.io.File>`.

We next elaborate on how MJ morphing supports adaptation to evolving program structures and its modular type safety properties through examples.

## 2 Adapting Structure to Changing Structures

The structure of an MJ generic class can evolve consistently with the structure of its type parameter. This property allows writing feature extensions and adaptations that are inherently evolvable. We illustrate the benefits of this property using a common design pattern: wrapper [6]. A wrapper class declares the exact same methods as the class it wraps. It delegates each method call to the wrapped class, adding functionality before or after the delegation. The `MethodLogger` class of the previous section is a classic wrapper.

For a real world exposition of the wrapper pattern, consider the class `java.util.Collections`, a utility class provided by the Java Collections Framework (JCF) [1]—the standard Java data structures library. `java.util.Collections` provides a number of methods that take a particular kind of data structure, and return that data structure enhanced with some additional functionality. For example, the method `synchronizedCollection(Collection<E> c)` takes a `Collection c`, and returns a synchronized version of that collection. The implementation of this method returns an instance of the wrapper class `SynchronizedCollection<E>`. For each method in the `Collection` interface, `SynchronizedCollection` defines a method with the exact same signature. The bodies for these methods all first synchronize on a mutex, and then delegate the call to the underlying `Collection` object. `java.util.Collections` offers similar methods that return synchronized versions of other kinds of data structures: `synchronizedSet(Set<E>)`, `synchronizedSortedSet(SortedSet<E>)`, `synchronizedList(List<E>)`, etc. Each of these methods, in turn, requires its own wrapper class definition: `SynchronizedSet<E>`, `SynchronizedSortedSet<E>`, `SynchronizedList<E>`, etc.

Note that these wrapper class definitions are tightly coupled with the interface of the data structures they are wrapping. If the interface of `Collection<E>` changes (e.g., a new method is added, or an existing method is now taking an extra argument), the class `SynchronizedCollection<E>` needs to be redefined, as well. Note also the redundancy at both the class and the method level. At the class level, all `Synchronized*` wrapper classes have the same structure, yet one wrapper class needs to be defined for each data structure to be synchronized. If the need to synchronize a new data structure arises, then a new wrapper class needs to be defined. At the method level, all methods within a `Synchronized*` class share a highly regular structure: first synchronize on a mutex, then delegate the call. Yet they still need to be defined individually. Java provides no way to modularly impose this structure on all methods.

With MJ, however, we can remove such dependency and redundancy, with a single MJ class:[2]

```
public class SynchronizeMe<interface X> implements X {
  X x;
  Object mutex;
  public SynchronizeMe(X x) { this.x = x; mutex = this; }

  //For each non-void method in X, declare the following:
  <R,A*>[m] for(public R m(A) : X.methods)
  public R m (A a) { synchronized(mutex) { return x.m(a); } }

  // Similarly for each void-returning method in X
  <A*>[m] for(public void m(A) : X.methods)
  public void m(A a) { sychronized(mutex) { x.m(a); } }
}
```

`SynchronizeMe` decouples the synchronization feature from the interfaces needing such a feature. Its definition adapts effortlessly to the interfaces it wraps. There is no need to modify `SynchronizeMe` when the underlying wrapped interface changes. `SynchronizeMe` can be instantiated with any interface to provide a synchronized implementation of that interface, thus replacing all `Synchronized*` wrapper classes. Additionally, `SynchronizedMe` removes method-level redundancy using a static iteration block to impose the same structure on all methods.

The full version of the above MJ class consists of less than 50 lines of code, replacing more than 600 lines of code in the JCF. Similar simplifications can be obtained for other nested classes in `java.util.Collections`, which account in total for some 2000 lines of code in the original JCF implementation: `UnmodifiableSet`, `UnmodifiableList`, `UnmodifiableMap`, etc. are replaced by a single morphed class, and the same is done for `CheckedSet`, `CheckedList`, `CheckedMap`, etc.

While adding logging or synchronization functionality is doable with AOP tools such as AspectJ, MJ allows the "morphing" of a wrapper class in much

[2] This example implementation uses `this` as the mutex. A more flexible implementation could provide a constructor that allows programmers to choose their own mutex. In fact, this is the strategy adopted in the JCF.

more interesting ways. For example, one can declare a MJ class `MakeLists<C>` such that, for each single-argument method of its type parameter `C`, `MakeLists<C>` has a method of the same name, but takes a *list* of the original argument type, invokes the original method on each element of the list, and returns a list of the original method's return values:

```
class MakeLists<C> {
  C c; // wrapped object.
  ... // constructor initializing c.

  <R,A>[m] for(R m (A) : C.methods)
  List<R> m (List<A> la) {
    ArrayList<R> rlist = new ArrayList<R>();
    if ( la != null )
      for ( A a : la ) { rlist.add(c.m(a)); }
    return rlist;
  }
}
```

This transformation is not expressible using AspectJ. Consider how this functionality can be implemented using plain Java: a `MakeListsSomeType` class would have to be declared for every `SomeType` that we want to have this extension for. Additionally, if the structure of `SomeType` changes, e.g., a new single-argument method is added, or an existing method changes its argument or return types, `MakeListsSomeType` would need to be modified to reflect those changes, as well. In contrast, the MJ generic class `MakeLists<C>` works for any class `C` without advanced planning of which types this extension can be added to. It also morphs with the structure of each `C`, without further programmer intervention.

## 3 Modular Type Checking

For an example of modular type checking, consider the following "buggy" class:

```
class CallWithMax<class X> extends X {
  <Y>[meth]for(public int meth (Y) : X.methods)
  int meth(Y a1, Y a2) {
    if (a1.compareTo(a2) > 0) return super.meth(a1);
    else return super.meth(a2);
  }
}
```

The intent is that class `CallWithMax<C>`, for some `C`, imitates the interface of `C` for all single-argument methods that return `int`, yet adds an extra formal parameter to each method. The corresponding method of `C` is then called with the greater of the two arguments passed to `CallWithMax<C>`. It is easy to define, use, and deploy such a generic transformation without realizing that it is not always valid: not all types `Y` will support the `compareTo` method. MJ detects such errors when compiling the above code, independently of instantiation. In

this case, the fix is to strengthen the pattern with the constraint `<Y extends Comparable<Y>>`:

```
<Y extends Comparable<Y>>[meth]for(public int meth (Y) : X.methods)
```

Additionally, the above code has an even more insidious error. The generated methods in `CallWithMax<C>` are not guaranteed to correctly override the methods in its superclass, `C`. For instance, if `C` contains two methods, `int foo(int)` and `String foo(int,int)`, then the latter will be improperly overridden by the generated method `int foo(int,int)` in `CallWithMax<C>` (which has the same argument types but an incompatible return type). MJ statically catches this error.

## 4 A Comparison to AOP

Morphing can be used to address some of the same issues as AOP. To be sure, morphing only relates to a small but central part of AOP functionality: aspect advice of structural program features, such as method before-, after-, and around-advice. Particularly, the logging and synchronization examples shown in previous sections are frequent use cases for AOP languages. Thus, it is worth delineating the similarities and distinct differences between morphing and AOP. We next compare MJ, the only reference morphing language, to AspectJ [8], a representative AOP tool for Java.

### 4.1 How Functionality is Added

Both MJ and AspectJ allow functionalities that cross-cut multiple class definitions to be defined in a modular way. For example, the method logging functionality can be defined in one MJ class, `MethodLogger`. However, the way such functionalities are added into a base class definition is one of the main differences between MJ and AspectJ. With MJ, cross-cutting functionality is added "into" a base class through explicit parameterization of the morphing class. The new functionality only exists in the parameterized morphing class, whereas the definition of the base class itself does not change. For example, parameterized morphing class `MethodLogger<java.io.File>` has the functionality that all `int`-returning methods are logged. However, the definition of `java.io.File` itself remains unchanged. In AspectJ, an aspect definition states the classes a functionality should be added to.[3] In this way, the new functionality is weaved with the code of the original class. The program cannot simultaneously use the separate notions of "original class" and "class with the cross-cutting functionality". One way to view the semantics of AspectJ is as changing the original class's definition. For instance, given an AspectJ aspect that adds logging code to each `int`-returning method of `java.io.File`, the class `java.io.File` itself can be thought of as changed after aspect application. Indeed this also happens to be the way current AspectJ compilers implement the semantics of weaving.

[3] In the case of generic aspects in AspectJ 5, the affected classes can be specified through parameterization of the aspect.

We view explicit parameterization in MJ as an important feature for two reasons. First, the ability to leave the original class definition untouched is an important one. For example, a programmer should be able to use both synchronized and unsynchronized versions of a data structure in the same program, depending on his needs. This is indeed the case with the MJ class `SynchronizeMe`, shown in Section 2. With AspectJ, however, the programmer must choose one or the other. AOP purists may hold the view that cross-cutting functionality enhancements, by definition, should be applied to all classes that need them. But as shown through the synchronization example, this is a very rigid requirement. Furthermore, if indeed all instantiations of a class should have a particular cross-cutting functionality, it should be possible to extend MJ with a global search-and-replace tool, replacing all instances of a class with the explicitly parameterized version of a morphing class. This is part of future work, however. Our current research focuses on the fundamental core of morphing, and we expect that usability enhancements will come later.

Secondly, explicit parameterization provides a way to clearly document and control the semantics of a program. This is particularly true when multiple, separately-defined functionality enhancements need to be added to a class. One of the much researched topics in AOP is aspect interaction. When one defines an aspect in AspectJ, there is no good way to specify the order of its application relative to all other aspects, some of which may be unknown to the aspect developer. Furthermore, an addition of another aspect unknown to the developer can change the program semantics in unexpected ways. This is an undesirable characteristic in terms of modularity. MJ, on the other hand, allows explicit control of functionality addition through instantiation order. The type-instantiation order gives a clear meaning as to where and how functionality is added.

### 4.2 Modular Type Safety and Trade-offs in Expressiveness

The other main difference between MJ and AspectJ is MJ's guarantee of modular type safety. In order to make such guarantees, we limited our attention to some specific features instead of adding maximum expressiveness to the language. Pattern matching in MJ is simple and high level by design. A programmer can only inspect classes at the level of method and field signatures: MJ pattern matching applies to reflection-level structural elements of a type. In contrast, AspectJ allows a programmer to match on a program's dynamic execution characteristics, using keywords such as `cflow` (for control flow) and `cflowbelow` in pointcuts.

Though MJ limits its pattern matching to the type signature level, it does allow matching using subtype-based semantic conditions, in contrast to the purely syntactic matching of signatures AspectJ offers. For instance, using pattern-matching type variables, MJ allows one to express a pattern that matches all methods that return *some* subtype of `java.lang.Comparable`. This is a pattern not expressible through AspectJ. The combination of pattern matching and the static `for` construct in MJ provides a controlled but useful kind of programmability in defining where a certain functionality should be introduced.

Although we make comparisions only to AspectJ, the arguments in this section generalize to other AOP tools, as well. AspectJ is representative in the way it applies aspects, and is perhaps the most expressive aspect language today.

## 5 Future Work: When Plain Morphing Isn't Enough

We have demonstrated through examples that morphing with MJ is particularly useful for defining classes whose structure must mirror the structure of another class or interface. However, there are some useful cases that MJ, with just pattern matching and static iteration, cannot express in a modularly type safe way. For example, it is often the case that each field in a class has its own getter and setter methods. These getter and setter methods must be manually defined by the developer. This seems to be the perfect use-case for morphing. We ought to be able to define a morphing class that defines a getter and a setter method for each field in its type parameter:

```
public class AddGetterSetter<class C> extends C {
  <F>[f] for( F f : C.fields )
  public F get#f () { return f; }

  <F>[f] for( F f : C.fields )
  public void set#f ( F newF ) { f = newF; }
}
```

Note that `AddGetterSetter` uses the MJ language construct `#`, which concatenates a constant prefix to a pattern matching name variable. For each field `SomeField` in `C`, `AddGetterSetter<C>` declares a getter method, `getSomeField`, and a setter method, `setSomeField`. However, this class is not modularly type safe. We cannot establish that the names of the methods being declared, `getSomeField` and `setSomeField`, whatever `SomeField` may be, do not conflict with methods in the superclass `C`. For example, `AddGetterSetter<Foo>` would not be a well-typed class if `Foo` is defined as follows:

```
public class Foo {
  int up;

  public int setup ( int i ) { ... }
}
```

`AddGetterSetter<Foo>` contains the method `void setup(int)`, which incorrectly overrides the method `int setup(int)` in its superclass `Foo`: `setup` in `AddGetterSetter<Foo>` has the same argument type as `setup` in `Foo`, but a non-covariant return type.

One possible way to express such functionality while keeping modular type safety is to put an additional condition on each element in the static iteration. We need to be able to express that we only want to iterate over those fields `f` of `C` for which a `set#f` (or `get#f`) method does not already exist in `C` itself. We

are currently considering an extension to the pattern language. For example, we could change the static iteration block defining the setter method to:

```
<F>[f] for( F f : C.fields;
            no set#f(F) : C.methods )
public void set#f ( F newF ) { f = newF; }
```

Note the extra clause: `no set#f(F) : C.methods`. This clause serves as the extra conditional needed to ensure that no conflicting method already exists in `C`.

This addition to MJ's pattern matching language might seem simple and innocuous at first, but the combination of iteration along with conditionals on types and their structures can easily yield undecidable type systems. We must take care to restrict the conditionals so that our type checker can still provide useful feedback to the programmers. Striking this balance between the need for this additional expressiveness and the tractability of the type system is our future focus.

## 6 Conclusion

Overall, we consider MJ and the idea of morphing to be a significant step forward in supporting software evolution. Morphing can be viewed as an aspect-oriented technique, allowing the extension and adaptation of existing components, and enabling a single enhancement to affect multiple code sites (e.g., all methods of a class, regardless of name). Yet morphing can perhaps be seen as a bridge between AOP and generic programming. Morphing allows expressing classes whose structure evolve consistently with the structures they mirror. Morphing strives for smooth integration in the programming language, all the way down to modular type checking. Thus, reasoning about morphed classes is possible, unlike reasoning about and type checking of generic aspects, which can typically only be done after their application to a specific code base. Morphing does not introduce functionality to unsuspecting code. Instead, it ensures that any extension is under the full control of the programmer. The result of morphing is a new class or interface, which the programmer is free to integrate in the application at will. We thus view morphing as an exciting new direction in supporting software evolution.

## References


1. *Java Collections Framework Web site, http://java.sun.com/j2se/1.4.2/docs/guide/collections/.* Accessed Apr. 2007.
2. J. Bachrach and K. Playford. The Java syntactic extender (JSE). In *Proc. of the 16th ACM SIGPLAN conference on Object Oriented Programming, Systems, Languages, and Applications*, pages 31–42, Tampa Bay, FL, USA, 2001. ACM Press.
3. J. Baker and W. C. Hsieh. Maya: multiple-dispatch syntax extension in Java. In *Proc. of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, pages 270–281, Berlin, Germany, 2002. ACM Press.

4. D. Batory, B. Lofaso, and Y. Smaragdakis. JTS: tools for implementing domain-specific languages. In *Proc. Fifth Intl. Conf. on Software Reuse*, pages 143–153, Victoria, BC, Canada, 1998. IEEE.
5. B. Burke et al. *JBoss AOP Web site, http://labs.jboss.com/portal/jbossaop*. Accessed Apr. 2007.
6. E. Gamma, R. Helm, and R. Johnson. *Design Patterns. Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional Computing Series. Addison-Wesley, 1995.
7. S. S. Huang, D. Zook, and Y. Smaragdakis. Morphing: Safely shaping a class in the image of others. In E. Ernst, editor, *To Appear: Proc. of the European Conf. on Object-Oriented Programming (ECOOP)*, LNCS. Springer-Verlag, July 2007.
8. G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. In *Proc. of the 15th European Conf. on Object-Oriented Programming*, pages 327–353, London, UK, 2001. Springer-Verlag.
9. G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In M. Akşit and S. Matsuoka, editors, *Proc. of the 11th European Conf. on Object-Oriented Programming*, volume 1241, pages 220–242. Springer-Verlag, Berlin, Heidelberg, and New York, 1997.
10. J. Palsberg and C. B. Jay. The essence of the visitor pattern. In *Proc. 22nd IEEE Intl. Computer Software and Applications Conf., COMPSAC*, pages 9–15, 19–21 1998.
11. T. Skotiniotis, J. Palm, and K. J. Lieberherr. Demeter interfaces: Adaptive programming without surprises. In *European Conference on Object-Oriented Programming*, pages 477–500, Nantes, France, 2006. Springer Verlag Lecture Notes.
12. E. Visser. Program transformation with Stratego/XT: Rules, strategies, tools, and systems in Stratego/XT 0.9. In C. Lengauer, D. Batory, C. Consel, and M. Odersky, editors, *Domain-Specific Program Generation*, pages 216–238. Springer-Verlag, 2004. LNCS 3016.

# AOP vs Software Evolution: a Score in Favor of the Blueprint.

Walter Cazzola[1] and Sonia Pini[2]

[1] Department of Informatics and Communication,
Università degli Studi di Milano, Italy
cazzola@dico.unimi.it
[2] Department of Informatics and Computer Science
Università degli Studi di Genova, Italy
pini@disi.unige.it

**Abstract.** All software systems are subject to evolution, independently by the developing technique. Aspect oriented software in addition to separate the different concerns during the software development, must be "not *fragile*" against software evolution. Otherwise, the benefit of disentangling the code will be burred by the extra complication in maintaining the code.
To obtain this goal, the aspect-oriented languages/tools must evolve, they have to be less coupled to the base program. In the last years, a few attempts have been proposed, the Blueprint is our proposal based on behavioral patterns.
In this paper we test the robustness of the Blueprint aspect-oriented language against software evolution. Keywords: AOP, Software Evolution, Fragile Pointcut Problem.

## 1 Introduction

All software systems are subject to evolution, they evolve over time as new requirements and functionality emerge, or adaption and extensions are to be made. Studies pointed out that up to 80% [13] of the system lifetime will be spent on maintenance and evolution activities. A program that is useful in a real-world environment necessarily must change or become progressively less useful in that environment [12].

Aspect-oriented programming has been designed with the intention of providing a better separation of concerns by modularizing concerns that would otherwise be tangled and scattered across the other concerns. This would made the software more maintainable, evolvable and understandable. Paradoxically, the major aspect-oriented techniques instead of improving software maintainability seem to restrict the evolvability of that software, as highlighted in [21]. This problem is due to the so called "fragile pointcut problem" [11].

Pointcuts are deemed fragile when seemingly innocent changes to the base program, such as renaming or relocating a method, break a pointcut such that it no longer captures the join points it is intended to capture. When code is added to a program and introduces new join points in the program, pointcuts are similarly considered fragile in the case some of these new join points should be captured by the pointcut but it fails to do so.

It implies that all pointcuts of each aspect need to be checked and possibly revised whenever the base program evolves, since they often break when the base program is re-factored. All pointcuts referring to the base program need to be examined both after an evolution and after a re-factoring, because they capture a set of join points based on some structural and syntactical properties, any change to the structure or syntax of the base program can alter the set of join points that is captured by the pointcuts. This problem exists both if a programmer uses wildcards and not.

In practice, the pointcut fragility derives form the *dependency* on the program syntax and the *coupling* between aspects and program [1, 7, 8, 20]. The fragile pointcut problem is a serious inhibitor to evolution of aspect-oriented programs. The critical element of the past generation of AO tool is the necessity to specify program elements names and the impossibility to select elements without using naming convention or regular expressions. In short, they have a *linguistic approach*, so aspect writers need to be completely aware of the base-code details and evolution, so each aspect become strictly bound to the application on who has been designed. To obtain a less fragile approach, it is necessary a new generation of aspect-oriented languages/tools, less coupled to the base program.

In the last few years, the main goal of the new generation of AO approaches is to get a more semantic join point selection mechanism to avoid the fragile pointcut problem. Some approaches are: the pointcut delta analysis [11, 20], the approach of Kellens et al. described in [8], the join point model proposed by Mohd Ali and Rashid in [14], the functional query language proposed by Eichberg et al. in [4], the graphical approach to model pointcuts described by Stein et al. in [19] and so on. In [3, 16] we have defined a new (visual) model-based join point selection mechanism. We tackle the fragile pointcut problem by eliminating the intimate dependency of pointcut definitions on the base program and by using a high level description of the program behavior during the join point selection.

In this paper we want to prove the robustness of the Blueprint against the evolution. The rest of the paper is organized as follows: in section 2 we shortly overview the Blueprint approach and elements, in section 3 we introduce our test case for the evolution, finally, in section 4 and in section 5 we face some related works and draw out our conclusions.

## 2 The Blueprint Language

The Blueprint framework is based on our previous work [2] and it is completely detailed in [16]. Moreover, a working prototype of the framework has been developed in Java.

The main goal of the Blueprint language is to overcome many problems of the past generation of AO language [1, 9, 11], such as, the granularity of the join point, the fragile pointcut problem, the semantic selection and so on.

The Blueprint is based on the idea that the description of the application behavior cannot be strictly coupled to the application syntactic details. It permits a *loose approach* to the description of the application behavior. This means that the aspect programmer can use different levels of detail during the description of a single join point blueprint

by using any possible combinations of *loose* and *tight elements*. This approach permits to describe a well identified behavior tightly coupled to the application code by specifying the names of the involved elements, and a less known behavior by using *meta-information* to abstract from the real application code.

The Blueprint is a novel aspect-oriented framework, its join point selection mechanism allows the selection of the join points *abstracting* from implementation details, name conventions and any other source code dependency. In particular the aspect programmer can select the interested join points by describing their supposed location in the application through UML-like descriptions (basically, activity diagrams) representing computational patterns on the application behavior; these descriptions are called blueprints. The blueprints are just patterns on the application behavior, i.e., they are not derived from the system design information but express properties on them. In other words, we adopt a sort of enriched UML diagrams to describe the application control flows or computational properties and to locate the join points inside these contexts.

The Blueprint uses a static quantification, i.e., it allows quantification over the abstract syntax tree of the program, hereby queries such as "print the value of a variable used in a loop test condition and modified in the loop body" are possible. This kind of quantification requires to access the source code of the application, because we need to obtain a parsed version of the underlying program, to run the transformation rules realizing the quantified aspects over that abstract syntax tree. The Blueprint language can be used on the bytecode as well since it can be univocally decompiled (modulo semantic equivalence) by apposite tools, e.g., by Jode.

In our approach, we do not need to use position qualifiers such as **`before`** and **`after`** advices to indicate where to insert the concern inside the base code, since we describe the context we are looking for, we can either locate the join points exactly where we want to insert the new code or, to highlight the portion of behavior we want to replace.

The Blueprint framework recalls the AspectJ terminology but some terms are used with a slightly different meaning. The Blueprint *join points* are *hooks where code may be added* rather than *well defined points in the execution of a program* as in AspectJ. In other words, the AspectJ join points are based on the idea that "when something happens, then something gets executed[3]". In this view a join point consists of things like method and constructor calls, method and constructor executions, object instantiations, field references and so on. While the Blueprint approach is that "join points can occur in any part of the code", this view permits of changing a single line of code. We use a *statement-level granularity* for the join point model whereas AspectJ uses an *operational level granularity* for the join point model. In particular we consider two different kinds of join points: the *local join points* that represent points in the application behavior where to insert the code of the concern, and *region join points* that represent portion of the application behavior that must be replaced by the code of the concern. The pointcuts are obtained as composition/enumeration of the join points selected by the join point blueprints rather than as the logical composition of queries on the application code. While *introduction*s and *advice*s keep their usual meaning.

To complete the picture of the situation, we have introduced some new concepts: *join point blueprint* and, *blueprint space*. The former is *a template (a blueprint) on*

[3] http://www.eclipse.org/aspectj/doc/released/progguide/index.html

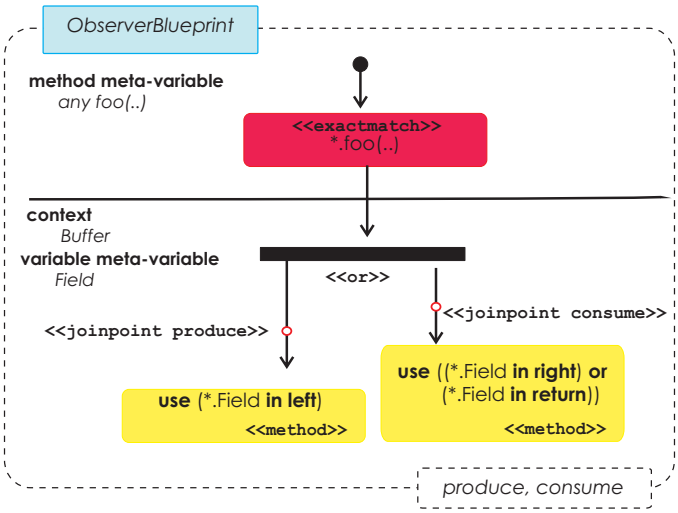



**Fig. 1.** Sample Join Point Blueprint.

*the application behavior identifying the join points in their context*; these blueprints describe where the local and region join points should be located in the application behavior. The blueprint does not completely describe the computational flow but only the portions relevant for selecting the join points. The latter is the set of all join points blueprints defined on the same application.

The key element of our approach are the *join point blueprints*, they graphically depicts where a join point (both local and region) should be in the application behavior. They look like an activity diagrams.

The diagram contextualizes the join point location by describing some crucial *events* that should occur close to the join point, these events will be used to recognize the join point. The frame gives some ancillary information, such as the blueprint name (at the top left corner), the join points name exposed by the blueprint (at the bottom right corner) and some meta-info used by the weaver to parametrize the context and to get values from the join point. The listed join points are the only exposed to the pointcut specification. The join point position is denoted by the ≪`joinpoint name`≫ stereotype (or by the couple ≪`startjoinpoint name`≫ and ≪`endjoinpoint name`≫ for the region join points).

To get a more expressive language and less coupled to the code syntax and structure, and by assuming that the aspect programmer does not necessarily know the implementation details of the code, we introduced the *meta-information* section inside the blueprint diagram. The meta-information is the textual portion of the blueprint, that allows the programmer of describing the context at the desired implementation detail, i.e., either by using real variable, method and field names or less precise information. For example, if the programmer does know the method name, but he knows its parameter types, he can use this information, in the meta-information section, by declaring a new

*method meta-variable* with a fantasy name,and indicating the right number and type of its parameters, and finally by using this new meta-variable in the blueprint to describe the sought behavior. The meta-information elements will be unified to variable names used into the application during the *weaving phase*.

Figure 1 shows a very simple join point blueprint that absolutely do not recall the whole expressivity allowed by the formalism. For a detailed and exhaustive description, please, refer to [16] chapter 4.

## 3 A Test-Bed for the Blueprint Robustness

So far, we have used the Blueprint to locate simple join points in toy-applications, like showed in [3,16], with few lines of code. Now, it is fundamental to test the robustness of the Blueprint language against the evolution by using a real application with thousand lines of code and a long time life cycle with many adaptation steps. To this goal, we adopted the Health Watcher (HW) system[4] developed at UPE and introduced by Soares et al. in [18].

HW is a typical web-based application that manages health-related information. It includes a variety of crosscutting concerns, such as concurrency, distribution, persistence, and so on. The same application has been previously used as test-bed by the Lancaster University (in [6]), that has created and compared one object-oriented (by using Java) and two aspect-oriented (by using AspectJ and CaesarJ) implementations of HW. Moreover, they introduced nine steps of evolution to the initial application.

### 3.1 HW Evolution

We consider the HW evolution from version 8 to version 9, which adds new functionalities to the application. Version 9 adds the following functionalities: insertion of new health unit, insertion of new medical speciality, insertion of new symptoms, searching for symptoms, updating of a symptom, searching for speciality by code, updating of medical speciality, and insertion of new disease types.

These new functionalities involve the creation of new records and repositories for *diseases* and *symptoms*. Potentially, they can introduce changes to the public interface and interfere with the correct working of the existing pointcuts.

Most of the AOP approaches use a join point model similar to that of AspectJ [10]. The AspectJ pointcut language offers a set of *primitive pointcut designators*, such as `call`, `get` and `set` specifying a method call and the access to an attribute. All the pointcut designators expect, as an argument, a string specifying a pattern for matching method or field signature. These string patterns introduce a real dependency of the syntax of the base code. Intuitively, since pointcuts capture a set of join points based on some *structural* or *syntactical* property, any change to the structure or syntax of the base program could also change the applicability of the pointcuts and the set of captured join points.

[4] The complete source code developed is available at http://www.comp.lancs.ac.uk/ greenwop/-tao

Aspect developer implicitly imposes some *design rules* that the base program developer has to follow when evolves his program to be compliant with the existing aspects and avoid of selecting more or less join points than expected. In this case, problems with evolution depend also of the need of guessing these, often silent, conventions. These rules derive from the fact that pointcuts often express *semantic properties* about the base program in terms of its *structural properties*.

First to present our approach, we present the problems encountered, also in this case, by the AspectJ aspects. In particular, we consider the aspect used for the HW synchronization of concurrent insertion and showed in Listing 1.1.

It is fairly evident that the pointcut definition takes in consideration only the method name of a particular class and not the behavior or the semantic to locate the interested join points. In this case, the aspect programmer has written a correct pointcut, and the corresponding aspect works as intended. When the code is changed (i.e., in version 9) by adding new *persistent entity*, i.e., `DiseaseRecord` and `SymptomRecord` despite the behavior added by these new entities is the same of the old entity, the `synchronizationPoints` pointcut (see Listing 1.2) has been changed in order to consider also the new methods.

**Listing 1.1.** The AspectJ pointcut in version 8

```
public pointcut synchronizationPoints(Employee employee) :
  execution(* EmployeeRecord.insert(Employee))
    && args(employee);
```

Note that this is only a possible way to write the pointcut, but in general the problems are the same. For example, in Listing 1.1, we can insert a wildcard in place of the class name (`EmployeeRecord`), in this case the pointcut is not broken by the evolution, but if the programmer decides to change the method name, e.g., from `insert` to `store` the pointcut does not work right, and (s)he must adapt the pointcut definition to locate all the right point in the application.

**Listing 1.2.** The AspectJ pointcut in version 9

```
public pointcut synchronizationPoints(Object o) :
  (execution(* EmployeeRecord.insert(Employee)) ||
  execution(* DiseaseRecord.insert(DiseaseType)) ||
  execution(* SymptomRecord.insert(Symptom)) )&& args(o);
```

Since, the problem of the evolution in aspect-oriented programs is mainly that the set of join points captured by a pointcut may change when changes are made to the base program, even though the pointcut definition itself remains unaltered. Then, to avoid this problem we need a *low coupling* of the pointcut definition with the source code. The aim of the Blueprint approach is just to overcome the AOP problem about software evolution, by allowing the selection of the join points abstracting from *implementation details*, *name conventions* and any other *source code dependency*.

In Figure 2 is showed the join point blueprint used to locate the methods that need synchronization, it describes a *relevant portion of a method behavior*. In particular, since all application methods that store records into repositories are composed by a check to

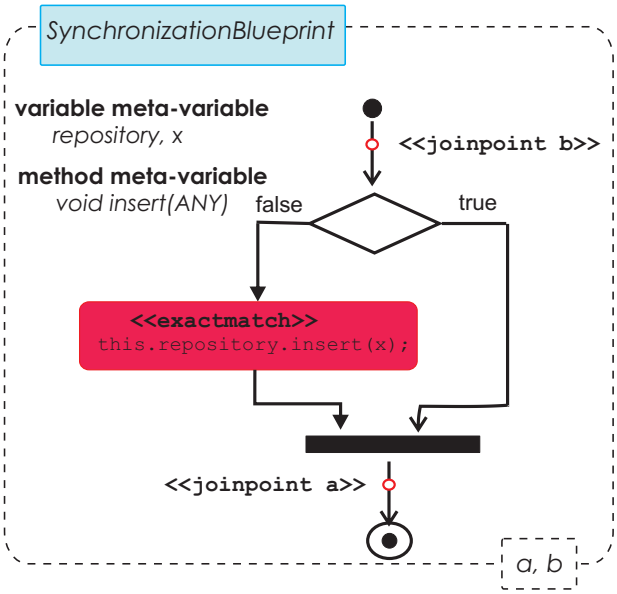



**Fig. 2.** Join Point Blueprint for Insertion Synchronization.

control if the record is already inserted or not inside the repository, we can search an `if` statement containing, in the false branch, the code to insert the record.

The ≪`joinpoint b`≫ locate the join point at the beginning of the method that contains the statement that match the relevant portion of behavior, while the ≪`joinpoint a`≫ locate the join point at the end of the method that contains the matched statements of the diagram. The diamond indicate that we are looking for a conditional statement, where the condition is not relevant for the context definition, like so, it is not relevant what is contained in the `true` branch.

The `repository` variable meta-variable used in the action (i.e., the red rounded rectangle) during the weaving process is unified to a class field. The `insert` method meta-variable represents a method that does not return nothing and that has only one parameter of any type Note that the name of the method meta-variable is completely independent from the name of the searched method, i.e., changing the name `insert` in `abcdef` the located behavior and unified variables do not change.

Since the new entities have almost the same behavior of the old one, like showed in Listing 1.3, the behavior described in the blueprint locate all the methods that need a synchronization point.

Our selection mechanism matches the `insert()` method of the `EmployeeRecord` class presents in version 8, and the `insert()` methods present in the `DiseaseRecord` and, `SymptomRecord` classes, added in version 9, without change anything. The `repository` variable meta-variable is respectively unified to `employeeRepository`, `diseaseRep` and `rep` application's field. The `void insert(ANY)` method meta-variable

**Listing 1.3.** The Application Implementation

```
// insert method of EmployeeRecord class
public void insert(Employee employee) // throws clause {
  if (employeeRepository.exists(employee.getLogin())) {
     throw // new exception ;
  } else {
     employeeRepository.insert(employee);
  }

// insert method of DiseaseRecord class
public void insert(DiseaseType td) // throws clause {
  if (diseaseRep.exists(td.getCode())) {
     throw // new exception ;
  } else {
     this.diseaseRep.insert(td);
  }

// insert method of SymptomRecord class
public void insert(Symptom symptom) // throws clause {
  if (rep.exists(symptom.getCode())) {
     throw // new exception );
  } else {
     rep.insert(symptom);
  }
}
```

is respectively unified to the `insert` method of the `EmployeeRepositoryArray`, `DiseaseTypeRepositoryArray`, and `SymptomRepositoryArray` class.

## 4 Related Works

The Blueprint framework is not the first attempt of dealing with the limitations of the current join point selection mechanisms. In the next of the section we report some of the most significant attempts, without pretending to be exhaustive.

In [8], Kellens et al. tackle the *fragile pointcut problem* by replacing the intimate dependency of pointcut definitions on the base program by a more stable dependency on a *conceptual model* of the program. This conceptual model provides an abstraction over the structure of the source code and classifies base program entities according to the concepts that they implement. The strength of the approach is on the definition of the conceptual model of the base program. The classification of source-code entities in the conceptual model is constructed using annotations in the source code and, defining extra design constraints that need to be respected by source-code entities, for the model to be consistent. This approach requires developers to describe a conceptual model of their program and its mapping to the program code, in this way, it breaks the *obliviousness* [5] property. Moreover, it is coupled with the structure of the base program,

but not coupled with its implementation, and only the program entities can be used to define pointcut, since they use the same join point of the AspectJ join point model. Finally, they still need a mechanism for automatically verifying the correctness of the classifications defined by the conceptual model.

In [4], Eichberg, et al. present the usage of functional query language for the specification of pointcuts. In their approach a pointcut is a set of nodes in a tree representation of the program's modular structure, and this set is selected by query on node attribute written in a query language. They created an XML-to-class file assembler/disassembler that can be used to create an XML representation of a class file and convert an XML file back into a class file on the basis of their bytecode framework. The query language is used on top of this XML representation of the program structure. Their join point model defines more join point of the AspectJ one, because bytecode structure permits to identify more point, e.g., the storing of a value in a local variable. Their query language is general enough to express a wide range of very different pointcut models.

In [17], Sakurai and Masuhara propose a new aspect-oriented programming language that uses unit test cases as interface of crosscutting concerns. A test-based pointcut matches join points in the execution of a target program that (potentially) have the same execution history as one of the unit test cases specified by the pointcut. This approach replace the fragile pointcut problem with the maintenance of unit test cases whose cost should anyhow be paid with practical software development.

In [15], Nagy et al. propose a new approach to AOP by referring to program unit through their design intentions to answer to the need of expressing semantic pointcuts. Design intention is represented by annotated design information, which describes for example the behavior of a program element or its intended meaning. Their approach instead of referring directly to the program, provide a new language abstraction to specify pointcuts based on some design information. Design information are inserted inside the base program using annotations and they are associated manually, derived on the presence of other design information and, through superimposition. The key benefit of this approach is that it reduces direct dependencies between the crosscutting concerns and the program source. Unfortunately, this approach breaks the *obliviousness* [5] property. This property is broken because certain design information has to be specified by the software engineer, and moreover the software engineer must use a consistent and coherent set of design information for each sub-domain of an application.

## 5 Conclusions

Current aspect-oriented approaches suffer from well known fragile pointcut problem. A common attempt to give a solution consists of creating a more semantic mechanism for the join points selection. This paper shortly describe the Blueprint framework, a novel approach to join points identification that permits to decouple aspects definition and base-code syntax and structure. Moreover, this paper presents a test-bed in order to evidence the robustness of the Blueprint pointcut against the software evolution.

## 6 Acknowledgements

The authors wish to thank the original developers of the HW application and Alessandro Garcia for sharing the HW code.

## References

1. Walter Cazzola, Jean-Marc Jézéquel, and Awais Rashid. Semantic Join Point Models: Motivations, Notions and Requirements. In *Proceedings of the Software Engineering Properties of Languages and Aspect Technologies Workshop (SPLAT'06)*, Bonn, Germany, on 21st March 2006.
2. Walter Cazzola and Sonia Pini. Join Point Patterns: a High-Level Join Point Selection Mechanism. In Thomas Khüne, editor, *MoDELS'06 Satellite Events Proceedings*, Lecture Notes in Computer Science 4364, pages 17–26, Genova, Italy, on 1st of October 2006. Springer. Best Paper Awards at the 9th Aspect-Oriented Modeling Workshop.
3. Walter Cazzola, Sonia Pini, and Massimo Ancona. Design-Based Pointcuts Robustness Against Software Evolution. In Walter Cazzola, Shigeru Chiba, Yvonne Coady, and Gunter Saake, editors, *Proceedings of the 3rd ECOOP Workshop on Reflection, AOP and Meta-Data for Software Evolution (RAM-SE'06)*, in 20th European Conference on Object-Oriented Programming (ECOOP'06), pages 35–45, Nantes, France, on 2nd of July 2006.
4. Michael Eichberg, Mira Mezini, and Klaus Ostermann. Pointcuts as Functional Queries. In *Proceedings of the 2nd ASIAN Symposium on Programming Languages and Systems (APLAS 2004)*, LNCS, Taipei, Taiwan, November 2004. Springer.
5. Robert E. Filman and Daniel P. Friedman. Aspect-Oriented Programming is Quantification and Obliviousness. In *Proceedings of OOPSLA 2000 Workshop on Advanced Separation of Concerns*, Minneapolis, USA, October 2000.
6. Phil Greenwood, Alessandro F. Garcia, Thiago Bartolomei, Sergio Soares, Paulo Borba, and Awais Rashid. On the Design of an End-to-End AOSD Testbed for Software Stability. In *Proceedings of the 1st International Workshop on Assessment of Aspect-Oriented Technologies (ASAT.07)*, Vancouver, Canada, March 2007.
7. Kris Gybels and Johan Brichau. Arranging Language Features for More Robust Pattern-Based Crosscuts. In *Proceedings of the 2nd Int'l Conf. on Aspect-Oriented Software Development (AOSD'03)*, pages 60–69, Boston, Massachusetts, April 2003.
8. Andy Kellens, Kris Gybels, Johan Brichau, and Kim Mens. A Model-driven Pointcut Language for More Robust Pointcuts. In *Proceedings of Software engineering Properties of Languages for Aspect Technologies (SPLAT'06)*, Bonn, Germany, March 2006.
9. Gregor Kiczales. The Fun Has Just Begun. Keynote AOSD 2003, Boston, March 2003.
10. Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeff Palm, and Bill Griswold. An Overview of AspectJ. In *Proceedings of the 15th European Conference on Object-Oriented Programming (ECOOP'01)*, pages 327–353, Budapest, Hungary, June 2001. ACM Press.
11. Christian Koppen and Maximilian Störzer. PCDiff: Attacking the Fragile Pointcut Problem. In *Proceedings of the European Interactive Workshop on Aspects in Software (EIWAS'04)*, Berlin, Germany, September 2004.
12. Meir M. Lehman. Programs, Life Cycles, and Laws of Software Evolution. *Proceedings of the IEEE*, 68(9):1060–1076, September 1980. Special Issue on Software Engineering.
13. Meir M. Lehman, Juan Fernandez-Ramil, and Goel Kahen. A Paradigm for the Behavioural Modelling of Software Processes using System Dynamics. Technical Report 2001/8, Imperial College, Department of Computing, London, United Kingdom, September 2001.

14. Noorazean Mohd Ali and Awais Rashid. A State-based Join Point Model for AOP. In *Proceedings of the 1st ECOOP Workshop on Views, Aspects and Role (VAR'05)*, in 19th European Conference on Object-Oriented Programming (ECOOP'05), Glasgow, Scotland, July 2005.
15. István Nagy, Lodewijk Bergmans, Wilke Havinga, and Mehmet Akşit. Utilizing Design Information in Aspect-Oriented Programming. In Robert Hirschfeld, Ryszard Kowalczyk, Andreas Polze, and Mathias Weske, editors, *Proceedings of 4th Annual International Conference on Object-Oriented and Internet-based Technologies, Concepts, and Applications for a Networked World (Net.ObjectDays)*, LNI 61, pages 39–60, Erfurt, Germany, September 2005.
16. Sonia Pini. *Blueprint: A High-Level Pattern Based AOP Language*. PhD Thesis, Department of Informatics and Computer Science, Università di Genova, Genoa, Italy, June 2007.
17. Kouhei Sakurai and Hidehiko Masuhara. Test-based Pointcuts: A Robust Pointcut Mechanism Based on Unit Test Cases for Software Evolution. In *Proceedings of Linking Aspect Technology and Evolution revisited (LATE'07)*, Vancouver, British Columbia, Canada, March 2007.
18. Sergio Soares, Eduardo Laureano, and Paulo Borba. Implementing Distribution and Persistence Aspects with AspectJ. In Mamdouh Ibrahim and Satoshi Matsuoka, editors, *Proceedings of the 17th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'02)*, pages 174–190, Seattle, Washington, USA, November 2002. ACM Press.
19. Dominik Stein, Stefan Hanenberg, and Rainer Unland. Modeling Pointcuts. In *Proceedings of the AOSD Workshop on Aspect-Oriented Requirements Engineering and Architecture Design*, Lancaster, UK, March 2004.
20. Maximilian Störzer and Jürgen Graf. Using Pointcut Delta Analysis to Support Evolution of Aspect-Oriented Software. In *Proceedings of the 21st IEEE International Conference on Software Maintenance (ICSM'05)*, pages 653–656, Budapest, Hungary, September 2005. IEEE Computer Society.
21. Tom Tourwé, Kris Gybels, and Johan Brichau. On the Existence of the AOSD-Evolution Paradox. In *Proceedings of the Workshop on Software-engineering Properties of Languages for Aspect Technologies (SPLAT'03)*, Boston, Massachusetts, April 2003.