Vrije Universiteit Brussel - Belgium
Faculty of Sciences
In Collaboration with Ecole des Mines de Nantes - France
and
Universidad de Chile - Chile
2003

Tool Support for Partial Behavioral Reflection

A Thesis submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
(Thesis research conducted in the EMOOSE exchange)

By: Peter Ebraert

Promoter: Prof. Theo D'Hondt (VUB)
Co-promoter: Dr. José Piquer (Universidad de Chile)

Abstract

More and more programmers use aspect-oriented programming and reflection in order to make a clean separation of concerns in their programs. Currently very few tool support is provided for any of those programming paradigms. It is obvious that good tool support is indispensable for promoting the application of these paradigms.

The goal of this thesis is to provide some tool support for developing reflective applications. Throughout the research, Reflex is used as a case study. It is a Java framework that allows a kind of reflection, permitting a clean separation of concerns in its way.

After a study of the problem environment, we explain the static and dynamic tool support developed in this thesis. Typically, static tool support will be used at development time, while dynamic tools will be used at runtime. At the static level, we help the programmer by easing the Reflex configurations and by detecting and resolving configuration conflicts. At the dynamic level we used Reflex itself in order to create a runtime monitor, allowing programmers to dynamically monitor both standard object-oriented and reflective Java applications built with Reflex.

Acknowledgement

First of all, I would like to thank **Prof. Dr. Theo D'Hondt** and **Isabel Michiels** for giving me the chance of studying the *European Master in Object Orientation and Software Engineering* program and for helping me with the application forms and all administrative formalities.

Next I want to thank **Prof. Dr. Jose Piquer** and especially **Eric Tanter**, for allowing me to investigate for my master thesis at the *Departamento de Ciencias de la Computación* in the *Universidad de Chile.* I also want to thank them, and especially Eric, for always being there whenever I needed help, for the punctual proof reading of several thesis drafts, and for the close cooperation during the development of my thesis.

For travelling with me to Nantes and Santiago and for her great support during the hardest days, I would like to thank my girlfriend **Sabine**, without whose care and special attention the work would have been a lot harder; and **my parents**, who granted me the opportunity to study and supported me during my studies. Also I want to thank Michael, who exerted his knowledge on the English language, in order to glue the words of this research report together.

Finally, a word of thanks goes to **Marisol Bravo**, for allowing us to stay at her house during the investigation period; and to all the new friends I met during the marvellous stay in Santiago de Chile.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Divide et impera! It was Julius Caesar who spoke those words, and applied them in order to manage the complexity of ruling the Roman empire. It is the same principle programmers have to practice in order to cope with the complexity of developing computer programs. The separation of concerns [33] is at the core of software engineering, as it incorporates the divide and conquer principle. Using that principle, all the different concerns should be split up into different modules [59] – separate chunks of code, preferably self-contained, or as self-contained as possible, which perform a certain task. This leads to less duplication of code and therefore smaller code which is easier to read, debug, maintain, improve and extend.

The problem is that in some cases it is very hard to separate all the different concerns, as we might be faced with *crosscutting concerns* [52], which act across many modules. This has led to the invention of many interesting, and effective, modularization approaches.

One of the most successful approaches is Aspect-Oriented Programming (AOP) [47], where every concern is modelled as a separate aspect. All aspects are then automatically composed in order to obtain the requested program behavior [50, 44, 46].

Another way of implementing a clean separation of concerns is through reflection [65, 53]. Reflection allows objects to *look* at their current state or behavior possibly in order to make some *changes* to that state or behavior [70]. These changes permit the programmer to separately implement the non-functional and crosscutting concerns [76].

The major drawbacks of reflection are its inherent complexity as well as its assumed cost in terms of performance. This is why *partial reflection* is very important as it allows highly selective use of reflection. A Java framework – *Reflex* – was developed that allows partial reflection for Java [72]. It offers appropriate interfaces for static and dynamic configuration of partial reflection at various levels.

In recent research, reflection has already been compared to AOP. [44]

classifies reflection into the category of aspectual decomposition paradigms. In [54], the authors state that reflection interacts with AOP in two main respects. First, reflective techniques appear among the most promising to build aspect weavers that would be both general and extensible. Second, AOP appears as a promising structuring tool for reflection as more and more facets come into play in reflective descriptions of complex systems and programming languages. [67] shows how to theoretically implement AOP using reflection.

1.1 Problem setting

More and more researchers are finding out the power of explicit separation of concerns. Because of that, aspect-oriented and reflective programming are getting quite popular. But – as more developers start using them – the need rises for good tool support.

At this time, AspectJ is the most popular implementation of AOP and Eclipse is the most popular Java development environment in research, as it provides an easy extensible plugin framework, is free of charge and completely open source. Because of those facts, the AspectJ team is currently developing an Eclipse plugin: AspectJ Development Tools (AJDT). AJDT will ease AOP and permit most developers to stay in their known programming environment.

Currently there is still no tool support for Reflex and the partial reflection it permits. The only support the developers get, lies in the source code documentation. This is why we want to provide good tool support for Reflex. Tool support comes in two dimensions: *static* support – which is typically used before the program is started – and *dynamic* support – which is used while the program is running. Having both a static and dynamic part, the tool support should facilitate the use of Reflex.

1.2 Goal

The main goal of this thesis is to assist the programmer in the development of reflective applications using Reflex. For that, we provide both static and dynamic tool support.

The static support comes in the form of an Eclipse plugin. It provides the appropriate editors and wizards for configuring Reflex. As conflicts can arise in such configuration, a mechanism for detecting and resolving those conflicts had to be established. We contributed in the development of the Reflex conflict detection and resolution framework. The Eclipse plugin offers the support for using that framework. Finally it also offers some IDE support for debugging reflective applications, having a functionality for stepping through the source code while running the application.

The dynamic support lies in the visualization of the applications execution. It is developed as a stand-alone monitor that can either be used together with or separate from the static part. The monitor provides a great help for debugging ordinary object-oriented or reflective programs. In addition to this, it allows the user to observe and/or manipulate the behavior of an object at runtime to some extent.

1.3 How to read this thesis

This research work starts with an overview and brief explanation of all the technologies that were used during the investigation period. As the main goal of this thesis is to assist the programmer in developing reflective and aspect-oriented applications, we begin by explaining those two paradigms. Respectively Chapters 2 and 3 deal with that matter.

The fourth chapter discusses the technologies that had to be used for implementing the tool support. We start by explaining how Java class loading works. Then we move to Javassist, the heart of the Reflex framework. As the tool support will be implemented as an Eclipse plugin, we then discuss Eclipse and the Standard Widget Toolkit (SWT). Also Sun Windowing (SWING) is discussed as it is used in the dynamic tool support. The chapter ends by clearing out AspectJ and AJDT.

After that theoretical part, we move to the practical part of this thesis and more precisely to the development and implementation of the Eclipse plugin for supporting the developers that are using Reflex. Chapters 6 till 9 each cope with the development explanation of one part of the Eclipse plugin. Chapter 10 handles the dynamic tool support. It explains how a runtime monitor was developed and implemented and which functionalities it currently provides.

We end by evaluating our work, listing some advantages and drawbacks of our approach, drawing some conclusions and stating possible future work (Chapters 11 and 12).

Part I

Preliminaries

Chapter 2

Reflection

The goal of this chapter is to briefly introduce the user in the world of reflection. For a more detailed explanation, the reader should take a closer look at [66, 72].

After giving a quick introduction on what reflection is and where it comes from, we discuss the different existing kinds. We will see that three main levels of reflection can be discerned.

First of all, we will clarify the distinction between static and dynamic reflection, giving an answer to the question *when* exactly reflection will occur. Then we will demonstrate the difference between structural and behavioral reflection, answering *what* will be reflective. Thirdly we compare introspection and intercession stating *which power* the reflection provides. We end by explaining what behavioral reflection is.

2.1 Introduction

According to the Oxford dictionary, *reflection* is the act of sending back a mirror image of something or somebody. Many things can be done with that image. It could be used, for example, to change a current state, like wiping that bit of mayonnaise from one's nose. One can also learn to act differently using the reflection. Just think about the dancers practicing in front of a mirror in order to improve their performance.

In order to better understand the concept of reflection, it may be useful to go back to the studies of *self-awareness* in artificial intelligence: "Here I am walking down the street in the rain. Since I'm starting to get drenched, I should open my umbrella." [66] This situation reveals a self-awareness of behavior and state. Moreover that self-awareness leads to a change in that selfsame behavior and state.

It would be desirable for computations to avail themselves of these reflective capabilities, examining themselves in order to make use of metalevel information in decisions about what to do next. Reflection allows objects to *look* at their current state or behavior maybe in order to make some *changes* to that state or behavior [70].

In real life we need a mirror (or a reflective material) in order to see our reflection. The same goes for computation, where an object is typically observed using some external, *metalevel* viewpoint. That is why in some programming languages reflection is a lot more intuitive than in others. For example in *Smalltalk*[51], the entire object model is based on entities which control other entities – metaentities. A Smalltalk class is in fact an entity which is responsible for managing its instance objects. The class itself is managed by a metaclass, which manages the class and so on. This way, a Smalltalk entity can always query its metaentity for information. As we can see, this architecture allows a very easy implementation of reflection. In *Java*, things get more complex as not all the Java entities are managed by a metaentity. A Java instance will always be able to query its class for information, but a Java class will not be able to do that, because classes in Java not implicitly have a metaentity watching over them.

Reflection is a widely ranging concept that has been studied independently in many different areas of science. It has long been investigated in philosophy and formalized to some extent in logic [39]. It arose naturally in Artificial Intelligence (AI), where it is intimately linked to the end goal itself. In AI, reflection is viewed as the emergent property responsible, at least in part, for what is considered an *intelligent behavior* [28].

Reflection in programming languages has emerged from the studies of Brian Smith around the foundations of self awareness and self-references, and his work around the application of these concepts to computer science [65]. The introduction of reflection to object-oriented programming is due to the doctoral studies of Pattie Maes [53], in which reflection refers to the ability of a system to observe and modify its computation. But also in other sub areas of programming languages, reflection has been applied. In [28] a fine comparison is given on the different applications of reflection in logic, functional and object-oriented programming.

2.2 Principles of Reflection

In a reflective system, code at the *base level* executes under control of code at the *metalevel.* The metalevel is acting as an interpreter of the base level, possibly changing the way the base-level computation is performed. But in order to allow the metalevel to control the base-level program, that base-level program must be *reified.* Reifying a program is providing a mechanism for modelling that program's execution state. This state can then be queried or modified in order to observe or change the corresponding program.

2.3 Reflection kinds

2.3.1 Static versus Dynamic

Reflection can be used both at run and development time. Depending on when the reflection is employed, it is said to be static or dynamic respectively. In static reflection, metalevel information is only used at compile time to produce class specific code. It does not involve runtime computations. Dynamic reflection is much more powerful as metalevel objects still exist at runtime. That way, it allows observing and modifying the base-program objects at runtime [16].

2.3.2 Structural versus Behavioral

Depending on which part of the base-level representation is accessed, the part describing the structure of the program, or the part describing its behavior, reflection is said to be structural or behavioral [72].

Structural reflection is the ability of a language to provide a complete reification of both the program currently executed as well as a complete reification of its abstract data.

Behavioral reflection is the ability of a language to provide a complete reification of its own semantics (processor) as well as well as a complete reification of the data it uses to execute that program [9].

2.3.3 Introspection versus Intercession

If a reflective system is observing the behavior of an application and is coping with answering questions on the state of a program, then we talk about *introspection*. When the system can also auto-modify its code, then we talk about *intercession*. [21].

Bobrow et al. give a nice definition that summarizes this matter in [20]. *"Reflection is the ability of a program to manipulate as data something representing the state of the program during its own execution. There are two aspects of such manipulation: introspection and intercession. Introspection is the ability for a program to observe and therefore reason about its own state. Intercession is the ability for a program to modify its own execution state or alter its own interpretation meaning."*

2.4 Behavioral Reflection

Behavioral reflection consists in adapting the behavior of applications. In order to modify the behavior of an application, that application should be reified, as we have seen in section 2.2. Reification of an object-oriented application consists in instrumenting the concerning operations with hooks. Those hooks are interception points that define where delegation to the

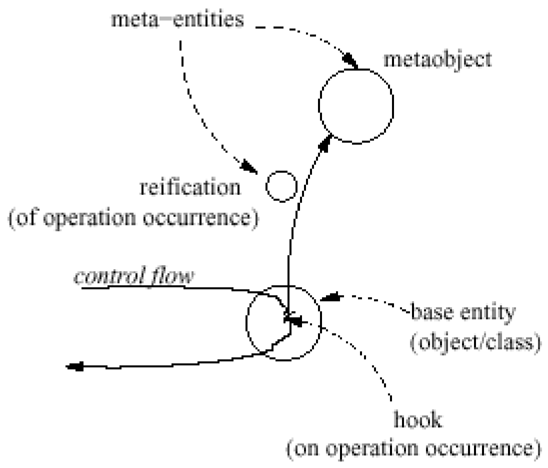


Figure 2.1: Behavioral reflection with metaobjects

metalevel will occur. The link between a base object and a metaobject is called a *metalink*, also known as *causal connection link*.

A *reflective object* is an object in which some operations (message send, instantiation, cast, ...) are reified and controlled by a metaobject. When an instance of a class is reflective, that class is said to be reflective as well.

Whenever control flow reaches a reified operation from a reflective base-level object, a *control flow shift* occurs, giving execution control to the metalevel (*shift up*). There, the hooked metaobject takes execution control and performs the necessary actions. After terminating these actions, another control shift occurs, giving execution control back to the base-level entity (*shift down*). Figure 2.1 clearly shows how behavioral reflection is working.

2.5 Conclusion

We started this chapter by introducing reflection and by stating from which ideas it has emerged. We saw that reflection was first studied in philosophy and artificial intelligence and that it was later introduced in programming languages.

A reflective program typically consists of two levels: the base level containing the base program and a metalevel containing the metaprogram. In order to let those two levels interact, the base program has to be reified. Reifying a program is establishing a mechanism in order to model its execution state in order to observe or modify that state.

There are many different kinds of reflection but basically reflection can be differentiated on three criteria: static versus dynamic, structural versus behavioral and introspection versus intercession. The first criterium involves when the reflection is taking place; static for reflection at development time and dynamic for runtime reflection. The second measure is based on which part of the representation is accessed; structural if the program structure is concerned and behavioral if the program behavior is affected. The last one deals with the ability to change the base program (intercession) or to only observe it (introspection).

This chapter ends by explaining behavioral reflection – adapting the behavior of applications. In order to do that, base programs must be instrumented by hooks to metaobjects, which get control whenever execution control reaches the reified operation of the reflective object.

Chapter 3

Aspect-Oriented Programming

This chapter introduces the paradigm of Aspect-Oriented Programming (AOP) [46, 63, 13, 14]. Depending on the time on which the separated concerns are recomposed, we can classify the different AOP implementations into two main categories: static or dynamic implementations, for compile time and runtime compositions respectively. AspectJ is the most popular implementation of AOP. The AspectJ Development Tools (AJDT) project will provide Eclipse platform based tool support for AOSD with AspectJ [73].

The differences between AOP and reflection are also stated in this chapter. While reflection is more powerful, AOP (as in AspectJ) will be less complex. However, reflection and AOP have a lot in common, as they can both be used in order to separate concerns. We will even see that we can implement AOP using reflection.

This chapter ends with a discussion on aspect conflicts and their resolution. Literature provides several solutions for solving the aspect conflicts. They are compared and discussed in order to solve the equivalent problem occurring in reflection (as we will see in chapter 9).

3.1 Introduction

Some aspects of system implementation, such as logging, error handling, standards enforcement and feature variations are notoriously difficult to implement in a modular way. The result is that code is tangled across a system and leads to quality, productivity and maintenance problems.

AOP is a programming paradigm that is mainly used for coping with that problem, providing ways for a clean separation of crosscutting concerns. A concern is said to be crosscutting if it cannot be cleanly separated into a separate module because it is affecting several modules [37].

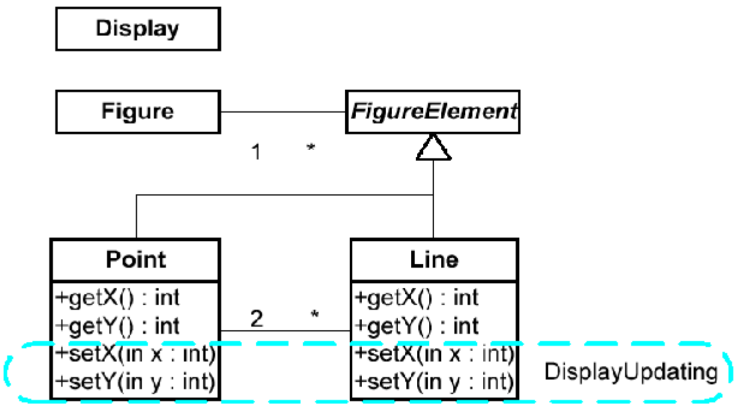


Figure 3.1: The UML diagram of a simple figure editor.

Consider the UML class diagram for a simple figure editor described in [37] (see figure 3.1). There are two concerns for the editor: keeping track of the position of each figure element (data concern) and updating the display whenever an element has moved (feature concern). The object-oriented design nicely decomposes the graphical element so that the data concern is neatly localized. However, the feature concern must appear in every movement method, crosscutting the data concern. The software could be designed around the feature concern; but then the data concern would crosscut concern for display update [50].

AOP aims at separating these different concerns into single units called aspects. An aspect is a modular unit with a crosscutting implementation. It encapsulates some behavior that affects multiple classes into reusable modules.

Aspectual requirements are concerns that introduce crosscutting in the implementation. Error checking and handling, synchronization, context-sensitive behavior, performance optimizations, monitoring and logging, debugging support and multi-object protocols are all aspects.

With AOP, each aspect can be expressed in a separate and natural form. When all aspects are declared, a *weaver* can be exerted to combine the aspects set and base-program files into the tangled application code (see figure 3.2). As a result of this principle, a single aspect can contribute to the implementation of a number of procedures, modules, or objects, increasing reusability of the source code.

Depending on the AOP implementation, the weaving can happen at different times. When it takes place at runtime, we can do both *dynamic* and *static* AOP. When the weaving happens at compile time, only static AOP is possible.

In order to allow this automatic weaving, from aspect code and base-

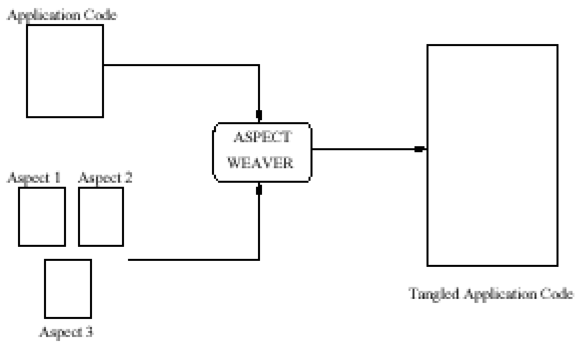


Figure 3.2: E.g. weaver: composing an application using Aspects and base program.

program code, we need some extra entities. Every AOP language has three critical elements for coping with this matter: *a join point model*, *a means of identifying join points*, and *a way of affecting the implementation at join points* [37, 45].

A join point is a certain point in the program structure or the execution of the program. The join point model provides the means to describe the joint points – the points where enhancements should be made. It also provides a mechanism to express sets and subsets of join points, to express common behavior.

After the join points are defined, we need to alter the implementation at those point in order to insert the crosscutting behavior. As we said above, this is also done by a weaver. The weaver parses the base code and whenever a join point is detected, it will insert the associated aspect code. For being able to detect join points, a standard on declaring join points must be established. For that, each AOP language must have a join point definition syntax.

3.2 Implementations

Aspect-oriented programming languages have reached an important milestone as developers are beginning to use them to build real commercial systems [62]. Just like in the early days of Smalltalk, many developers can get great advantage from using these new technologies despite the rough edges that are always present in a first-generation technology. But just as in the early days of object-oriented programming there remain many interesting problems to be solved to further improve the usability and power of this new technology and to maximize the benefits it can provide to developers.

3.2.1 Static AOP

Currently *AspectJ* [4] is the most used AOP language. AspectJ is an aspect-oriented extension to Java that enables the modular implementation of a wide range of crosscutting concerns. Each of those crosscutting concerns is modularized by an *aspect*. Aspect composition – or weaving – in AspectJ happens at compile time, only permitting static AOP.

Another Java extension to static AOP is *Hyper/J* [7]. It supports a "multi-dimensional" separation and integration of concerns. Just like AspectJ, every concern is modelled in a separate entity – a *hyperslice*. However, there is a slight difference between aspects and hyperslices, as the latter may also be used to model non-crosscutting concerns. Just like AspectJ, composition of the different separated concerns also happens at compile time. [75] shows an interesting comparison of both.

Most of efforts on AOP are made on the Java platform. This is not a coincidence, but rather reflects the usefulness of this platform for developing new programming language technologies in a form that is accessible to real-world developers. Nevertheless, the lessons learned from designing these languages on top of Java appear to generalize well to other languages. They have inspired many other projects which extend languages. *AspectR* [5], *AspectC* [2], *AspectC++* [3] and *AspectS* [6] are AspectJ-like implementations of AOP for respectively Ruby, C, C++ and Smalltalk. *Apostle* [1] is another AspectJ-like aspect-oriented extension to Smalltalk. Obviously the Smalltalk extensions Apostle and AspectS also provide dynamic AOP, as in Smalltalk development time is the same as runtime.

3.2.2 Dynamic AOP

Examples of dynamic AOP implementations include *Prose* (PROgrammable Service Extensions) [8] and *EAOP* (Event-based Aspect-Oriented Programming) [35]. Both are Java extensions that allow Java programs modification at runtime.

In PROSE, aspects are regular Java objects that can be inserted in the Java Virtual Machine (JVM). Once an aspect has been inserted in a JVM, any occurrence of relevant events results in the execution of the corresponding aspect advice. If an aspect is withdrawn from the JVM, the aspect code is discarded and the corresponding interception(s) will no longer take place.

In EAOP, a global monitor keeps track of all base-level *events*. That way, the monitor will always be able to execute something extra when a certain event – operation – occurred. It will also be very easy to keep track of control flows because all information is centralized. The monitor can in that way be seen as one big metaobject which is responsible for managing the execution of the base program and the aspects. An aspect can be seen as an event transformer. In fact – in EAOP – an aspect is a Java object

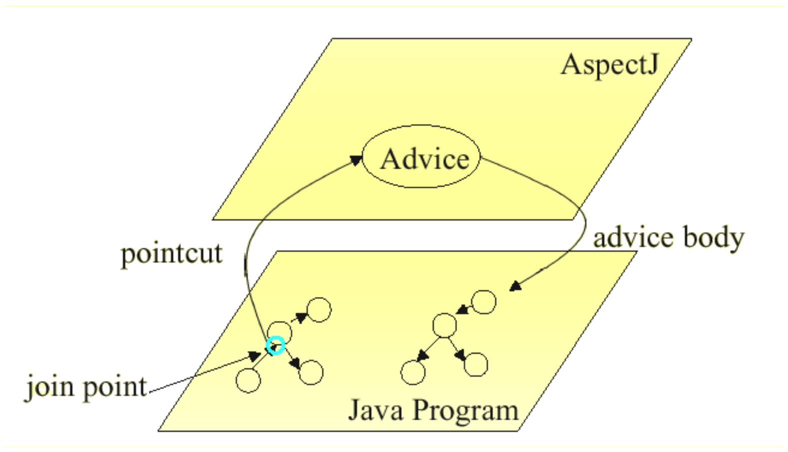


Figure 3.3: The AspectJ model

which takes an event as a parameter, performs some computation or action (which may modify the argument event), and waits for the next event.

Of course there are a lot more implementations and technologies related to AOP, but it is not our purpose to list all implementations. We merely want to give the reader an idea of the current success of AOP and its integration in different programming languages. Precisely this integration will be the challenge for all AOP languages despite the fact that its solutions are likely to be platform specific. Figuring out how best to build usable and extensible tools is a problem that any serious programming language project faces – it is impossible to judge the usefulness of a programming language in the absence of real developers using it, and it is impossible to convince real developers to use a language without high quality tools. In the next section we will discuss the available tool support for AOP [41].

3.3 Tool support for AOP

As we could see in the previous section, AOP has been applied to many programming languages. In general, very few tool support – or none at all – is provided for any of those AOP implementations. The most interesting tool support is in fact provided for AspectJ.

3.3.1 AspectJ

AspectJ adds to Java just one new concept, a *join point* – and that is really just a name for an existing Java concept. It also adds to a few new constructs: *point cuts*, *advices*, *Inter-type declarations* and *aspects*.

Join points state the points in the base program where we want aspects to take control. They can be declared on the following operations:

- Method Call
- Class Boundary
- Method Execution
- Access or Modification of member variable
- Exception Handlers
- Static and Dynamic Initialization

Many join points can be assembled to a set of joint points – *point cuts.* To each of the point cuts, we then attach an *advice*, which states the actions to take when one of the point cuts is reached. The body of the advice is specified in ordinary Java code. The *inter-type declarations* allow the programmer to modify a program's static structure, namely, the members of its classes and the relationship between classes. *Aspects* are the unit of modularity for crosscutting concerns. They behave somewhat like Java classes, but may also include point cuts, advice and inter-type declarations.

In AspectJ, compilation is done in two phases. First there is the weaving phase, in which all aspect code is woven into the base program code at the appropriate places. This results in new java files which can be used in an ordinary java compilation. At runtime, when at a certain point in the execution, a pointcut holds, the attached advice is executed, as we can see in figure 3.3.

The base code and the aspect code must be written using an external Java editor, as no editor is included in the distribution of AspectJ. Luckily things are changing here, as an Eclipse plugin is being developed for easing AspectJ's use: The AspectJ Development Tools for Eclipse (AJDT).

3.3.2 AJDT

The goal of AJDT is to deliver a user experience that is consistent with the Java Development Tools (JDT) when working with AspectJ projects and resources. This will be accomplished by developing an integration layer between the AspectJ Development Environment Framework and Eclipse/JDT extension points. Currently, AJDT provides the tools, explained in table 3.1.

Drawbacks

- By default, AJDT suppresses automatic builds on resource modification (for AspectJ projects only) since AspectJ does not yet support

Tool part	**Explanation**
Popup menus	Allowing the user to execute specific actions
Editors	Properties File Editor (for specifying the AspectJ configuration), AspectJ Editor (an ordinary java editor extended with some keywords (like *aspect*)
Wizards	New AspectJ project, new AspectJ Aspect, new AspectJ configuration file
Actions	Build button, for allowing partial compilation.
Views	Event trace view (displaying all events related with the plugin), Aspect Visualiser Views (none are really useful yet).
Markers	A marker for showing where aspects are applying
Jump Action	For going to the impacting aspect advice.

Table 3.1: The tools provided by the AJDT Eclipse plugin

incremental compilation. This will force the programmer to always compile entire projects and so cause a serious loss of time.

- Currently there is no eager parsing support for AJDT. A consequence of this is that the editors outline view does not update as you edit, but instead is updated as a result of performing a build (after each compilation). This might be annoying because simple spelling mistakes will consequently not be spotted until after compilation.

- Debugging aspect-oriented AspectJ programs is quite hard. One can set breakpoints in both the base program and aspects. But while the debugger will be able to stop the programs execution, it is not yet capable of finding the corresponding source code. Hence stepping through the program in conjunction with the source is not possible. A solution for this problem is a high priority item for the AJDT project.

3.4 AOP versus reflection

While AOP [47] enables the modularization of crosscutting concerns, reflection [65] is commonly recognized as a valuable instrument for separation of concerns [27] as reflective programming languages have always focused on the execution and representation aspects of software.

The main benefit of both programming styles lies in allowing programmers to actively participate in the implementation choices that were traditionally only the responsibility of the language implementors. Although reflective notions can be introduced in practically all programming languages, representing all programming paradigms, object-oriented programming itself

Definition	**AOP term**	**Reflection term**	
A point in the program (static or execution) where some enhancement should be made.	Join point	Hook	Behavioral Reflection
A reference to a group of program points.	Point cut	Set of hooks	
The action to take when program points are reached.	Advice	Metaobject	
The code that implements the action.	Advice body	Metaobject body	
Declarations that modify the static program structure	Inter-type declarations	Structural intercession	Structural Refl.

Table 3.2: Mapping AOP to reflection

gets a prominent role in the field. Objects provide the necessary structuring tool to organize and manage the complexity of metamodels and their protocols. Reflective object-oriented models place a clear frontier between application and descriptive code. The application code appears in the base level of the standard objects whilst the descriptive one appears at the metalevel [54].

So, as we can see, both technologies provide ways to make a clean separation of concerns and consequently aim at maximizing program modularization and reuse. However, there are some significant differences as well. While AOP stresses on language support, ease of use and performance, reflection focuses on power: it does not only provide a separation of concerns but also allows dynamic and generic programming [74]. All this power tends to bring along complexity, which is the main drawback of reflection.

3.5 Implementing AOP with Reflection

In the previous section we have seen that both AOP and reflection are quite similar, as they can both be used to separate concerns. Knowing that AOP is all about separating concerns into aspects and then reassembling all that code again in an automated compilation phase, we know out of intuition that we can implement aspect-orientation using reflection.

Table 3.2 shows all constructs introduced by AOP, their definitions and how we can map them to constructs from the reflection world. This way, table 3.2 proves how we can implement aspect orientation with reflection. Join points, point cuts and advices provide ways for adapting the behavior of an application. They are equivalent to behavioral reflection, while inter-type declarations are equivalent to structural reflection.

In [29] authors state that a combination of the two should be used in order to obtain the best of both worlds. The proposed prototype shows that AspectJ and runtime reflection in Java can be combined together to support runtime weaving, allowing runtime adaptability. Of course this will also have some consequences on the program's performance.

3.6 Aspect conflicts and composition

As we have stated before, implementations of aspect-oriented programming always incorporate two steps. The first is the decomposition of crosscutting concerns in aspects. The second is the recomposition of those aspects into one application. However, problems may rise doing this recomposition, as conflicts between the separate aspects may occur.

When two or more aspects are impacting on the same point of the base program, an aspect composition should be defined in order to avoid conflicts. This composition states the way in which the aspects and the base program are supposed to work together in order to form an application with the desired behavior.

Imagine that both a logging and an encryption aspect are impacting on the same program point. For instance, if we would apply the logging aspect *after* the the security aspect we could generate a log for system administrators. If we would apply the logging *before* the encryption, we could ensure that logging is encrypted. This example clearly shows why aspect compositions should be declared by the programmer.

The same kind of problems might occur in reflective programming, as several metaobjects might be impacting on the same point of the base program. The AOP literature provides different solutions for that issue, as we will see next.

3.6.1 Solutions from the literature

Most of the AOP implementations do not provide a solution for the composition of different aspects impacting on the same base-program point concerns or use a very primitive solution for it. Literature shows three main strategies for coping with problems of that kind: through composition filters, logical meta programming or through a composition framework.

In Prose [61], JAC [60], Hyper/J [7], composition filters are used for coping with conflicting aspects. In AspectJ [4] the aspect precedence must be declared in an aspect composition language.

EAOP also allows more than one aspect to impact on the same execution point. The monitor manages a tree whose nodes are composition operators and leaves are aspects. The composition is done by using a composition language where the allowed composer operators are: `seq`, `fst`, `cond` and `any`, as explained below.

Composition Filters

In [42], Hunleth et al. propose the use of composition filters for composing different concerns. Composition filters [18, 11, 12] provide a model through which crosscutting concerns can be encapsulated into separate software modules. Here, all method invocations are treated as if they were passing messages. Filters can then be set up to monitor and modify messages depending on such things as the sender of the message, the recipient, or the contents of the message. Additionally, filters can be attached to classes as enhancements. Doing this, some aspect code can be introduced when certain message sends occur.

The aspect composition is specified by the order to which the filters are attached to the classes providing only a way of sequentially composing aspects. The sole notion of aspect precedence is however not sufficient as we also might want the composition to be more complex (see 9). [17] states that there is still a lot of research to be done towards possible composition of crosscutting concerns.

Logical Meta Programming

In [30, 32, 31, 23] a logical metalanguage (SOUL or TyRuBa) is used for declaring the program points that need to be enhanced. Logical rules are defined in order to detect the affected join points. When a join point is found, the needed code (of that aspect) is inserted in the base code. When all rules have been applied on the base program, the new source code (extended with all aspects code) is available to be run. Using this idea, one can express aspects as logical rules and compose them by producing new rules that call some of the logical rules in a certain sequence.

This is a very powerful approach, but is not applicable in our case as we do not have any logical language but only Java. Introducing a logical metalanguage in order to provide concern composition, would be too much overhead. Even if we would replace the logical metalanguage by a procedural language (e.g. Java) and apply the same concepts, it would still require a lot of extensions to the existing Reflex framework.

Composition Language and Framework

In [34] the authors state that normally the user – programmer – is responsible for solving the possible conflicts. A framework is then presented that helps the programmer to solve the conflicts. This is done by using a three phase model. The first step is to write all the base and aspect code. The second phase is conflict detection. The last phase includes the conflict resolution. The second and third phase can be iterated on, till all conflicts are resolved.

A conflict occurs when two (or more) aspects, are interacting with each other. I.e. if at least one of their crosscuts match the same join points. But in fact, this definition is too strong. Because it is possible that there is no conflict at all, and that the aspects could be executed one after another without affecting each other. Therefore the authors distinguish between two kinds of independence: strong and weak independence. Two aspects are said to be strongly independent if none of their crosscuts have common join points. In that case, they will never impact on the same program points and never cause any conflicts. Two aspects are said to be weakly independent if they have crosscuts with common join points but if the aspects themselves can be composed to a single aspect. In that case, the aspects will certainly impact on – at least – one common program point. Here, however, a simple aspect composition could solve the conflict. That composition can be specified using one of the following composers:

- A1 `seq` A2: specifies that when both aspects have a common cross point, A1's actions will be done and then A2's actions will be done.
- A1 `fst` A2: propagates the execution control to A1, and, if and only if A1 did not detect a crosscut, the execution control is given to A2.
- A1 `any` A2: propagates the execution control to A1 and A2 in an arbitrary order.
- A1 `cond` A2: propagates the execution control to A1, and, if and only if A1 detects a crosscut, the event is forwarded to A2.

3.7 Conclusion

We started this chapter by introducing the paradigm of aspect-oriented programming (AOP). We saw that this paradigm's goal is to provide a clean separation of concerns in computer programs. AOP typically incorporates two steps: the separation of the different concerns and the composition of the separated concerns into one running application with the desired behavior.

The different implementations of AOP can be classified in two main categories depending on the kind of AOP they provide: static AOP when the composition occurs at development time, and dynamic AOP when the composition is done at runtime. In contrast to the former, the latter permits new aspects to be introduced when the application is already running.

We saw how AOP is gaining more and more popularity and how it is being applied to a lot of programming languages. Still, there is a lot that still needs to be done, as good tool support is indispensable for convincing programmers to use AOP, and very few tool support is provided. Recently AJDT was released. This Eclipse plugin provides good support for developing aspect-oriented applications. Nevertheless, there are still a lot of immaturities in this plugin.

The differences between AOP and reflection were then stated. While reflection is more powerful, AOP (as in AspectJ) will be less complex. Reflection and AOP still have a lot in common, though, since they can both be used in order to separate concerns.

We ended this chapter by discussing the recomposition of the separated concerns. This recomposition can result in conflicts as some concerns can be impacting on the same base-program points. For that, a composition must be defined that states in which way the separate concerns and the base program are supposed to work together in order to obtain the desired behavior. The literature provides three different solutions for that matter. The most primitive solution lies in the use of composition filters, as applied in In Prose, JAC and Hyper/J. Another solution is provided by using logical metaprogramming. The third solution lies in the definition of a composition language and the development of a composition framework, as applied in EAOP and AspectJ.

Chapter 4

Study of environment

In this chapter an overview will be given of the technologies that were used or studied during this research: Java, Javassist, Reflex, Eclipse, SWING and SWT. The concepts of and relations between these technologies form the basis of the this thesis. Therefore, we will not try to explain every small detail of these six technologies, but rather focus on the their concepts, purposes and basic properties.

4.1 Java

A distinction has to be made between *development time*, *compile time*, *load time* and *runtime*. Development time typically is when the programmer is writing the source code of the program. Compile time occurs when the source code is compiled into byte code. Java byte code is stored in a binary file called a class file. Each class file contains one Java class or interface. When all the code has been developed and compiled, the programmer has to run his program. Whenever a class is needed, that class has to be loaded in the memory of the Java Virtual Machine (JVM). That is done by a *class loader* at load time. Every JVM has such a class loader, which can be subclassed in order to customize the class loading process. Once the class is alive in the JVM, it is said to be at runtime.

Java already has some basic reflection support. It comes in a standardized API that we find in the `java.lang.reflect` package. It makes dynamic, behavioral and structural reflection possible. Yet, no modifications can be made to the program behavior, as it only allows introspection of classes and objects. This permits the programmer to query a Java class about its properties, methods, constructors, etc... It is Java's late binding that permits the reflection, making sure that the names and properties of functions are available at execution time. Those names are indispensable for an implementation of reflection [22].

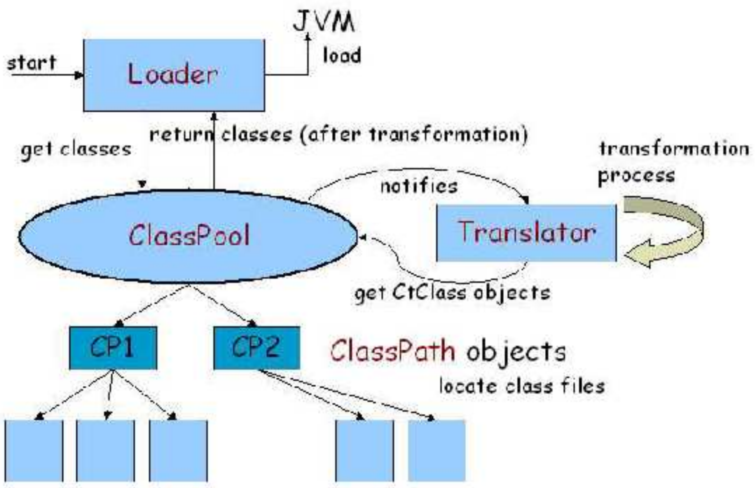


Figure 4.1: The Javassist architecture

4.2 Javassist

Javassist [26] is a class library for dealing with Java byte code. The goal of Javassist is to allow intercession – modifications of compiled Java files; class files – both at compile and load time.

Javassist modifies the class loading process in order to adapt the (byte code of the) loaded classes before they are really loaded into the JVM. In order to do that, Javassist has two special entities: a *class loader* and a *translator*. The Javassist class loader loads the classes from a *class pool* – a set of classes that is known by the class loader. Given a certain name, it must be able to localize or generate the data that corresponds with that class definition.

Before the class is actually loaded into the JVM, the translator gets notified and reifies the class that is being loaded. The result of that transformation is a CtClass object. CtClass objects offer the same introspection capabilities than those of the standard reflection API of Java, plus intercession capabilities (e.g., adding/modifying a member, changing the superclass, altering method bodies...). 4.1 shows how the entire Javassist class loading process works.

Using that architecture, Javassist allows the modification of the class files at load time – before they are actually used. Doing so, we can add some extra behavior to the classes and even modify the class structure down to the method definition.

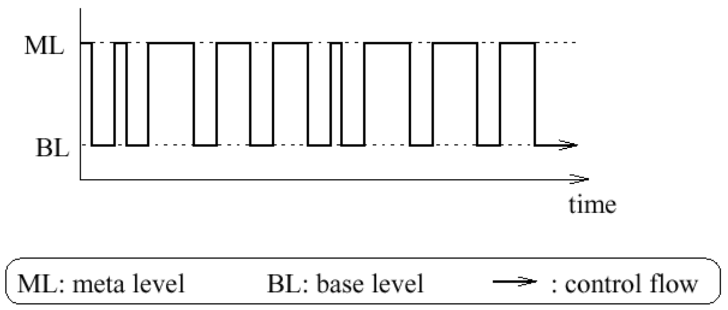


Figure 4.2: Reflectogram of reflective application: illustration of the evolution of the control flow in a reflective application.

4.3 Partial Behavioral Reflection for Java: Reflex

As we have seen in chapter 2, in a reflective application, code at the *base level* executes under control of code at the *metalevel*. Both levels are working together in order to provide the desired application behavior. Because of that, the execution control of such a program, typically has an execution graph that always jumps from base level to metalevel and back (see figure 4.2).

4.3.1 Partial Behavioral Reflection

Partial reflection [40] addresses the issue of flexibility versus efficiency by limiting to the greatest extent the number of control shifts occurring at runtime. Indeed, shifting to the metalevel is powerful but costly. Such a shift consists of first reifying the operation occurrence, and then delegating (at least a part of) its interpretation to the metaobject.

Reification is an important cause for performance degradation. For instance, in the case of a method invocation, reification implies, at least, wrapping all the arguments of the invocation into an array of objects and retrieving a reference to a method object. This data may further be encapsulated into a unique method call object. From an efficiency viewpoint, it is therefore crucial to limit the number of shifts and pay the price of reification only when effectively needed.

Partial reflection makes it possible to balance the effects of compilation, which embeds a set of assumptions (a specialization) and reflection, which retracts some of these assumptions (a generalization). A typical control flow diagram of a partially reflected base program is shown in figure 4.3. One can see that a lot less shifts are made between base and metalevel.

Partiality has two dimensions answering precisely *what* will be reflective and *when* it is reflective. The following two subsections explain each of those

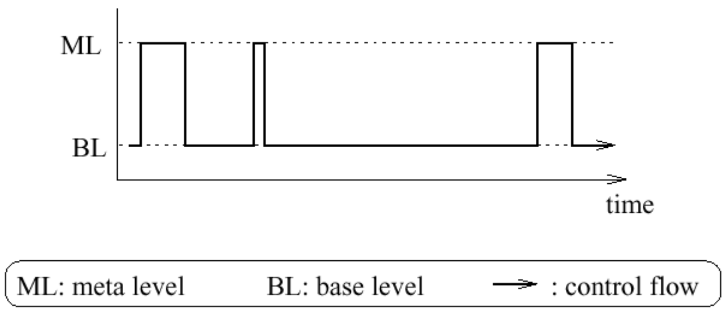


Figure 4.3: Reflectogram of partial reflective application: illustration of the evolution of the control flow in a partial reflective application.

dimensions.

Spatial selection

Spatial selection consists of precisely selecting what will be reified in an application and what will not. Note that spatial selection can be done statically or dynamically. Typically, there are three different levels of spatial selection:

- **Entity selection** refers to the selection of the reflective classes and objects. For instance, specifying that classes `A` and `B`, and instance `c` of class `C` are reflective, whereas other classes and objects are left intact.

- **Operation selection** refers to the possibility of selecting which operations are reified, for a given reflective entity (i.e., a class or an object). For instance we may want to reify message receive and field access for class `A`, and only message send for class `B`.

- **Intra-operation selection** refers to the possibility of performing fine-grained selection with respect to a particular operation. This selection may be based on characteristics of specific occurrences; for instance, we may want to limit message receive reification to message foo on class `A`. For caller-side operations, the selection can also be based on the method or constructor where such occurrence is found; for instance, we may want to reify only message sending of foo (on instances of class `A`) that occur in all public methods of `B`. This last one provides a very fine grained control over the reification process, and makes it possible to significantly improve performance.

Temporal selection

Temporal selection consists of selecting when reifications are effectively active. That way, it is in fact the ability to change the reflective properties over time with a minimal extra cost.

During the lifetime of an object, the reflective needs may change: a reification may have to be turned off so that metalevel behavior does not apply anymore, or conversely, some external condition may require activating a reification. Obviously, for temporal selection to be worthwhile, the cost of a deactivated reification should be less than that of an activated reification with an empty metaobject. In [72] some benchmarks show that this approach has indeed a better performance.

4.3.2 Reflex

Reflex [72] is the first Java extension that fully supports partial reflection in a portable manner, and that seamlessly integrates load time and runtime reflection, along with many static optimizations. Because of its concern with the concrete applicability of behavioral reflection, the concretization of its architecture takes the form of a portable Java library. Note that its architecture is dedicated to Java, but the concepts and underlying ideas of this architecture are language-independent.

Reflex makes classes reflective through byte-code transformation at load time. It uses the Javassist framework, in order to do that. Before they are actually loaded into the JVM, the classes are instrumented with a mechanism that allows the metalevel to *take control* of the execution. Entities of the metalevel will then be able to reason and act upon base-level computation. Doing the byte-code modifications at load time, makes sure that Reflex is applicable to binary components and in settings where all classes are not known until they are actually loaded (e.g., open distributed systems).

In Reflex, a group of execution points of interest can be put together in a *hookset*. Every hookset has a metaobject associated with it. That way they provide a customizable means to conceptually group execution points, and manipulate, possibly dynamically, the metaobjects associated with them [70, 72].

This thesis is based on Reflex 1.0. In this version, partiality is provided by the use of configuration files for static configurations and by the use of a runtime API as some aspects may have to be configured while the application is running. For instance, the program assembler may know that a given reification needs to be activated all the time. On the contrary, a metaprogrammer may want to activate some reification on runtime events. Because Reflex is based on byte code manipulations, it only provides static spatial selection, only allowing the user to specify what will be reified in an application before the program is started. Reflex provides an activation

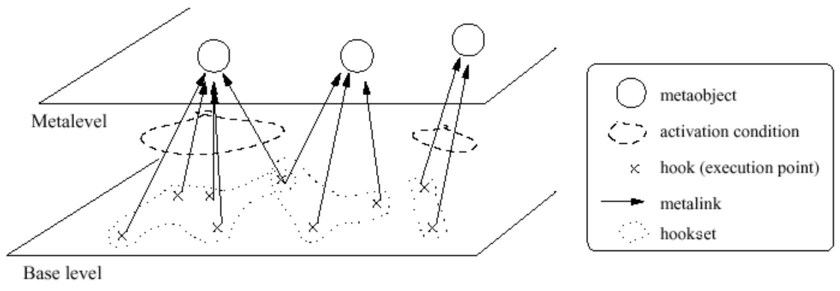


Figure 4.4: The Reflex model: In this example, three hooksets (two of which are activatable) are represented with their associated metaobjects.

mechanism in order to overcome this limitation, as explained below.

In figure 4.4, we see how a metaobject is attached to some execution point(s). For doing that, one has to assemble all the execution points of interest into a hookset. This is done in the assembler configuration file (as we will see in the next section). To that hookset, we then attach a metaobject. When the programs execution reaches a point that has a hook to a metaobject, the control is given to the attached metaobject.

In both of the configuration files, metaobject attributes can be given. The six available attributes are presented in table 4.1. The metaobject activation clause is an intermediate layer, whose advantages are twofold. First it permits us to activate some hooksets at some time and deactivate them later (at runtime), making a real partial behavioral reflection possible. The second advantage lies in the performance, since the performance overhead of a deactivated metaobject is a lot smaller then an activated one. So when reflection is not needed anymore for a certain hookset, we can turn it off to win on performance.

4.3.3 Reflex configuration

In Reflex, four different roles were defined in the development process of a reflective application [71]. While the *base and metaprogrammer* have to implement respectively the base and the metalevel of the reflective application, the metalevel architect and the assembler are each specifying a part of the static configuration of the framework. The *metalevel architect*, specifies the structure of the metalevel and the *assembler* has to compose the results of the previous three, for example by specifying where which metaobjects must be applied, and when.

Separating the static configuration in two levels, can be useful in the following way. The architect level can be used to impose some constraints

attribute	values	description
control	before^d after before-after replace	specifies the semantics of the meta-object, that is, if it acts before and/or after operation occurrences or if it replaces them.
scope	object^d class hookset	specifies the scope of the metaobject, that is, if the metaobject is instance-, class-, or hookset-specific.
mintype	*list of qualified class/inter-face names*	enforces type compatibility rules over metaobjects: when setting a metaobject reference, the Reflex runtime ensures that the given metaobject is compatible with the specified types.
updatable	true^d false	specifies whether a metaobject reference can be changed at runtime or not.
activation	disabled^d startOn startOff	specifies whether the underlying reifica-tions of a hookset can be activated and deactivated dynamically. If activation is enabled, one must specify whether the hookset starts as activated or not.

^{d} = default value

Table 4.1: Reflex metaobject attributes

to the metalevel, to which the assembler has to comply in his assembler level configuration. This configuration can for example be used to impose a security framework [25], or to provide a common metaobject composition framework [71, 58].

Architect configuration file

This is the configuration file for the metalevel architect. The main purpose of this file is to allow the metalevel architect to define the operations which can be reified. It also allows the architect to declare or enforce some metaobject attributes.

For an operation to be reifiable, the metalevel architect must associate a static operation class with a corresponding operation name, and a reification component, which is the code transformation entity capable of reifying such an operation. For instance, the following XML code properly declares support for two operations, cast and message send:

```
<operations>
   <operation
      class="reflex.operation.MsgSend"
      name="MsgSend"
      reificationComponent="reflex.reifcomp.MsgSendRC"/>
   <operation
      class="reflex.operation.Cast"
      name="Cast"
      reificationComponent="reflex.reifcomp.CastRC"/>
</operations>
```

Assembler configuration file

Through this file, the assembler is allowed to define hooksets, and, optionally, class selector sets.

A hookset consists of hookset parts. Every part specifies the events that the metaobject is interested in by stating a class selector and operation selector (respectively responsible of defining which classes and which operation occurrences that are to be be reified). Also the control of the part – stating whether the metaobject is given control before, after, before and after an operation occurrence or if it can replace it – is related to such specification (as we might want to declare parts with a different control) and can therefore be included in the parts definition.

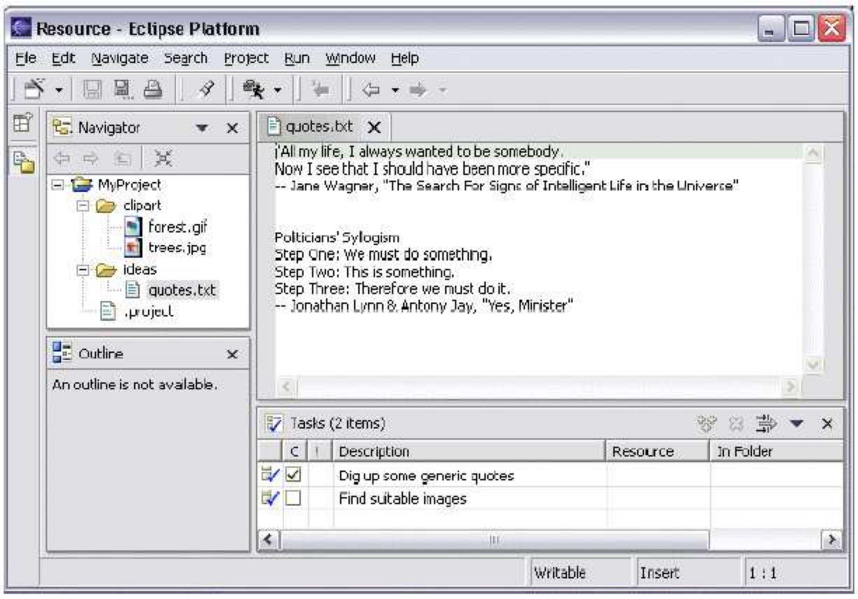
Theoretically, a hookset specifies a set of hooks which are pieces of code that are inserted into a base entity. They define a reification of an occurring operation and make sure the associated metaobject gets control whenever that target operation gets executed. The metaobject attributes (see table 4.1 are declared in this configuration file and may not be conflicting with enforced attributes declared in the architect configuration file. If none are declared, the default attributes from the architect configuration file are used.

A class selector set states which classes must be reified at which condition. They provide a mechanism for allowing partiality, stating which classes must be reified and which not. The undermentioned code shows the definition of an assembler configuration XML file.

```
<reflexConfig-assembler>
   <classSelectorSet id="CSset1">
      <classSelector name="ClassSelectorA"/>
      <classSelector name="ClassSelectorB"/>
   </classSelectorSet>
   <hookset id="hookset1">
      <part
         id="part1"
         operation="reflex.operation.MsgSend"
         classSelector="CSset1"
         operationSelector="OpSelectorA"/>
      <part
         id="part2"
         operation="reflex.operation.Cast"
         classSelector="ClassSelectorC"
         operationSelector="OpSelectorB"/>
      <metaobject>
         <definition
            setter="MetaobjectSetterA"/>
         <attributes
            control="before"
            activation="startOn"/>
      </metaobject>
   </hookset>
</reflexConfig-assembler>
```

4.3.4 Applications

Behavioral reflection has been applied to many domains, in particular distributed systems [56, 24, 48], adaptable mobile object systems [49, 69, 19], concurrent systems [55] and fault tolerant systems [38]. The main strength of behavioral reflection is to provide the means to achieve a clean separation of concerns [33, 59], including dynamic ones, and hence to offer a modular support for adaptation in software systems [19, 64].



An outline is not available.

```
"All my life, I always wanted to be somebody.
Now I see that I should have been more specific."
-- Jane Wagner, "The Search For Signs of Intelligent Life in the Universe"


Polticians' Sylogism
Step One: We must do something.
Step Two: This is something.
Step Three: Therefore we must do it.
-- Jonathan Lynn & Antony Jay, "Yes, Minister"
```

C	!	Description	Resource	In Folder
✓		Dig up some generic quotes		
☐		Find suitable images		

Writable Insert 1 : 1

Figure 4.5: The Eclipse platform user interface

4.4 Eclipse, SWING and SWT

The Eclipse Platform is designed for building Integrated Development Environments (IDEs) that can be used to create applications as diverse as web sites, embedded Java programs, C++ programs, and Enterprise JavaBeans. SWING is a Java library that copes with the graphical issues of making ordinary Java applications. The Standard Widget Toolkit (SWT) is a SWING equivalent library. But as Eclipse is made using SWT, it is mandatory to use SWT in order to create Eclipse plugins.

4.4.1 Eclipse

The Eclipse platform is an IDE for anything, and for nothing in particular. Figure 4.5 shows a screen shot of the main workbench window as it looks with only the standard generic components that are part of the Eclipse platform.

Although the Eclipse platform has a lot of built-in functionality, most of that functionality is very generic. It takes additional tools to extend the Platform to work with new content types, to do new things with existing content types, and to focus the generic functionality on something specific.

The Eclipse platform is built on a mechanism for discovering, integrating, and running modules called *plugins*. A tool provider writes a tool as a separate plugin that operates on files in the workspace and surfaces its tool-specific User Interface (UI) in the workbench. When the Platform is launched, the user is presented with an integrated development environment composed of the set of available plugins. Even the Java Development Tools are developed as an Eclipse plugin.

The quality of the user experience depends significantly on how well the tools integrates with the Platform and how well the various tools work with each other.

Eclipse is open source, free of charge and easy to extend because of its plugin framework and because it is written in Java. Those reasons make Eclipse the far most popular Java development platform in the research community.

4.4.2 Sun Windowing, SWING

A common issue in widget toolkit design is the tension between portable toolkits and platform integration. The Java AWT (the Abstract Window Toolkit) provides platform integrated widgets for lower level widgets such as lists, texts, and buttons, but does not provide access to higher level platform components such as trees or rich text. This forces application developers into a "least common denominator" situation in which they can only use widgets that are available on all platforms.

The Swing toolkit attempts to address this problem by providing non-native implementations of high level widgets like trees, tables, and text. This provides a great deal of functionality, but makes applications developed in Swing stand out as being different. Platform look and feel emulation layers help the applications look more like the platform, but the user interaction is different enough to be noticed. This makes it difficult to use emulated toolkits to build applications that compete with shrink-wrapped applications developed specifically for a particular OS platform.

4.4.3 Standard Widget Toolkit, SWT

The Standard Widget Toolkit (SWT) is a widget toolkit for Java developers that provides a portable API and tight integration with the underlying native Operation System GUI platform. It is equivalent with SWING, but is specifically used for making Graphical Eclipse plugins. Even Eclipse itself was developed using SWT.

Many low level UI programming tasks are handled in higher layers of the Eclipse platform. For example, the plugin.xml markup for UI contributions specifies menu and toolbar content without requiring any SWT programming. Additionally, JFace viewers and actions provide implementations for the common interactions between applications and widgets. However, knowledge of the underlying SWT architecture and design philosophy is important for understanding how the rest of the platform works.

Internally, the SWT implementation provides separate and distinct implementations in Java for each native window system. The Java native libraries are completely different, with each surfacing the APIs specific to the underlying window system. Because no special logic is buried in the natives, the SWT implementation is expressed entirely in Java code. Nevertheless, the Java code looks familiar to the native OS developer. Any Windows programmer would find the Java implementation of SWT for Windows instantly familiar, since it consists of calls to the Windows API that they already know from programming in C. Likewise for a Motif programmer looking at the SWT implementation for Motif. This strategy greatly simplifies implementing, debugging, and maintaining SWT because it allows all interesting development to be done in Java. Of course, this is of no direct concern for ordinary clients of SWT since these natives are completely hidden behind the window system-independent SWT API.

4.5 Conclusion

In this chapter we introduced all the needed technologies for making the Reflex plugin, as stated in the thesis goals. We saw how class loading works in Java, and how that Javassist modifies this process in order to provide load-time behavioral reflection for Java.

We have seen how partial behavioral reflection limits the control shifts occurring at runtime, and so making reflective programs more efficient. Reflex uses Javassist to make load-time byte-code modifications in Java programs. Doing so, Reflex provides functionality to do partial, dynamic and behavioral reflection for both introspection and intercession.

Reflex separates four roles in the development process. The base-level and metalevel programmer respectively implement the base and metalevel source code. The metalevel architect designs the metalevel, possibly imposing some limits. The assembler uses the production results of the three former ones, in order to composes a partial reflective application. Both the architect and the assembler are supposed to define their configurations into two separate XML configuration files.

At this time, Eclipse is the most popular tool for Java development. SWT and SWING are both Java libraries for the development of graphical applications. The main difference between both lies in the target environment of the application being developed. While SWING is perfectly suited for general user interfaces, it is mandatory to use SWT for the development of Eclipse plugins, as Eclipse itself is developed using SWT.

Part II

Static tool support

Chapter 5

Introduction

The Reflex tool support can be divided up in two major parts. First there is the static tool support, that is used at development time. Second there is the tool support that is to be used at runtime. This part explains what the static part incorporates and how it was developed. The next part will cope with the dynamic tool support.

5.1 Why Eclipse?

In section 4.4.1, we have seen that Eclipse is written in Java, which suites Eclipse's extensibility well. The Eclipse project oversees development of the Eclipse IDE platform and the Java Development Tooling (JDT) – a Java development environment built on the Eclipse platform. It also sets the code and specifications for the plugin development environment and provides enough documentation for doing that.

Eclipse is not the best, though the cheapest Java Development Tool, as it is free of charge. Next to that, it is open source, which is very much appreciated in the research community. These two facts make Eclipse a very frequently used Java development environment for researchers. As Reflex is a framework which is currently still targeting the research group, it was an obvious choice that the tool support for it also had to target that same user group.

These two motivations made us choose Eclipse as the environment for which to develop the Reflex tool support.

5.2 Plan of attack

The Reflex development phase incorporates four main fazes. First the metalevel architect designs the reflective application. Then, both base- and metalevel code should be implemented. Thirdly, the assembler specifies how the base and metalevel have to work together in order to provide the desired application behavior. The last faze of the development process consists in testing and debugging the new application.

The plugin can be divided into four parts, each providing support for some faze(s) of the development process. Each part will be discussed in one chapter.

Chapter 6 discusses the plugin part responsible for the management of the static configuration files – the files in which the architect and assembler specify their part. It will incorporate wizards and editors for the corresponding tasks of the configuration creation and modification. In Chapter 7, we discuss the ability to launch (run or debug) Reflex programs.

The final two chapters of this part (8 and 9) explain how we will assist the programmer in understanding and configuring their applications properly. First we provide and explain the functionality of detecting where hooks are inserted in the base program. Then we explain how conflicts in those impacting hooksets are detected and how they are to be resolved.

Chapter 6

Configuration files

Reflex uses two levels of static configuration. After explaining both levels and their associated configuration files, we explain why and how we developed the tool support for creating and modifying those files. Sections two and three give a detailed overview on the working of respectively the wizards and editors.

6.1 Introduction

Each time a class is loaded by the Java Virtual Machine, Reflex has to determine which hooksets are associated to that class by applying the class selectors. Then it has to see where to insert hooks in the class byte code by applying the operation selectors (see also section 4.3. The static configuration of Reflex is done through two configuration files, one for the metalevel architect and one for the assembler.

First, the metalevel architect has to specify the structure of the metalevel through some global parameters, some of them directly related to the configuration file of the assembler. For instance, the configuration attributes for the metaobjects can be enforced here. Also the different operations that will possibly be reified must be declared in this file 4.3.3.

On the second level (the assembler level) the hooksets and their parts are declared 4.3.3. It is on that same level that the programmer will be allowed to define the hookset part compositions using the composition language, as we will see in Chapter 9. Also the class selectors are to be declared in this level. They are provided as a convenient way to group various class selectors under a unique identifier, which can then be used in the hookset definitions. They provide ways to state the partiality of the reflection, as seen in section 4.3.1.

The configuration of each level has to be defined in a separate XML file. Theoretically there is no name constraint on those files. But in order to ease the identification of the configuration files, we propose the name convention

Config file	**File name**
Architect configuration	whatever-name.arc
Assembler configuration	whatever-name.asc

Table 6.1: Reflex configuration file name convention

stated in table 6.1.

In order to facilitate the definition of the configuration in the two XML files, we chose to develop both configuration wizards and editors. Each one has its strengths and weaknesses, so we want to leave the choice to the programmer, which one to use. While the wizards are a lot more user friendly, the editors will be a lot faster to work with. So at the beginning, a programmer might feel more secure working with the wizards. But after gaining some experience, the editors might probably be preferable.

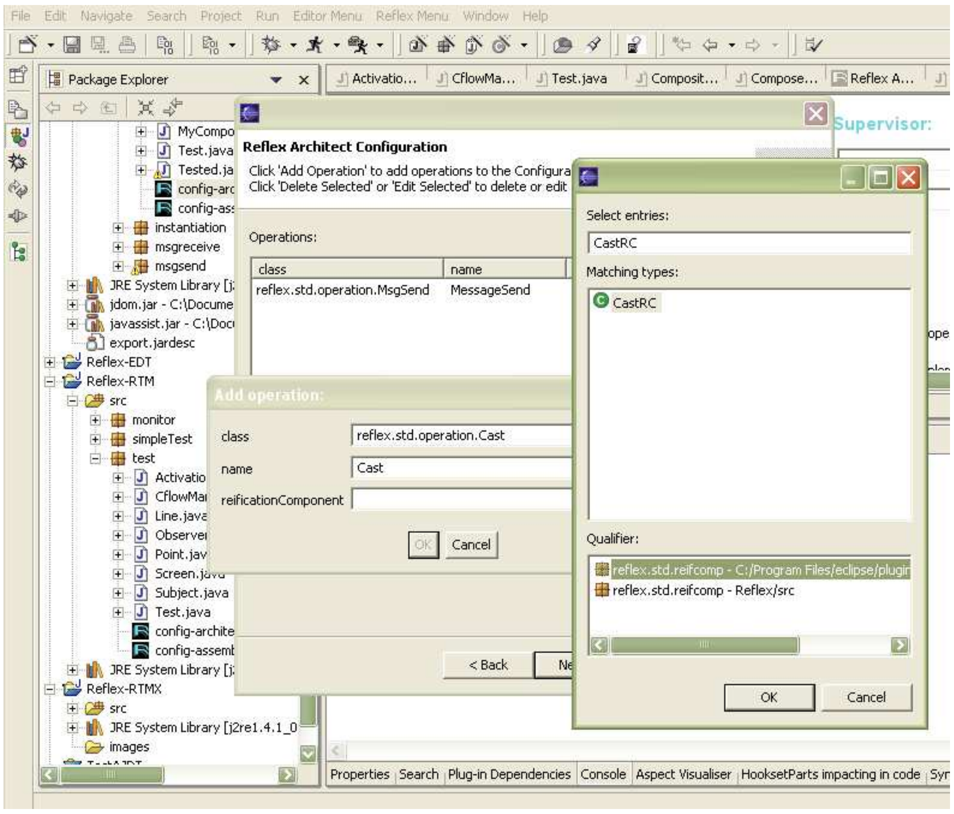
A common advantage of both editors and wizards lies in the automatic XML code generation. All know it is quite painful to program in XML, as it requires a lot of self-discipline for avoiding bugs. Using the wizards or editors, the programmer will avoid these inconveniences and ensure that no mistakes are made against the XML schema. The automatic code generation also makes sure that the same layout format (same amount of tabs, spaces and returns, etc.) is used in all produced configuration files, bringing along better comprehensibility.

6.2 Configuration wizards

A wizard is a series of structured dialog boxes that ask questions. The answers to these questions produce the unique desired result. Wizards are designed specifically to accomplish a task. Learning may be a product of a wizard – but not its main goal, because that is the accomplishment of the task itself. Studies claim that wizards enhance user's performance and reduce the time to complete a task. They also reduce training time, since the knowledge to complete a task is embedded in the wizards, and the user may no longer need to know or be trained in that knowledge [15].

As we have seen in the introduction of this chapter, there are two different kinds of configuration files in Reflex. Consequently two wizards were developed in order to assist the programmer in creating the architect and assembler configuration files. Figure 6.1 shows a screen shot of the architect configuration wizard in action.

While the wizards are perfectly suited for starting out a configuration file, editors are obviously better suited for modifying existing configuration files.



Reflex Architect Configuration

Click 'Add Operation' to add operations to the Configura
Click 'Delete Selected' or 'Edit Selected' to delete or edit

Operations:

class	name
reflex.std.operation.MsgSend	MessageSend

Add operation:

class	reflex.std.operation.Cast
name	Cast
reificationComponent	

Select entries:
CastRC

Matching types:
CastRC

Qualifier:
reflex.std.reifcomp - C:/Program Files/eclipse/plugir
reflex.std.reifcomp - Reflex/src

Figure 6.1: Screen shot of the architect configuration wizard.

6.3 Configuration editors

Once the programmer is feeling familiar enough with the configuration file syntax, he can start using the tools editors. They provide a faster and still graphical way of modifying or creating the static Reflex configuration files. As there are two different kinds of configuration files, two different editors were developed.

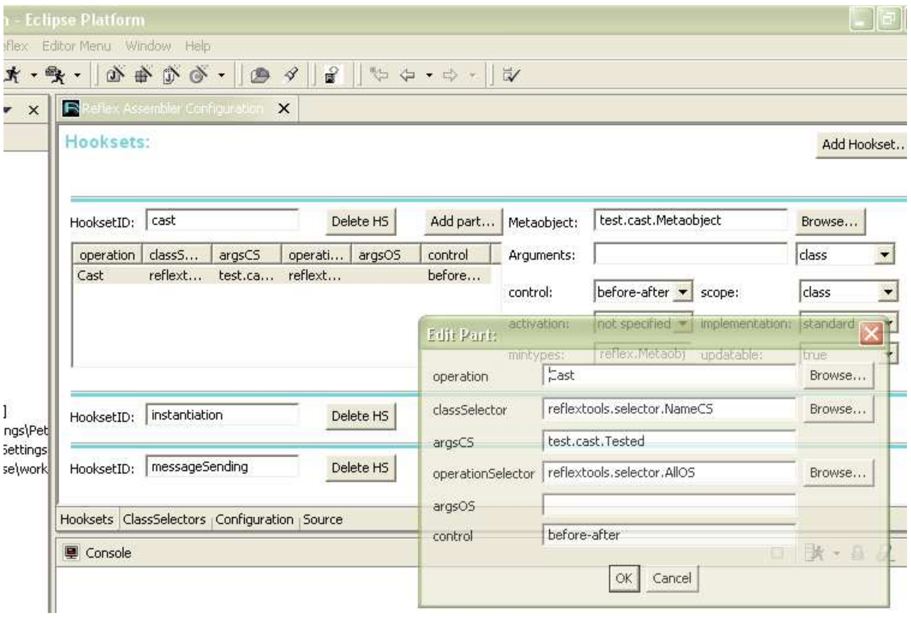
The editors use tabs for separating the different parts of the configuration data. The architect configuration file editor is the easiest, as it only has two tabs. The first tab shows a graphical overview on the enforced and default attributes, the reified operations, and the reification supervisor (as defined in [72]). Text fields, radio buttons, tables and drop down boxes are used to represent the configuration data. The second tab shows the XML code that is equivalent to the graphical configuration specified in the first tab.

The assembler configuration file editor is a bit more complicated, as every assembler configuration file must be related to some architect configuration file, specifying some constraints for the first. Because of that, the assembler configuration file editor has four tabs. The first tab contains a graphical representation of all the data related to the hooksets and their parts. The second tab contains all data on defined class selectors. The third tab contains the data on the associated architect configuration file. The fourth – and last – tab contains the associated XML source code of that configuration file. Again text fields, radio buttons, tables and drop down boxes are used to represent the configuration data.

When the user opens a certain assembler configuration file for the first time, he will be asked for the reference to the associated architect configuration file. This reference is needed in order to verify the correctness of the assembler file, as the architect file might impose some constraints to which the assembler file must comply. The information on the associated architect configuration file is stored in a file on disk (in the same directory as the edited file) in order to avoid that – every time the programmer opens an assembler file – the question would be raised asking for the reference to that file. The user will be able to modify this association through the third tab of the assembler editor.

The editors are implemented in such a way, that the source code will always be up to date with the graphical representation. Thanks to that, the user might change the configuration in the graphical part, and the corresponding XML code will be automatically updated. The other way around, it also works.

Every time a user changes tabs, his changes are checked and if not correct, a warning message containing more details on the mistake is displayed to the user. The user is then asked to correct the mistake before the tab-change will be permitted. Figure 6.2 shows a screen shot of the assembler editor in action.



HooksetID: cast
Metaobject: test.cast.Metaobject

operation	classS...	argsCS	operati...	argsOS	control
Cast	reflext...	test.ca...	reflext...		before...

Arguments: class
control: before-after scope: class
activation: not specified implementation: standard
mintypes: reflex.Metaobj updatable: true

Edit Part:
operation: Cast
classSelector: reflextools.selector.NameCS
argsCS: test.cast.Tested
operationSelector: reflextools.selector.AllOS
argsOS:
control: before-after

HooksetID: instantiation
HooksetID: messageSending

Figure 6.2: Screen shot of the assembler configuration editor.

Due to time constraints, Undo/Redo information is not maintained. The editor makes use of a temporary file (containing a copy of the original file) for allowing the user not to save the changes made. When opening a file for editing, the temporary file will be automatically created in the same directory as the edited file. When closing the edited file, the temporary file will be automatically removed. If the user does not wish to save his changes, the contents of that temporary file are placed in the real file. That way, the original information is preserved.

6.4 Conclusion

The Reflex configuration can be divided up in two levels: the architect and assembler level. On the architect level, the metalevel architect is defining which operations that will be reified, and stating the default and enforced attributes of the metaobjects. On the assembler level, the cooperation between the base and metalevel is specified.

For each of the two different levels there is one XML configuration file, specifying the configuration of that level. We presented both a wizard and editor for each level, responsible for aiding the user in declaring and adapting the configuration file of that level.

The configuration wizards, are best suited for insecure users that are starting to work with the Reflex framework. They provide clean and user friendly ways of declaring configuration files. The configuration editors provide a fast and clear way for modifying existing configuration files. Both wizards and editors make sure that the resulting XML files will never conflict with the XML schemas. They will also ensure that always the same layout is used in the produced configuration files, increasing their readability and comprehensibility.

Chapter 7

Launcher and debugger

The ability to launch (run or debug) code under development is fundamental to an Interactive Development Environment (IDE). But because Eclipse is more of a tools platform than a tool itself, Eclipse's launching capabilities depend entirely on the current set of installed plug-ins. This chapter describes how we built a launcher and debugger for the Reflex framework. It is based on [68], an article which was indispensable for the development of this part of the plugin.

7.1 Introduction

A launcher is a set of Java classes that live in an Eclipse plug-in that performs launching. A debugger is an equivalent set that allows the programmer to run his applications in debug mode (step-by-step running, jumping into a method call, return, etc.). Support for that kind of debugging is in fact also a part of the dynamic support the plugin provides.

The first rule of implementing a launcher is to reuse as much as possible. As the Reflex launcher and debugger extend the Java launcher and debugger respectively, there is a large amount of code – that is part of the public Eclipse API – that may be reused.

7.2 Defining the launch configuration type

The first step in creating our launcher is declaring a configuration type, as shown in the following XML snippet from our plug-in's `plugin.xml` file:

```
<extension
 point="org.eclipse.debug.core.launchConfigurationTypes">
   <launchConfigurationType
      name="Reflex"
      delegate=
         "reflex.edt.launcher.ReflexLaunchConfiguration"
      modes="run, debug"
      id="reflex.edt.launchconfig">
   </launchConfigurationType>
</extension>
```

The most important part of this declaration is the delegate attribute which specifies the fully qualified name of a class implementing the launch method, which launches a specified config. `ReflexLaunchConfiguration` extends the `AbstractJavaLaunchConfiguration` delegate and overwrites only the launch method, which is called by Eclipse when the programmer asks to run an application of the `reflex.edt.launchconfig` configuration type.

The modes attribute specifies both run and debug, as we want this configuration to be used for both the launcher and the debugger.

7.3 Defining tab groups

The most important piece of User Interface to consider when developing a new launcher is the 'tab group'. A tab group specifies a number of tabs which will each hold some part of the configuration data. The user will be able to reach the new tab group as can be seen in figure 7.1.

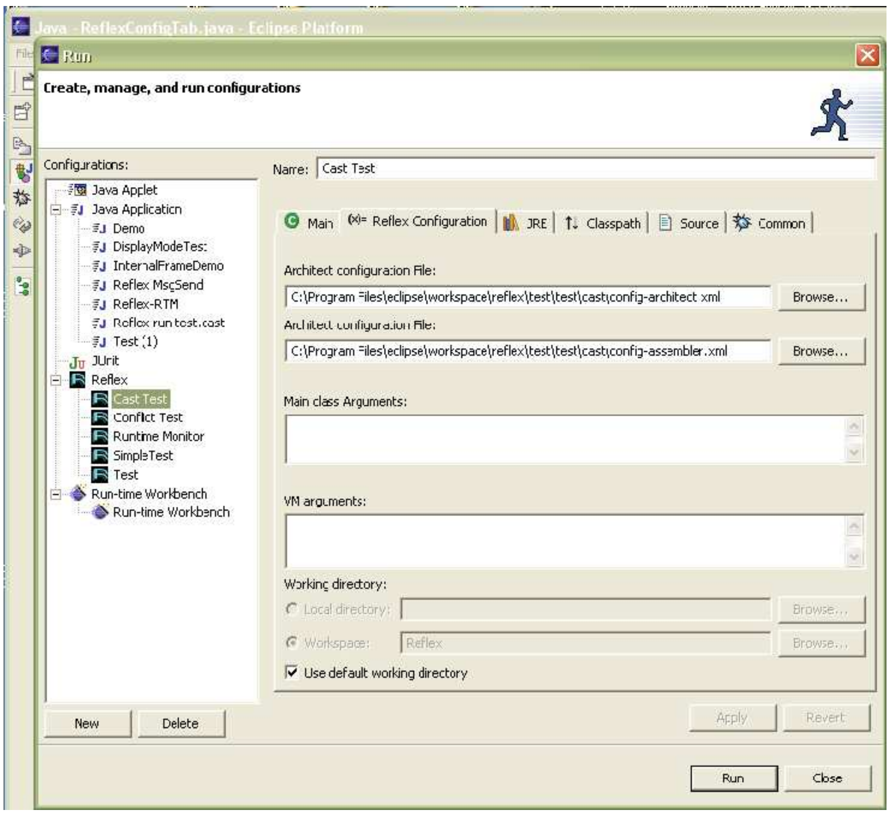
The following XML code declares the tab group in the `plugin.xml` file.

```
<extension point=
      "org.eclipse.debug.ui.launchConfigurationTabGroups">
   <launchConfigurationTabGroup
      type="reflex.edt.launchconfig"
      class="reflex.edt.launcher.ReflexTabGroup"
      id="reflex.edt.launchConfigurationTabGroup">
   </launchConfigurationTabGroup>
</extension>
```

All that this declaration really does is associate a tab group implementation with a configuration type. Any time a config of the specified type is selected by the user, the class named by the class attribute will be used to provide the content of the tabbed folder.

The `ReflexTabGroup` declares a group of tabs that will be included in the tab group associated to Reflex applications. It reuses all Java tabs:

- `JavaMainTab` containing the general info on the application E.g. its project and its main class
- `JavaJRETab` holding the path to the Java Runtime Environment which is to be used for running the application



Create, manage, and run configurations

Name: Cast Test

Architect configuration File: C:\Program Files\eclipse\workspace\reflex\test\test\cast\config-architect.xml

Architect configuration File: C:\Program Files\eclipse\workspace\reflex\test\test\cast\config-assembler.xml

Main class Arguments:

VM arguments:

Working directory:

Local directory:

[✓ Workspace:] Reflex

[✓] Use default working directory

Figure 7.1: Screen shot of the Reflex launch Tab group

- `JavaClasspathTab` housing the class path information
- `JavaSourceLookupTab` containing the information where the attached source code is to be found
- `CommonTab` providing extra configuration possibilities

The only new tab is `theReflexConfigTab`. And still, that tab extends the standard `JavaArgumentsTab`. We only overwrote some of its methods for covering with two new fields: the name of the Architect Configuration file and the name of the Assembler Configuration file.

7.4 Defining launch shortcuts

As useful as the configuration tabs are, there are many times when users do not want to bother with them. They simply want to launch a program. This is where *launch shortcuts* come in. A launch shortcut allows users to identify a resource in the workbench (either via selection or the active editor) and launch that resource with a single click without bringing up the configuration tabs. Launch shortcuts are useful when the user is happy with default values for all config attributes and wants to launch in a hurry. This will therefore only work when there are an architect and assembler file – which are following the naming convention – specified in the same directory as the application the user wants to run.

As the previous paragraph stated, it is not mandatary to have launch shortcuts, but it is clear that they make user's live easier by allowing them to create a config, set all attributes to default values and launch that config in a single mouse click. The following XML code gives an idea on how the declaration of the Reflex shortcuts in four perspectives looks like.

```
<extension point="org.eclipse.debug.ui.launchShortcuts">
   <shortcut
      label= "Reflex"
      icon= "icons/reflex.bmp"
      helpContextId= "launcher.launch_shortcut"
      modes= "run, debug"
      class= "ReflexLaunchShortcut"
      id= "reflex.edt.reflexShortcut">
         <perspective id="JavaPerspective"/>
         <perspective id="JavaHierarchyPerspective"/>
         <perspective id="JavaBrowsingPerspective"/>
         <perspective id="DebugPerspective"/>
   </shortcut>
</extension>
```

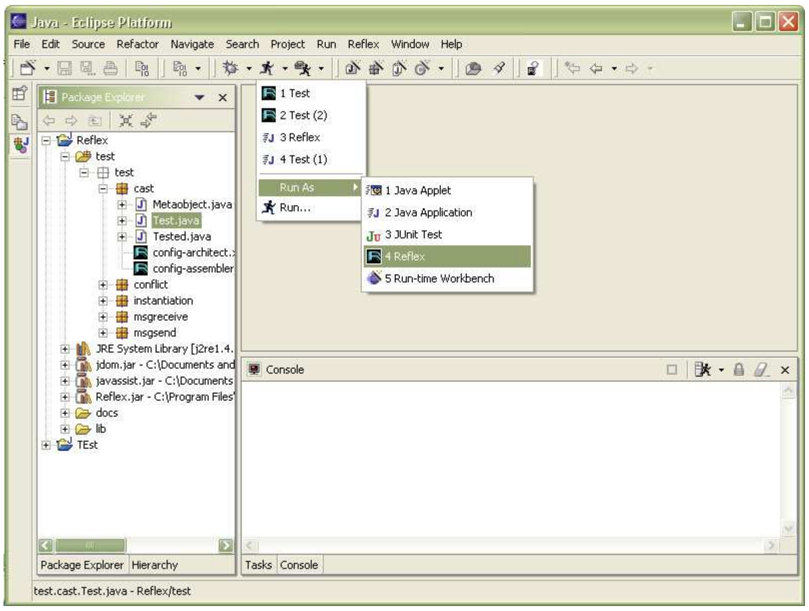


Figure 7.2: Screen shot of the Reflex launch Shortcuts

`ReflexLaunchShortcut` is a Java class that implements the interface `ILaunchShortcut`. It has twelve methods that cope with the fast launching of a Reflex application. Figure 7.2 shows a screen shot of the Reflex Shortcuts. The icons were also developed and linked to the Reflex launcher using the `plugin.xml` file. The related XML code is not included here, but is available in the open source code of the plugin. Shortcuts for debugging just work in a similar way and are therefor also left out of this written document.

7.5 Conclusion

This chapter discussed the different steps that were taken for developing a *launcher* and *debugger* for Reflex. A launcher is a set of Java classes that live in an Eclipse plug-in that performs launching. A debugger is an equivalent set that allows the programmer to run his applications in debug mode (step-by-step running, jumping into a method call, return, ...).

This can be split up in three steps. First we must define the new launch configuration type. Secondly, we must define the tab group of the launcher. This tab group presents a graphical interface for allowing the user to state his launching configuration. The last step is an optional step which we chose to take in order to provide a better service to the user. It consists in defining launch shortcuts, which are useful if the user is in a hurry.

Each of those steps is executed in two phases. In the first phase, we must define the extension points in the `plugin.xml` file – as always in developing an Eclipse plugin. These declarations always refer to some Java classes, which are implemented in the second phase.

Chapter 8

Hook detection

This chapter explains the part of the Eclipse plugin that makes it possible for the user to automatically detect the points where hooks will be inserted in the base code. It also allows the user to jump to the metaobjects associated with the detected hooks.

8.1 Introduction

A Reflex application typically has three program layers. First there is the base level, then there is the activation level, and finally there is the metalevel 8.1. While the base level houses the ordinary program, the metaclasses are defined on the metalevel. The remaining and intermediate level – the activation level – is used for allowing temporal selection. The activation API offers services to control the activation of reifications during execution (provided that they have been declared as *activatable* in the static configurations).

As explained before, when a base level class is loaded, Reflex verifies whether that class must be reified. If yes, hooks to the metalevel are inserted in the base code. When the program execution reaches such a hook, the activation of that hook is checked, and – if activated – program execution is given to the associated metalevel object.

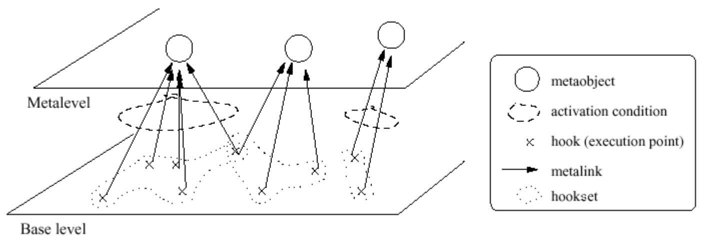


Figure 8.1: The different Reflex program layers

The places in the base program where the hooks are inserted, are crucial to the programmer. As the program execution might shift to the metalevel at those points. Therefore it would be very useful if we could mark those points and provide a way for the programmer to jump to the impacting metaobject code. These functionalities are similar to the ones provided by AJDT, where the join points are marked and a jump action is provided for jumping to corresponding advices 3.3.2.

8.2 Detecting hooks

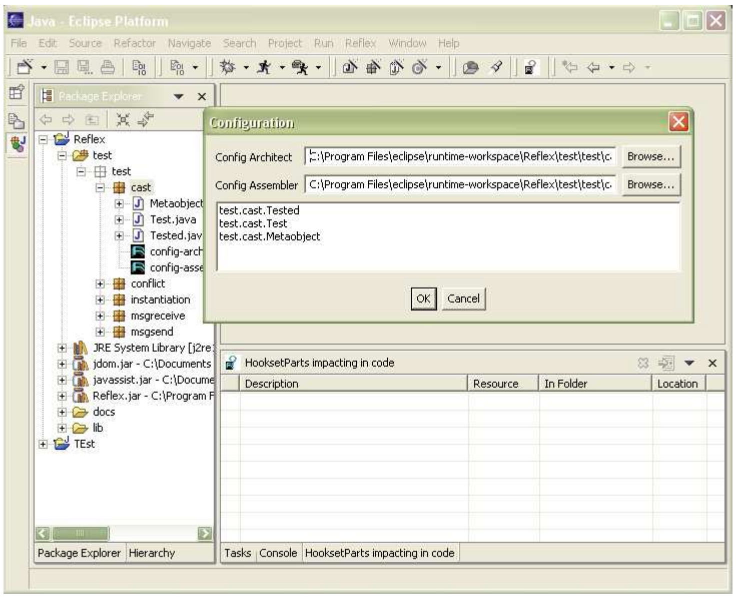
Detecting hooks in a base program is implemented by an Eclipse *action*. The action defines a new menu item and an icon, and the actions that have to be taken when the action is selected for execution.

8.2.1 Declaring the Eclipse action

As always in developing an Eclipse plugin, we are starting by declaring the extension in the `plugin.xml` file. The following XML extract shows how the "show hooks" action was declared.

```
<extension point="org.eclipse.ui.actionSets">
   <actionSet label="Reflex Hookset Parts"
              visible="true"
              id="reflex.edt.xmleditor.actionSet">
      <menu label="Reflex" id="ReflexMenu">
          <separator name="ReflexGroup"></separator>
      </menu>
      <action
          label="&amp;Find Reflex hooks"
          icon="icons/find.bmp"
          tooltip="Show points where hooks are inserted"
          class="reflex.edt.actions.ShowHooks"
          menubarPath="ReflexMenu/ReflexGroup"
          toolbarPath="ReflexMenu"
          id="reflex.edt.actions.ShowHooks">
      </action>
   </actionSet>
</extension>
```

Above code correctly declares a new popup menu item and an icon, on which the user can click in order to execute the action `ShowHooks`. Next subsection discusses which algorithm is applied in the `ShowHooks` class.


Config Architect: C:\Program Files\eclipse\runtime-workspace\Reflex\test\test\c

Config Assembler: C:\Program Files\eclipse\runtime-workspace\Reflex\test\test\c

```
test.cast.Tested
test.cast.Test
test.cast.Metaobject
```

Figure 8.2: Screen shot of a hook detection.

8.2.2 Defining the associated detection code

When the user wants to detect hooks, he must first select some Java entities (classes, packages or projects) for which he wants to perform such detection. Then the user has to click on the actions icon or menu item in order to start the looking process.

The detection process starts with the generation of a list that contains all the classes that occurred in the selected entities, `theTypes`. Because hook detection depends on which configuration files are used, a question asking for a reference to both configuration files will be raised before actual detection is started. This is the point where the screen shot – shown in 8.2 – is taken.

The application will always try to suggest the use of some configuration files depending on which Java entities were selected by the user before starting the looking process. For instance, if a package was selected, and the package contained Reflex configuration files, they will be suggested.

Reflex has support for specifying a *hook collector*: an object that will be informed whenever a hook is detected. We use that functionality for plugging a hook collector called `AllHookCollector` into the Reflex framework. While running the application, and whenever a hook is to be inserted in the base code, the `newHook()` method of this `HookCollector` is called. That method will collect all the available data of the hooks and store it for later use.

As hooks are inserted when a class is loaded, we can only detect them

Reflex parameters	**Explanation**
`theArchitectFilePath`	The path to the architect configuration
`theAssemblerFilePath`	The path to the assembler configuration
`reflex.edt.actions.ClassLoader`	The main class of the base level application
`theTypes`	The list of all class names to load

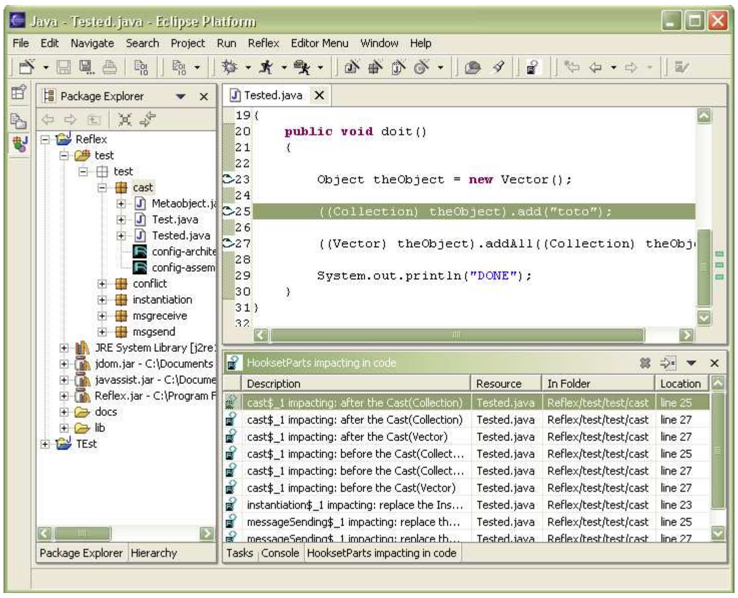
Table 8.1: Reflex configuration for hook detection.

at that time. Therefor we must make sure that all the classes for which we want to detect hooks are actually loaded. The `ClassLoader` class has one `main` method. It takes in a list of class names and tries to load the associated class for each of those class names. Using that class as the main class of the base level application, and passing the `theTypes` list to it, ensures that all the classes that were selected by the user, get loaded. Consequently all hooks of those classes are also detected and their information stored in the `AllHookCollector`. Table 8.1 shows the Reflex configuration for detecting hooks in certain classes.

When the `Classloader` finishes loading all classes that were passed to it, the application finishes. At that point, the `AllHookCollector` contains all information of the hooks that had to be inserted in the selected – and loaded – classes. This information can then be used for marking the lines in the base program code where hooks occurred, as we will see in the next section.

8.3 Marking hooks in the source code

The Eclipse workbench has a central mechanism for managing resource annotations. They are called *markers*. A marker is like a yellow sticky note stuck to a resource. It can record information about a problem. First we have to define the new marker we will be using. Then we associate an image to the defined marker to state how it will look like. Then we extend the standard Java editor with the needed behavior for handling the markers. And finally, we declare a view that will be able to list all markers of our new type. Figure 8.3 shows a screen shot of the marked hooks and the view that is listing all the detected hooks.


Java - Tested.java - Eclipse Platform
File Edit Navigate Search Project Run Reflex Editor Menu Window Help
Package Explorer
Reflex
test
test
cast
Metaobject.ja
Test.java
Tested.java
config-archite
config-assem
conflict
instantiation
msgreceive
msgsend
JRE System Library [j2re
jdom.jar - C:\Documents
javassist.jar - C:\Docume
Reflex.jar - C:\Program F
docs
lib
TEst
Package Explorer Hierarchy
Tested.java
HooksetParts impacting in code
Tasks Console HooksetParts impacting in code

```
19 {
20     public void doit()
21     {
22
23         Object theObject = new Vector();
24
25         ((Collection) theObject).add("toto");
26
27         ((Vector) theObject).addAll((Collection) theObj
28
29         System.out.println("DONE");
30     }
31 }
32
```

Description	Resource	In Folder	Location
cast$_1 impacting: after the Cast(Collection)	Tested.java	Reflex/test/test/cast	line 25
cast$_1 impacting: after the Cast(Collection)	Tested.java	Reflex/test/test/cast	line 27
cast$_1 impacting: after the Cast(Vector)	Tested.java	Reflex/test/test/cast	line 27
cast$_1 impacting: before the Cast(Collect...	Tested.java	Reflex/test/test/cast	line 25
cast$_1 impacting: before the Cast(Collect...	Tested.java	Reflex/test/test/cast	line 27
cast$_1 impacting: before the Cast(Vector)	Tested.java	Reflex/test/test/cast	line 27
instantiation$_1 impacting: replace the Ins...	Tested.java	Reflex/test/test/cast	line 23
messageSending$_1 impacting: replace th...	Tested.java	Reflex/test/test/cast	line 25
messageSending$_1 impacting: replace th...	Tested.java	Reflex/test/test/cast	line 27

Figure 8.3: Screen shot of the detected hooks.

8.3.1 Defining new markers

The definition of a new marker happens in the `plugin.xml` file. It holds information o its attributes. Attributes contain information that is held by the marker. The following code defines a new marker type extending the standard bookmark type.

```
<extension id="HooksetPartMarker"
          point="org.eclipse.core.resources.markers">
   <super type="org.eclipse.core.resources.bookmark" />
   <attribute name="PartId"></attribute>
   <attribute name="MetaLevelSourceLocation"></attribute>
   <persistent value="false"></persistent>
</extension>
```

As our new `HooksetPartMarker` extends the marker of type bookmark, it inherits all its attributes. The attributes `MetaLevelSourceLocation` and `PartId` and will be respectively used for storing the hookset part ID to which the hook belongs and the location of the associated metaobject source code.

The false persistent tag defines that the platform will not save the markers state between sessions. If one declares a markers'type as persistent, the state of markers of that type will automatically be saved and restored between sessions by the platform.

8.3.2 Associating an image to the marker

As markers are items to graphically assist the programmers, it is appropriate to provide a good image that explains what the marker stands for. The following XML snippet of the `plugin.xml` file declares the image that will be associated with markers of type HooksetPartMarker.

```
<extension point="org.eclipse.ui.markerImageProviders">
   <imageprovider
      markertype="reflex.edt.HooksetPartMarker"
      icon="icons/hook.gif"
      id="reflex.edt.HooksetPartMarkerProvider">
   </imageprovider>
</extension>
```

8.3.3 Extending the Java editor

In order to cope with the markers behavior (displaying them and jumping to the metalevel code), it is needed to declare a new editor or extend the existing Java editor. The following code declares the new editor and associates files with the ".java" extension to it.

```
<extension point="org.eclipse.ui.editors">
   <editor
      name="Reflex Java Editor"
      default="true"
      icon="icons/jcu_obj.gif"
      extensions="java"
      contributorClass="editors.MultiPageEditorContributor"
      class="reflex.edt.javaEditor.ReflexJavaEditor"
      symbolicFontName
         ="org.eclipse.jdt.ui.editors.textfont"
      id="reflex.edt.javaEditor.ReflexJavaEditor">
   </editor>
</extension>
```

The `ReflexJavaEditor` extends the `CompilationUnitEditor` (the standard Java editor). Because displaying markers is already supported by the standard Java editor, no new definition of an editor would be needed if no jumping behavior would be required. But – as we will see in the next section – we need to reference this editor for providing the jumping behavior.

8.3.4 Listing all markers in a view

In the Eclipse platform a view is typically used to navigate a hierarchy of information, open an editor, or display properties for the active editor. We will use the Eclipse view to list all the detected markers and its associated hooks information.

```
<extension point="org.eclipse.ui.views">
   <category name="Reflex" id="reflex.edt"></category>
   <view
      name="Hooks/Conflicts in code"
      icon="icons/find.bmp"
      category="reflex.edt"
      class="reflex.edt.views.HooksetPartsView"
      id="reflex.edt.views.HooksetPartsView">
   </view>
</extension>
```

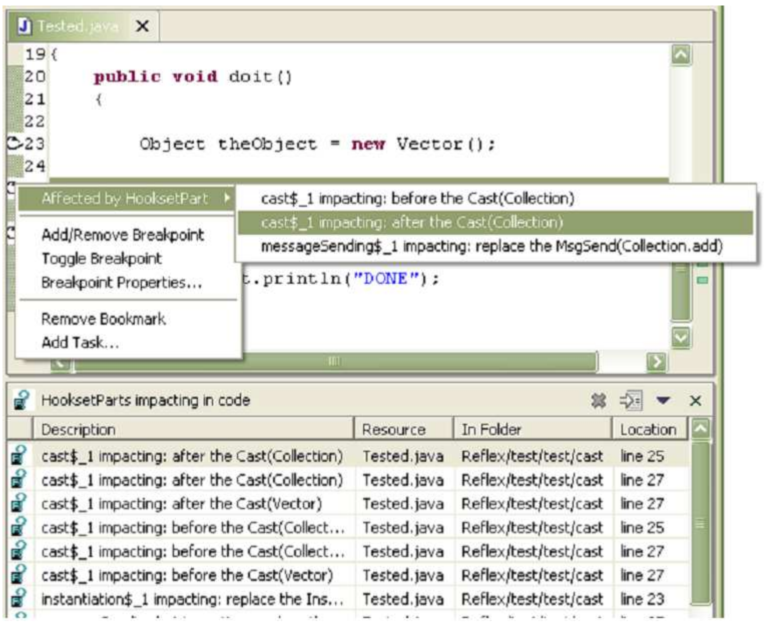
The above stated code snippet from the `plugin.xml` file correctly declares the new view. The `HooksetPartsView` is responsible for declaring the view's behavior. As the behavior is just the same as the standard bookmark viewers behavior, the definition is not that complicated. The only difference lies in the information that is displayed in the list, as we can see in figure 8.3.

8.4 Jumping from marker to associated code

When the user has detected all the impacting hooks in his base code, he can right click the markers to jump to the impacting metalevel code. This is a practical option for both developing and debugging purposes. In Eclipse, such a functionality is established by the use of an editor action. The following `plugin.xml` code shows how we associated the extra action to the `ReflexJavaeditor` – which we defined in the previous section.

```
<extension point="org.eclipse.ui.editorActions">
   <editorContribution
      targetID="reflex.edt.javaEditor.ReflexJavaEditor"
      id="reflex.edt.javaEditor.HooksetPartRulerActions">
      <action
        label="ImpactingHook"
        class="reflex.edt.javaEditor.
                           HooksetPartActionDelegate"
        id="reflex.edt.HooksetPartActionDelegate" />
   </editorContribution>
</extension>
```

When the user right clicks in the ruler (the left margin of the editor, where the markers are situated) the `HooksetPartActionDelegate` takes control. It has two important methods. The first one – `menuAboutToShow()`


Tested.java
Affected by HooksetPart
cast$_1 impacting: before the Cast(Collection)
cast$_1 impacting: after the Cast(Collection)
messageSending$_1 impacting: replace the MsgSend(Collection.add)
Add/Remove Breakpoint
Toggle Breakpoint
Breakpoint Properties...
Remove Bookmark
Add Task...
HooksetParts impacting in code

```
{
    public void doit()
    {

        Object theObject = new Vector();

t.println("DONE");
```

Description	Resource	In Folder	Location
cast$_1 impacting: after the Cast(Collection)	Tested.java	Reflex/test/test/cast	line 25
cast$_1 impacting: after the Cast(Collection)	Tested.java	Reflex/test/test/cast	line 27
cast$_1 impacting: after the Cast(Vector)	Tested.java	Reflex/test/test/cast	line 27
cast$_1 impacting: before the Cast(Collect...	Tested.java	Reflex/test/test/cast	line 25
cast$_1 impacting: before the Cast(Collect...	Tested.java	Reflex/test/test/cast	line 27
cast$_1 impacting: before the Cast(Vector)	Tested.java	Reflex/test/test/cast	line 27
instantiation$_1 impacting: replace the Ins...	Tested.java	Reflex/test/test/cast	line 23

Figure 8.4: Screen shot of a user who is about to jump to one of the impacting hooks associated code.

– is called to see if this action delegate wants to influence the menu before it is displayed. In the case of our plugin, we have to check if there is a hook marker affecting on the line in which the user has right clicked. If there is one, we add an '*affected by*' line to the context submenu that will appear.

By going through the submenu and selecting an impacting hook, the user will force the editor to jump to a particular file and location – as the `run()` method will be invoked by that. First we will use the information of the `MetaLevelSourceLocation` attribute for building a Jump marker that can be passed to `openEditor()` to force Eclipse to open the right source file and jump to the right location in that file showing the exact method that is impacting. Figure 8.4 shows a screen shot of a user which is about to jump to one of the impacting hooks associated code.

8.5 Conclusion

This chapter explained the development of the plugin part that is responsible of detecting the impacting hooks in the base code. We started out motivating that this part of the plugin is really useful for a programmer to see where the metalevel will take control.

After that, we explained how we manage the impacting hook detection. First we declared a new Eclipse action, which can be executed to start up the lookup process. Then we explained how the marking is done, using newly defined markers. The discovered hooks are also listed in a view, showing the information that is held by the hooks marker.

Finally we explained that the users are able to jump to the metaobject code which will be executed when that hook is reached. This option is proved to be useful, as it fastens up both debugging and development.

Chapter 9

Conflict detection and resolution

This chapter starts by introducing what hookset conflicts are. After that, we present the solution we provide for coping with them. We tackle the composition problem by defining a language and framework that allow programmers to compose conflicting hooksets. This solution was borrowed from the literature, as we could see in Chapter 3.

This chapter ends by explaining the part of the tool support that implements the detection of configuration conflicts and provides support for resolving them.

9.1 Introduction

Hooksets are a reference to a group of specific points during a program execution where you wish to take some extra metalevel action, as we have seen in 4.3.2. It might occur that – at some point in the program – several hooks are to be inserted at the same place. In such a case conflicts might occur and a correct hook composition should be established for avoiding unexpected application behavior. For coping with that, we contributed in the development of a composition framework and language.

The tool support for doing the composition lies in the detection of conflicts and the suggestion of a composition rule for coping with the conflict. This chapter ends with an explanation on how the conflicts are detected and how we support resolving them.

9.2 Theoretic approach

In section 4.3.3, we have seen that hooksets are composed of hookset parts. It is on that level of granularity that we will detect and solve conflicts.

9.2.1 Hookset conflicts

Just like in [34], two parts are said to be conflicting if they are impacting on the same program point. This definition might be too strict, because it might easily occur that two hookset parts that are impacting on the same point are not resulting in a real conflict at all. Imagine we have two hookset parts that are impacting on the same operation e.g. cast. While the first one is adding some behavior before the cast, the second is adding some extra behavior after the cast. In such a case, there is not really a conflict. Though, our approach would still see this as a conflict. But that is not bad, because it is perfectly possible that the programmer might want to specify a special composition in such a case: only adding the behavior before the cast, only adding the behavior after the cast, adding both behaviors or adding none.

9.2.2 Hookset composition

We chose to develop a composition language and framework, because this solution was already successfully applied in the literature (see 3.6) and it lies the closest to our needs. The programmer will be able to use the composition language in the static configuration files in order to resolve hookset conflicts.

The composition language

Appendix A shows the complete grammar of the composition language as it now exists. Next to this, the basic keywords – *if* and *then* should be used in the rules for separating the condition and the action parts of the rule.

The *IF* part of the rule states an *identifier_declarator*: a list of hooksets and/or hookset parts that must be impacting for the rule to apply. The shape of identifying a hookset part is: $h4\$p1$. As the $ symbol is used as a separator, it might not be used in the hookset nor part id's.

The *THEN* part of the rule states an *expression*: a composition of the impacting hookset parts. Table 9.1 shows which operators might be used in the composition language. The *THEN* part might also reference a Java class which is implementing the `Composer` interface. In that case, that class will be responsible of solving the occurring conflict. Its `resolve` method takes in a `Composition` object and outputs an adapted `Composition` object. That object typically holds three lists of hookset part id's, which are all impacting on the affected operation. While the first list holds the id's of the parts impacting before the operation, the second one has holds the id's of the parts that are impacting after the operation. The third list holds the part id's which metaobjects are going to replace the affected operation.

At the end of the composition definitions, the programmer must specify a default rule. Again in that rule, the programmer is free to use the composition language or to refer to a Java class. The only difference is that this

Function	Definition
`seq(H1,H2)`	First apply $H1$, then apply $H2$
`skip(H1)`	Don't do $H1$'s actions – implicit: we can just drop $H1$ from the THEN part
`wrap(H1,H2)`	Wraps the actions of H1 around the actions of $H2$. This is only useful if they are of a before-after control
`rule(H1,H2)`	Apply the action of the rule that handles $H1$ and $H2$

Table 9.1: Hookset composition functions and their explanations

rule should cover all possible conflict cases, as it will be applied in case no other rules matched the conflict situation.

The composition framework

It is on the assembler level that we will allow the programmer to declare hookset part compositions using the composition language.

The compositions can be declared in simple rules – using the composition language – or by referring to a Java class that will be responsible for the composition. Next code sample shows the XML code of a composition example. The most specific matching rule in a conflict situation will be exerted. If no matching rule was found, the default composition rule will be applied. The programmer must make sure that this rule covers all possible conflict situations. If not, the standard `seq` operator will be used and a warning will be shown to the user.

```
<composition>
   <rule
      if="h1,h2$p2"
      then="SEQ(h1, h2$p2)" />
   <rule
      if="h6$p1, h7, h8, h9"
      then="SEQ(h6$p1, WRAP(h7, h8, h9))" />
   <rule
      if="h1,h2$p2,h3"
      then="SEQ( WRAP( RULE(h1, h2$p2), h1), h3)" />
   <rule
      if="h1$p1, h2"
      then="h1$p1" />
   <default composer="test.conflict.MyComposer"/>
<composition>
```

The above example shows the declaration of four basic rules and one default rule. The condition part of the rules lists a number of occurring hooksets (e.g. $h1$) and/or hookset parts (e.g. $h6\$p1$). $h6\$p1$ refers to the hookset part with id $p1$ from the hookset with id $h6$. Knowing that, we can easily deduce that the first rule implies something like:

"If the second part of hookset $h2$ *and any part of hookset* $h1$ *are impacting on the same point of execution, then we first execute the code of the metaobject associated with the part of* $h1$ *and then the associated code of the metaobject of the second part of* $h2$*."*

Once all composition rules are declared in the configuration file, the programmer can start his reflective program. When a class is needed for the first time, it gets loaded and instrumented with the impacting hooks (see 4.3). When several hooks have to be inserted at the same point, a `Composition` object is created. It has three lists: a before list, a replace list and an after list as we can see below.

```
Composition {
   before: h1$p1, h2$p2
   replace:
   after: h1$p1, h2$p2
}
```

Basically this object says that there is a conflict between $h1\$p1$ and $h2\$p2$. Then the composition rules are exerted. When there is a rule that matches the conflicting entities, that rule gets executed over the `Composition` object. It returns an adapted `Composition` object with three ordered lists, stating how the hooksets parts have to be composed. The resulting `Composition` object will be used for generating the hook code which will be inserted in the class before it gets actually loaded for use. Doing so, once the class is loaded, the composition is fixed and cannot be modified anymore.

When the execution control reaches that particular point, the related metaobjects will be given control of the execution in the order specified by the lists of the `Composition` object. First the metaobjects from the before list are given execution control. When they finished their execution, the same is done for the metaobjects from the replace list and the after list. The return value of the last executed replace metaobject body will be the return value of the reified operation.

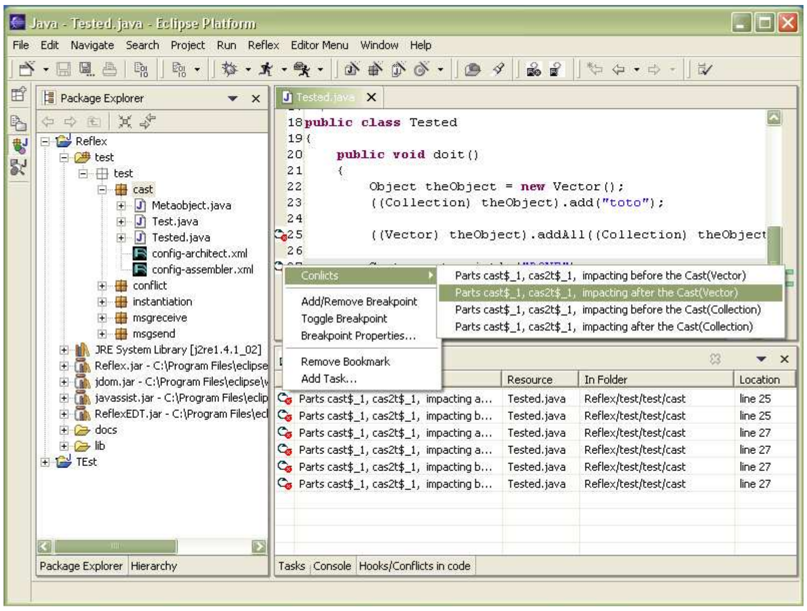
In case a reference to a Java class is made in the then part of rule (as in the last rule of the example), that object will be responsible for solving the matching conflicts. Therefore, it must implement the `Composer` interface. That interface has a `resolve` method which takes a `Composition` object as input, modifies that `Composition` object, and outputs the modified object, which will be representing the new composition.

9.3 Support for conflict detection

Conflict detection is implemented as an Eclipse action, just like hook detection. Indeed, conflict detection is an extension of the hook detection process presented in section (back ref). This extension aims at signaling conflicting hooks (i.e. acting on the same program point). Just like the detection of the hooks, the user will have to press a button in order to start the detection. Because the conflicts are detected by running the reflective program, first the Reflex configuration should be stated by referencing the two configuration files. Once that is done, the reflective program starts running and all the conflicts that are found are stored using a hook collector (as seen in section 8.2). On termination, the base-program points where conflicts occur are marked by *conflict markers.*

```
<extension point="org.eclipse.ui.actionSets">
   <actionSet label="Reflex Hookset Parts"
              visible="true"
              id="org.reflex.ui.xmleditor.actionSet">
      <menu label="Reflex" id="ReflexMenu">
         <separator name="ReflexGroup"></separator>
      </menu>
      ...
      <action
            label="&amp;Find Reflex conflicts"
            icon="icons/findC.bmp"
            tooltip="Show all conflicting hooks"
            class="reflex.edt.actions.ShowConflicts"
            menubarPath="ReflexMenu/ReflexGroup"
            toolbarPath="ReflexMenu"
            id="reflex.edt.actions.ShowConflicts">
      </action>
   </actionSet>
</extension>
<extension id="HooksetPartConflictMarker"
           point="org.eclipse.core.resources.markers">
   <super type="org.eclipse.core.resources.bookmark"/>
   <attribute name="AssemblerFileLocation"></attribute>
   <persistent value="false"></persistent>
</extension>
```

The previous XML snippet shows how the action and marker or defined. The three dots indicate the place where the previous defined action should be stated. The `ShowConflicts` class implements the behavior of detecting the conflicts and of marking the points in the base code.


Java - Tested.java - Eclipse Platform
File Edit Navigate Search Project Run Reflex Editor Menu Window Help
Package Explorer
Reflex
test
test
cast
Metaobject.java
Test.java
Tested.java
config-architect.xml
config-assembler.xml
conflict
instantiation
msgreceive
msgsend
JRE System Library [j2re1.4.1_02]
Reflex.jar - C:\Program Files\eclipse
jdom.jar - C:\Program Files\eclipse\
javassist.jar - C:\Program Files\eclip
ReflexEDT.jar - C:\Program Files\ecl
docs
lib
TEst
Package Explorer
Hierarchy
Tested.java
Conlicts
Add/Remove Breakpoint
Toggle Breakpoint
Breakpoint Properties...
Remove Bookmark
Add Task...
Parts cast$_1, cas2t$_1, impacting before the Cast(Vector)
Parts cast$_1, cas2t$_1, impacting after the Cast(Vector)
Parts cast$_1, cas2t$_1, impacting before the Cast(Collection)
Parts cast$_1, cas2t$_1, impacting after the Cast(Collection)
Tasks
Console
Hooks/Conflicts in code

```java
public class Tested
{
    public void doit()
    {
        Object theObject = new Vector();
        ((Collection) theObject).add("toto");

        ((Vector) theObject).addAll((Collection) theObject
```

	Resource	In Folder	Location
Parts cast$_1, cas2t$_1, impacting a...	Tested.java	Reflex/test/test/cast	line 25
Parts cast$_1, cas2t$_1, impacting b...	Tested.java	Reflex/test/test/cast	line 25
Parts cast$_1, cas2t$_1, impacting a...	Tested.java	Reflex/test/test/cast	line 27
Parts cast$_1, cas2t$_1, impacting a...	Tested.java	Reflex/test/test/cast	line 27
Parts cast$_1, cas2t$_1, impacting b...	Tested.java	Reflex/test/test/cast	line 27
Parts cast$_1, cas2t$_1, impacting b...	Tested.java	Reflex/test/test/cast	line 27

Figure 9.1: Screen shot of a user who is about to resolve one of the detected conflicts.

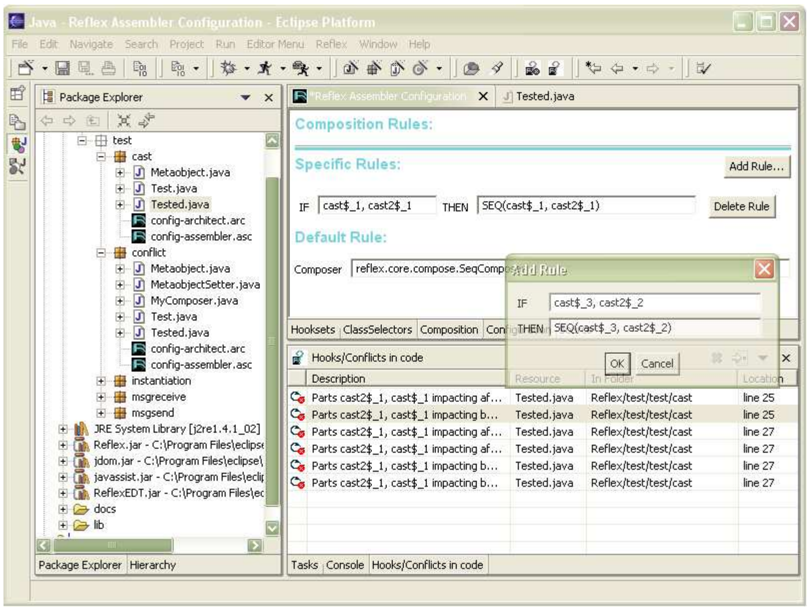
Just like all the impacting hooks, all the detected conflicts are listed in an Eclipse view. For that we use the same view as defined in section 8.3.4. Only the image that comes with every listed item is different as we can see in figure 9.1.

9.4 Support for conflict resolution

Conflicts can be resolved by using the developed composition framework presented earlier in this chapter. In that framework the user is expected to declare a composition rule in the assembler configuration file using the defined composition language. For allowing the user to do that, the assembler editor was extended.

9.4.1 The assembler editor

Until now, the assembler editor had four tabs (the hooksets tab, the class-selectors tab, the configuration tab and the source code tab). Each of them allows the assembler to cope with one of his tasks. As of now, the assembler has the extra task of defining the composition rules, an extra tab – *the composition tab* – was provided. It consists of a number of basic rules and of one default rule. Figure 9.2 shows a screen shot of the composition tab of the enhanced assembler editor.


Composition Rules:

Specific Rules:

IF cast\$_1, cast2\$_1 THEN SEQ(cast\$_1, cast2\$_1)

Default Rule:

Composer reflex.core.compose.SeqComp

Add Rule

IF cast\$_3, cast2\$_2

THEN SEQ(cast\$_3, cast2\$_2)

Hooks/Conflicts in code

Description	Resource	In Folder	Location
Parts cast2\$_1, cast\$_1 impacting af...	Tested.java	Reflex/test/test/cast	line 25
Parts cast2\$_1, cast\$_1 impacting b...	Tested.java	Reflex/test/test/cast	line 25
Parts cast2\$_1, cast\$_1 impacting af...	Tested.java	Reflex/test/test/cast	line 27
Parts cast2\$_1, cast\$_1 impacting af...	Tested.java	Reflex/test/test/cast	line 27
Parts cast2\$_1, cast\$_1 impacting b...	Tested.java	Reflex/test/test/cast	line 27
Parts cast2\$_1, cast\$_1 impacting b...	Tested.java	Reflex/test/test/cast	line 27

Figure 9.2: Screen shot of the enhanced assembler editor.

9.4.2 The marker action

Every conflict marker has an action attached to it. Executing that action, will open the assembler configuration file in the appropriate editor. Figure 9.1 shows a screen shot of a user who is going to resolve a detected conflict.

A new rule will be suggested to the user covering all the hooks that were conflicting at that point. The user can either accept that rule and save the new assembler configuration file, or modify the rule using the composition language defined earlier in this chapter. Figure 9.2 shows the rule that was suggested for resolving that conflict.

9.5 Conclusion

This chapter started by introducing what hookset conflicts are. Basically, when more then one hook has to be inserted at the same point of the base program, we say we have a hookset conflict.

Resolving those conflicts consists in providing hookset compositions, stating the order in which the different hooks are inserted. A composition language (containing four composition operators) was defined and a composition framework was developed for allowing the programmer to declare those compositions and for letting the system act properly upon those declarations.

Finally, we explain the part of the tool support that implements the detection of those configuration conflicts and provides support for resolving them. Detecting the conflicts is implemented as an Eclipse action, which can be executed to start up the lookup process. The conflicts are then marked in the source code and listed in the adequate view.

Every conflict marker has an action attached to them for resolving that conflict. When the user chooses to resolve a certain conflict, a composition rule is proposed. The user can then accept that rule or modify it according to his needs.

Part III

Dynamic tool support

Chapter 10

Runtime monitoring

This chapter consecutively discusses the motivation for, the requirements of and the development of a runtime monitor. The monitor is able to observe both standard object-oriented and reflective applications.

10.1 Introduction

Modern software development is inconceivable without tools to inspect running programs. Runtime inspection covers not only exhaustive querying of program state but also controlling its execution, e.g., pausing and resuming. Tools range from debuggers, tracing, test, and monitoring tools to program comprehension and reverse engineering tools. This chapter discusses the development of a tool that is able to monitor running programs. This dynamic monitor is able to observe both basic and Reflex programs.

10.2 Requirements for runtime inspection

An interactive environment for runtime inspection has several requirements related to software visualization issues [43]. Although the core of this chapter is not about visualization itself (these aspects would require more research), we believe at least two important issues deserve early consideration: visual load, and synchronization.

As soon as we are interested in realistic applications, a tough issue in visualization is the control of the visual load, that is, the possibility to ensure that not too much information is displayed at a given time, so that the user cognitive charge is not excessive and the user can avoid getting lost. A user should be able to select what exactly is of interest to him and possibly be given the chance to adjust the visualization layout. Another important requirement deals with the synchronization between the executing application and the inspection environment. Our approach is based on a synchronous means of control [57]. The idea here is to provide the user with a feeling of direct interaction with the running application, offering the possibility to suspend execution, to adjust settings or interact with the application, before resuming execution. To sum up, it seems fundamental to provide a very fine-grained control over what is inspected, along with a high-level of interactivity.

10.3 Monitoring an object-oriented program

According to the above-mentioned requirements, we are convinced that Reflex is particularly well-suited to provide an interactive environment for runtime inspection. In order to back up that statement, we developed a monitoring tool that uses Reflex for allowing runtime inspection.

Visually, the user gets confronted with a Graphical User Interface (GUI hereafter). This multi-windowed system permits visual load to be controlled (e.g. windows can be minimized, resized, closed and moved). Through the GUI, the user can interact with the runtime API of Reflex. Among other things, this API can be used to dynamically assign new activation conditions to hooksets and to change metaobjects associated with hooksets both at the desired granularity level (i.e., object, class, hookset, or global). In addition to static configuration files, this makes it possible for the user to precisely control, down to the finest granularity level, which parts of the system and which particular execution points will be observed/manipulated. In Reflex, those points are grouped into hooksets. To every hookset we attach a small control window which, at present, simply offers terminal-like output and basic interaction features. As of now, these features include inspection of hookset arguments and control over the activation of a hookset (at the various levels: object, class, hookset and global). We can simply activate or deactivate a monitor, but we can also make the activation depend on the activation of the underlying level (hereafter, `SUB`). For instance, if the activation of a class-scope hookset is set on `SUB`, the activation of each class – affected by that hookset – will be evaluated in order to decide whether hooks in that class are active. It is even possible to make activation totally custom, as one can specify its own activation object on a certain level.

For each hookset, we have a window responsible of monitoring its operations. Depending on the scope of the hookset, we are using different types of windows.

- For a hookset-scope hookset, the window is a simple hookset-scope window that monitors all operations affected by the hookset.
- For a class-scope hookset HS, we use a window containing a class-scope window for each of the class `C` affected by that hookset. Every one of those windows is assigned to monitoring the operations of instances of the corresponding class `C` affected by hookset HS.

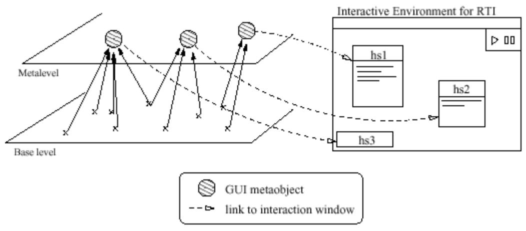


Figure 10.1: Monitoring an object-oriented program

- For an object-scope hookset HS, there are three levels of nested windows. The hookset-scope window contains a class-scope window for each affected class `C`, which contain a object-scope window for each instance `O` of class `C`. Those windows monitor all operations of the corresponding object `O`, which are instances from a matching class `C` affected by the hookset HS.

Closing a window on a certain level closes all its nested windows and stops the monitoring of that hookset, class or object. This allows visual load to be limited as the user can really select what he wants to monitor. Figure 10.1 shows how the interface looks like. Obviously, further versions could illustrate program dynamics in a more elaborated fashion.

Synchronization between the user and the running application can be controlled thanks to the fact that base-level operations are reified in order to give control to the metalevel. At the base level, we are running the Java program we want to monitor. At the metalevel, there is one metaobject for each hookset which is responsible for the monitoring of the set of operations specified by that hookset. Every time an operation – we want to monitor – occurs at the base level, the execution control will be passed to the corresponding metaobject. When that metaobject gets control, it monitors the base level operation and gives back the control to the base level. That way – whenever an operation occurs we are monitoring – the interactive environment can synchronize with the running application. For instance, if the user asks to suspend the program execution, this suspension will take effect upon the next entrance to the metalevel – that is to say when a monitored operation occurs.

10.4 Monitoring a reflective application

As we have seen before, Reflex main goal is supporting separation of concerns (SOC) through behavioral reflection. Indeed, it can serve as a generic platform for aspect-oriented development as argued in [72]. In such a case, an application runs at the base level while crosscutting and/or non-functional

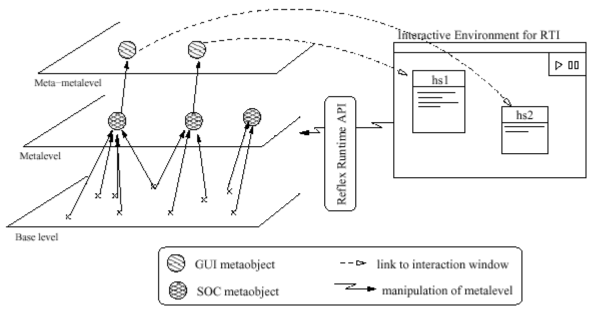


Figure 10.2: Monitoring an aspect-oriented program

concerns are implemented modularly as metalevel entities (SOC metaobjects).

In order to help in the prototyping, development, and debugging of such a concern metalevel, our approach to runtime inspection could provide a valuable support. In such a scenario, the inspection environment actually runs at the meta-metalevel, allowing for the manipulation of the metalevel 10.2. The interaction with the runtime API of Reflex would then be much more powerful than in standard application inspection, since it would not simply include hookset activation and basic synchronization features, but would also provide the means to dynamically change the bindings between base-execution points and SOC metaobjects.

Note that plugging a metalevel on top of an existing metalevel does not raise any problem since metaobjects are implemented with standard Java classes, and are thus also subject to the selective Reflex reification.

10.5 Conclusion

In this chapter we first motivated why runtime monitoring is indispensable for good software development. We have seen that the requirements for a good monitoring tool, are twofold. First there is the visualization aspect, and secondly, there is the synchronization aspect. In order to develop a good tool, both aspects must be taken into account.

Then we saw how the developed tool copes with the above mentioned aspects. Visualization can be controlled by allowing the user to open, close, resize and minimize the different monitor windows. That way he can precisely choose what information is displayed, and what is not. Synchronization is established by allowing the users to pause, continue – at a chosen speed – and step through monitored applications.

Finally, we have shown that the recursivity of the model of behavioral reflection makes it possible to apply such a graphical tool to inspect a metalevel layer implementing crosscutting and/or non-functional concerns. This would certainly prove highly useful in assisting the prototyping, development and debugging of applications making use of runtime SOC.

Part IV

Evaluation and Conclusion

Chapter 11

Evaluation

In this thesis, tool support was developed for assisting programmers in the process of creating and testing dynamic and reflective programs. In this chapter we evaluate the problems and limitations of our work. Note that while Java – and in particular Reflex – was used for the case study, the underlying thesis concepts are language-independent.

11.1 Static tool support

The static tool support was created in the shape of an Eclipse plugin, as Eclipse is the most used Java interactive development environment in the research community. This part of the tool support incorporates both wizards and editors for coping with the Reflex configuration. Both editors and wizards have their advantages and drawbacks. While the wizards are more easy to use, the editors provide faster ways to modify or create configuration files.

11.1.1 Configuration editors

When the user is modifying a configuration file and is changing between two editor tabs, following algorithm is followed to ensure that the resulting configuration will never be inconsistent.

The first step consists in making backups of all the configuration data: the *hooksets*, the *classSelectorSets*, the *composition rules* and the corresponding XML code. This is done as a security measure for being able to cope with possible problems later on. Then, we generate the XML code that is equivalent to the graphical configuration, and save it to disk.

At that time we want to verify that the resulting configuration is not conflicting, so we parse the configuration file using the appropriate Reflex parser. If it manages to finish without any conflicts, all the widgets of the target tab are recreated and filled with the new configuration data. Note that the recreation of all widgets is needed as it is possible that the number of items (*Hooksets*, *ClassSelectorSets* or *Composition rules*) may have changed, modifying the amount of widgets to be created on that tab page.

If the parser does not finish parsing successfully, we restore the configuration data from the backups we made and warn the user that a configuration error was detected. Using this algorithm, the user will not be able to change between tabs until the error was corrected.

Currently, the widgets themselves are used to store the configuration data. This has a serious drawback, as it makes us update all tabs and their widgets, on every page change. Obviously this has a negative impact on the performance of the editors. However, adding an external model for storing the configuration data, would make us create new parsers or at least change the existing Reflex parsers.

11.1.2 Hook detection

Another dimension of the static support lies in the automatic detection of the places where shifts from the base to the metalevel will occur. At those points, the user can follow a link that leads to the source code of the associated metaobject, allowing faster development and debugging.

In order for being able to mark the detected hooks, the hook collector needs to obtain the *localization* information of those hooks. Sometimes – if the reified operation does not implement the `LocalizableOperation` interface – that information is not available to the hook collector. In that case, the user gets warned that there was a certain number of hooks with an unknown location. The user is also told that the reified operations he uses should implement the `LocalizableOperation` interface in order to be localizable. That interface, part of the Reflex framework, has four methods which are each able to retrieve some localization information on the location of the reified operation occurrence:

- `getFileName()` returns the simple file (without the full path) name of the file where the operation occurs (or null if not available)
- `getLineNumber()` returns the line number in the file where the operation occurs (or -1 if not available)
- `getPackageName()` returns the name of the package where the operation occurs
- `getSimpleClassName()` returns the simple (without package) name of the class where the operation occurs.

In Reflex, a certain hook always has a metaobject attached. Yet in some cases, a *metaobject setter* is assigned to initialize this metaobject at load time (e.g. the factory design pattern). In such a case, it is not possible to statically jump to the associated metaobject – as it is not known at development time. In that case we allow the user to jump to the metaobject setter definition. Nevertheless, more could be done for helping the programmers in this regard. In some simple cases, we might be able to determine the appropriate metaobject definition (e.g. if the metaobject setter systematically creates the same metaobject). However, such a task would require some kind of program analysis, out of the scope of this thesis work. Furthermore, in most cases, the metaobject will be retrieved dynamically, making it impossible to know statically what it will be.

11.1.3 Conflict detection and resolution

The last aspect of the static support is the hookset conflict detection and composition. In our approach, conflicts are said to occur when two or more hooks are to be inserted at the same point in the base program. This is an overstrict definition, though, as it is perfectly possible that those hooks will not raise a real conflict as we have seen in chapter 9. But of course, you are better off locating all potential conflict situations, and being able to make adjustments when it is still possible, than finding too few and being faced with unexpected behavior at runtime.

Nonetheless, the detection algorithm could be significantly enhanced so that it would no longer detect such border cases as actual conflicts. In [34] the authors distinguish *strong* and *weak* independence of concerns. Two concerns are said to be strongly independent if none of their hooksets have common hooks. Two concerns are said to be weakly independent if they have hooksets with common hooks but if the concerns themselves can be composed to a single concern. Using the same approach, we can redesign the metalevel and hookset definitions – maybe joining some of them – in order to avoid conflicts from being detected among weakly dependent concerns.

11.2 Dynamic tool support

The developed dynamic support lies in monitoring reflective, or ordinary object-oriented applications. For that we developed a runtime monitor using Reflex itself. The monitor copes with the two main criteria for runtime monitoring, which are coping with the information overload problem and ensuring synchronization between the monitor and the monitored application.

The information overload problem is countered by two features. The first feature is allowing the user to precisely declare what to monitor and what not. This will make monitoring a lot faster and stop unwanted information to be shown. Secondly, the users can organize the different monitor windows which are each monitoring some aspect of the application. Not a lot attention was given to the look-and-feel and the graphical layout of the monitor as it is a preliminary exploratory version which focussed more on functionality issues. Still, for improving the user experience, a good look-and-feel is indispensable.

Synchronization is permitted whenever a monitored operation occurs in the base program. So when a program is executing, and no monitored operation occurs, the monitor can never synchronize with the monitored program. This might seem as like drawback, but then, one can assume that the programmer should reconfigure the monitor for making it follow up on more operations.

Currently, the monitor does not have a lot of functionalities for allowing intercession on reflective applications as only the metaobject activation is controllable (permitting activation or deactivation of certain metaobjects). The development of an extended GUI that interacts with the Reflex runtime API would be very useful to that extent, as it would bring along more functionalities for runtime intercession in reflective applications.

Chapter 12

Conclusion and future work

12.1 Conclusion

The main goal of this thesis was to provide ways for assisting the programmer in the development of reflective applications using Reflex. For that, both static and dynamic tool support were provided.

The static support comes in the form of an Eclipse plugin and has four major parts, each providing assistance for some phases of the development process: base- and metalevel design, base- and metalevel implementation, assemblage of both levels and application tests. The plugin provides the appropriate editors and wizards for configuring Reflex and so assisting in both the first and third phase of the development process.

The second part of the plugin, provides a launcher and debugger for Reflex applications. A launcher is a set of Java classes that live in an Eclipse plug-in that performs launching. A debugger is an equivalent set that allows the programmer to run his applications in debug mode (step-by-step running, jumping into a method call, return, ...). This part provides support for the testing phase in the development process.

The third part of the plugin provides some support for both the implementation and testing phase, as it has functionalities for detecting hooks in base-level code and for jumping between base and metalevel objects where those hooks occur.

As conflicts can arise in this configuration, a mechanism for detecting and resolving those conflicts had to be established. We contributed in the development of the Reflex conflict detection and resolution framework. The last part of the Eclipse plugin offers the support for using that framework, assisting the programmer in the assemble phase.

The dynamic support was partially developed in the Eclipse plugin and partially as a stand alone monitor. The dynamic support of the Eclipse plugin lies in the debug functionality it provides in its second part. The other part of the dynamic tool support lies in the visualization of the applications execution. It is developed as a stand alone monitor that can either be used together with or separate from the static part. The monitor provides a great help for debugging ordinary object-oriented or reflective programs. In addition to this, it allows the user to observe and/or manipulate the behavior of an object at runtime to some extent.

Throughout this thesis, Reflex was used as a case study. By using it, providing tool support for it and comparing it to AspectJ, some bugs and shortcomings came up. Correcting those bugs, providing the tool support and extending the Reflex framework wherever needed, resulted in a more stable framework, with more covered programming features and with good tool support for helping Reflex users to develop and debug reflective applications.

12.2 Future work

As Reflex is an ongoing research, it was modified a lot during this research period. Those modifications lead to a new version, that will be presented at OOPSLA 2003 [72]. In the new Reflex version, the assembler configuration will have changed significantly. Both the wizards and the editors should be adapted for coping with those changes. The rest of the developed tool support is not affected by those changes.

Regarding the dynamic support, there are also some extensions we want to suggest for future work. Currently, the monitor offers some support for changing the behavior of the running application. This is rather limited, though, as it only consists in activating or deactivating metaobjects. Of course, Java does not permit a lot of changes to a running program, but more research could maybe show ways for doing that. Also the GUI of the runtime monitor can possibly be improved. More research on that topic will have to demonstrate any further possible applications.

Also, an implementation of partial behavioral reflection in Smalltalk would open many doors, as Smalltalk is better suitable for runtime changes. It would allow a very powerful runtime monitoring with a lot of possibilities for runtime modifications. Another application of this Smalltalk framework would lie in the runtime management of web services, permitting maintenance of web services without taking them offline.

Currently, the composition of conflicting hooks is done at load time. Providing ways for doing that at runtime would provide yet another way to dynamically modify the behavior of a running application. More research on that topic must demonstrate the further applications.

Bibliography

[1] Apostle. http://www.cs.ubc.ca/labs/spl/projects/apostle/.

[2] AspectC. http://www.cs.ubc.ca/labs/spl/projects/aspectc.html.

[3] AspectC++. http://www.aspectc.org/.

[4] AspectJ. http://aspectj.org.

[5] AspectR. http://aspectr.sourceforge.net/.

[6] AspectS. http://www.prakinf.tu-ilmenau.de/∼hirsch/Projects.

[7] Hyper/J. www.research.ibm.com/hyperspace/HyperJ/HyperJ.htm.

[8] Prose. http://prose.ethz.ch/Wiki.jsp?page=AboutProse.

[9] *Workshop on Metamodeling in OO*, 1995.

[10] Advancing the State-of-the-Art in Run-Time Inspection. Darmstadt, Germany, July 2003. http://www.st.informatik.tu-darmstadt.de/pages/workshops/ASARTI03/index.html.

[11] M. Akşit and B. Tekinerdoğan. Aspect-oriented programming using composition filters. In S. Demeyer and J. Bosch, editors, *Object-Oriented Technology, ECOOP'98 Workshop Reader*, page 435. Springer Verlag, July 1998.

[12] M. Akşit, K. Wakita, J. Bosch, L. Bergmans, and A. Yonezawa. Abstracting object-interactions using composition-filters. In R. Guerraoui, O. Nierstrasz, and M. Riveill, editors, *Object-Based Distributed Processing*, pages 152–184. 1993.

[13] Mehmet Akşit. Composition and separation of concerns in the object-oriented model. In *ACM Computing Surveys*, volume 28A, 1996.

[14] Mehmet Akşit. Issues in aspect-oriented programming. In *Workshop on Aspect Oriented Programming (ECOOP 1997)*, June 1997.

[15] Leewood Associates. Why we should use wizards. http://www.pcd-innovations.com/downloads/Leewood_Sample.pdf.

[16] Giuseppe Attardi and Antonio Cisternino. Template metaprogramming an object interface to relational tables. *Lecture Notes in Computer Science*, 2192:266–267, 2001.

[17] L. Bergmans and M. Akşit. Composing multiple concerns using composition filters. http://trese.cs.utwente.nl/composition_filters/.

[18] L. Bergmans, M. Akşit, and B. Tekinerdoğan. Aspect composition using composition filters. pages 357–382. Kluwer Academic Publishers, 2001.

[19] G.S. Blair, G. Coulson, A. Andersen, L. Blair, M. Clarke, F. Costa, H. Duran, N. Parlavantzas, and K. Saikoski. A principled approach to supporting adaptation in distributed mobile environments. *Symposium on Software Engineering for Parallel and Distributed Systems*, 2000.

[20] D.G. Bobrow, Gabriel R.G., and White J.L. The clos perspective, chapter clos in context - the shape of the design space. *MIT Press*, 1993.

[21] Noury Bouraqadi-Saâdani. Java et la réflexion. *Rapport interne, EMN*, 2000.

[22] Mathias Braux and Jacques Noyé. Towards partially evaluating reflection in java. In *Proceedings of the 2000 ACM SIGPLAN workshop on Partial evaluation and semantics-based program manipulation*, pages 2–11. ACM Press, 1999.

[23] Johan Brichau, Kim Mens, and Kris De Volder. Building composable aspect-specific languages using logic metaprogramming. In *Proceedings of the Generative Programming and Component Engineering (GPCE) Conference*, LNCS. Springer Verlag, 2002.

[24] D. Caromel, W. Klauser, and J. Vayssiere. Towards seamless computing and metacomputing in java. *Concurrency Practice and Experience*, 10, 1998.

[25] D. Caromel and J. Vayssiere. Reflections on mops, components, and java security. In *ECOOP 2001. Volume of LNCS.,Budapest, Hungary*, volume 2072 of *Knudsen, J.L.*, pages 256–274. Springer-Verlag, 2001.

[26] S. Chiba. Load-time structural reflection in java. In *ECOOP2000 - Object-Oriented Programming - 14th European Conference*, number 1850, pages 313–336. Springer-Verlag, 2000.

[27] Pierre Cointe, editor. *Proceedings of Reflection'99*, LNCS 1616. Springer Verlag, July 1999.

[28] François Nicola Demers and Jacques Malenfant. Reflection in logic, functional and object-oriented programming: a short comparative study. In *proceedings of the IJCAI'95 Workshop on Reflection and Metalevel Architectures and their Applications in AI Montreal*, volume 27, pages 29–38, 1995.

[29] Pierre-Charles David, Thomas Ledoux, and Noury M. N. Bouraqadi-Saâdani. Two-step weaving with reflection using AspectJ. In *Workshop on Advanced Separation of Concerns in Object-Oriented Systems (OOPSLA 2001)*.

[30] Kris De Volder. Aspect-oriented logic meta programming. In *Workshop on Aspect Oriented Programming (ECOOP 1998)*, June 1998.

[31] Kris De Volder and Theo D'Hondt. Aspect-oriented logic meta programming. In Cointe [27], pages 250–272.

[32] Kris De Volder, Tom Tourwé, and Johan Brichau. Logic meta programming as a tool for separation of concerns. In ECOOP-AOP00 [36].

[33] E. Dijkstra. The structure of THE multiprogramming system. *Communications of the ACM 11*, pages 341–346, 1968.

[34] Rémi Douence, Pascal Fradet, and Mario Südholt. A framework for the detection and resolution of aspect interactions. Course slides EMN, Irisa.

[35] Rémi Douence, Mario Südholt, and Pierre Cointe. A testbed for the design and implementation of aspect-oriented programming.

[36] *Workshop on Aspects and Dimensions of Concerns (ECOOP 2000)*, June 2000.

[37] Tzilla Elrad, Mehmet Akşit, Gregor Kiczales, Karl Lieberherr, and Harold Ossher. Discussing aspects of AOP. *Communications of the ACM*, 44(10):33–38, October 2001.

[38] J.C. Fabre, V. Nicomette, T. Pérennou, R.J. Stroud, and Z. Wu. Implementing fault tolerant applications using reflective object-oriented programming. *Proceedings of FTCS-25.*

[39] S. Feferman. Transfinite recursive progressions of axiomatic theories. *Journal of Symbolic Logic*, 27:259–316, 1962.

[40] Ibrahim M. H. Report of the workshop on reflection and metalevel architectures in object-oriented programming. *OOPSLA/ECOOP'90*, 1990.

[41] Jim Hugunin. The next steps for aspect-oriented programming languages (in java). *Xerox Palo Alto Research Center.*

[42] Frank Hunleth, Ron Cytron, and Christopher Gill. Building customizable middleware using aspect oriented programming. In *Proceedings of OOPSLA 2001 Workshop on Advanced Separation of Concerns in Object-Oriented Systems.* to appear, 2001.

[43] M. H. Brown J. Stasko, J. Domingue and B. A. Price. Software visualization. 1998. The MIT Press.

[44] G. Kiczales. Aspect-oriented programming. volume 28, page 154. ACM Press, 1996.

[45] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W.G. Griswold. Getting started with AspectJ. *Communications of the ACM*, 44(10):59–65, October 2001.

[46] Gregor Kiczales and Erik Hilsdale. Aspect-oriented programming. In *Proceedings of the 8th European Software Engineering Conference held jointly with 9th Acm Sigsoft Symposium on Foundations of Software Engineering*, page 313. ACM Press, 2001.

[47] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In Mehmet Akşit and Satoshi Matsuoka, editors, *11th Europeen Conf. Object-Oriented Programming*, volume 1241 of *LNCS*, pages 220–242. Springer Verlag, 1997.

[48] T. Ledoux. OpenCorba: a reflective open broker. In *Reflection'99, Saint-Malo, France*, volume 1616 of *LNCS*. Springer Verlag, 1999.

[49] T. Ledoux and M.N. Bouraqadi-Saadani. Adaptability in mobile agent systems using reflection. *Workshop on Reflective Middleware RM2000*, april 2000.

[50] Ken Wing Kuen Lee. An introduction to aspect-oriented programming. The Hong Kong University of Science and Technology.

[51] Chamond Liu. *Smalltalk, Objects, and Design.* Iuniverse.com, 1996.

[52] C. Lopes, E. Hilsdale, J. Hugunin, M. Kersten, and G. Kiczales. Illustrations of crosscutting. In ECOOP-AOP00 [36].

[53] Pattie Maes. Concepts and experiments in computation reflection. *PhD thesis, Artificial intelligence laboratory, VUB*, 1987.

[54] J. Malenfant and Pierre Cointe. Aspect-oriented programming versus reflection: a first draft. Position Statement for the OOPLSA 96 AOP meeting.

[55] H. Masuhara, S. Matsuoka, and A. Yonezawa. An object-oriented concurrent reflective language for dynamic resource management in highly parallel computing. *IPSJ SIG Notes*, 94:18, 1994.

[56] J. McAffer. Meta-level programming with CodA. *Proceedings of the 9th Conference on Object-Oriented Technologies*, 952:190–214, 1995.

[57] Katharina Mehner and Awais Rashid. Towards a standard interface for runtime inspection in aop environments. In *Workshop on Tools for Aspect-Oriented Software Development (OOPSLA 2002)*, November 2002.

[58] Malenfant J. Cointe P. Mulet, P. Towards a methodology for explicit composition of metaobjects. In *Proceedings of OOPSLA'95*, pages 316–330. ACM Press, 1995.

[59] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Comm. ACM*, 15(12):1053–1058, dec 1972.

[60] R. Pawlak, L. Seinturier, L. Duchien, and G. Florin. Jac: A flexible solution for aspect-oriented programming in java. In Yonezawa and Matsuoka [76], page 2192.

[61] Andrei Popovici, Thomas Gross, and Gustavo Alonso. Dynamic weaving for aspect-oriented programming. In Gregor Kiczales, editor, *Proc. 1st International Conf. on Aspect-Oriented Software Development (AOSD-2002)*, pages 141–147. ACM Press, April 2002.

[62] R. Price. Real-world AOP tool simplifies OO development. *JavaReport*, September 2001.

[63] A. Rashid. Aspect-oriented and component-based software engineering. In *IEE Proceedings-Software*, volume 148, pages 87–88, June 2001.

[64] B. Redmond and V. Cahill. Supporting unanticipated dynamic adaptation of application behavior. In B. Magnusson, editor, *Proc. ECOOP 2002*, volume 2374, pages 205–230. Springer Verlag, June 2002.

[65] B.C. Smith. Reflection and semantics in Lisp. *Proceedings of the 14th Annual ACM Symposium on Principles of Programming Languages*, 1984.

[66] Jonathan Sobel and Daniel Friedman. An introduction to reflection-oriented programming. *Reflection 96*, 1996.

[67] Gregory Sullivan. Aspect-oriented programming using reflection. *OOPSLA 2001*, 2001.

[68] Joe Szurszewski. We have lift-off: The launching framework in eclipse. january 2003. http://www.eclipse.org/articles/index.html.

[69] E. Tanter, M. Vernaillen, and J. Piquer. Towards transparent adaptation of migration policies. *8th ECOOP Workshop on Mobile Object Systems*, 2002.

[70] Eric Tanter. Reflexión, metaprogramación y programación por aspectos. 2002. Slides available at http://www.dcc.uchile.cl/∼etanter/tanter-jcc02.pdf.

[71] Eric Tanter, N. Bouraqadi, and Jacques Noyé. Reflex: Towards an open reflective extension of java. In Yonezawa and Matsuoka [76], pages 25–43.

[72] Eric Tanter, Jacques Noyé, Denis Caromel, and Pierre Cointe. Partial behavioral reflection: Spatial and temporal selection of reification. *OOPSLA*, 2003.

[73] AspectJ Team. Programming guide. 2002. http://dev.eclipse.org/.

[74] Michael A. Trick. A tutorial on dynamic programming. 1997. http://mat.gsia.cmu.edu/classes/dynamic/dynamic.html.

[75] Christina von Flach G. Chavez, Alessandro Fabricio Garcia, and Carlos J.P. de Lucena. Some insights on the use of aspectJ and hyper/J. *Tutorial and Workshop on Aspect-Oriented Programming and Separation of Concerns*, August 2001.

[76] A. Yonezawa and S. Matsuoka, editors. *Metalevel Architectures and Separation of Crosscutting Concerns 3rd International Conf. (Reflection 2001), LNCS 2192*. Springer-Verlag, September 2001.

Part V

Appendix

Appendix A

Composition language grammar

```
Helpers
    letter = ['a'..'z'] | ['A'..'Z'] | '_';
    digit = ['0'..'9'];
    non_zero_digit = ['1'..'9'];
    letter_or_digit = letter | digit;

Tokens
    blank  = (' ' | '\t')+;
    comma = ',';
    l_par = '(';
    r_par = ')';
    dollar = '$';
    star   = '*';
    seq = 'SEQ';
    wrap = 'WRAP';
    rule = 'RULE';
    skip = 'SKIP';
    identifier = letter letter_or_digit* ;
    identifier2 = letter_or_digit+;

Ignored Tokens
   blank;

Productions
   expression =
      {name} name |
      {function} function;
```

```
name =
   {hookset} identifier |
   {hookset_part} identifier dollar [identifier2]:identifier;

function =
   {wrap} wrap l_par parameter_declarator r_par |
   {seq} seq l_par parameter_declarator r_par |
   {skip} skip l_par identifier_declarator r_par |
   {rule} rule l_par identifier_declarator r_par;

identifier_declarator =
   {identifier} name |
   {comma} identifier_declarator comma name;

parameter_declarator =
   {expression} expression |
   {comma} parameter_declarator comma expression;
```

Appendix B

Paper on the ASARTI workshop (held at ECOOP 2003)

ASARTI stands for Advancing the State-of-the-Art in Run-Time Inspection.

Citing the workshop organizers [10]: "Modern software development is inconceivable without tools to inspect running programs. Runtime inspection covers not only exhaustive querying of program state but also controlling its execution, e.g., stopping and resuming. Tools range from debuggers, tracing, test, and monitoring tools to program comprehension and reverse engineering tools. New applications incorporate runtime inspection as a programming concept in the style of event-condition-action rules. Configurable software, generative programming, plug and play components, and internet software such as web pages, applets or the WebServices standard emphasize the need to deal with runtime information at different levels of abstraction and the need to integrate heterogeneous runtime information. Lacking well-established models for representing and accessing program dynamics, tools must use ad hoc mechanisms. This limits reuse and interoperability. De facto standards for runtime inspection such as the Java Platform Debugger Architecture (JPDA) have clearly contributed to improve the situation but do not cope with all requirements. Implementors seeking to create debugging environments for ubiquitous computing are faced with even greater difficulties. The workshop seeks to identify best practice and common requirements, to specify conceptual data and control models for runtime inspection and to discuss practical issues such as standardized APIs and data exchange formats."