Crosscutting Revision Control System

Research Thesis

In Partial Fulfillment of the Requirements for the Degree of

Master of Science in Computer Science

Sagi Ifrah

The Research Thesis Was Done Under
the Supervision of Prof. David H. Lorenz
in the Dept. of Mathematics and Computer Science
The Open University of Israel

Submitted to the Senate of the Open University of Israel
Elul 5772, Raananna, August 2012

Acknowledgements

I thank my adviser Prof. David H. Lorenz, for his professional guidance, for many meeting that led us to publish a paper based on this thesis in ICSE 2012, for stimulating and invoking interests in several AOP issues, yet giving me the space to explore and research the domains I found interesting, and for his administrative support.

I am grateful to my brother, Shahar, and to Oren Mishali, AOP expert and a postdoc, who provided encouraging and constructive feedback.

I thank Gennady Agranov, Shai Koenig, Boaz Rosenan, and the anonymous reviewers for their useful comments. This research was supported in part by the Israel Science Foundation (ISF) under grant No. 926/08.

I would like to thank Dr. Anat Lerner and Dr Tal Hasner for many interesting and challenging meetings and discussions. I would like to thank Dr Miray Avigail, and Dr. Ela Zur for their administrative support, and to the staff of the Open University of Israel.

I wish to thank my dear friends for believing in me, and providing a great environment for growing.

I would like to dedicate this thesis to my parents Judit and Simon Ifrah that always encourage us to get high education, and to my beloved sister Hadas that starts her quest for high education the next year, good luck dear.

Abstract

Large and medium scale software projects often require a *revision control system (RCS)*. Unfortunately, a revision control system does not perform well with obliviousness and quantification found in aspect-oriented code. When classes are oblivious to aspects, so is the RCS. The crosscutting effect of aspects is not tracked, managed or controlled. The RCS does not track which revisions of a class were advised by which aspect revisions. Consequently, comparing revisions of a class does not reveal functionality changes caused by the advising aspects. The problem is aggravated by quantification, that enables a single piece of advice to advise multiple classes. Examining the differences in the functionality of the RCS for aspect-oriented programs versus its functionality for legacy programs without aspects reveals additional RCS-related setbacks. These setbacks endanger the development processes that rely on RCS services.

This work studies this problem in the context of using AspectJ (a standard AOP language) with Subversion (a standard RCS). We identify and analyze RCS-related setbacks, and offer a way to adapt the RCS for managing versions of AOP code. We introduce a *crosscutting revision control (XRC)* approach that features enrichment, persistence, comparison and display of crosscutting effects. The realization of the XRC approach is proposed through a flexible architecture based on the MVC architectural pattern.

Another contribution of this work is the implementation of an Eclipse plug-in (named XRC) that extends the JDT, AJDT, and SVN plug-ins for Eclipse to provide support for crosscutting revision control (XRC).

List of Publications

S. Ifrah and D. H. Lorenz. Crosscutting revision control system. In *Proceedings of the 34th International Conference on Software Engineering (ICSE 2012)*, June 2-9, 2012, Zurich, Switzerland, pages 321–330. IEEE, 2012.[18]

Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

Since the early days of *Aspect-Oriented Software Development (AOSD)* **(author?)** [11, 8, 24], the question "[h]ow do I know what aspect affects my code?" [19] has been asked. The question has been partially resolved by *aspect-oriented* enhancements to the *Integrated Development Environment (IDE)*. For example, AJDT (*AspectJ Development Tools)* [5] is an Eclipse plug-in for the AspectJ language [21, 22], that visualizes the "crosscutting effect" of aspects by displaying *markers* on a vertical ruler in the editor, to denote advice, inter-type declarations (itds), annotations, and softened exceptions. When hovering the mouse cursor over these markers, a *hint message* with additional information is displayed.

Unfortunately, AJDT does not support the tracking of these markers in previous versions of the software. Consequently, it is difficult to discover which aspects advised previous versions of a class. It is especially difficult to compare two versions and review any changes to the crosscutting effect.

Viewing a previous version ("*history*") and comparing versions ("*diff*") are two of the most common operations provided by a revision control system (RCS).[1] Use of RCS is very common in organizations that develop code, and is often integrated

[1]In this thesis we use the term RCS to refer to a revision control system in general, not to be confused with the specific RCS tool called *RCS* [32].

with the IDE. Indeed, the Eclipse IDE provides a GUI for common RCS services. In Eclipse, *diff* displays the text of the compared revisions of the file, highlighting the differences in a Compare view. However, the Compare view displays neither AJDT markers (advice, itds, annotations, softened exception) nor JDT[2] markers (break points, tasks, warnings, etc.). Needless to say, there is no support for comparing markers.

[2]JDT (Java Development tools) is an extension that enhances the Eclipse IDE with support for Java development.

1.1 AOP and Revision Control Systems

Many revision control systems follow the scheme found in Tichy's RCS tool [32] in managing revisions of programs as (text) documents organized in files. They store and display (textual) differences between successive revisions. They also offer facilities for merging parallel revisions by reconciling the differences. However, AOP (*Aspect Oriented Programming*) by nature defies these principles. First, concerns crosscut the file structure. Second, obliviousness [12] leaves certain crosscutting effects undetected in the (textual) display of files and changes. Third, quantification [13] complicates version dependency management.

For example, let C be a versioned class, and let A be a versioned aspect that advises C. Let C' and A' be newer versions of C and A, respectively. Assume that revision v_1 comprises the set of files $\{C, A\}$; and that revision v_2 comprises the set of files $\{C, A'\}$. Assume that revision $v_{2'}$ comprises the set of files $\{C', A\}$, modified in parallel to revision v_2; and that revision v_3 comprises the files $\{C', A'\}$, by merging revisions of v_2 and $v_{2'}$. Let $\Delta(v_1, v_3)$ denote the "delta" between the two revisions v_1 and v_3.

In a RCS, inspecting $\Delta(v_1, v_3)$ typically amounts to displaying $\mathit{diff}(C, C')$ and $\mathit{diff}(A, A')$, where *diff* is an external tool for comparing two text files. However, thanks to obliviousness, A' may have an effect on (the woven behavior of) C' that is not visible in $\mathit{diff}(C, C')$. Similarly, thanks to quantification, A' may have an effect on C' that is not visible in $\mathit{diff}(A, A')$. Let $\mathit{eff}(A, C)$ denote the "invisible" crosscutting effect that A has on C. Since neither $\mathit{eff}(A, C)$ nor $\mathit{eff}(A', C')$ are observable in code, the difference between $\mathit{eff}(A, C)$ and $\mathit{eff}(A', C')$ is, too, not observable. In addition, $\mathit{eff}(A', C)$ and $\mathit{eff}(A, C')$ may be of interest but are also "invisible."

1.2 Contribution

This work introduces XRC, which stands for *Crosscutting Revision Control.* The XRC approach. that was introduced in ICSE 2012 [18], provides better support for revision control of AOP programs, including an Eclipse prototype implementation. Intuitively, the core idea of XRC is to persist to the RCS the otherwise transient $\mathit{eff}(A, C)$ as *crosscutting metadata (XMD)* associated with C. This, combined with new RCS and IDE capabilities for displaying and comparing this XMD, provides crosscutting revision control for AOP programs that is currently unavailable.

The inherent difficulty in bridging RCS and IDE services for AOP stems from their respective biases. Traditional IDE support for AOP is associated with the build process (compilation) in order to provide the necessary weaving information, while RCS is associated with the storing and retrieving of unwoven files and is not aware of any weaving information.

The general problem may be broken down into four sub-problems: metadata mining and persistence, metadata comparison, versioned metadata display (viewing a previous version and displaying the differences), and integration with the software development process. As a proof-of-concept, we prototyped an XRC Eclipse plug-in for AspectJ and Subversion (SVN). Specifically, the work contributes:

- *Diff Engine.* We introduce a semantic *diff* that is adapted for comparing XMD. Obviously, a standard *diff* is not good enough, because the crosscutting effect is mostly invisible in the text.

- *Visual Enhancement*s. Because markers and hint messages are currently ignored by the RCS and by the IDE compare tool, ways to visually display a comparison with a previous version were never considered. Thus, new visual markers are introduced for displaying the differences between the compared XMD.

- *RCS–IDE Integration.* A traditional RCS treat the file as the unit of revision.

> However, XMD by essence crosscuts the file structure. For example, the XMD may change, while the text of the file remains unchanged. This requires to adapt the RCS–IDE processes for save, build, check-in, display version, compare versions, etc.

We provide a solution for XMD. However, the solution can be generalized to handling metadata that represents other kinds of markers. Support for save, compare, and display of metadata might be useful for errors and warnings as well, or any other metadata that is associates with the source code.

Chapter 2

Background

This chapter briefly reviews the terms, technologies, systems and components that are involved in the problem and in the solution.

2.1 Aspect Orientated Programming

Aspect Oriented Programming (AOP) is a programming paradigm which "aims to increase modularity by allowing the separation of Cross-Cutting Concerns"[1]. A *Cross-Cutting Concern* (CCC) is a concern that cross-cuts the system's functionality, such as logging, monitoring, and synchronization. Typically, the CCC implementation is scattered throughout the code, resulting in tangled code [23]. AOP was motivated by the following programming problems that accompany CCCs:

Scattered code. Multiple pieces of code scattered in multiple classes in order to handle a single concern. E.g., code of logging concern scattered in many classes that are designed for other concerns.

Tangled code. A single piece of code (e.g., a method) handles multiple concerns. Listing 2.1 illustrates the tangled code problem.

[1] *http://en.wikipedia.org/wiki/Aspect-oriented_programming*

```java
public void withdraw(int sum){
    log("withdraw starts");//Log: log start
    openDBConnection(); //DB: open DB connection
    checkUserPermissions();//Security: check permissions
    balance -=sum;//*** actual withdraw logic ***
    closeDBConnection(); // DB: close DB connection
    log("withdraw ends"); //Log: log end
}
```

Listing 2.1: **Tangled code is illustrated in a withdraw method.** The actual one line logic is tangled with code of other concerns.

AOP improves the *Separation of Concerns* (SoC) by introducing techniques for isolating and concentrating a CCC implementation into a single modular unit, named an *aspect*. *AspectJ* is an aspect-oriented extension to Java. AspectJ applies SoC by allowing to attach advice (method-like construct) of a specific CCC, to its pointcuts (collection of join points, well defined points in the execution of the program) [22]. AspectJ and Java constructs are used to centralize the CCC implementation in aspects. *Aspect Oriented Software Development* (AOSD) extends the AOP paradigm to the level of software engineering, with respect to the different activities of SW development (e.g., requirements, design, implementation, testing and integration).

2.1.1 Obliviousness and Quantification

AOP improves the SoC elegantly by combining obliviousness and quantification. *Obliviousness* implies that a class is unaware of its advising aspects, whereas *quantification* refers to the ability of a single piece of advice to affect multiple elements in the program [14]. Due to the oblivious nature of AOP, $\mathit{eff}(A, C)$ is hidden ("invisible") when viewing[2] the code of class C. Consequently, the developer who is editing C might not be aware that the aspect A advises it. Quantification leads to a similar problem with respect to A. When editing the code of A, it is hard to keep track all

[2]This section refers to viewing and editing text only, without the assistance of AJDT.

```
public aspect A {
    before(): execution(void *()){
        System.out.println("before: " + thisJoinPoint.toString());  }
}
```

Listing 2.2: **Obliviousness and Quantification**. The aspect A advises C without modifying the source of C, due to *AspectJ's obliviousness.* A single piece of advice in A advises multiple methods in several classes (that matches the signature void *()), due to *AspectJ's quantification*

its advised classes. Listing 2.2 demonstrates obliviousness and quantification.

2.2 RCS – Revision Control System

A *revision control system* (RCS), also known as a *version control system* (VCS) is used to track and manage project source files (and other project resources) as they evolve [31].

The idea of versioning [9] is simple: a *revision* is created each time a file is changed. The software and each file thus evolves as a succession of revisions. The most basic services RCS provides (since early 80s) are history, diff (also known as delta), multi-user management and merging facilities.

RCS enables multiple users to work on a software concurrently. The multi-user management may prevent overlapping changes by check-in and check-out mechanisms that acquire and release a lock on a file. The change management and tracking use versions and branches to follow and control the changes that occur on the code through the product evolution. RCS assists to evaluate the product quality on its different versions, document on which version a new feature or a new bug has been introduced, when has a bug been solved, and what version did it influence.

2.2.1 Subversion (SVN)

Subversion (SVN) is an advanced and free RCS. Its development started at 2000 by CVS developers, with the intention of replacing CVS. CVS have been released at 1990, it is an enhancement (and expansion) of RCS that was released in 1982. Every system is more advanced and sophisticated than its predecessor. RCS manages version for single files, CVS manages whole products, and Subversion adds advanced features such as supporting properties per file.

SVN comprises of a repository, and a server[3]. The repository stores the data, and the server provides the different services required from a RCS. The services include cat (view the contents of a revision of a file in the repository), diff (displays the differences between two revisions or paths) ,check-in, update, check-out, import, status and so on.

2.2.2 SVN properties

SVN supports version properties per file. A file may have properties, and SVN stores them with the file. From the SVN point of view, modifications to the properties are similar to modifications to the file, and the properties are checked-in with the file. This feature allowed XRC to attach metadata to the file easily, letting this metadata be versioned with the file.

2.2.3 RCS Support for Software Development Processes

RCS supports, among others, the following software development processes [31]:

Code review is done on previous versions of a file (via history), as well as on current versions as part of the change control. Code review [15] refers to the ability to

[3]http://svnbook.red-bean.com

examine the source code, in order to inspect functional and structural correctness or to investigate a failure.

Change control provides control over the changes that are checked-in to the RCS. Before code is checked-in to the RCS, it is considered a good practice to have the code reviewed by another programmer ("review before merge" policy [31, 2]). This is typically done by examining the files that have been changed. This practice keeps the RCS "safe" with relatively stable code, since all checked-in changes have been reviewed. RCS provides operations that support change control: (i) Synchronize with repository identifies the files that have been changed. (ii) Check-in the files that have been reviewed and approved.

Tracking code evolution is required for maintenance and debugging. It enables to track a change, and to understand the functionality changes over time. This means to find when was the functionality of a class changed (e.g. new, modified or removed functionality), when were a new feature or a new bug introduced, who made the change, and why. RCS provides operations that support change tracking: (i) Viewing the history versions of a file (since the evolution of code is reflected in its history). (ii) Diff (compare revisions) is useful for finding out about changes, and understand the differences between two versions. Eclipse displays the changes in the Compare view.

Revert to a stable revision may be required due to problems and instability in the current version. Tracking a class's evolution may help to find a previous stable version. Once such a version is found, RCS enables to revert to this version (using check-out).

Support team development is essential for teams. RCS might enable two developers to check-out and modify the same class. RCS will typically warn the second developer that will try to check-in, that his code is not updated, and allow him to update, merge or solve conflicts, in order to prevent overlapping changes.

2.3 Related Work

2.3.1 AspectJ Development Tools

Eclipse AspectJ Development Tools (AJDT) “provides tool support for editing, building, and debugging AspectJ programs on the Eclipse platform” [5]. The AJDT Eclipse plug-in adds to the IDE new capabilities for visualizing crosscutting effects. It computes the crosscutting metadata and introduces advice markers and hint messages about $\mathit{eff}(A, C)$ when editing the current version of C or A. Eclipse uses a vertical ruler in the editor in order to display these markers. Hint messages are displayed when rolling the mouse cursor over the markers. However, these markers are not saved with the file and thus not stored with checked-in revisions of C and A. AJDT displays this information only for the current version. When viewing previous versions of C and A, the markers (which are likely to be different, since different versions of aspect could advise different versions of classes) are not shown.

Interestingly, early versions of AJDT did include facilities for crosscutting comparison and changes, which are currently not functioning.

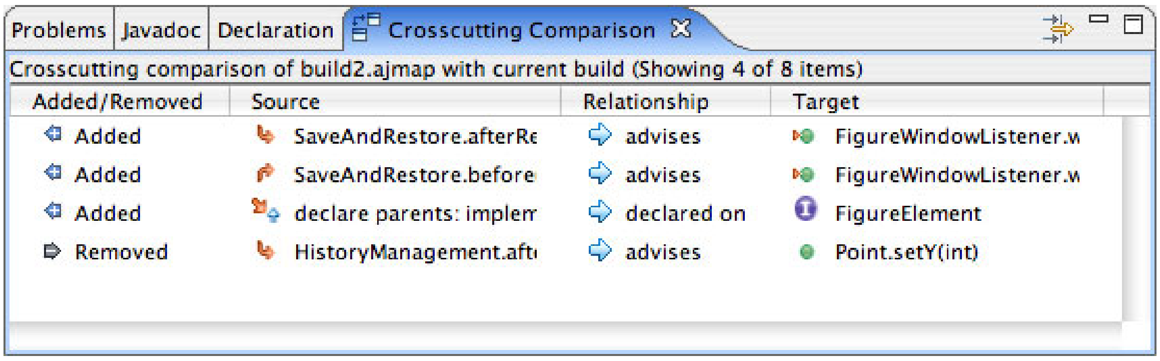
2.3.2 Obsolete Features of AJDT

2.3.2.1 Crosscutting Comparison

A *Crosscutting Comparison* capability was part of the AJDT 1.2.1 and 1.3 releases[4]. The feature was introduced as following:

> "Crosscutting Comparisons allows you to take a snapshot of the crosscutting relationships in your project and then compare the snapshot with relationships present in a later version of the project." [4]

[4]http://www.eclipse.org/ajdt/demos/comparison.html, http://www.eclipse.org/ajdt/demos/#COMPARISON-DEMO


Problems | Javadoc | Declaration | Crosscutting Comparison

Crosscutting comparison of build2.ajmap with current build (Showing 4 of 8 items)

Added/Removed	Source	Relationship	Target
Added	SaveAndRestore.afterRe	advises	FigureWindowListener.w
Added	SaveAndRestore.before	advises	FigureWindowListener.w
Added	declare parents: implem	declared on	FigureElement
Removed	HistoryManagement.aft	advises	Point.setY(int)

Figure 2.1: Crosscutting Comparison view

The results of the comparison were displayed in a special designated view, as shown in Figure 2.1 [4]

In AJDT 1.6.1, the crosscutting model enhancements[5], the AJDT's internal representation of the crosscutting model became redundant and removed in order to improve the performance of the edit, save, and build operations. The Crosscutting Comparison functionality and view could not work without this model, and thus the feature become obsolete, until it would be adapted to the new model and re-enabled.

In comparison to XRC, the AJDT Crosscutting Comparison obsolete feature had the following disadvantages. First, the Crosscutting Comparison view was not consistent with the Eclipse Compare view. It did not integrate the differences in a Compare view. Second, it managed all the relationships of the project in a single file. Specifically, it did not support to save or examine the differences per class or aspect. Third, it required to save the snapshots manually. Last but not least, it did not support the enhanced crosscutting model of Eclipse. Eclipse version 3.4 and higher only works with AJDT version 1.6.1 and later, which no longer supports this functionality.

[5]http://www.eclipse.org/ajdt/whatsnew161/index.html

2.3.2.2 Crosscutting Changes (Obsolete Feature)

AJDT 1.5 introduced another relevant feature, named *Crosscutting Changes*[6]. With this feature, as posted on march 07:

> "... advice markers are highlighted when the crosscutting has changed, such when a method is advised for the first time, or when there has been a change in the set of places affected by some advice. The reference point for the comparison can be changed using the new drop-down on the Cromsscutting Comparison view. The choices are to use the last build (of any type), the last full build, or a crosscutting map file in the project."[7]

This feature become obsolete as well, due to the crosscutting model enhancements in AJDT 1.6.1, until it will be adapted and re-enabled.

The Crosscutting Changes feature worked with the Crosscutting Comparison, which is obsolete. It did not support the resolution of versioned classes and aspects from a RCS.

2.3.2.3 Ideas Borrowed from AJDT

- AJDT introduces advice markers in order to display metadata on the aspect/-class relationships in current version. We reuse the AJDT markers in order to display the same kind of metadata also for viewing a previous version and when viewing on diff.

- As many other diff tools, Crosscutting Comparison displayed the essence of the change: was a piece of advice added or removed. We added in the Compare view a ruler and designated markers for this purpose (see add and remove markers in Figure 4.2).

[6]http://www.eclipse.org/ajdt/newfeatures/#ccchanges,
http://www.eclipse.org/ajdt/demos/AJDT-short-talk.html,
http://www.eclipse.org/ajdt/demos/#ECLIPSECON07

[7]http://www.eclipse.org/ajdt/newfeatures/#ccchanges

- Crosscutting Changes inspired us to reuse the highlight changed-advice marker.

2.3.3 Specialized Differencing Tools

Two of the main concerns of comparison tools are comparing special types of files, and tracking evolution by file versions comparison. For example, diffPDF[8] allows to compare PDF files textually or visually, diff-doc[9] compare MS Word/Excel/Powerpoint, HTML, XML, and other document files, and other tools such as Beyond Compare[10] compare images, directories, binaries, compressed files and more. The Crossutting Comparison described in Section 2.3.2.1 specialized in comparing crosscutting effects, yet it is inferior to XRC and is obsolete. Many tools support software evolution tracking via sophisticated features, such as showing in-line changes, merge tools, 3-way comparison, structured comparison, semantic differencing, identifying moving lines and others. Ldiff [3], is an enhanced *diff* tool that is capable of tracking line movement. Its output details in addition to regular diff output, the line numbers of text suspected to be moved to another line. Type-safe diff for families of datatypes [28] proposes a type-safe, structural diff-like library. It can compare tree-like data (e.g. the abstract syntax tree). SemDiff [6, 7] deals with the problem of identifying, analyzing, and understanding the evolution of code. SemDiff addresses client programs adaptations to the evolution of frameworks. It refers mainly to breaking a client program because of framework API changes. Lsdiff [29, 25] is a comparison technique and Eclipse plug-in that "identifies systematic structural differences as logic rules" [29]. It analyzes and displays the structural differences, helping to understand the essence of the change (refactoring, deleting a method etc.). iDiff [30] addresses a setback in file-based comparison. File-based comparison cannot handle well some of the program entities manipulation. iDiff utilize the interactions between program

[8]http://www.qtrac.eu/diffpdf.html
[9]http://www.softinterface.com/MD/Document-Comparison-Software.htm
[10]http://www.scootersoftware.com/index.php

entities to enhance the comparison.

Some tools reveals the software evolution using reverse engineering: UMLDiff [34] detects high level structural changes, and Aspect Oriented Development Visualization (AODVis) [26] uses Eclipse plug-ins for AspectJ reverse engineering and visualization.

The XRC supports evolution tracking of crosscutting effects. It shares with these tools the aspiration to enhance the simple diff textual comparison tools to a semantic diff, ignoring the text and order in favor of the structure and semantics. In contrast, XRC adds to the RC system repository new XMD information, which is readily available in the IDE but not tracked. This extra information may help mining tools and programmers detect more crosscutting inconsistencies and avoid potential bugs. The XRC diff is designated for XMD comparison, and have semantic and structural characteristics.

2.3.4 Crosscutting Configuration Management

Arimoto et al. address the Configuration Management (CM) of Crosscutting Framework (CF). They define CFs as “a type of Aspect-Oriented Framework, which includes only one crosscutting concern, such as persistence, distribution or security.” [1]. CFs may be reuse in different applications, have different versions, and similarly to regular components, requires versioning.

> “As there may be evolutions and modifications in both CFs and the application, producing different versions of this software, there must be an appropriate version control which manages the framework -or frameworks -versions, which are related to each version of the application.” [1]

TOFRA [1] is a tool that addresses this configuration management (CM) problem, and manages the dependencies of versioned CF for versioned applications. In comparison, XRC provides RCS support for AOP code at the file level. TOFRA may complement XRC and provide CM support at the CFs and application level.

Chapter 3

The Problem

The RCS support for AOP is incomplete, and deficient. The first section in this chapter reviews shortly the development process, emphasizing the roles of RCS and the effects on AOP in practice. The second section discusses naive solutions. To better understand the problem, it is illustrated in the third section. The last section analyze and explains the setbacks in the RCS support for in the AOP development process of.

3.1 RCS and AOP in a Practice

During the development process, usually several developers collaborate using RCS. Among other activities, they are programming, reviewing, testing and debugging code. Examining a typical development scenario might help to understand the RCS roles in the development process, and to find where and how AOP defies RCS.

Figure 3.1 depicts a sequence diagram for a typical development scenario. In this scenario, a developer has to do the following:

1. **check-out** or update the class files of a component she intends to work on,

2. work on and **modify** some source files,

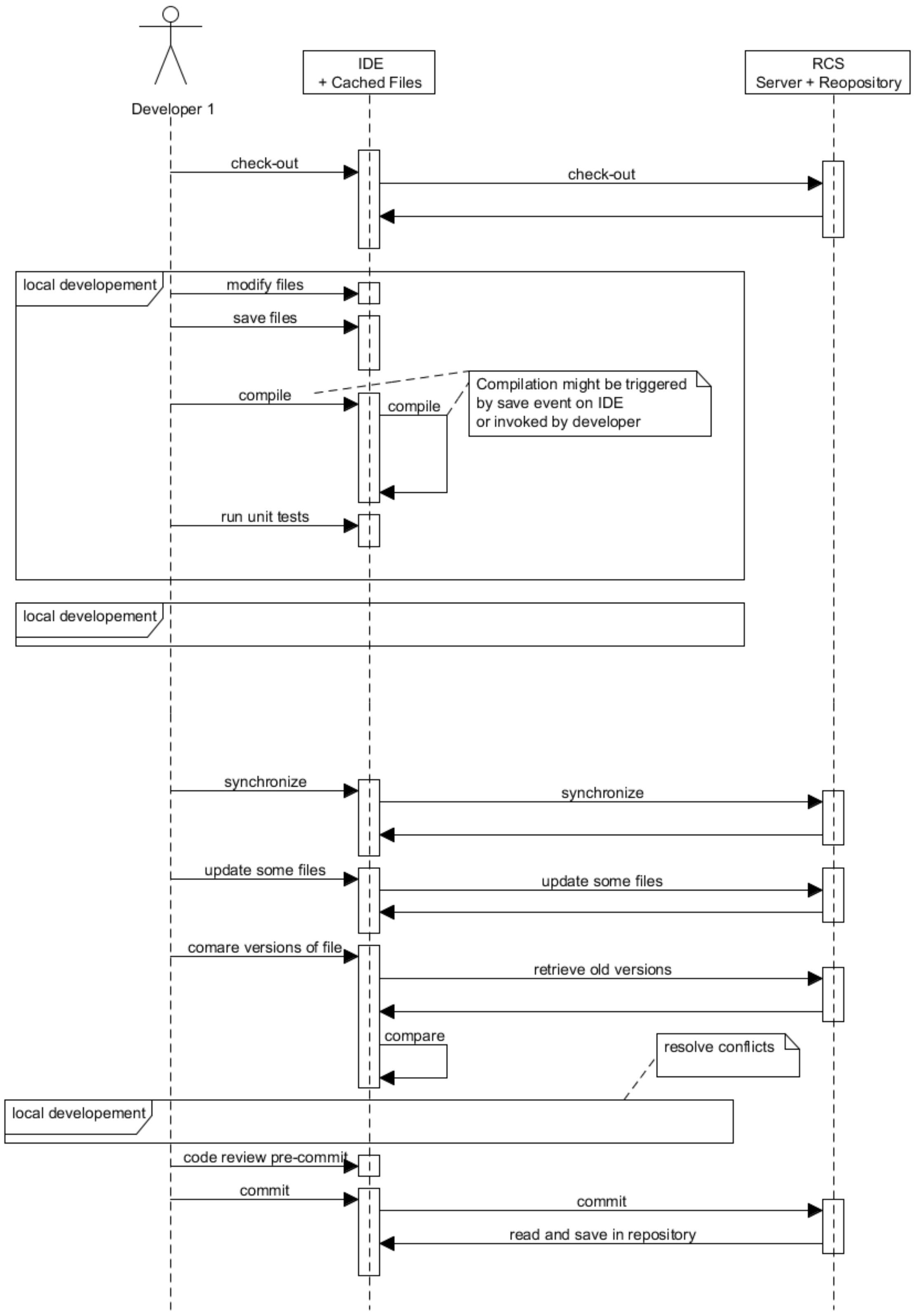


Figure 3.1: A typical development scenario (simplified)

3. **save** the files,

4. **build** and compile the source code (might be initiated by the developer, or by the IDE as response to the save event),

5. **test** her code by running unit tests,

6. repeat steps 2-5 several times as necessary, (concurrently, steps 2-5 might be done by other developers as well, on their own local working copies),

7. **synchronize** the files to see if they were modified by other developers,

8. **update** the files that don't conflict,

9. **handle overlapping code** and **solve conflicts** by **comparing files** with their previous versions, and by **code review** of previous versions with an emphasis on the differences, in order to understand the evolution,

10. repeat steps 2-5 again in order to solve conflicts (if necessary),

11. **review** the changed code, (ask also the team leader, or other developer to participate or perform code review),

12. **check-in** the changes.

To understand the role AOP plays in the typical development scenario, and the loss of support by RCS, let C_{local} be a class in the local working-copy, and let A_{local} be an aspect in the working-copy that advices C_{local}. In steps 2, 9, and 11, the developer might explore $\mathit{eff}(A_{local}, C_{local})$, aided by the AJDT advice markers displayed for A_{local} or for C_{local}. In steps 2 and 9, the developer might modify $\mathit{eff}(A_{local}, C_{local})$ by:

- Modifying A_{local} to advise differently. I.e, A_{local} would start or stop to advise a class (e.g., by modify a pointcut), or change the code of an advice. This modifies the crosscutting effect of C_{local} without modifying C_{local} file.

- Modifying C_{local} to be advised differently. I.e., modifying C_{local} in a way that A_{local} will start or stop to advise it (e.g., by renaming a method so it is no longer matches a pointcut), or change the code of the advised element in C_{local}. This modifies the crosscutting effect of A_{local} without modifying A_{local} file.

In step 7, the synchronization with the RCS repository finds the files that were modified textually in the repository or locally. RCS cannot identify files that their crosscutting effect was changed, but textually remained the same. In step 11, only modified files are being reviewed. A class or an aspect that was not modified textually, and its crosscutting effect was modified would probably be missed, and not be reviewed at all. In steps 2, 3, 4, AJDT updates the markers to reflect the aspects advice, as accurate as it can. However, neither the crosscutting effects nor the AJDT advice markers are persisted in the RCS. Consequently, comparing versions of a file, or displaying a previous version (in step 9) does not display any AJDT markers.

3.2 Naive Solutions

This section suggests naive solutions that may allow tracking and comparison of crosscutting effects.

3.2.1 Checking-in Woven Code

Save the woven class, in order to save and display the whole data. Saving the woven class in the RCS, means to save not only the class and aspect sources, but also the woven, compiled class. Obviously, the compiled class contains the woven $\mathit{eff}(A, C)$. This approach is complex and inefficient. The compiled class contains not only the woven$\mathit{eff}(A, C)$, but also C itself. Saving duplications implies for a waste. Assume we have C and the compiled and woven $C_{compiled}$ in order to extract $\mathit{eff}(A, C)$ we need to find the differences between C, $C_{compiled}$. This diff is non-trivial and even not well defined, since C is a source file, and $C_{compiled}$ is a byte-code file. Even if we were to solve it, and do it somehow in the source and not in the byte-code level (to avoid the woven $\mathit{eff}(A, C)$, and get the source of $\mathit{eff}(A, C)$), we now have the union of of all the aspects that advise C, $\bigcup\limits_{A} \mathit{eff}(A, C)$, and don't know which code came from which aspect. In order to find it, we need to do a kind of a smart comparison between $U_A \mathit{eff}(A, C)$ and the suspected aspects, and to be more precise, with the right version of them, or all the versions. This smart comparison should find for a versioned aspect A if it is one that caused part of $U_A \mathit{eff}(A, C)$. This will finally give the data required in order to display which versioned aspect is responsible for each piece of advice in C. This process is very complex, and has some black holes in it, and requires a lot of processing, thus a lot of time.

3.2.2 IDE Support (e.g., Eclipse AJDT)

AJDT introduces markers to hint on $\mathit{eff}(A, C)$ as described in Section 2.3.1. It assists the comprehension of the code, yet it supports only the current version, not the versioned files from the RCS. Section 2.3.1 describe two obsolete features that handled crosscutting changes and comparison. Fix or reestablish these feature might be considered as a naive solution to the problem. The limitations described in Section 2.3.1, clarify in detail why it is not a good comprehensive solution.

3.3 Illustration

To illustrate the problem and its consequences, consider a bank account application, comprising an *Account* class with several methods, among them a method named *withdraw*. A developer might misspell the name of the method as "withdr**o**w" ("**o**" instead of "a").

3.3.1 Scenario 1: Being unaware of an aspect

Suppose that another developer notices the typo and in the next revision, *Account′*, corrects the spelling everywhere, except for an occurrence in the code of an aspect (named *BlackListAdvice*), a reasonable oversight when using the *rename* refactoring tool in Eclipse. This oversight will often go unnoticed. On comparing the class source code to its previous versions,

$$\mathit{diff}(\mathit{Account}, \mathit{Account}')$$

the only visual difference is the spelling correction.

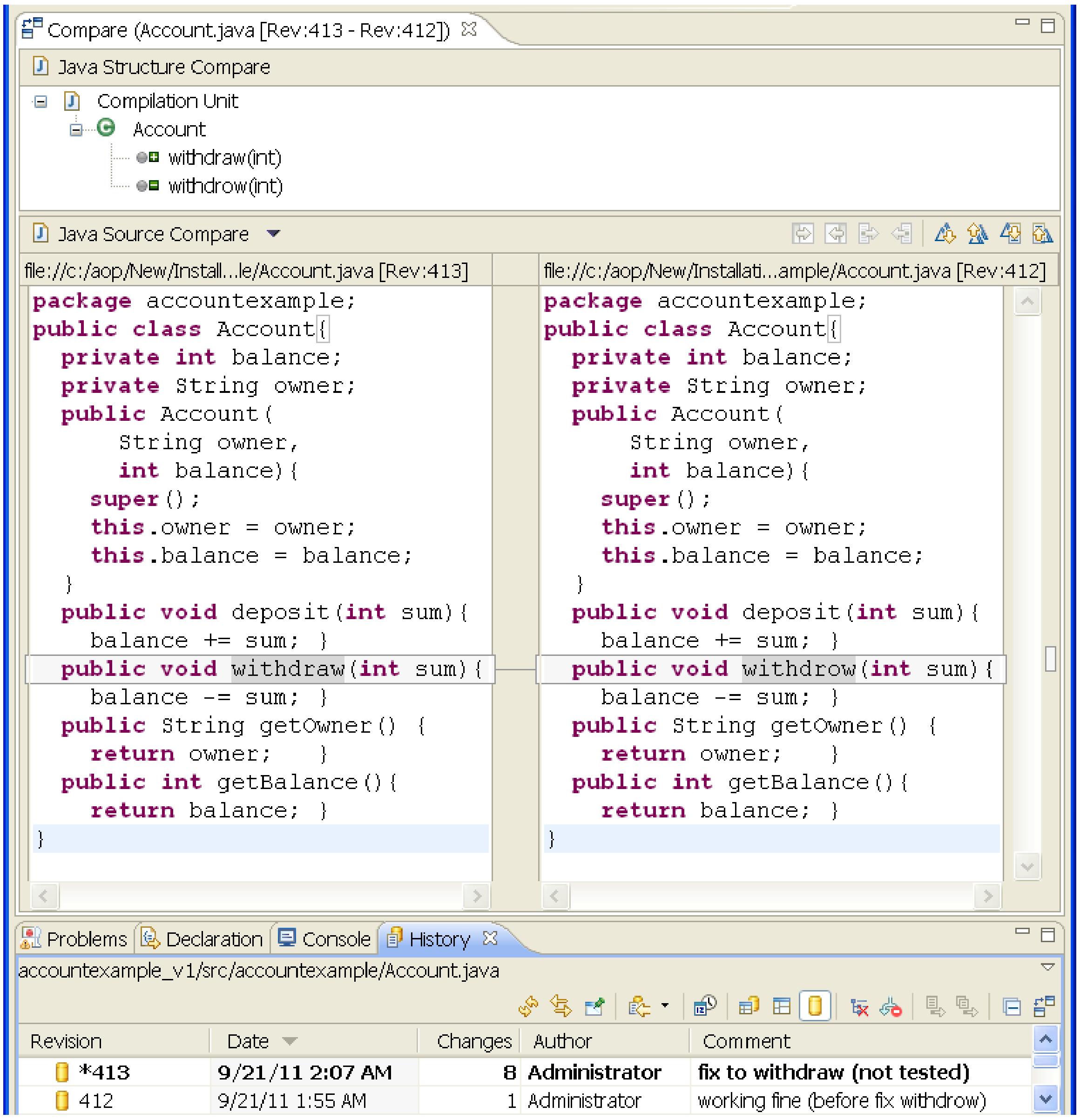
Figure 3.2 depicts the Compare view, comparing *Account* with *Account′*. There is no indication that an aspect affects the code or that the crosscutting effect has changed. Actually, there is no visible reason to expect a different behavior when executing the method.

Hopefully, if we were to use JUnit, a test might now fail and yield an error (otherwise this bug would be very hard to discover).

In Figure 3.3a, *Account* is shown with the typo:

1. AJDT markers indicate the aspect's crosscutting effect.

2. JUnit runs successfully, indicated by a green bar. A limited user was identified, and the account has \$50.



Compare (Account.java [Rev:413 - Rev:412])

Java Structure Compare

- Compilation Unit
  - Account
    - withdraw(int)
    - withdrow(int)

Java Source Compare

file://c:/aop/New/Install...le/Account.java [Rev:413]

```
package accountexample;
public class Account{
  private int balance;
  private String owner;
  public Account(
      String owner,
      int balance){
    super();
    this.owner = owner;
    this.balance = balance;
  }
  public void deposit(int sum){
    balance += sum; }
  public void withdraw(int sum){
    balance -= sum; }
  public String getOwner() {
    return owner;    }
  public int getBalance(){
    return balance; }
}
```

file://c:/aop/New/Installati...ample/Account.java [Rev:412]

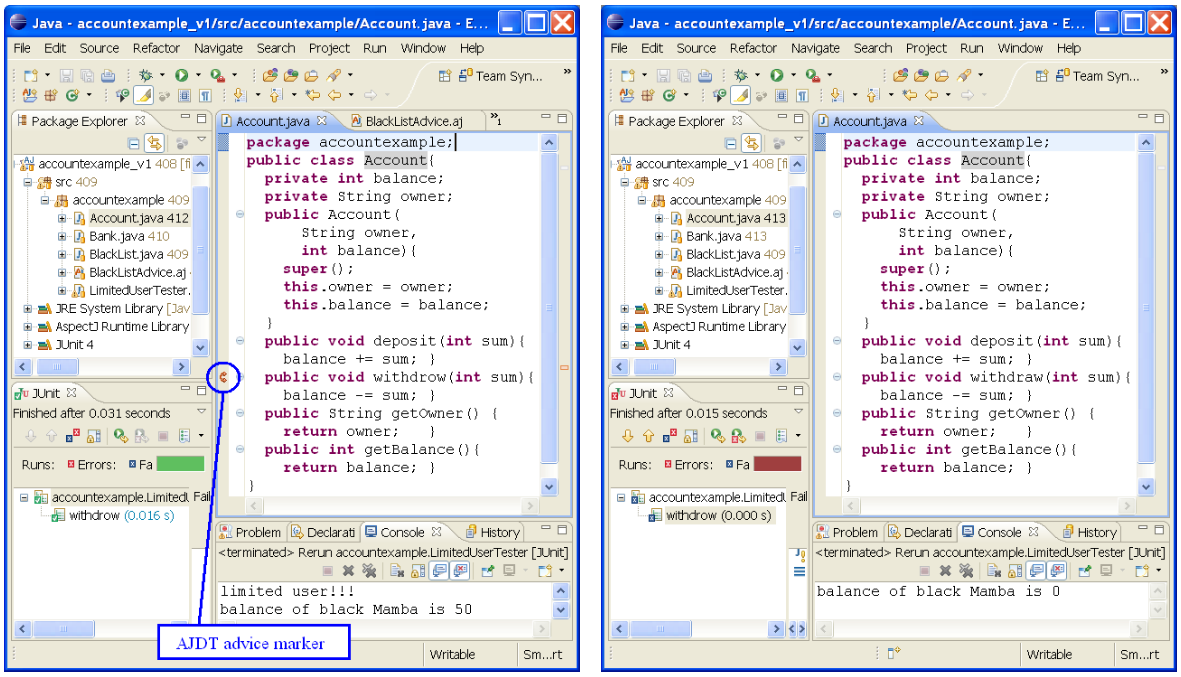
```
package accountexample;
public class Account{
  private int balance;
  private String owner;
  public Account(
      String owner,
      int balance){
    super();
    this.owner = owner;
    this.balance = balance;
  }
  public void deposit(int sum){
    balance += sum; }
  public void withdrow(int sum){
    balance -= sum; }
  public String getOwner() {
    return owner;    }
  public int getBalance(){
    return balance; }
}
```

Problems | Declaration | Console | History

accountexample_v1/src/accountexample/Account.java

Revision	Date	Changes	Author	Comment
*413	9/21/11 2:07 AM	8	Administrator	fix to withdraw (not tested)
412	9/21/11 1:55 AM	1	Administrator	working fine (before fix withdrow)

Figure 3.2: Diff between [Rev:413] and [Rev:412] of Account class. No markers are shown despite the existence of an aspect that advises “withdrow” in [Rev:412]



```
package accountexample;
public class Account{
  private int balance;
  private String owner;
  public Account(
      String owner,
      int balance){
    super();
    this.owner = owner;
    this.balance = balance;
  }
  public void deposit(int sum){
    balance += sum; }
  public void withdrow(int sum){
    balance -= sum; }
  public String getOwner() {
    return owner;    }
  public int getBalance(){
    return balance; }
}
```

```
<terminated> Rerun accountexample.LimitedUserTester [JUnit]
limited user!!!
balance of black Mamba is 50
```

```
package accountexample;
public class Account{
  private int balance;
  private String owner;
  public Account(
      String owner,
      int balance){
    super();
    this.owner = owner;
    this.balance = balance;
  }
  public void deposit(int sum){
    balance += sum; }
  public void withdraw(int sum){
    balance -= sum; }
  public String getOwner() {
    return owner;    }
  public int getBalance(){
    return balance; }
}
```

```
<terminated> Rerun accountexample.LimitedUserTester [JUnit]
balance of black Mamba is 0
```

(a) With the typo "withdr<u>o</u>w" in the method name (b) After correcting the typo in the method name

Figure 3.3: Account code displayed in Eclipse with the AJDT plug-in

In Figure 3.3b, *Account* is shown after correcting the typo:

1. There is no indication of the aspect's effect. Seemingly, everything should be fine. Note that also in the project view, there is no warning or error, since the situation is perfectly legal.

2. JUnit fails, indicated by a red bar. A withdraw that should have been rejected was committed successfully.

However, inspecting the test code in Listings 3.1, 3.2, 3.3 provides no indication as to why the test has failed, or, for that matter, to why the test has passed in the first place. Without the LimitedUserTester, the error would probably have been found too late, only when trying to get a limited user to repay debts that he probably cannot pay. Yet, even with the LimitedUserTester JUnit, we only see that a user was added to the black-list, and somehow withdraw should have been blocked with an error. There is nothing to assist the developer in finding and fixing the bug. There is

```java
public class LimitedUserTester{
@Test
public void withdraw(){
    String user = "black Mamba";
    Account account = new Account(user, 50);
    BlackList.getInstance().add(user);
    account.withdraw(50);//expect to display error,
                        //and avoid the withdraw.
    System.out.println("balance of "+account.getOwner()
        +" is "+account.getBalance());
    assertTrue(account.getBalance()==50);
    }
}
```

Listing 3.1: LimitedUserTester class after the fix

no indication (in the IDE) of any aspect advising the code. Should we discover the aspect that ceased to advise, only then the mysterious bug might be revealed.

3.3.2 Scenario 2: Fixing only the aspect

Suppose that the typo was fixed in the *BlackListAdvice* aspect (but not in the advised classes), and the aspect was checked-in to the RCS. Consequently, *BlackListAdvice* ceases to advise *Account*, resulting in an error that, similarly to the previous scenario, is not observed when inspecting the class *Account* or its revisions.

3.3.3 Scenario 3: Concurrent development

A similar scenario could occur in concurrent development. Assume two developers, Sagi and Dave, participate in a project. Sagi works on a Business Logic that is not a crosscutting concern, while Dave is responsible for the Security concern, which is a crosscutting one. Dave deletes an aspect or changes a pointcut that causes some target in the Logic domain to stop being advised. Dave checks-in his code. The check-in did not involved checking-in of Logic classes, although it changed their functionality. Sagi might then find out that his code is broken. It could be very hard for him to

```java
public class Account {
    private int balance;
    private String owner;
    public Account(String owner, int balance){
        super();
        this.owner = owner;
        this.balance = balance;
    }
    public void deposit(int sum){ balance += sum; }
    public void withdraw(int sum){ balance -= sum; }
    public String getOwner() { return owner; }
    public int getBalance(){ return balance; }
}
```

Listing 3.2: Account class after the fix

```java
public class BlackList {
    private static BlackList instance = new BlackList();
    public static BlackList getInstance(){ return instance; }
    private Set<String> blackSet = new HashSet<String>();
    public void add(String user){ blackSet.add(user); }
    public void remove(String user) {blackSet.remove(user);}
    public boolean contains(String user) {
        return blackSet.contains(user);
    }
}
```

Listing 3.3: BlackList class was not changed

```java
@Test  public void testVipAccount(){
   Account vipAccount = new Account("King David", 1000);
   VipValidator vipAccountValidator = new VipValidator(vipAccount);
   //vip has no problem (before fix alarms, after fix should not
      alarm)
   vipAccount.setValidator(vipAccountValidator);
   vipAccount.withdraw(2500);
   assertTrue(vipAccount.getBalance()==-1500);
   //vip can't withdraw
   vipAccount.setValidator(vipAccountValidator);
   vipAccount.withdraw(2500);
   assertTrue(vipAccount.getBalance()==-1500);
   assertTrue(vipAccountValidator.getCounter() == 1);//count failures
```

Listing 3.4: testVipAccount method from ValidatorsTester JUnit class

discover the reason, since he does not have a way to identify aspects that advised the previous version but stopped to advise the current version of his class, (and so he sets off to develop XRC).

3.3.4 Scenario 4: Identify Inheritance change

The version 2.00 of the bank account application requires three kinds of users: regular, special, and vip. The regular user cannot have an overdraft. Trying to withdraw more money than he owns in his account fails by a $UserValidator$, and a failures counter increases. The special user can have an overdraft of 1000\$. Trying to withdraw more money than he owns in his account turn an alarm on (whether the withdraw fails or succeeds). The vip user can have an overdraft of 2000\$, and have the same kind of alarm as the special user.

A new requirement is to avoid the alarm for the vip user. Masy, the developer that worked on the code got fired because he was too messy, and the completion of this task has been assigned to a new developer named Dave. Dave runs the $ValidatorTester$ unit test, which fails on the last assertion of testVipAccount in Listing 3.4.

Dave understands that the unit test fails because something is wrong with the getCounter() method in the $VipValidator$ class. He compares the last versions of $VipValidator$, but it was not changed recently generally, and by Masy specifically. How should Dave find out that Masy changed a *declare parents* advice, in the aspect $VipValidatorAspect$ that advise $VipValidator$? The inheritance has been changed, without any trace in the $VipValidator$ class. Dave reaches a dead-end.

3.4 RCS-Related Setbacks

A RCS plays an important role in supporting a healthy software development process. When the RCS does not function optimally, the process is hurt.

3.4.1 Loss of code review capability

AOP's obliviousness limits the code review to the code that is written explicitly in the examined class. The functionality that is added or modified through aspects implicitly is hidden in the aspects. The full behavior and functionality are obscure, and very hard to discover. E.g. viewing C, no clue to $\mathit{eff}(A, C)$ can be found. Scenario 1 illustrates that viewing the previous version of *Account* class (a version that worked), does not reveal the complete behavior of *withdrow* (i.e., the aspect *BlackListAdvice* advises *withdrow* and handles limited users).

3.4.2 Loss of change control

RCS relates to aspects that advise other classes as regular files. Affecting the behavior and functionality of a class through an aspect occurs without change control on the affected class. When an aspect is checked-in, the changes to the advised classes will be part of the revisioned sources. The RCS does not ask for approval, nor does the RCS require to check-in the implicitly affected classes. These classes thus escape code review and bypass change control. E.g., when A' was checked-in, $\mathit{eff}(A', C)$ changed the behavior of C, with no change management, change control or change tracking over C. Moving this burden to the developer is unwise. Thanks to AOP's obliviousness and quantification, the task of finding all the changes might be very hard and tedious. The functionality changes are no longer limited to the modified files, since every class that is advised by a new or modified aspect might have functionality changes. Moreover, an aspect deletion or modification that removes old advice causes a functionality change that is even harder to find. Furthermore, AOP's quantification can cause multiple changes in multiple classes via simple short statements in single aspect file. In Scenario 1, *Account* was developed previously to *BlackListAdvice*. The developer that added the *BlackListAdvice*, actually changed the functionality of *Account* class, and specifically the logic of *withdrow* method. Yet, when he checked-in his code, there was no change control from the *Account* class point of view. Find changes did not found *Account* as changed. Check-in was not required to *Account* class. It stayed in RCS with the same version "hiding" the change in its behavior. In Scenario 2, after the fix of the *BlackListAdvice* aspect, there was no need to check-in the *Account* class, and therefore lose of change control occurred again. No version exists for the previous state, where it worked.

3.4.3 Loss of the class evolution tracking

RCS provides operations that support change tracking (i) viewing history versions of a file (ii) diff - compare versions of a file. Both operations are disturbed in several manners for AOP code:

- The lose of change control may lead to lose in evolution tracking as well. A change of an aspect that advise a class, does not require to check-in the class. If the class is not checked-in, it has no version for the change, which is especially difficult to track. In the example (Scenario 2), after the aspect BlackListAdvice stopped to advise Account, and was checked-in, the class Account stays with the same version. The behavioral change of Account is an evolution of the class that went unnoticed

- Due to AOP's obliviousness, a versioned class file has no information on its advising aspects. A class functionality change might appear only in its advising aspect. This change is "invisible" to the change tracking RCS operations. E.g.,

(i) viewing history version C will not display $\mathit{eff}(A, C)$. (ii) $\mathit{diff}(C, C')$ will not display the difference between $\mathit{eff}(A, C)$ and $\mathit{eff}(A', C')$ (i.e. any change that results from A and A' different advice). In the example, Figure 3.2 does not show the change in the behavior of Account. BlackListAdvice stopped to affect Account, and the only visual change is the name of the method. No hint for the real class, byte-code or functionality change.

- Suppose that the aspects that advised the versioned classes, are known. The order of the changes might still be obscure. For AOP code, the evolution of a class is not sequential anymore. Prior to AOP, every change in the class, was checked-in in a single version. It is obvious that this change did not affected previous version, and affects the next version. The changes are explicit. AOP's obliviousness decouples the class that is advised from the aspect that advise it. The versions of the class and aspect are independent. Their evolution is parallel. Even with timestamps and version numbers that expose the check-in order, still one can not know the real history. Example: lets assume aspect A advises class C, both were checked-in, than evolved to A', C' and checked-in, and on next phase, to C'' and A''. Viewing the history, if the check-in order was $A \rightarrow C' \rightarrow A'$, we will not know if C' worked with A or with A'. If the check-in order was $A' \rightarrow C' \rightarrow A''$, we will not know if C' worked with A'or A''.

- Suppose a stronger assumption, that "everything" is known - the aspects that advised the versioned classes, and their versions. The changes of the class might be spread over many versioned aspects, and the class versions. Tracking the changes with no smart visual aid is a difficult and tedious task that requires to navigate through multiple files back and forth. Eclipse Compare view can no longer display all the changes in a single view.

3.4.4 Loss of reversion to a stable revision

This is a direct result of the previous problems. It's hard to find a stable point in the history, in order to revert to it. In the example (Scenario 2), as described previously (in evolution lose), a developer that view the history versions of Account, will find no lead to a stable revision, to check-out. Scenario 4 provides another illustration for this problem.

3.4.5 Loss of team development support

AOP's obliviousness let different developers work on different aspects that advise the same class, and on the class itself. This might hide overlapping changes. The RCS is unaware of it. The aspects will be checked-in with no error or warning from the RCS, since they are different files. The overlapping changes will be revealed only during build (in case of failure), in deep code review, or on run-time. In the example (Scenario 3), Sagi and Dave changed the functionality of the same class in the Logic domain. The changes were overlapping. RCS did not detect for none of them that the class is not updated, and did not prevent the changes overlapping.

We saw that the basic problems are because we don't know what aspect revisions advise a class, and that even if this information is achieved, still there are complicated problems.

Chapter 4

The XRC Approach

4.1 Solution Overview

The main requirement for XRC is adding revision control support for the crosscutting metadata (XMD). The root of the problems is that RCS is unaware of the XMD, and does not manage it. An instance of the problem is that the Eclipse IDE with the AJDT and SVN plug-ins does not manage, store, or display the XMD.

To provide revision control for XMD, there is a need to define, persist, compare and display XMD. The required components for the solution are:

1. An *abstract data type (ADT)* for representing XMD, and the ability to associate and store the XMD with a source code file.

2. A *diff* engine that enables to compare XMD of two files (or two versions of the same file).

3. *Visual enhancements* that enable to display markers, (e.g., markers that the Java editor displays for advice) when comparing Java source files and when viewing a previous version.

4. *RCS–IDE integration* that, together with the other components, provides a

mechanism for saving comparing, and displaying XMD.

4.1.1 Crosscutting Metadata (XMD)

In Section 1.1 we denoted by $\mathit{eff}(A, C)$ the crosscutting effect of A on C. Since C might be advised by multiple aspects, its XMD sums to:

$$\mathit{XMD}(C) = \bigcup_{A} \mathit{eff}(A, C)$$

where A ranges over all aspects in the project. Similarly,

$$\mathit{XMD}(A) = \bigcup_{C} \mathit{eff}(A, C)$$

where C ranges over all classes in the project. It should be noted that $\mathit{XMD}(C)$ and $\mathit{XMD}(A)$ include concrete information, such as the line numbers of specific pieces of advice and effected program element. For conciseness and for symmetry considerations, we focus in this paper mainly on $\mathit{XMD}(C)$, and here after denote it XMD. The XMD is the data that we want to preserve.

AJDT metadata The AJDT plug-in displays XMD information visually using markers and hint messages in the Eclipse editor. Internally, AJDT represents the XMD with a map ADT:

$$Map < Integer, \ List < IRelationship >>$$

A key in this map represents a line number in the source code. A value in this map is a list of *IRelationship* objects, where *IRelationship* is an interface defined in AJDT. An implementation of *IRelationship* contains the relevant data of a single piece of advice: source, target list, name, and kind of the advice. This map is the XMD in

AJDT.

XRC metadata The XMD that XRC writes to and reads from the RCS is essentially the AJDT XMD, enriched with data that XRC requires: the version of the advising aspect, a flag in case the advising aspect was modified locally and differs from its version in the RCS repository, and the details of the parent in a declare parents advice. The enrichment is done by modifying the existing members of the *IRelationship* implementation.

4.1.2 Diff Engine

The *diff* engine takes the XMD of two class versions, compares them, and returns a data structure with the differences. Comparing only the icons is obviously not good enough, because the same icon may represent different kinds of advice or advice from different aspects.

A simple straightforward comparison would be to compare the pieces of advice per line, and mark the differences. However, this solution gives false positive results. For example, adding a single empty line at a beginning of a class, checking it in, and comparing it with its previous version will yield that all the advice after this empty line and on are different from the previous version. A developer will have to inspect each one in order to understand that the crosscutting effect stayed the same.

Comparing the pieces of advice sorted according to their their line number may still gives false positive results, e.g., for a piece of advice that is moved before another piece of advice.

In order to avoid such false positives, a piece of advice that has moved from one line to another with the same content is considered unchanged. This is adapted from ldiff [3], an enhanced *diff* tool that is capable of tracking text that was moved to another line. However, ldiff cannot be used as is to compare XMD, since XMD is not text. Every line might have a list of *IRelationship*, each comprising multiple targets that the piece of advice affects.

The XRC *diff* engine compares *XMD*(C) with *XMD*(C') by first "flattening" the data, then computing the differences, and finally "inflating" the result back into a format that Eclipse understands:

$$\mathit{diff}\,(\mathit{XMD}(C), \mathit{XMD}(C')) = \lceil \Delta\,(\lfloor \mathit{XMD}(C) \rfloor\,, \lfloor \mathit{XMD}(C') \rfloor) \rceil$$

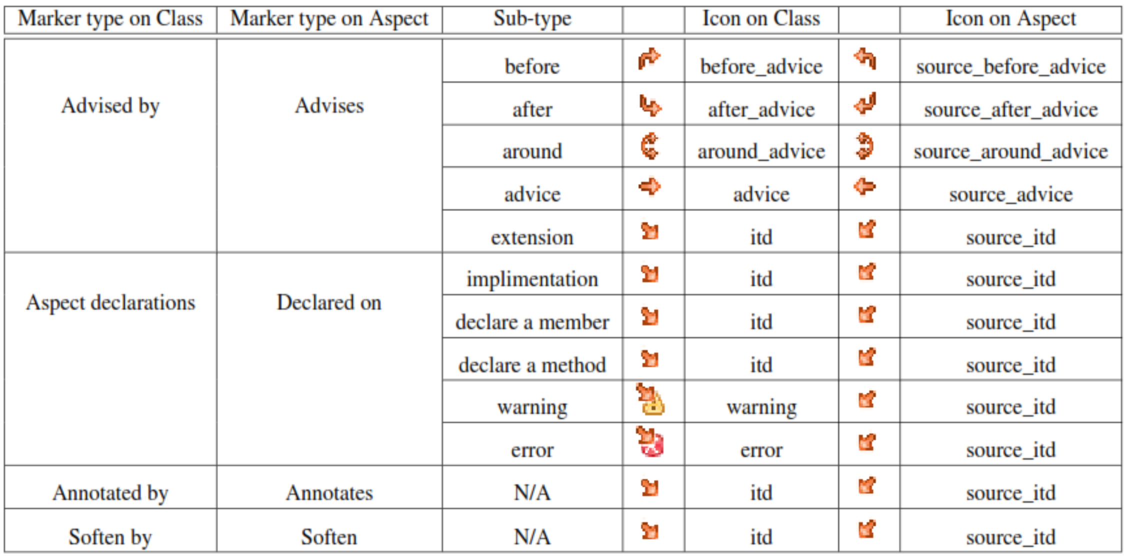
For this, XRC uses two methods that translate the XMD representation required for display to the representation required for comparison, and back:

- $flatten$: $\mathit{XMD} \to \mathit{Set} < \mathit{IRelationship} >$, denoted $\lfloor . \rfloor$, takes an XMD object, and returns a $\mathit{Set} < \mathit{IRelationship} >$ based on the *IRelationship* objects in the XMD. It breaks each *IRelationship* with multiple targets into single-target *IRelationship* objects, and removes the line numbers.

- $inflate$: $(\mathit{XMD}, \mathit{Set} < \mathit{IRelationship} >) \to \mathit{XMD}$, denoted $\lceil . \rceil$, takes an XMD object, and a $\mathit{Set} < \mathit{IRelationship} >$. It constructs and returns a new XMD with the *IRelationship* objects from the input set restored to their original structure and line numbers, accumulating targets of the same line to a list in order for Eclipse to be able to use the data for displaying the XMD.

The XMD representation required for display is of the *map-form:* $Map < \mathit{Integer}, \mathit{List} < \mathit{IRelationship} >>$. The flatten method translates it into the *set-form:* $\mathit{Set} < \mathit{IRelationship} >$ where each *IRelationship* has a single-target. The set-from representation is used for the comparison.

Comparing two *IRelationship* objects sums to comparing their members (source, target list, name, and kind of the advice). If their members except the target lists are equals, and the target lists have at least one target that is equal, there is a problem of "partially-equals" objects. The pieces of advice that the *IRelationship* objects



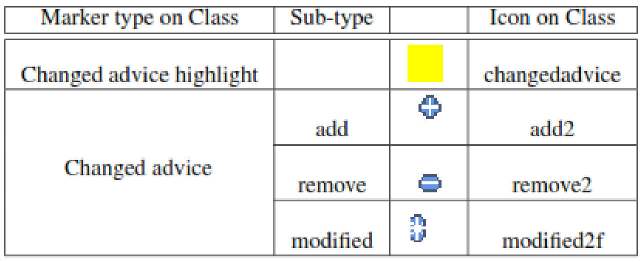
Marker type on Class	Marker type on Aspect	Sub-type		Icon on Class		Icon on Aspect
Advised by	Advises	before	↱	before_advice	↰	source_before_advice
		after	↳	after_advice	↲	source_after_advice
		around	↺	around_advice	↻	source_around_advice
		advice	➡	advice	⬅	source_advice
		extension	↘	itd	↗	source_itd
Aspect declarations	Declared on	implimentation	↘	itd	↗	source_itd
		declare a member	↘	itd	↗	source_itd
		declare a method	↘	itd	↗	source_itd
		warning	⚠	warning	↗	source_itd
		error	✖	error	↗	source_itd
Annotated by	Annotates	N/A	↘	itd	↗	source_itd
Soften by	Soften	N/A	↘	itd	↗	source_itd

Figure 4.1: AJDT Markers

represent differ, yet one of their targets does not differs. The flatten method and the set-form eliminate this problem. When an *IRelationship* has a single target it is "flat", its equal method is well defined, and set manipulations such as subtraction and union become trivial. Consequently, the set-form is an elegant and useful form for manipulations required for the comparison while ignoring the lines. The DTOs and the data manipulations are illustrated in the Developer Guide (Appendix B).

4.1.3 Visual Enhancements

AJDT Markers and Rulers AJDT displays the markers in the editor, via a ruler to the left of the source, as shown in Figure 3.3a. Table 4.1 summarizes the AJDT marker types, sub-types and their icons. A marker represents a relationship between the class being edited in the editor and the aspects that advise it (or vice versa, a relationship between the aspect being edited in the editor and classes that it advises). Since every relation on one side implies a relation on the other side, we see a kind of symmetry, both in the marker types, and in their icons. AJDT also provides navigation from the aspect to the class or vice versa via a pop-up menu by clicking on a marker. AJDT markers are being created and updated on build.



Marker type on Class	Sub-type		Icon on Class
Changed advice highlight		🟨	changedadvice
Changed advice	add	⊕	add2
	remove	⊖	remove2
	modified	⟳	modified2f

Figure 4.2: XRC Markers

XRC Markers and Rulers In order to display the differences in a Compare view, we use rulers similar to the one used in the editor. Unfortunately, the Eclipse JDT infrastructure does not support rulers in a Compare view. We therefor had to extend it. A side benefit is that these rulers can also be used for other tasks that involve displaying markers in the Compare view. On a ruler, named *advice-ruler*, we display the AJDT markers, and use a *Changed advice highlight* marker (Figure 4.2) to emphasize differences. This marker changes the background of a marker that has changed, so it can superimpose existing markers.

On a second ruler, named *diff-ruler*, inspired by the obsolete AJDT Crosscutting Comparison view (see Section 2.3.2.1), we mark the essence of the change. Two addition markers, *add* and *remove* (Figure 4.2), are used to indicate advice addition and removal, respectively. A fourth marker, named *modified-marker* (Figure 4.2), is used to warn the programmer that the advising aspect (for the marked advice) is not a versioned one, but its state was nonetheless modified when the advised class was checked-in.

To understand the need for the *modified-marker* consider the following scenario. Let C be a versioned class. Let A be a versioned aspect that advise C. Let A' be a modified version of A that was not checked-in yet, but advises C differently than A. Suppose C is being reviewed and checked-in as C', with the metadata that represents $\mathit{eff}(A', C)$. Let A'' be a modified version of A' that advise C differently than A or A'. A'' is checked-in, while A' was never checked-in. This scenario leads to a situation that the metadata of C is allegedly deceptive. $\textit{diff}(C, C')$ will display the *modified-marker*, since A' is unattainable, and it can only hint on A as its predecessor. Therefore, we need the *modified-marker* to discriminate the situation where the version of the aspect is known and trusted, from the situation where it is not, and only the previous version of the aspect that had been checked-in is known.

The XRC changed-advice markers (*add*, *remove* and *modified*) in Figure 4.2 are designed in a special way to enable overlapping. The *diff-ruler* can display for a single line the three changed-advice markers.

4.1.4 RCS–IDE Integration

XRC is integrated with the IDE and modifies its save, build, check-in, history, and compare processes.

Save Save is done in three steps:

- *Execute JDT's save process.* When the developer saves a file, the JDT's save process runs, and at some point XRC gains control over the save process.
- *Extract and save the markers.* XRC extracts the markers from the file into a Serializable object, and saves it in the RCS as a property associated with the file.
- *Create and register an AJBuildListener for the saved resource.* An AJBuildListener is created and registered for the file, in order to handle it on the next build.

Build Build is done in four steps:

- *Execute JDT's and AJDT's build processes.* XRC works in post build, when the models in JDT and AJDT are already updated. XRC hooks to the postAJBuild hook by using AJBuildListener. The listeners are registered when saving a file.

- *Find affected files.* XRC compares the XMD of the file with its predecessor, and analyzes which of the affected files differ.
- *Extract and Enhance the XMD.* For each affected file, XRC uses JDT and AJDT to find the AJDT XMD and enriches it with the XRC XMD (adding parents details, target revisions, etc.).
- *Write the XMD.* XRC writes the enhanced XMD as a property of the working copy of the file. The file is then marked as dirty. SVN marks the file as dirty automatically when its properties are modified.

Check-in Check-in saves with the file the properties of the file, where the XMD is stored. Since SVN checks-in the properties of a file with the file, no change is required for SVN.

Display a previous version When reading a file from the RCS, XRC reads the XMD from the properties of the file, and displays the markers accordingly.

Compare versions When comparing two previous versions:

- *Execute JDT's compare process.* When the developer compares files with a “java” or “aj” extension, XRC takes control over the compare process.
- *Read the XMD.* XRC reads the XMD for each of the compared versions.
- *Run the diff engine.* The *diff* engine compares the XMD, and returns an object with the differences.
- *Create and display the markers.* XRC creates markers according to the differences. The markers are assigned to the advice-ruler and to the diff-ruler, in each side of the Compare View.

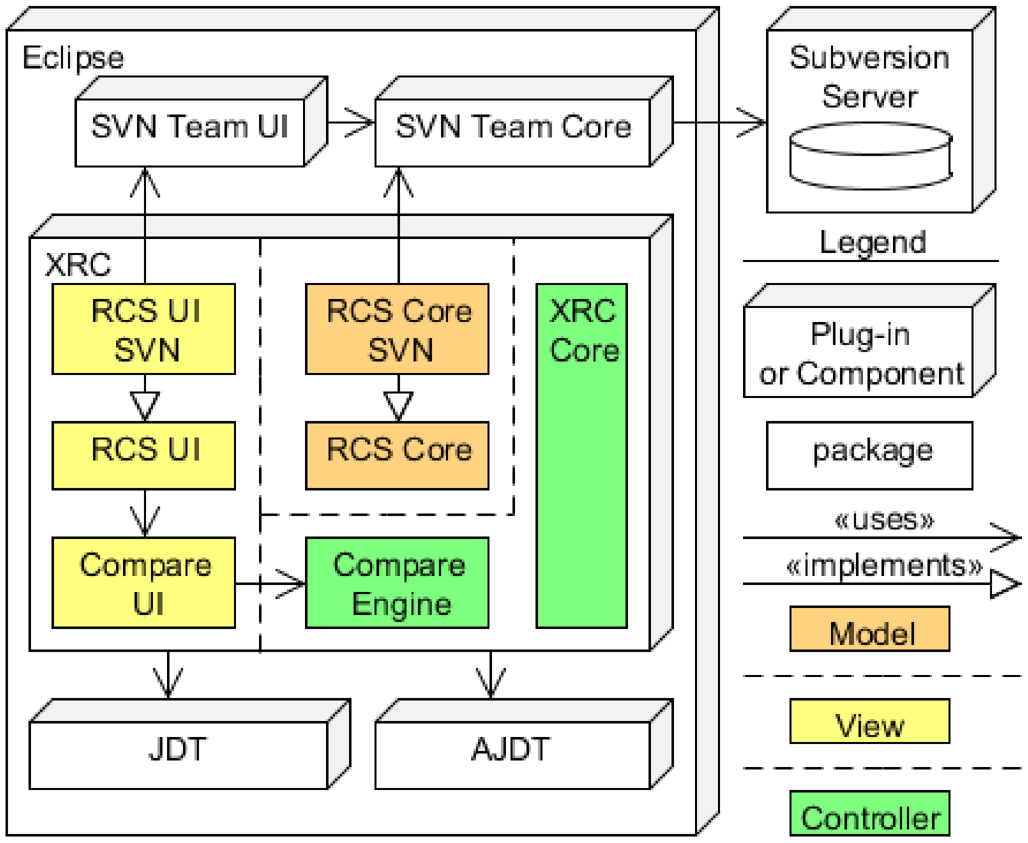


Figure 4.3: XRC Architecture

4.2 Architecture

The XRC plug-in extends JDT and AJDT, and uses the SVN plug-ins for Eclipse to provide the XRC solution for Subversion. The interface with the Eclipse IDE is done through extension points. XRC is incorporated into the IDE and RCS using the model-view-controller (MVC) architectural style [27] depicted in Figure 4.3. The SoC in the proposed design is according to the OO principles: encapsulation, high cohesion, and low coupling. It provides reusable components to ease implementation for other RCS or for other kind of metadata comparison. The XRC implementation comprises the following components:

4.2.1 Model

The model is the XMD. The XMD is the crosscutting metadata of a class derived from the versioned aspects that advise it.

- *RCS Core* is an API and infrastructure for writing and reading metadata, markers information, etc., from the RCS.
- *RCS Core SVN* is an implementation of the RCS Core interface for SVN.

4.2.2 View

The view provides visual enhancements for displaying XMD, e.g., displaying the crosscutting effect of A on C, both when viewing C and when comparing C and C'.

- *RCS UI* is an API and infrastructure for displaying AJDT and XRC markers.
- *RCS UI SVN* is an implementation of the RCS UI interface for SVN.
- *Compare UI* is an API for displaying metadata on versioned files comparisons and on open versions of a file, using markers.

4.2.3 Controller

The controller performs *diff* to compare XMD on request, and modifies the IDE and RCS policies for save, check-in, build, and compare. By tracking and by managing the crosscutting effect of aspects, it updates the XMD "at the right time.".

- *Compare Engine* is an API for comparing metadata and the visual information displayed by markers.
- *XRC Core* is an infrastructures layer above the JDT and AJDT layers, integrated with the save and build processes, and manages the markers model for XRC. It intercepts and modifies RCS processes, such as save and build.

Chapter 5

Evaluation

5.1 Magnitude of the Problem

The importance and magnitude of the problem in practical settings can be learned from related studies. Ferrari et al. [10], for example, conducted an exploratory analysis of twelve releases of three medium-sized real-word aspect-oriented systems taken from different application domains. Their analysis examined how obliviousness influences the presence of faults in evolving aspect-oriented programs. They found that obliviousness facilitates the emergence of faults under software evolution conditions. Their analysis confirmed, with statistical significance, that "the lack of awareness between base and aspectual modules tends to lead to incorrect implementations" [10]. To regain crosscutting awareness, XRC enables integrated RC support for crosscutting metadata.

5.2 Assessment of the Tool

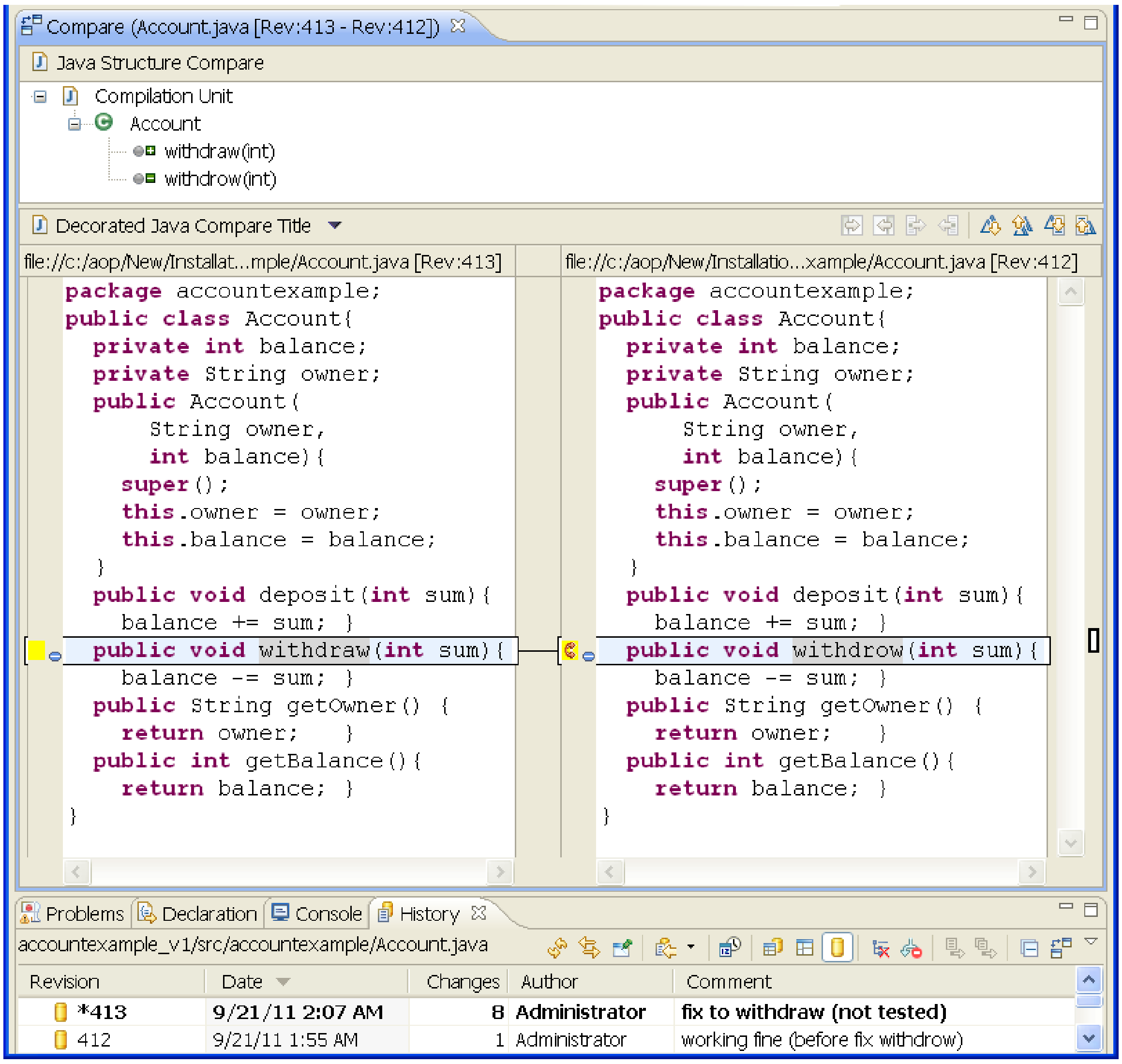
We implemented an XRC plug-in for Eclipse in order to demonstrate the *feasibility* of the approach, and for evaluation. To assess the *ability* and *efficiency* of XRC in tracking down inconsistency problems, we performed coverage tests and examined the behavior of XRC on several small examples qualitatively as well as on a larger open-source project. AJHotdraw [33] is an aspect-oriented refactoring of JHotDraw, a relatively large and well-designed open source Java framework for technical and structured 2D graphics. We reviewed the AJHotdraw code with XRC, including: modifying, checking-in, viewing previous versions, and comparing versions of aspects and classes. We noticed no apparent degradation in *performance* and we confirmed that the overall *user experience* is consistent with that of JDT. Regarding time and space efficiency, XRC does not require to save the woven classes. Its *space* efficiency stems from persisting for each file only the source and its relevant XMD. The XRC is efficient in *run-time*, since retrieving the XMD requires a single read from the RCS. There is no need for any complicated processing or re-weaving. To show how XRC helps to deal with obliviousness and solve the problems described earlier, we revisit the scenarios presented in Section 3.3.

5.2.1 Comparing Two Advised Versions

With the Eclipse plug-in for XRC (Chapter 4), comparing the *Account* to its previous version immediately reveals that something is different with the advice (Figure 5.1). Rolling the mouse cursor over the marker shows a hint message stating that the *BlackListAdvice* aspect ceased to advise. Obviously, fixing the typo in the *BlackListAdvice* aspect (Listing 5.1) will eliminate the bug.

5.2.2 Viewing an Advised Revision from SVN History

With the Eclipse plug-in for XRC, markers are displayed also for old versions of *Account*. In Figure 5.2, the selected title tab confirm that we are looking at [Rev:320] of MyClass. The figure displays a variety of markers that represent XMD. Note that without the XRC plug-in, none of these markers would be displayed in the left ruler.

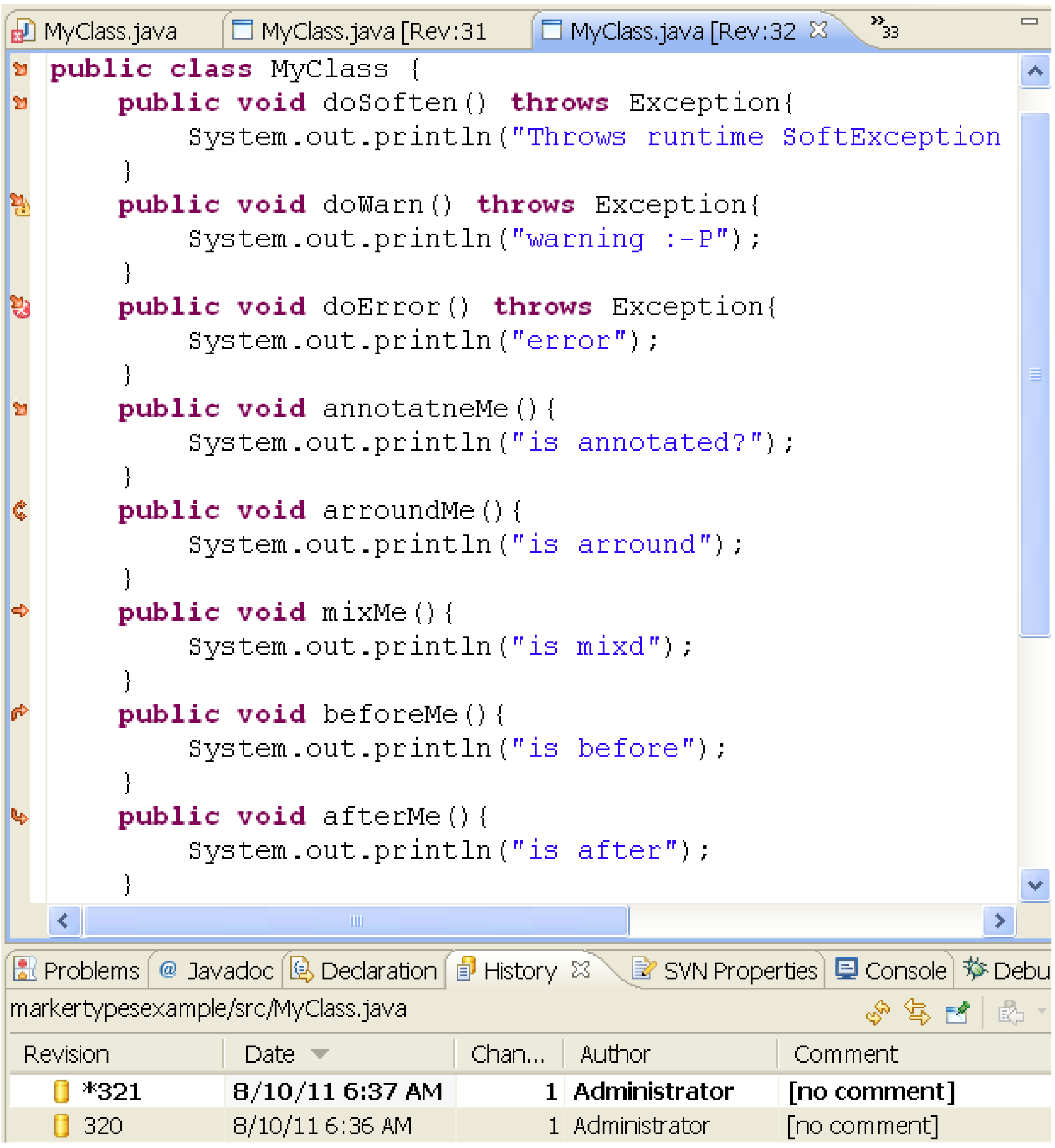

Compare (Account.java [Rev:413 - Rev:412])
Java Structure Compare
Compilation Unit
Account
withdraw(int)
withdrow(int)
Decorated Java Compare Title
file://c:/aop/New/Installat...mple/Account.java [Rev:413]
file://c:/aop/New/Installatio...xample/Account.java [Rev:412]
Problems
Declaration
Console
History
accountexample_v1/src/accountexample/Account.java

```
package accountexample;
public class Account{
  private int balance;
  private String owner;
  public Account(
      String owner,
      int balance){
    super();
    this.owner = owner;
    this.balance = balance;
  }
  public void deposit(int sum){
    balance += sum; }
  public void withdraw(int sum){
    balance -= sum; }
  public String getOwner() {
    return owner;    }
  public int getBalance(){
    return balance; }
}
```

```
package accountexample;
public class Account{
  private int balance;
  private String owner;
  public Account(
      String owner,
      int balance){
    super();
    this.owner = owner;
    this.balance = balance;
  }
  public void deposit(int sum){
    balance += sum; }
  public void withdrow(int sum){
    balance -= sum; }
  public String getOwner() {
    return owner;    }
  public int getBalance(){
    return balance; }
}
```

Revision	Date	Changes	Author	Comment
*413	9/21/11 2:07 AM	8	Administrator	fix to withdraw (not tested)
412	9/21/11 1:55 AM	1	Administrator	working fine (before fix withdrow)

Figure 5.1: Diff between [Rev:413] and [Rev:412] of Account class with XRC. In comparison with Figure 3.2, markers are shown and compared



```
public class MyClass {
    public void doSoften() throws Exception{
        System.out.println("Throws runtime SoftException
    }
    public void doWarn() throws Exception{
        System.out.println("warning :-P");
    }
    public void doError() throws Exception{
        System.out.println("error");
    }
    public void annotatneMe(){
        System.out.println("is annotated?");
    }
    public void arroundMe(){
        System.out.println("is arround");
    }
    public void mixMe(){
        System.out.println("is mixd");
    }
    public void beforeMe(){
        System.out.println("is before");
    }
    public void afterMe(){
        System.out.println("is after");
    }
```

markertypesexample/src/MyClass.java

Revision	Date	Chan...	Author	Comment
*321	8/10/11 6:37 AM	1	Administrator	[no comment]
320	8/10/11 6:36 AM	1	Administrator	[no comment]

Figure 5.2: Viewing a previous version [Rev:320] of a class retrieves the XMD from SVN and displays the AJDT markers in the left ruler

```
public aspect BlackListAdvice {
    void around(int sum, Account account):
            execution(void withdrow(int))
            && this(account) && args(sum){
        if(BlackList.getInstance()
                    .contains(account.getOwner())){
            System.out.println("limited user!!!");
        } else {
            proceed(sum, account);
        }
    }
}
```

Listing 5.1: BlackListAdvice with the typo

5.2.3 Regaining RCS Support for SW Development Processes

Regaining code review Viewing a previous version of Account with XRC now displays every advice that affected that version. The hint messages help understand the way the code used to work.

Regaining change control Repeating Scenario 2 (Section 3.3.2) with XRC, once the *BlackListAdvice* aspect is fixed, the XMD of the *Account* class is updated during build and flags the class for check-in. A code review of *Account* will probably expose the bug before check-in. However, even a check-in without fixing the bug will distinguish the previous version that worked from the new version that does not work.

Regaining code evolution tracking With XRC, the developer has to check-in the *Account* class whenever the aspect is modified. This check-in enables to track the code evolution. The *diff* in Figure 5.1 shows the change in the behavior of *Account*. On both sides of the Compare view, a marker indicates that an aspect ceased to advise. Rolling the mouse cursor over the marker displays a more detailed hint message that explains why.

Regaining reversion to a stable revision Checking-in a revision of *Account* also helps to later identify a stable revision to be checked-out.

In scenario 4, lets assume that Masy worked with XRC. When he changed the way that the aspect $VipValidatorAspect$ advise $VipValidator$, XRC requires to check-in also the $VipValidator$ class (due to the change control regained by XRC). In his investigation, Dave notices this recent change, and compares the last versions of $VipValidator$. The text is identical, yet the markers reveal that $VipValidatorAspect$ previously advised $VipValidator$ to extend $SpecialUserValidator$, and now it extends $UserValidator$. This change avoids the alarm but fails the counting. Dave reverts to the previous stable state easily by overriding with the previous version of the aspect, verify that the test now passes, and continue to work on the feature.

Regaining team development support When change control is regained, RCS will detect and prevent overlapping changes to the XMD of a class, without first resolving any conflicts.

5.3 Functionality

To verify the correctness of the implementation, and the tool's functionality, we designed and ran the coverage test suite described in Table 5.1, and observed the expected results described in the right-most column of Table 5.1.

5.4 Limitations and Threats to Validity

The XRC tool was build for and tested to work with a specific RCS and IDE, namely: SVN version 1.6.3, Eclipse version 3.7 (Indigo) for 32bit, AJDT version 2.1.3. However, the approach should be applicable in general. Implementing plug-ins for other RCS tools is left for future work.

Test name	Test steps	Expected results
AJDT markers coverage test (Figure 4.1)	Open MyClass (MyAspect) and verify that all AJDT markers are displayed. Modify the code of MyClass (MyAspect), save, build, and commit. View the previous version, and compare the two last versions.	All AJDT markers are displayed in the previous version (Figure 5.2), and in the Decorated Compare view. Rolling the mouse cursor over each marker displays a suitable hint message, including the version of the aspect that is advised.
XRC markers coverage test (Figure 4.2)	Change the names of two methods in MyClass (MyAspect), so that one that was advised (advise) will stop to be advised (to advise) and another that was not advised (advise) will start to be advised (to advise). Save, build, check-in MyClass (MyAspect), and compare the last two versions.	The markers associated with the modification are highlighted in the advice-ruler. Added (and removed) markers are displayed in the diff-ruler on both sides of the Compare view, at the line of the method.
Special modified marker test	Modify MyAspect (MyClass), but do not check-in it in. Repeat the XRC coverage test.	The special modified marker is displayed in the diff-ruler.
Change control test	Change the name of a method in MyClass (MyAspect), so that it will stop to be advised by MyAspect (advised MyClass) and will start to advised (to advise) by AnotherAspect (AnotherClass). Save, build, but do not check-in.	The three files are marked as modified in the Eclipse Package Explorer.
Multiple targets test	Run the change control test. Check-in the involved files.	The modified markers are displayed in the previous version, and in the Compare view. The hint message refers to both files.
Build failure test	"Break" the code (e.g., by adding a space in the middle of a method name in MyClass). Save and build.	No other file (or XMD) is modified. Change control is not required.

Table 5.1: XRC testing suite

We did not conduct users case study to substantiate the correctness and usability of XRC, since it requires several AspectJ developers that will use the XRC plug-in. However, my rich experience as a developer, the test-suit and the experiments I run during development, and for the examples support the necessity, usability and correctness of the XRC plug-in. We intend to publish the plug-in and get feedback from the open source community.

The implementation is for SVN, and allegedly relay on its properties features. Persisting the XMD without this feature requires mechanisms to associate metadata with a file, and to "touch" a file and mark it as unsynchronized with the RCS repository. Associating metadata can be done easily, e.g., by storing it in a file with the same name and other extension (.xrc). Touching a file is also trivial (if a simple touch command does not exist, adding a space to the file should do the work).

AOP languages other than AspectJ were not explored in this paper. However, obliviousness and quantification are the essence of AOP, they empower AOP and are in the roots of the problems and solution. Consequently, the XRC approach should work also for other AOP languages.

Visualization of the XMD is part of the XRC approach . The given implementation extends the Eclipse abilities for this visualization and reuse the AJDT. The XRC approach is not limited to Eclipse. Using other IDE or other diff/merge tool obviously requires adaptations for XMD visualization.

The RCS-IDE integration requires interfering with the processes in the IDE. It is done mainly by extending Eclipse plug-ins. The intervention is post-process. Hooking after build or after save should be enabled also in other IDEs. If no hooking mechanism is at hand, implementing the XRC approach might require its creation, or access to the IDE source.

A limitation of the approach is that the change control provided via XRC depends on the build process to keep the XMD up-to-date. Recall that AJDT updates its markers on build, and XRC is base on AJDT. Consequently, build must be done before check-in. Eclipse requires to perform team > cleanup for a project or a folder in order to synchronize changes with the SVN. This should be done after build, and before code reviews, in order to gain the XRC change control.

However, this limitation can be minimized by setting *auto-build* in the Eclipse configuration (which it is the also the default setting in Eclipse). The recommended development process is thus: save, build (if auto-build configuration is not set), review all the changes, and check-in all the files that have been changed.

Chapter 6

Conclusion

Traditional RC systems predate AOP, and external diff tools, code history views, and other elements of the RC system and its integration with the IDE were never fully adapted to AOP. Since the development of any software product, medium or large, requires revision control, the lack of appropriate support is an obstacle that hinders the use of AOP.

In this work we introduce crosscutting revision control—a novel approach, presented at ICSE 2012 [18]—that improves the revision control of AOP code. We also contribute a supporting XRC plug-in for Eclipse that provides the essential means for persisting, comparing, and displaying crosscutting metadata (XMD). The XMD is maintained and checked-in with the code. The persisted XMD is then used by the Eclipse IDE to mark with marginal icons the effect of aspects on previous versions of the code and to indicate whether or not that effect has changed.

XRC reintroduces RC to the aspect-oriented software development process, and identifies the gap that RC systems should bridge in order to improve RC support for evolving aspect-oriented programs. The approach, however, is not limited to AOP. It may be applied to breakpoints, warnings, and other markers and metadata, as well.

Future Work There are several directions for future work. A "divide and conquer" approach suggests to investigate each component further and deeper, analyze and improve its performance, and features. For example, enhancing the diff engine analysis to identify and avoid false positive results when different aspect versions apply the same advice to a class, however with different syntax. E.g., when refactoring an anonymous pointcut to a declared one.

A different interesting idea is to support a more complex comparison. Let A_i,C_i be versioned aspects and classes, respectively. When comparing the versioned classes C and C', one may be interested in the differences between $\mathit{eff}(\{A_1, A_2, ...A_m\}, C)$ and $\mathit{eff}(\{A'_1, A'_2, ...A'_n\}, C')$. This may help to understand and simulate the changes in C with each set of versioned aspects. Similarly, when comparing aspects A and A', the differences between $\mathit{eff}(A, \{C_1, C_2, ...C_k\}))$ and $\mathit{eff}(A', \{C'_1, C'_2, ...C'_l\}))$ may help to discover the effects of the versioned aspect on the given classes. However, this requires (re)weaving of the versioned classes and aspects.

This work identifies the problems and proposes the XRC approach solution and architecture. It focuses on the novel approach, its correctness and its feasibility. A related study [10] helps us to understand the magnitude of the problem in practice. Analyzing what percentage of the faults reported in the study could supposedly be avoided had XRC been used is a leading topic for future work. Other study should explore and apply the XRC approach on other domains, breakpoints, warnings, code recommenders and other markers and metadata.

Bibliography

[1] M. M. Arimoto, M. I. Cagnin, and V. V. d. Camargo. Version control in crosscutting framework-based development. In *Proceedings of the 23rd Annual ACM Symposium on Applied Computing (SAC'08)*, pages 753–758, Fortaleza, Ceara, Brazil, 2008. ACM.

[2] A. Begel and B. Simon. Novice software developers, all over again. In *Proceedings of the 4th International Workshop on Computing Education Research (ICER'08)*, pages 3–14, Sydney, Australia, 2008. ACM.

[3] G. Canfora, L. Cerulo, and M. Di Penta. Ldiff: an enhanced line differencing tool. In ICSE'09 [16], pages 595–598.

[4] M. Chapman. Aop@work: New ajdt releases ease aop development. Technical report, IBM Developer Works, August 2005.

[5] A. Clement, A. Colyer, and M. Kersten. Aspect-oriented programming with AJDT. In J. Hannemann, R. Chitchyan, and A. Rashid, editors, *Analysis of Aspect-Oriented Software (ECOOP 2003)*, July 2003.

[6] B. Dagenais and M. P. Robillard. SemDiff: Analysis and recommendation support for API evolution. In ICSE'09 [16], pages 599–602.

[7] B. Dagenais and M. P. Robillard. Recommending adaptive changes for framework evolution. *ACM Trans. Softw. Eng. Methodol.*, 20(4):19:1–19:35, Sept. 2011.

[8] T. Elrad, R. E. Filman, and A. Bader. Aspect-oriented programming. *Comm. ACM*, 44(10):29–32, October 2001.

[9] J. Estublier. Software configuration management: a roadmap. In A. Finkelstein, editor, *Proceedings of the 22th International Conference on Software Engineering (ICSE'00 FoSE), Future of Software Engineering Track*, pages 279–289, Limerick, Ireland, May 2000. ACM.

[10] F. Ferrari, R. Burrows, O. Lemos, A. Garcia, E. Figueiredo, N. Cacho, F. Lopes, N. Temudo, L. Silva, S. Soares, A. Rashid, P. Masiero, T. Batista, and J. Maldonado. An exploratory study of fault-proneness in evolving aspect-oriented programs. In ICSE'10 [17], pages 65–74.

[11] R. E. Filman, T. Elrad, S. Clarke, and M. Akşit, editors. *Aspect-Oriented Software Development.* Addison-Wesley, Boston, 2005.

[12] R. E. Filman and D. P. Friedman. Aspect-oriented programming is quantification and obliviousness. In P. Tarr, L. Bergmans, M. Griss, and H. Ossher, editors, *Proceedings of the OOPSLA 2000 Workshop on Advanced Separation of Concerns.* Department of Computer Science, University of Twente, The Netherlands, 2000.

[13] R. E. Filman and D. P. Friedman. Aspect-oriented programming is quantification and obliviousness. In Filman et al. [11], pages 21–35.

[14] R. E. Filman and D. P. Friedman. Aspect-oriented programming is quantification and obliviousness. In Filman et al. [11], pages 21–35.

[15] M. Höst and C. Johansson. Evaluation of code review methods through interviews and experimentation. *Journal of Systems and Software*, 52(2-3):113–120, June 2000.

[16] *Proceedings of the 31st International Conference on Software Engineering (ICSE'09)*, Vancouver, Canada, May 2009. IEEE Computer Society.

[17] *Proceedings of the 32nd International Conference on Software Engineering (ICSE'10)*, Cape Town, South Africa, May 2010. ACM.

[18] S. Ifrah and D. H. Lorenz. Crosscutting revision control system. In *Proceedings of the 34th International Conference on Software Engineering (ICSE'12)*, pages 321–330, Zurich, Switzerland, June 2012. IEEE.

[19] M. Kersten. AO tools: State of the (AspectJ) art and open problems. In M. C. Chu-Carroll, G. C. Murphy, S. Clarke, J. Estublier, A. Finkelstein, B. Harrison, and E. Newman, editors, *Workshop on Tools for Aspect-Oriented Software Development (OOPSLA 2002)*, November 2002.

[20] G. Kiczales. AspectJ: aspect-oriented programming using Java technology. JavaOne, June 2000.

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

[22] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. Griswold. An overview of AspectJ. In *Proceedings of the 15th European Conference on Object-Oriented Programming (ECOOP'01)*, number 2072 in Lecture Notes in Computer Science, pages 327–353, Budapest, Hungary, June 18-22 2001. Springer Verlag.

[23] G. Kiczales, J. Irwin, J. Lamping, , J.-M. Loingtier, C. V. Lopes, C. Maeda, and A. Mendhekar. Aspect-oriented programming. *ACM Computing Surveys (CSUR)*, 28(4es):154, Dec. 1996. Special issue: position statements on strategic directions in computing research.

[24] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In M. Akşit and S. Matsuoka, editors, *Proceedings European Conference on Object-Oriented Programming*, volume 1241, pages 220–242. Springer-Verlag, Berlin, Heidelberg, and New York, 1997.

[25] M. Kim and D. Notkin. Discovering and representing systematic code changes. In ICSE'09 [16], pages 309–319.

[26] J. Koch, S. Bohra, R. Goel, S. Pagade, and K. Cooper. AODVis: leveraging eclipse plugins to reverse engineer and visualize AspectJ/Java source code. In *Proceedings of the 1st Workshop on Developing Tools as Plug-ins (TOPI '11)*, pages 24–27, Waikiki, Honolulu, HI, USA, 2011. ACM.

[27] G. E. Krasner and S. T. Pope. A cookbook for using the model-view controller user interface paradigm in Smalltalk-80. *J. Object Oriented Program.*, 1(3):26–49, Aug./Sept. 1988.

[28] E. Lempsink, S. Leather, and A. Löh. Type-safe diff for families of datatypes. In *Proceedings of the 2009 ACM SIGPLAN workshop on Generic programming (WGP '09)*, pages 61–72, Edinburgh, Scotland, 2009. ACM.

[29] A. Loh and M. Kim. LSdiff: a program differencing tool to identify systematic structural differences. In ICSE'10 [17], pages 263–266.

[30] H. A. Nguyen, T. T. Nguyen, H. V. Nguyen, and T. N. Nguyen. iDiff: Interaction-based program differencing tool. In *ASE '11: Proceedings of the 26th IEEE/ACM International Conference on Automated Software Engineering*, pages 572–575. IEEE Computer Society, November 2011.

[31] B. O'Sullivan. Making sense of revision-control systems. *Queue*, 7(7):30:30–30:40, Aug. 2009.

[32] W. F. Tichy. RCS – a system for version control. *Softw. Pract. Exper.*, 15(7):637–654, July 1985.

[33] A. van Deursen, M. Marin, and L. Moonen. AJHotDraw: A showcase for refactoring to aspects. In T. Tourwé, A. Kellens, M. Ceccato, and D. Shepherd, editors, *Linking Aspect Technology and Evolution*, March 2005.

[34] Z. Xing and E. Stroulia. UMLDiff: an algorithm for object-oriented design differencing. In *ASE '05: Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*, pages 54–65, Long Beach, CA, USA, November 7-11 2005. ACM Press.

Appendix A

User Guide

A.1 The XRC Installation Kit

The XRC Installation Kit comprises two installation flavors for users (*Standalone installation* and *Embedded Installation*), and another installation to allow developers enhance, reuse and explore the XRC internals (*XRC-Developer Installation*). The kit may be downloaded from the SVN Repository of the XRC project[1] or from the site[2]. The XRC installation kit comprises:

- An Eclipse folder for the *Standalone installation*
- A repos folder for the SVN repository. The SVN repository contains:
  - The XRC plug-in source code and additional installation files (xrc and xrc_supplemental)
  - Two Bank Application examples (bankapplication_v1 and bankapplication_v2)
  - Test Suite (testsuite)

[1]??
[2]??

- XRC folder structured as follows:
    - unistallShield.bat
    - configuration/org.eclipse.equinox.simpleconfigurator/bundles.info
    - configuration/org.eclipse.equinox.simpleconfigurator/bundlesXRCEntries.info
    - artifacts.xml
    - artifactsXRCEntries.xml
    - plug-ins
        - org.eclipse.compare_3.5.200.xrc.jar
        - org.eclipse.jdt.ui_3.7.0.xrc.jar
        - xrc_2.0.0.xrc.jar

A.2 The XRC User Installation Procedures

A.2.1 Standalone Installation

Standalone Installation assumes you have Subversion Server version 1.6.3 (else download and install it as described below). The installation procedure is simply to copy the Eclipse with the plug-ins already installed from the Eclipse folder in the XRC installation kit.

A.2.2 Embedded Installation

Embedded Installation allows installing the XRC plug-in on an already installed Eclipse IDE.

For the *Embedded Installation*, note that the plug-in was developed and tested for Eclipse for windows 32-bit, version 3.7, and specific versions of the other components

(AJDT, SVN and its plug-ins) as mentioned in the following installation procedure. It may work for other versions, as well, but it was not tested.

1. Install Eclipse if it is not installed. Installations can be found in:
   http://www.eclipse.org/downloads/
   Older versions can be found in:
   http://archive.eclipse.org/eclipse/downloads/ and in:
   http://wiki.eclipse.org/Older_Versions_Of_Eclipse
   Version 3.7 can be found in:
   http://archive.eclipse.org/eclipse/downloads/drops/R-3.7-201106131736/

2. Install Subversion Server version 1.6.3 if not already installed. It can be found in
   http://subversion.tigris.org/ds/viewMessage.do?dsMessageId=2364143&dsForumId=445

3. Download and install SVN plug-ins for Eclipse 3.7

   (a) Eclipse >> Help >> Install New Software >>
   Work with: Indigo - http://download.eclipse.org/releases/indigo >> Collaboration >> Subversive SVN Team provider (Incubation) >> at the end of the installation, restart Eclipse when asked to.

   (b) Open SVN perspective. Eclipse will request to select connector installation for SVN. Select the SVN kit 1.3.2 (1.3.5 might work but was not tested). It will install the Subversive SVN connectors and the SVNKit 1.3.3 implementation.

4. Download and install AJDT plug-in for Eclipse 3.7 (AJDT version 2.1.3).

   (a) Eclipse >> Help >> Install New Software >>
   Work with: http://download.eclipse.org/tools/ajdt/37/update >> close eclipse

5. Create Uninstall Shield by copy the unistallShield.bat to %Eclipse (eclipse root folder), and run it. The uninstall shield may help to uninstall the XRC plug-in, yet if other plug-ins were installed later, it will require to understand the changes, and fix manually the artifacts.xml and bundles.info. The uninstall shield is a simple batch file that will:
    (a) Create a folder uninstall_XRC in the %Eclipse.
    (b) Copy the following files to the uninstall_XRC folder:
        i. %Eclipse/artifacts.xml
        ii. %Eclipse/configuration/org.eclipse.equinox.simpleconfigurator/bundles.info
    (c) Move the following files to the uninstall_XRC folder:
        i. %Eclipse/plugins/org.eclipse.compare_%VER.jar
           e.g. org.eclipse.compare_3.5.201.jar
        ii. %Eclipse/plugins/org.eclipse.jdt.ui_%VER.jar
           e.g. org.eclipse.jdt.ui_3.7.0.201109141219.jar
6. Install the XRC plug-in
    (a) Copy the plug-in jars (compare, jdt and xrc) to the %Eclipse/plugins folder.
    (b) Fix eclipse files to use the plug-in:
        i. %Eclipse/artifacts.xml
            A. Add 1 to the value in "artifacts".
            B. Fix the 3 entries according to the entries in the artifacts.xml of the plug-in (for convenience they can be found at artifactsXRCEntries.xml)
        ii. %Eclipse/configuration/org.eclipse.equinox.simpleconfigurator/bundles.info

A. Fix the 3 entries according to the entries in the artifacts.xml of the plug-in (for convenience they can be found at bundles XRCEntries.info)

7. Now you can explore the demos (Appendix A.4) or simply start using the plug-in.

Enjoy :-).

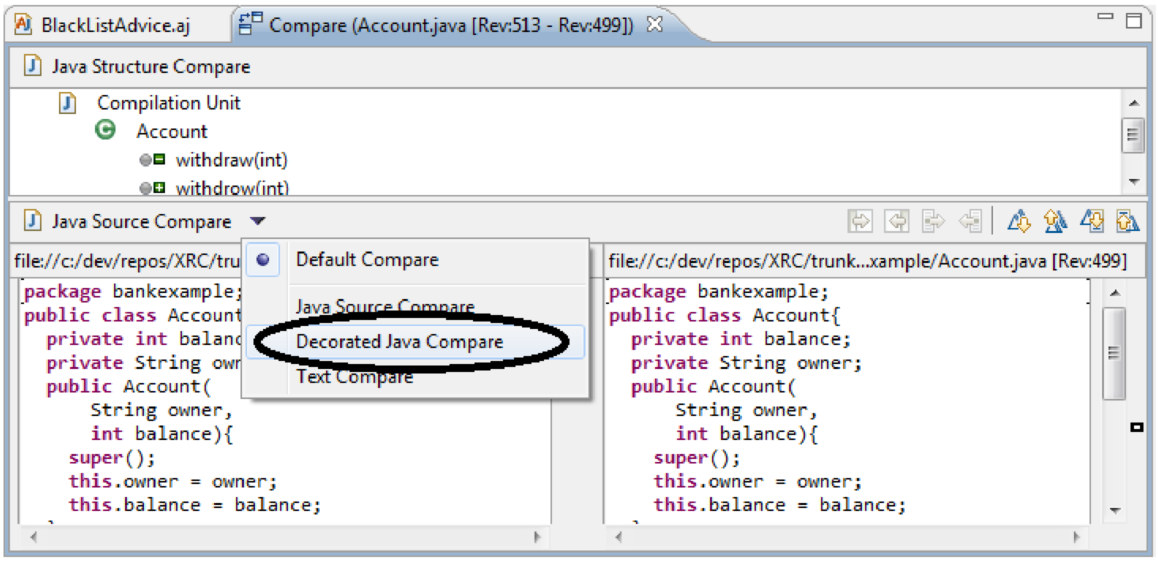
A.3 The XRC plug-in Experience

The XRC plug-in for eclipse changes and improves the user experience:

1. On opening a previous version of a class or an aspect, XRC displays AJDT markers to indicate the aspectual effect in the version (assuming the XRC plug-in was in use while checking-in the file, and the metadata was persisted to the RCS).

2. On comparing versions of an aspect or a class, the compare view opens. The user should select the *Decorated Java Compare*, as showed in Figure A.1 , in order to view the differences in the aspectual effects (assuming the effects were persisted by the XRC)

3. The user should be aware that when checking-in a file (of a class or an aspect) that changes the aspectual effect of other files, their properties in the working copy for the SVN will be modified, so the files will be marked as modified. In Eclipse, select Team >> cleanup to assure the display reflects the status of the files. The user is expected to check-in these files, after reviewing the aspectual effects.



file://c:/dev/repos/XRC/tru

```
package bankexample;
public class Account
  private int balanc
  private String owr
  public Account(
      String owner,
      int balance){
    super();
    this.owner = owner;
    this.balance = balance;
```

file://c:/dev/repos/XRC/trunk...xample/Account.java [Rev:499]

```
package bankexample;
public class Account{
  private int balance;
  private String owner;
  public Account(
      String owner,
      int balance){
    super();
    this.owner = owner;
    this.balance = balance;
```

Figure A.1: Selecting Decorated Java Compare

Some demos and an XRC testing suite are included in the XRC installation kit. To start experimenting with the XRC we encourage checking-out of the code, exploring the demos, and running the XRC Test Suite.

A.4 Explore the Demos

The SVN repository in the XRC installation kit contains several demos. Following are the deployment instructions:

1. Copy the repository folder "repos" to some path: %REPOS

2. Add the repository to SVN in Eclipse:

   (a) Open the *SVN Repository Exploring* perspective

   (b) Create new repository location

   (c) Insert the repository URL, e.g. *file://c:/dev/repos*

3. Select from the XRC trunk in the SVN repository the demos: bankexample_v1, bankexample_v2, testsuite, and check them out from the repos.

4. Explore each of the demos. Feel free to play with the code: check-out, build, save, compare, view previous version, check-in. Remember you can always revert to the version you checked-out, that is stable. If by an accident you find a bug or unexpected behaviour, you are welcome to send a bug report with full description, including how to reproduce the bug, and we will try to explain and/or fix the bug.

A.4.1 Bank Application Version 1

1. Check-out the code of bank application version 1 (bankapplication_v1) as described in A.4.
2. Run the LimitedUserTester unit test. It should pass.
3. Rename the method withdraw in class Account, to withdrow, using the eclipse rename refactoring.
    (a) Team >> cleanup might be required to show that BlackListAdvice was modified as well. Notes:
        i. BlackListAdvice is marked as checked-out, even though it was not modified textually. This is thanks to XRC, that modified its XMD.
        ii. A review of the BlackListAdvice might discover the bug.
4. Check-in the files.
5. Run the test. It is expected to fail.
6. Compare the two last versions of the class Account with regular compare. Only the typo fix is visible.
    (a) To do this comparison: select the Account class >> Team >> Show History >> select the two last versions >> compare with each other

7. Comare the two last versions of the class Account with XRC.
   (a) Switch compare viewer, and select the Decorated Java Compare, by clicking the down-arrow near Java Source Compare \/.
8. Hover the mouse over the AJDT marker. The XMD of the previous version is displayed.
9. Open the previous verion of class Account. The previous version is displayed with the AJDT marker.
   (a) Do it by double-clicking the version in the History pane.
10. Hover the mouse on the marker to see the XMD.
11. Fix the typo to withdraw in the BlackListAdvice, cleanup (Team >> cleanup) to see that the Account class was modified, and commit both classes.
12. Run the test. It will now pass.

A.4.2 Bank Application Version 2

1. Check-out the code of bank application version 2 (bankapplication_v2).
2. Run the ValidatorsTester unit test. It should pass.
3. Illustrate the scenario decribed in Section 3.3.4by changing VipValidatorAspect:
   (a) Comment out line 6 (the first parent declaration), and uncomment line 7 (the second parent declaration), so the VipValidator will extend the UserValidator.
   (b) Comment out line 14 (the statement *_alarmLimit = -10000;* in the declare constructor)

4. check-in the files.

5. Run the test. It is expected to fail.

6. Compare the two last versions of the class VipValidator with regular compare, and later with the XRC. Hover the mouse on the markers to see the XMD. Notice the declare parents details in the hint message were changed, VipValidator extends SpecialUserValidator in the old version, and UserValidator in the newer version.

7. Open the previous version. See the marker, and hover the mouse over it, to see the XMD.

8. Fix the aspect back by undo step 3.

9. Run the test. It is expected to pass.

A.4.3 XRC Test Suite

The XRC Test Suite demonstrates and tests the XRC. Future work is to automate the XRC Test Suite. The manual testing procedures are as following:

1. Verify that a build failure does not modify the XMD:

   (a) Select testsuite > team > cleanup and verify nothing is dirty (marked as modified, with '>' before the filename in the package explorer)

   (b) Modify MyClass: change 'afterMe' to 'af terMe' (with a space), to have a compilation error

      i. Note: the aspects (MyAspect, OtherAspect) actually stop to advise MyClass)

   (c) Save MyClass

   (d) Select testsuite > team > cleanup, and verify no aspect (MyAspect, OtherAspect) is dirty
   (e) Revert all to clean and return to the state as before the test

2. Verify that the XRC modifies the XMD of an advising aspect on a class change that ceased to be advised:
   (a) Select testsuite > team > cleanup, verify nothing is dirty
   (b) Modify MyClass: change declareError to undeclareError, so MyAspect would no longer advise it
   (c) Save MyClass
   (d) Select xrctestsuite > team > cleanup
   (e) Verify MyClass and MyAspect are dirty, and nothing else is dirty (e.g., OtherAspect)

3. Test coverage of the advice markers:
   (a) Open MyClass
   (b) Verify all markers are displayed, and with expected icons
   (c) Modify MyClass (e.g. add a space)
   (d) Save and commit MyClass
   (e) Select team > history and open last revision by clicking it
   (f) Verify all markers are displayed, and with expected icons
   (g) Verify the hints for the markers are making sense when hovering the mouse on the AJDT marker

4. Compare two previous versions of MyClass. Open *Decorated Java Compare* and examine the markers. Verify they make sense

5. Repeat tests 3,4 for MyAspect

6. Verify that the XRC modifies the XMD of an advised class on an aspect change that ceased to advise (Similar to test 2, but this time we change the aspect):

   (a) Change an advice so it will stop to advise a class

   (b) Verify the class was modified

   (c) Save and commit

   (d) View the results in the compare view

7. Play with the code, and verify both on a class, and on an aspect that all the changed advice markers from Figure are displayed

8. Multiple files relationships: verify that XRC functions properlly for a relationship that has more than one target. The XMD for both is saved and displayed

   (a) MyClass is advised by both MyAspect & OtherAspect

   (b) MyAspect advice both MyClass and MyInterface

Appendix B

Developer Guide

This guide explains top down the implementation of the XRC. Code snippets and sequence diagrams illustrates the intricate parts. It comprises installation and release procedures, for XRC developers, as well.

B.1 Implementation of the Architecture

The XRC plug-in extends JDT and AJDT, and uses the SVN plug-ins for Eclipse to provide the XRC solution for Subversion. The interface with the Eclipse IDE is done through extension points. XRC is incorporated into the IDE and RCS using the model-view-controller (MVC) architectural style [27] depicted in Figure B.1.

The XRC implementation reflects its architecture. Development level abstractions and parallel refinements in the plug-in software and in the paper lead to minor differences, mostly in names. Table B.1 maps the components to the packages to ease orientation.

The XRC implementation comprises the following components:

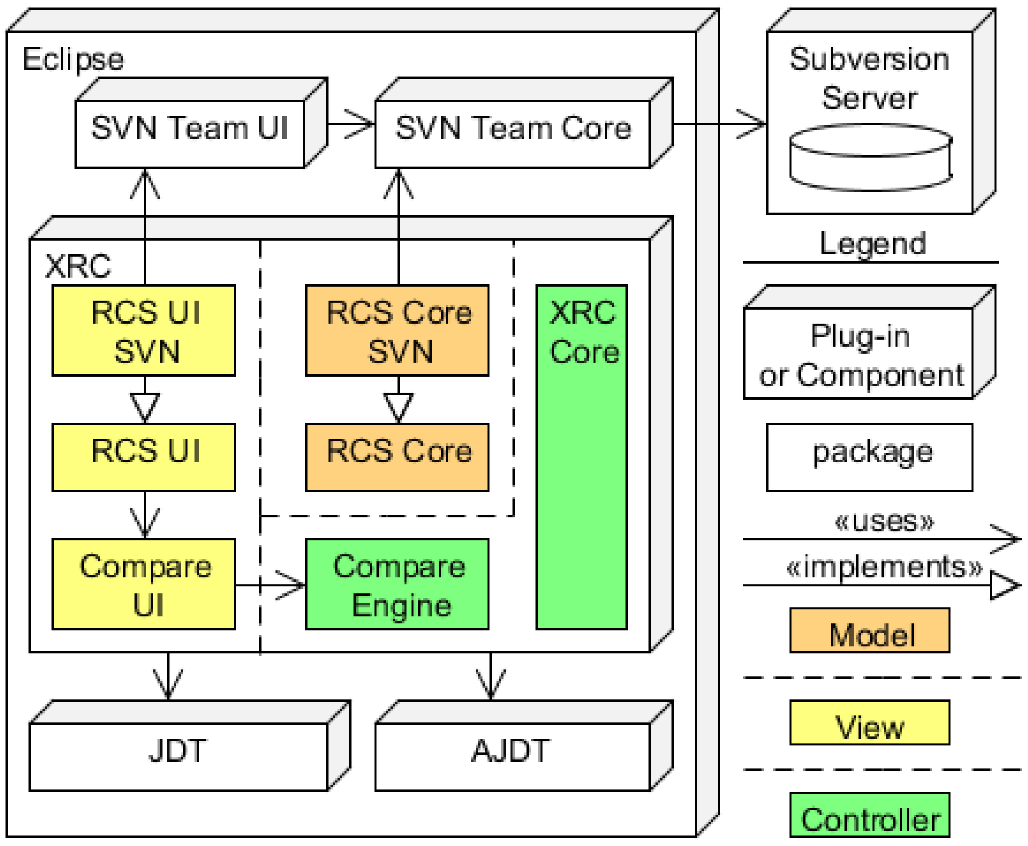


Figure B.1: XRC Architecture

MVC Component	Architecture Component	Software Package under xrc.api	Description
Model	RCS Core	rc	An API and infrastructure for writing and reading metadata, markers information from the RCS
		ide.eclipse.core.dto	The XMD data trasfer object (DTO) for the eclipse IDE
	RCS Core SNV	rc.svn	An implementation of the RCS Core interface for SVN
View	RCS UI	ide.eclipse.io	An API and infrastructure for displaying AJDT and XRC markers
	RCS UI SVN	ide.eclipse.io.svn	An implementation of the RCS UI interface for SVN
	Compare UI	ide.eclipse.plugin-.compare.ui	An API for displaying metadata on versioned files comparisons and on open versions of a file, using markers.
Controller	Compare Engine	compare	An API for comparing metadata and the visual information displayed by markers.
	XRC Core	api.ide.ecplise.core	An infrastructures layer above the JDT and AJDT layers, integrated with the save and build processes, and manages the markers model for XRC. It intercepts and modifies RCS processes, such as save and build.

Table B.1: Components and packages

B.1.1 Model - XMD and Its Persistence

The model defines an *abstract data type (ADT)* for representing XMD, and provides the ability to associate and store the XMD with a source code file. The XMD is the crosscutting metadata of a class (an aspect). It is derived from the versioned aspects (classes) that advise it (are advised by it). Both AJDT and XRC use internally a map ADT to represent the XMD information:

$$Map < Integer, List < IRelationship >>$$

A key in this map represents a line number in the source code. A value in this map is a list of *IRelationship* objects, where *IRelationship* is an interface defined in AJDT. An implementation of *IRelationship* contains the relevant data of a single piece of advice: source, target list, name, and kind of the advice. This map is the XMD in AJDT. The XRC enrich this XMD by modifying the existing members of the *IRelationship* implementation with: the version of the advising aspect, a flag in case the advising aspect was modified locally and differs from its version in the RCS repository, and the details of the parent in a declare parents advice. The class xrc.api.ide.eclipse.core.dto.RelationshipsInfo implements the XMD. The XRC model not only defines and represents the data, but also provides mechanisms to persist data to the RCS:

- *RCS Core* is an API and infrastructure for writing and reading metadata, markers information, etc., from the RCS.
- *RCS Core SVN* is an implementation of the RCS Core interface for SVN.

B.1.2 View - Visual Enhancements

The view provides *Visual enhancements* for displaying XMD. It enables to display markers, (e.g., markers that the Java editor displays for advice) when comparing Java source files and when viewing a previous version. I.e., it enables displaying the crosscutting effect of A on C, both when viewing C and when comparing C and C'.

- *RCS UI* is an API and infrastructure for displaying AJDT and XRC markers. The package api.ide.eclipse.io contains classes to read, and parse metadata from Eclipse IResource file objects , to display markers, and vice versa (parse and write this data).
- *RCS UI SVN* is an implementation of the RCS UI interface for SVN. It is coded in the package api.ide.eclipse.io.svn.
- *Compare UI* is an API for displaying metadata on versioned files comparisons and on opened versions of a file, using markers. Its code is in the package api.ide.eclipse.plugin.compare.ui.

B.1.3 Controller

The controller performs *diff* to compare XMD on request, and modifies the IDE and RCS policies for save, check-in, build, view previous version and compare. By tracking and by managing the crosscutting effect of aspects, it updates the XMD "at the right time.". It consists of two architectural components: the *Compare Engine* , and the *XRC Core.*

B.1.3.1 Compare Engine

The *diff* engine enables to compare XMD of two files (or two versions of the same file). It is an API for comparing metadata and the visual information displayed by markers. The interface xrc.api.compare.IRelationshipsInfoDiffEngine declares the diff method, which is implemented in the class xrc.api.compare.engine.RelationshipsInfoDiffEngine. The XRC *diff* engine compares $XMD(C)$ with $XMD(C')$ by first "flattening" the data, then computing the differences, and finally "inflating" the result back into a format that Eclipse understands:

$$diff\,(XMD(C), XMD(C')) = \lceil \Delta\left(\lfloor XMD(C) \rfloor, \lfloor XMD(C') \rfloor\right) \rceil$$

For this, XRC uses two methods that translate the XMD representation required for display to the representation required for comparison, and back:

- $flatten : XMD \rightarrow Set < IRelationship >$, denoted $\lfloor . \rfloor$, takes an XMD object, and returns a $Set < IRelationship >$ based on the $IRelationship$ objects in the XMD. It breaks each $IRelationship$ with multiple targets into single-target $IRelationship$ objects, and removes the line numbers. The implementation is in the IRelationshipsInfoDiffEngine class, and the method signature is Set<IRelationship> flat(Map<Integer, List<IRelationship>> data).

- $inflate : (XMD, Set < IRelationship >) \rightarrow XMD$, denoted $\lceil . \rceil$, takes an XMD object, and a $Set < IRelationship >$. It constructs and returns a new XMD with the $IRelationship$ objects from the input set restored to their original structure and line numbers, accumulating targets of the same line to a list in order for Eclipse to be able to use the data for displaying the XMD. The implementation is in the IRelationshipsInfoDiffEngine class, and the method signature is RelationshipsInfo inflate(RelationshipsInfo relationshipsInfo, Set<IRelationship> relationshipSet).

The XMD representation required for display is of the *map-form:* $Map < Integer, List < IRelationship >>$. The flatten method translates it into the *set-form:* $Set < IRelationship >$ where each $IRelationship$ has a single-target. The set representation is used for the comparison.

```
XMD createDeltaXMD(XMD minuendXMD,
            XMD subtrahendXMD){
    return inflate(minuendXMD,
        (flatten(minuendXMD)
        .subtract(
            flatten(subtrahendXMD)))
}
```

Listing B.1: The method createDeltaXMD from the class XMDDiffEngine

```
public aspect A {
    before(): execution(void m1*()){ System.out.println("before m1*");
         }
}
```

Listing B.2: Aspect A, version 430

Compare Engine Internals Illustration This section illustrates XMD in map-form, and set-form, and a subtraction of XMD.

The *diff* engine uses createDeltaXMD (Listing B.1) and other set manipulations, and returns an object representing the differences.

Listings B.2 and B.3 introduce an aspect A that advises a class C. In the following revision of C as seen in Listing B.4, the method m12 has changed to m2, and the methods switched locations.

Listing B.5 illustrates the XMD of each file in a simplified map-form, ignoring the name and kind fields of the *IRelationship*. Listing B.6 illustrates XMD subtraction using translation from map-form to set-form and back.

```
public class C {
    public void m11(){ System.out.println("in method m11"); }
    public void m12(){ System.out.println("in method m12"); }
}
```

Listing B.3: Class C, version 430

```
public class C {
    public void m2(){ System.out.println("in method m2"); }
    public void m11(){ System.out.println("in method m11"); }
}
```

Listing B.4: Class C, version 431. Method m12 has changed to m2, and the methods switched locations

```
XMD(A430): {2=>(IRelationship[source=A.a, targetList=(C>430.m11,C>430.m12
    )])}
XMD(C430): {2=>(IRelationship[source=C.m11, targetList=(A>430.a)]),
             3=>(IRelationship[source=C.m12, targetList=(A>430.a)])}
XMD(C431): {3=>(IRelationship[source=C.m11, targetList=(A>430.a)])}

Data Structure Legend:
Map: {key=>value, key2=>value2, ...}
Set: {element1, element2, ...}
List: (element1, element2, ...)
Values of field members in an object O: O[member1=valueOfMember1, member2=
    valueOfMember2, ...]
Enrichment of a target with a version: >NNN
```

Listing B.5: XMD in a map-form (simplified)

```
flatten(XMD(A430))={IRelationship[source=A.a, targetList=(C>430.m11)],
                     IRelationship[source=A.a, targetList=(C>430.m12)]}
f430=flatten(XMD(C430))={IRelationship[source=C.m11, targetList=(A>430.a)],
                          IRelationship[source=C.m12, targetList=(A>430.a)]}
f431=flatten(XMD(C431))={IRelationship[source=C.m11, targetList=(A>430.a)]}

f430−f431 = {IRelationship[source=C.m12, targetList=(A>430.a)]}
inflate(XMD(C430), f430−f431)={3=>(IRelationship[source=C.m12, targetList=(
    A>430.a)])}
```

Listing B.6: Illustration of XMD manipulations. Translate XMDs to a set-form, subtract them, and translate back to a map-form

The diff result ADT (IXMDDifference) In order to support a fine grained presentation, the diff engine returns an IXMDDifference, an abstract data structure that contains eight sets of metadata, each is XMD in a map-form:

v_1, $v_1 Removed$, $v_1 AddedToExist$,$v_1 AddedToNonExist$ for the older version in the comparison, v_1 and symmetrically:

v_2, $v_2 Added$, $v_2 RemovedToExist$,$v_2 RemovedToNonExist$ for the newer version v_2.

v_1 - enable to display all the markers on the side of v_1.

$v_1 Removed$ - identifies the markers that exist on the side of v_1, and do not exist in the the side of v_2 (enable to display icon for removed).

$v_1 AddedToExist$ - identifies the markers that were added on the side of v_2, yet there were markers on the same line in the the side of v_1 (enable to display icon for add).

$v_1 AddedToNonExist$ - identifies the markers that were added on the side of v_2, where no markers existed on the same line in the side of v_1 (enable to display icon for add).

The discrimination between the last two items allows to display them differently. E.g., to highlight the advice markers in $v_1 AddedToExist$, while keeping the ruler not highlighted (since no advice markers are to highlight) in $v_1 AddedToNonExist$. We chose in the current XRC version to highlights both (highlight without an advice marker for $v_1 AddedToNonExist$), since we find it more convenience. The XMDs for v_2 are symmetrical.

The diff algorithm Listing B.7 shows the creation of the IXMDDifference (the diff result ADT). The metadata is manipulated using previous results, and reusing generic methods, leading to a clean, elegant and efficient implementation.

```
IXMDDifference diff(XMD v1XMD, XMD v2XMD){
  XMD v1RemovedXMD = createDeltaXMD(v1XMD, v2XMD);
  XMD v2AddedXMD = createDeltaXMD(v2XMD, v1XMD);
//the relationships that have been changed on v1 side
  XMD v1AddedToExistXMD = createModifiedExist(v1XMD, v2AddedXMD );
//the relationships that have been changed on v2 side
  XMD v2RemovedToExistXMD = createModifiedExist(v2XMD, v1RemovedXMD );
//the relationships that are new on v1 side (where they don't exist
    yet)
  XMD v1AddedToNonExistXMD = createModifiedNonExist(v1XMD, v2AddedXMD,
       v1AddedToExistXMD);
//the relationships that have been removed on v2 side (where they no
    longer exist)
  XMD v2RemovedToNonExistXMD = createModifiedNonExist(v2XMD,
      v1RemovedXMD, v2RemovedToExistXMD);
  IXMDDifference xmdDifference = new XMDDifference(v1XMD, v1RemovedXMD
      , v1AddedToExistXMD, v1AddedToNonExistXMD, v2XMD, v2AddedXMD,
      v2RemovedToExistXMD, v2RemovedToNonExistXMD);
  return xmdDifference;
}
```

Listing B.7: The method diff from the class XMDDiffEngine

Listing B.8 illustrates subtraction of XMD, After flattening the arguments to set-form, they are subtracted as sets, and then inflate back.

The method createModifiedExist in Listing B.9, tests for each IRelationship in the XMD representing otherSideDelta, if it is also belongs to the XMD that represents the first side. If the answer is true, the IRelationship is added with its line to an XMD named sideModifiedExist. This identifies the lines in which there is a marker

```
XMD createDeltaXMD(XMD minuendXMD,
      XMD subtrahendXMD){
  return inflate(minuendXMD,
    (flat(minuendXMD)
    .subtract(
       flat(subtrahendXMD)))
}
```

Listing B.8: The method createDeltaXMD from the class XMDDiffEngine

```java
XMD createModifiedExist(XMD side, XMD otherSideDelta){
  XMD sideModifiedExist = new RelationshipsInfo();
  for (IRelationship otherSideDeltaRelationship : flat(otherSideDelta)
     ){
    line = side.findLineForSourceHandle(otherSideDeltaRelationship.
        getSourceHandle());
    sideModifiedExist.add(line, otherSideDeltaRelationship);
  }
  return sideModifiedExist;
}
```

Listing B.9: The method createModifiedExist from the class XMDDiffEngine

exists in the $v1$ (“before”) side, and it is modified.

createModifiedNonExist is implemented by reusing the createDeltaXMD method and sets manipulations, and we will not bother with the details here (the implementation code is clear enough).

B.1.3.2 XRC Core - RCS–IDE Integration

The *XRC Core* is an infrastructure layer above the JDT and AJDT layers, integrated with the save, build, check-in, history, and compare processes, and manages the markers model for XRC. It intercepts and modifies RCS processes, such as save and build.

save The save sequence diagram in Figure B.2 details the XRC's intervention in the save process. Figure B.3 depicts a more accurate and detailed sequence diagram.

1. **Execute JDT's save process**. In the plugin.xml of the XRC plug-in, CompilationUnitDocumentProvider4XRC is registered in the extension point org.eclipse.core.filebuffers.annotationModelCreation. When the developer saves a file, the JDT's save process runs. During its run, when the eclipse core calls the commit, it uses the commit implementation from CompilationUnitAnnota-

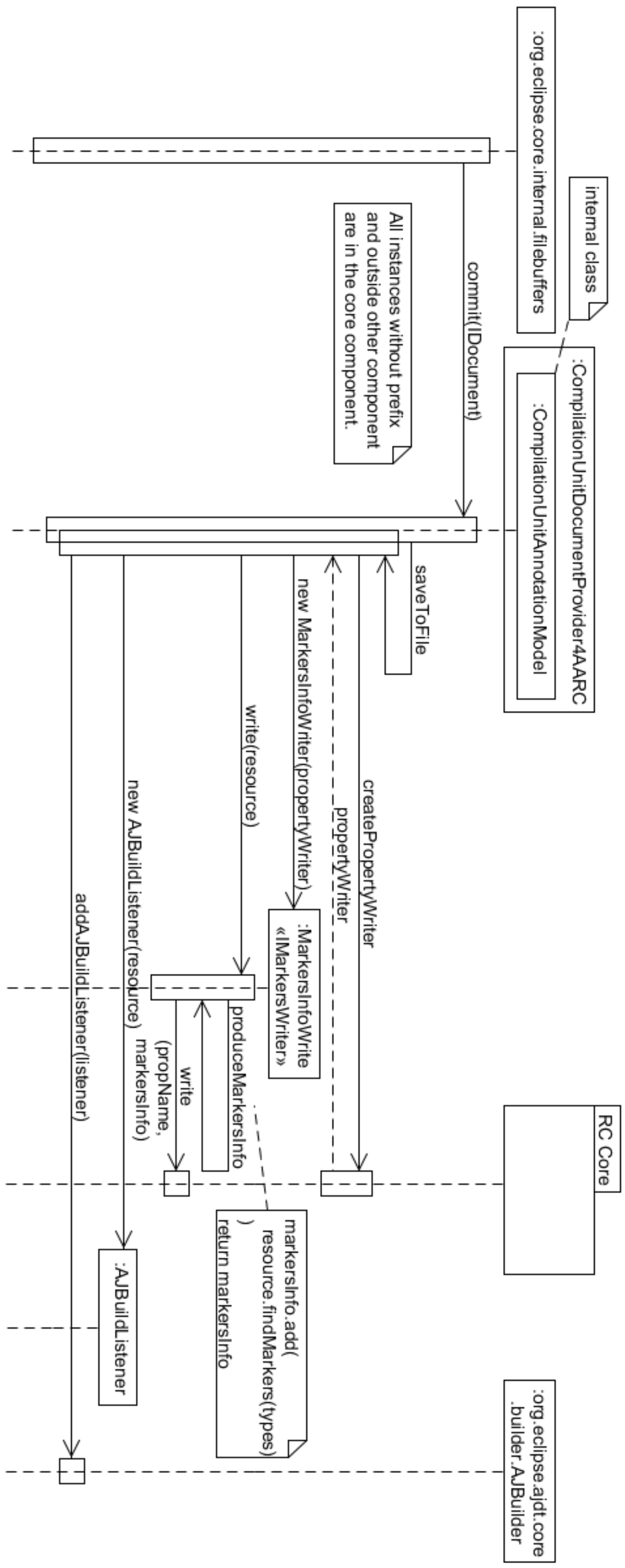
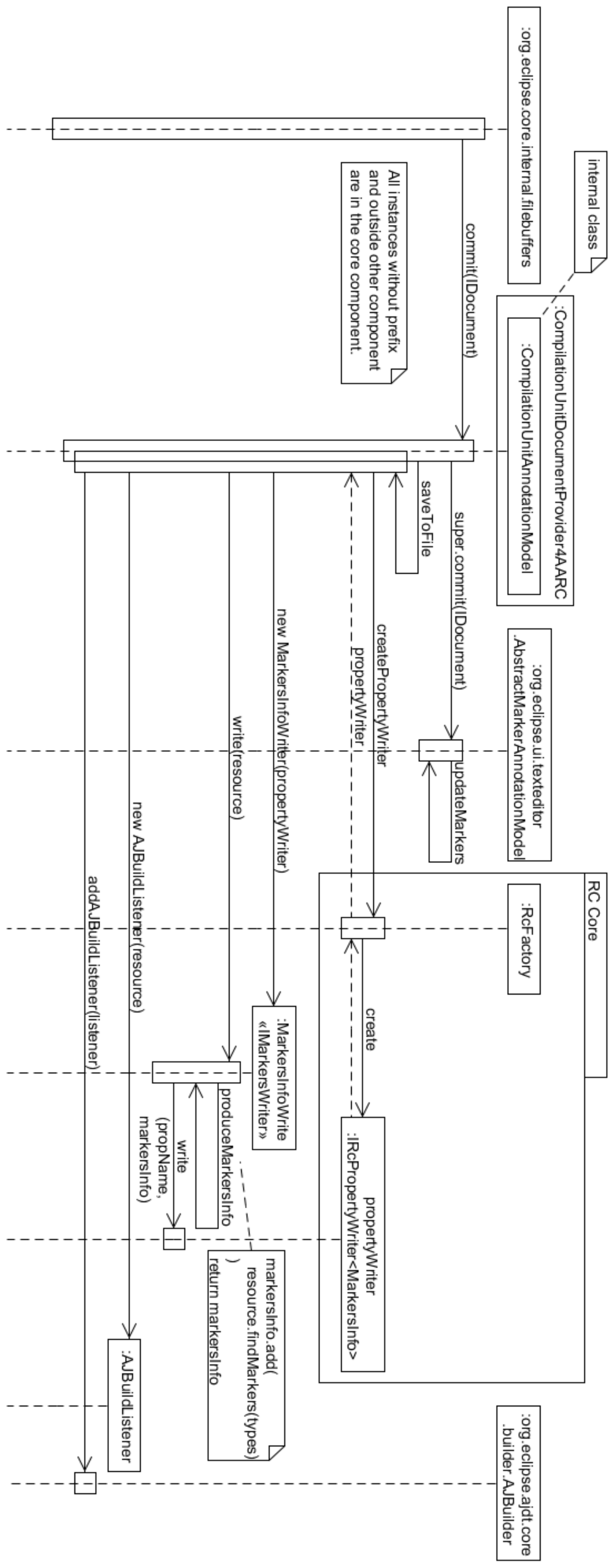

:org.eclipse.core.internal.filebuffers
internal class
All instances without prefix and outside other component are in the core component.
commit(IDocument)
:CompilationUnitDocumentProvider4AARC
:CompilationUnitAnnotationModel
saveToFile
createPropertyWriter
propertyWriter
new MarkersInfoWriter(propertyWriter)
write(resource)
new AJBuildListener(resource)
addAJBuildListener(listener)
:MarkersInfoWrite «IMarkersWriter»
produceMarkersInfo
write (propName, markersInfo)
RC Core
markersInfo.add(resource.findMarkers(types)) return markersInfo
:AJBuildListener
:org.eclipse.ajdt.core .builder.AJBuilder

Figure B.2: Save sequence diagram (simplified)



:org.eclipse.core.internal.filebuffers
internal class
All instances without prefix and outside other component are in the core component.
commit(IDocument)
:CompilationUnitDocumentProvider4AARC
:CompilationUnitAnnotationModel
super.commit(IDocument)
saveToFile
createPropertyWriter
propertyWriter
new MarkersInfoWriter(propertyWriter)
write(resource)
new AJBuildListener(resource)
addAJBuildListener(listener)
:org.eclipse.ui.texteditor .AbstractMarkerAnnotationModel
updateMarkers
RC Core
:RcFactory
create
propertyWriter :IRcPropertyWriter<MarkersInfo>
:MarkersInfoWrite «IMarkersWriter»
produceMarkersInfo
write (propName, markersInfo)
markersInfo.add(resource.findMarkers(types)) return markersInfo
:AJBuildListener
:org.eclipse.ajdt.core .builder.AJBuilder

Figure B.3: Save sequence diagram

tionModel, which is internal class of CompilationUnitDocumentProvider4XRC (as seen in Figure B.2). This is the way we gain control over the save process.

2. **Extract and save the markers.** We use the RC Core to get a propertyWriter (note that XRC core uses interfaces and abstractions, and it is independent of the specific implementation of the RC Core interfaces, replace of RC Core SVN is transparent to the core). The CompilationUnitAnnotationModel creates MarkersInfoWriter, and call it to write the resource (for the file being saved). The MarkersInfoWriter extract the markers from the resource, into MarkersInfo object, and save it in the RCS, using the propertyWriter. The save is in a temporary manner, as saving local file. Only on commit the XMD in the MarkersInfo will actually be versioned, with the file it belongs to.

3. **Create and add AJBuildListener with the saved resource.** The CompilationUnitAnnotationModel creates an AJBuildListener for the resource, and register in on the AJBuilder, in order to handle this resource on the next build. (see Section B.1.3.2).

build The build sequence diagram in Figure B.4 details the XRC's intervention in the build process. Figure B.5 depicts a more accurate and detailed sequence diagram.

1. **Execute JDT's and AJDT's build processes.** XRC's work is post build, when the models in JDT and AJDT cores are already updated. XRC hooks to the postAJBuild hook by using an AJBuildListener. The listeners are registered when saving a file.

2. **Find advice targets.** XRC compares the XMD of the file with its predecessor, and analyzes which of the affected files differ. The RelationshipsInfoWriter uses JDT and AJDT cores, and the RelationshipParser to find the relationships, and the targets of the advice, i.e., the advised classes.

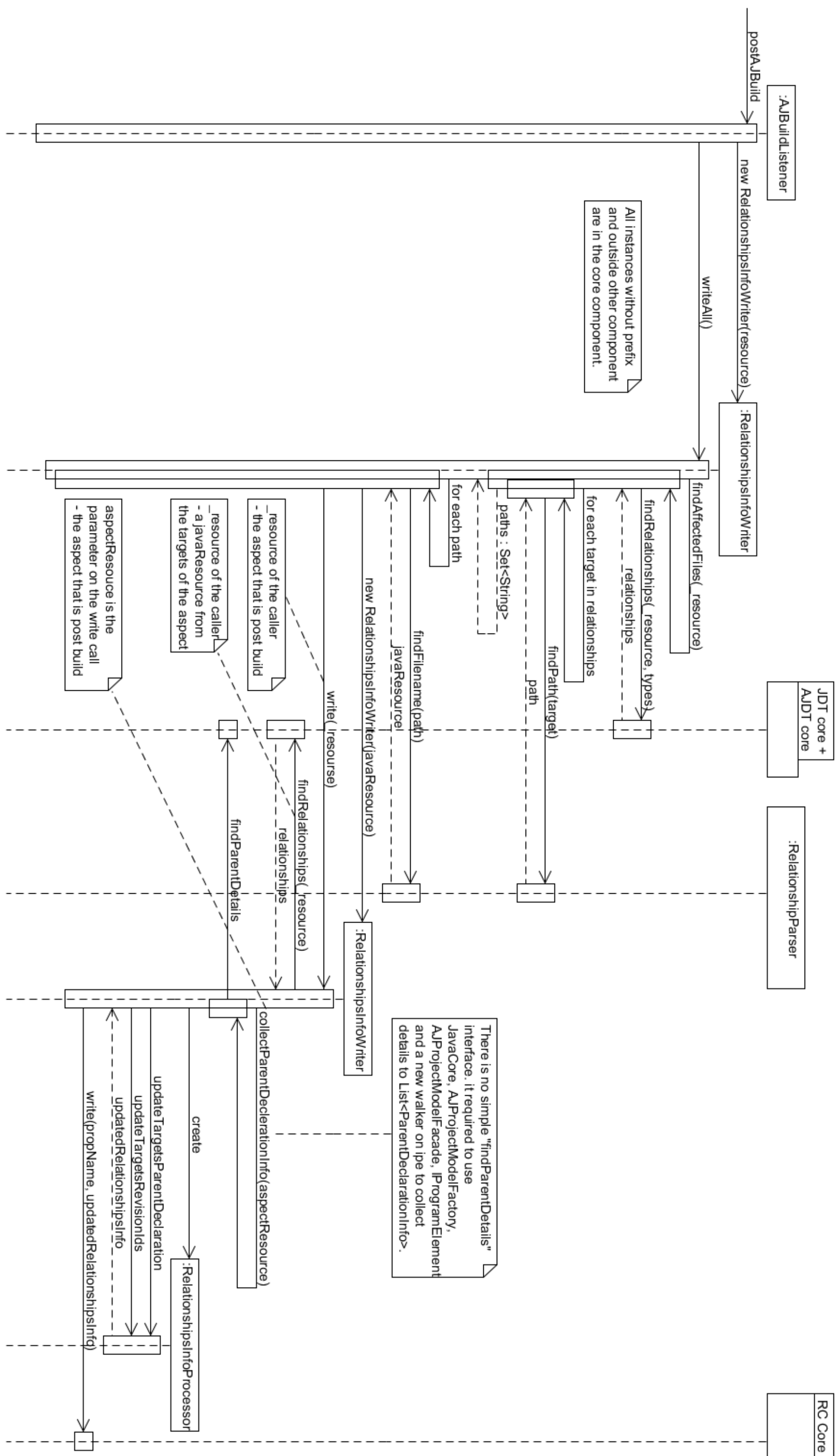

:AJBuildListener
postAJBuild
new RelationshipsInfoWriter(resource)
writeAll()
All instances without prefix and outside other component are in the core component.
:RelationshipsInfoWriter
findAffectedFiles(_resource)
findRelationships(_resource, types)
relationships
for each target in relationships
findPath(target)
path
paths : Set<String>
for each path
findFilename(path)
javaResource
new RelationshipsInfoWriter(javaResource)
write(_resourse)
resource of the caller - the aspect that is post build
_resource of the caller - a javaResource from the targets of the aspect
aspectResouce is the parameter on the write call - the aspect that is post build
JDT core + AJDT core
:RelationshipParser
:RelationshipsInfoWriter
findRelationships(_resource)
relationships
findParentDetails
collectParentDeclarationInfo(aspectResource)
There is no simple "findParentDetails" interface: it required to use JavaCore, AJProjectModelFactory, AJProjectModelFacade, IProgramElement and a new walker on ipe to collect details to List<ParentDeclarationInfo>.
create
:RelationshipsInfoProcessor
updateTargetsParentDeclaration
updateTargetsRevisionIds
updatedRelationshipsInfo
write(propName, updatedRelationshipsInfo)
RC Core

Figure B.4: Build (simplified)

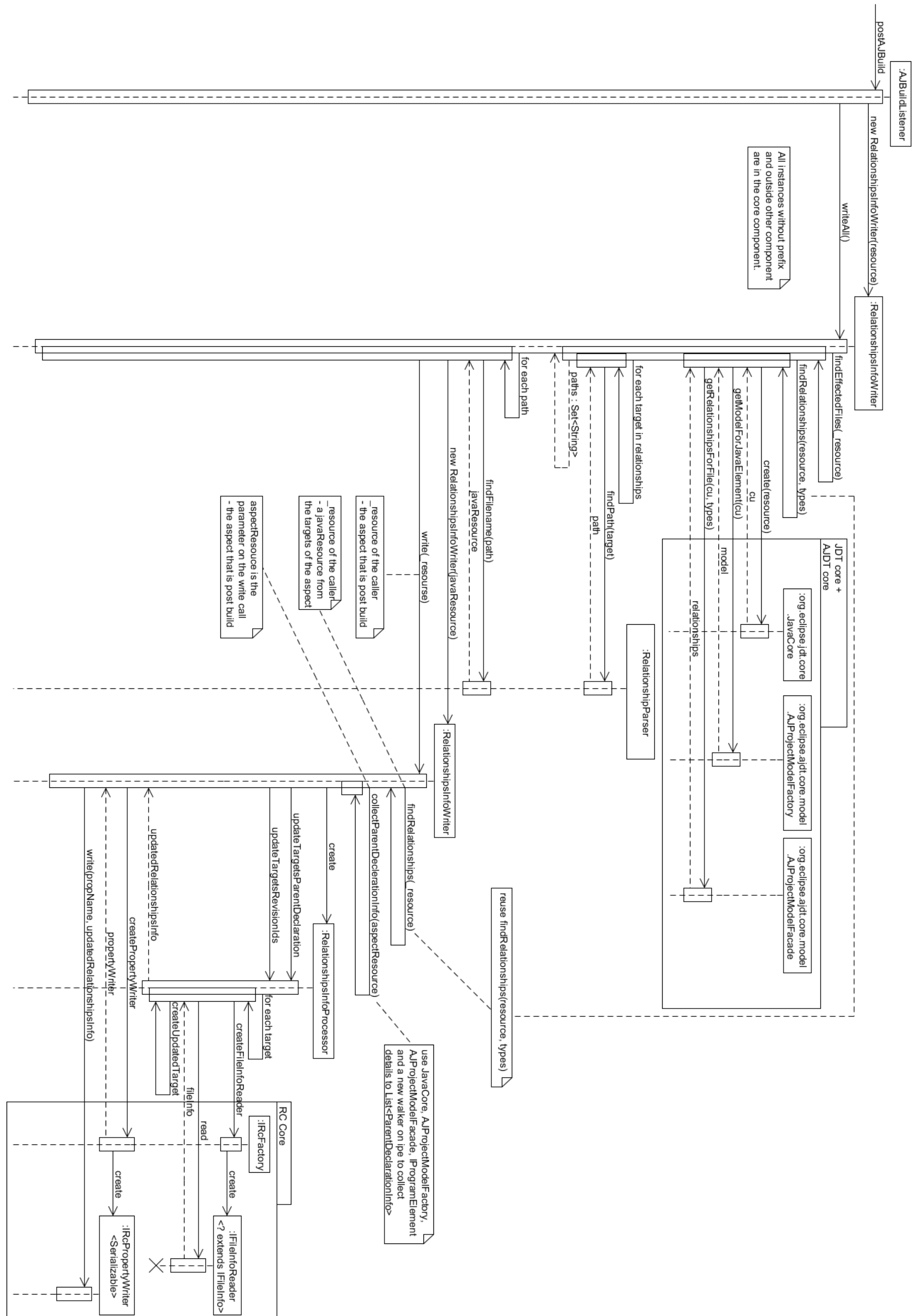

:AJBuildListener
postAJBuild
new RelationshipsInfoWriter(resource)
writeAll()
All instances without prefix and outside other component are in the core component.
:RelationshipsInfoWriter
findEffectedFiles(_resource)
findRelationships(resource, types)
create(resource)
cu
getModelForJavaElement(cu)
model
getRelationshipsForFile(cu, types)
relationships
for each target in relationships
findPath(target)
path
paths : Set<String>
for each path
findFilename(path)
javaResource
new RelationshipsInfoWriter(javaResource)
write(_resourse)
JDT core + AJDT core
:org.eclipse.jdt.core.JavaCore
:org.eclipse.ajdt.core.model.AJProjectModelFactory
:org.eclipse.ajdt.core.model.AJProjectModelFacade
:RelationshipParser
_resource of the caller - the aspect that is post build
_resource of the caller - a javaResource from the targets of the aspect
aspectResouce is the parameter on the write call - the aspect that is post build
:RelationshipsInfoWriter
findRelationships(_resource)
collectParentDeclarationInfo(aspectResource)
create
updateTargetsParentDeclaration
updateTargetsRevisionIds
updatedRelationshipsInfo
createPropertyWriter
propertyWriter
write(propName, updatedRelationshipsInfo)
reuse findRelationships(resource, types)
use JavaCore, AJProjectModelFactory, AJProjectModelFacade, IProgramElement and a new walker on ipe to collect details to List<ParentDeclarationInfo>
:RelationshipsInfoProcessor
for each target
createFileInfoReader
fileInfo
createUpdatedTarget
read
RC Core
:IRcFactory
create
create
:IFileInfoReader <? extends IFileInfo>
:IRcPropertyWriter <Serializable>

Figure B.5: Build

3. **Extract and Enhance the XMD.**

   (a) Create RelationshipsInfoWriter per target. A new RelationshipsInfoWriter is assigned for each advised class, in order to find its XMD, enhance and persist it. The RelationshipsInfoWriter Extracts the XMD from the resource. Meaning, find the advised class's relationships, via JDT and AJDT cores, and use it to create the RelationshipsInfo object, that represents the XMD.

   (b) The RelationshipsInfoProcessor assist to enhance it with XRC's additional data. This data is gathered both by the RelationshipsInfoWriter, which collects additional details on the XMD using JDT and AJDT cores (e.g., parents details); and both by the RelationshipsInfoProcessor and RC Core that aggregate information on the revisions. The RelationshipsInfoProcessor updates this XRC's relevant data in the XMD (parents details, target's revisions, and dirty flag).

4. **Write the XMD.** The RelationshipsInfoWriter use the RC Core interfaces to write the enhanced XMD as a property of the working copy's file. The file is then marked as dirty. SVN marks the file as dirty automatically when its properties are modified.

Note: The post build assumes save and build made the file and AJDT's XMD correct.

compare The compare sequence diagram in Figure B.6 details the XRC's intervention in the process of comparison between two versions. Figure B.8 depicts a more accurate and detailed sequence diagram. Figure B.7 spans the type hierarchy of DecoratedJavaMergeViewer.

1. **Execute JDT's compare process**. In the plugin.xml of the XRC plug-in,

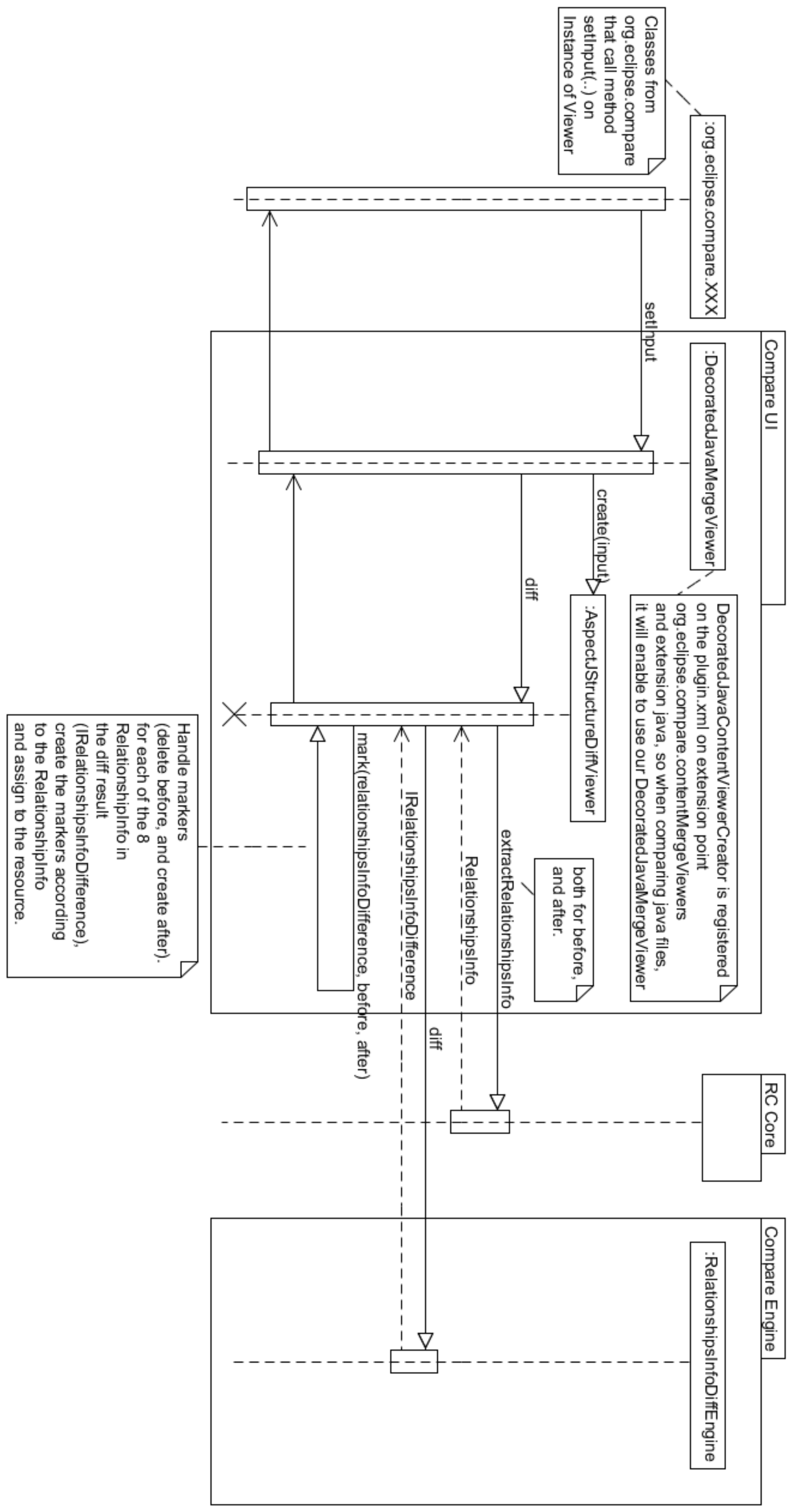

:org.eclipse.compare.XXX
Classes from org.eclipse.compare that call method setInput(..) on Instance of Viewer
setInput
Compare UI
:DecoratedJavaMergeViewer
create(input)
diff
:AspectJStructureDiffViewer
DecoratedJavaContentViewerCreator is registered on the plugin.xml on extension point org.eclipse.compare.contentMergeViewers and extension java, so when comparing java files, it will enable to use our DecoratedJavaMergeViewer
extractRelationshipsInfo
both for before, and after.
RelationshipsInfo
IRelationshipsInfoDifference
diff
mark(relationshipsInfoDifference, before, after)
Handle markers (delete before, and create after). for each of the 8 RelationshipInfo in the diff result (IRelationshipsInfoDifference), create the markers according to the RelationshipInfo and assign to the resource.
RC Core
Compare Engine
:RelationshipsInfoDiffEngine

Figure B.6: Compare (simplified)

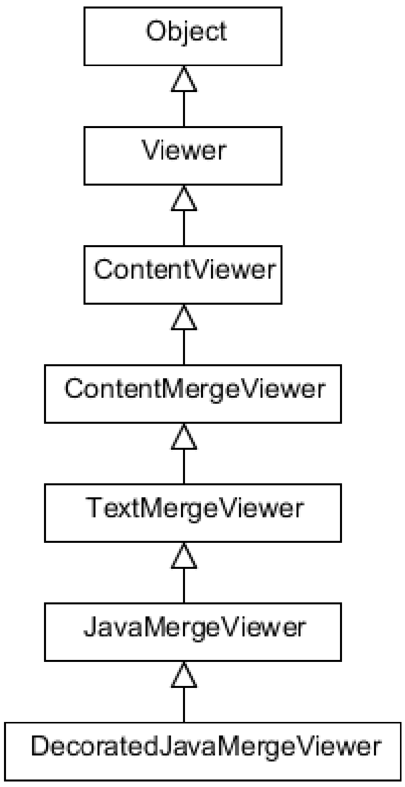
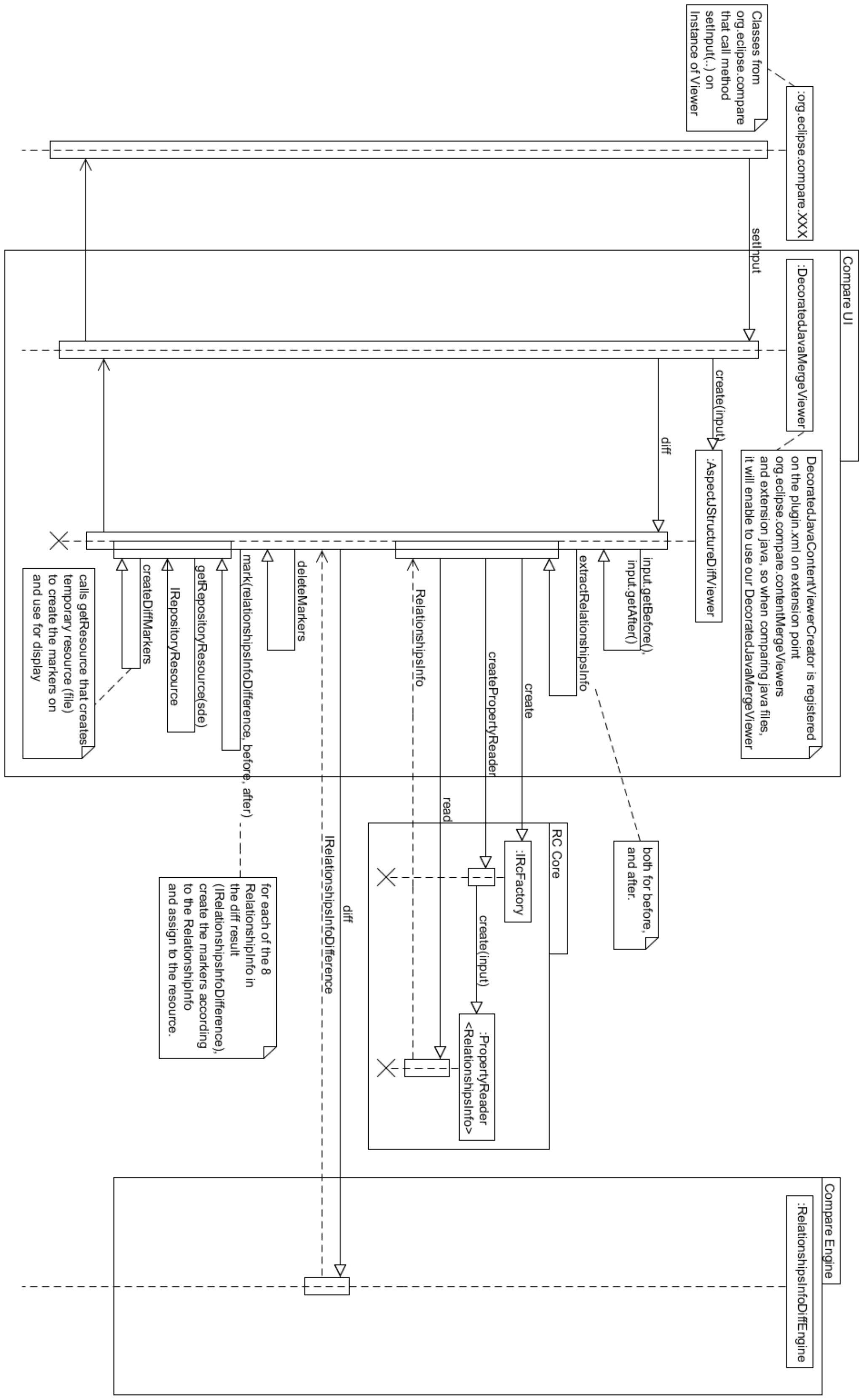


Figure B.7: Type hierarchy of DecoratedJavaMergeViewer. DecoratedJava-ContentViewerCreator is registered in the plugin.xml on extension point org.eclipse.compare.contentMergeViewers and extensions .java and .aj, consequently, when comparing java files, Eclipse enables using DecoratedJavaMergeViewer



:org.eclipse.compare.XXX
Classes from org.eclipse.compare that call method setInput(..) on Instance of Viewer
setInput
Compare UI
:DecoratedJavaMergeViewer
create(input)
diff
:AspectJStructureDiffViewer
DecoratedJavaContentViewerCreator is registered on the plugin.xml on extension point org.eclipse.compare.contentMergeViewers and extension java, so when comparing java files, it will enable to use our DecoratedJavaMergeViewer
input.getBefore(), input.getAfter()
extractRelationshipsInfo
create
createPropertyReader
RelationshipsInfo
read
deleteMarkers
mark(relationshipsInfoDifference, before, after)
getRepositoryResource(sde)
IRepositoryResource
createDiffMarkers
calls getResource that creates temporary resource (file) to create the markers on and use for display
both for before, and after.
RC Core
:IRcFactory
create(input)
:PropertyReader <RelationshipsInfo>
diff
IRelationshipsInfoDifference
for each of the 8 RelationshipInfo in the diff result (IRelationshipsInfoDifference), create the markers according to the RelationshipInfo and assign to the resource.
Compare Engine
:RelationshipsInfoDiffEngine

Figure B.8: Compare

DecoratedJavaContentViewerCreator is registered in the extension point org.eclipse.compare.contentMergeViewers and extensions .java, and .aj, so when comparing java files, it will enable use DecoratedJavaMergeViewer with the rulers.

2. **Read the *XMD*.** Read the XMD from RC Core for each version.

3. **Run the diff engine.** The diff engine compares the *XMD* of two files, and return an IXMDDifference with the differences.

4. **Create and display the markeres.** The AspectJStructureDiffViewer creates markers according to the IRelationshipsInfoDifference. The markers are assigned to the rulers, both for the advice ruler, and for the diff ruler in each side of the Compare View.

misc.

Check-in. SVN check-in also the properties, no change is required .

Display a previous version. When reading a file from SVN, XRC reads also the markers from SVN, and display them with the old version.

B.2 The XRC Developer Installation Procedures

The *XRC Developer Installation* supports developers that would like to enhance the XRC plug-in, to explore its internals, to reuse some of its code, to migrate it for other Eclipse version, etc.

1. Preconditions: Eclipse and Subversion are installed

2. Check-out the xrc from SVN, as described in Section A.4 (Note that errors exist because the plugins of aspectJ and svn are not installed).

3. Install AJDT:

   (a) Eclipse >> Help >> Install New Software >>
   Work with: http://download.eclipse.org/tools/ajdt/42/update

4. Install SVN subversive plugins:

   (a) Eclipse >> Help >> Install New Software >>
   Work with: Juno - http://download.eclipse.org/releases/juno under Collaboration,

   (b) On re-openning Eclipse, Subversive Connector discovery opens (open SVN perspective if it doesn't happen automatically). Select the svn connector: SVN Kit 1.3.7, and Native JavaHL 1.6.15

5. Prepare to import the compare and jdt plug-ins:

   (a) Open the %Eclipse/plugins/org.eclipse.compare_%VER.jar, look at the META-INF/MANIFEST.MF, and find the following entries:

      i. Eclipse-SourceReferences, e.g.
      scm:git:git://git.eclipse.org/gitroot/platform/eclipse.platform.team.git.
      From this value we will derive the repository URI, e.g.
      git://git.eclipse.org/gitroot/platform/eclipse.platform.team.git. Lets denote its value %SOURCE_URI

      ii. Bundle-Version, e.g. 3.5.300.v20120522-1148, lets denote its value %VER

      iii. Troubleshooting:

A. %VER can be found in the suffix of the name of the plugin, as well

B. Known %SOURCE_URI for a new plug-ins are:

- git://git.eclipse.org/gitroot/platform/eclipse.platform.team.git (for the compare plug-in)
- git://git.eclipse.org/gitroot/jdt/eclipse.jdt.ui.git (for the jdt plug-in)

C. The cvs properties for old versions are:

- Connection type: pserver
- User: anonymous
- Host: dev.eclipse.org
- Port: default
- Repository path: /cvsroot/eclipse
- Module: org.eclipse.compare/plugins/org.eclipse.compare (for the compare plug-in)
- Module: org.eclipse.jdt.ui (for the jdt plug-in)

6. Import the compare and jdt plug-ins code as projects:

(a) If requiered, for git URI, install EGIT:

i. Eclipse >> Help >> Install New Software >>
Work with: http://downs versions:load.eclipse.org/egit/updates/ Eclipse GIT Team Provider >> Eclipse GIT

(b) Use cvs or git to import the plug-ins as projects:

i. Use EGIT:

A. Open GIT perspective

```java
public synchronized void setCompilationUnitDocumentProvider(
    ICompilationUnitDocumentProvider cudp){
  fCompilationUnitDocumentProvider = cudp;
}
```

Listing B.10: The method setCompilationUnitDocumentProvider in class JavaPlugin

B. Paste the %SOURCE_URI, and select the branch (after Deselect all)

C. Open the tags node, select the tag according to the %VER and check it out

D. Select the plug-ins (org.eclipse.compare, org.eclipse.jdt.ui) and import the project.

ii. Use CVS:

A. Open CVS perspective

B. Create new repository locations with the properties

C. Open Java perspective

D. Import the projects (org.eclipse.compare, org.eclipse.jdt.ui) from CVS

7. Fix the jdt source code in the checked-out *org.eclipse.jdt.ui* project:

   (a) Change in the MANIFEST.MF the Bundle-Version value to be the version with .xrc suffix instead of its tag, e.g. Bundle-Version:3.8.0.xrc

   (b) Add the method in Listing B.10 at the end of org.eclipse.jdt.ui.JavaPlugin.

   (a) Comment out in plugin.xml the *extention* section with the property point="org.eclipse.core.filebuffers.annotationModelCreation"

   (b) Add public modifier for class declaration of the following classes:

      i. org.eclipse.jdt.internal.ui.compare.EclipsePreferencesAdapter

    ii. org.eclipse.jdt.internal.ui.compare.JavaNode

    iii. org.eclipse.jdt.internal.ui.compare.JavaStructureDiffViewer

    iv. org.eclipse.jdt.internal.ui.javaeditor.JavaEditorMessages

   (c) Add public modifier for the following methods:

    i. In org.eclipse.jdt.internal.ui.compare.JavaCompareUtilities:

     A. static JavaTextTools getJavaTextTools()

     B. static IDocumentPartitioner createJavaPartitioner()

    ii. In org.eclipse.jdt.internal.ui.javaeditor.JavaMarkerAnnotation:

     A. static final boolean isJavaAnnotation(IMarker marker)

8. Fix the compare source code in the checked-out *org.eclipse.compare* project:

   (a) Change in the MANIFEST.MF the Bundle-Version value to be the version with .xrc suffix instead of its tag, e.g. Bundle-Version:3.5.300.xrc

   (b) Compare the TextMergeViewer from the compare project, to the one from the installation kit, in the xrc_supplemental in the svn repository. The expected difference is switching of a small section (4 lines) of the code. Do this change in the compare project.

9. Fix the XRC source code in the checked-out *xrc* project:

   (a) Fix the dependencies in the MANIFEST.MF for the compare and jdt plug-ins to be with the specific version of the projects (with the .xrc suffix). Eclipse allows to do so in a nice way via Dependencies tab >> required plug-ins when editing the MANIFEST.MF.

   (b) Fix the value of the string log4jPath in the class ConfigManager to reflect its location (it is a constant that in future version will be calculated automatically).

10. Create a Run Configuration for the XRC project with the updated plug-ins:

    (a) Eclipse >> play (|> button) >> Run Configurations >>Eclipse Application >> new

        i. Name: xrc_trial

        ii. Arguments >> VM arguments: -Xms40m -Xmx512m -XX:+CMSClassUnloadingEnabled -XX:+CMSPermGenSweepingEnabled -XX:+UseConcMarkSweepGC -XX:MaxPermSize=128m

11. Run:

    (a) Eclipse >> play (|> button)

    (b) Add svn repository and check out, for example, bank application ver1 (Appendix A.4).

    (c) Open an aspect file (with .aj suffix) to verify the aspectJ perspective is active.

    (d) Explore the example.

Troubleshooting:

1. If the markers are not displayed, in the new Eclipse that is opened on Run, open AspectJ perspective.

2. If the decorated view does not display markers, debug the compare, put a breakpoint before the code that was moved (in TextMergeViewer class) and verify this point is reached on diff of two versions from history, when openning the Decorated Compare View.

3. In some old versions of Eclipse, modification of the Plugins tab in the Run Configuration might be required.

4. In old Eclipse versions, it might be required to install plugins in the new Eclipse instance that was opened via the Run configuration (e.g., AJDT, SVN plug-ins).

5. In migration to Eclipse 4.2, we have noticed that if there is a project checked out from the svn when running the XRC (via the new Eclipse instance), there might be a problem. Delete and check out projects after re-openning. It wasn't a problem on Eclipse 3.7, and should be investigated if it appears again.

6. On XRC, open MANIFEST.MF, check for errors and solve problems. See the errors in the xml, choose between following solutions:

   (a) Download new jars to fit the requirement, or:

   (b) Go to the Dependencies tab, click the item, click properties, and select the on available. This might require to change also code.

   (c) Eclipse > help > check for updates.

7. Compare the plug-in versions that worked with XRC to the new ones

B.3 Create XRC Installation Kit

When a new version of the Eclipse IDE is released, the XRC plug-in should be adapted. The XRC may evolve and introduce new features, as well. In these cases a new XRC Installation Kit is required. Following are instructions for its creation.

1. Make the *XRC-Developer Installalation*:

   (a) Check-in the XRC plug-in source code into the SVN repository, and copy the SVN repository folder (repos) to the Installation Kit folder denoted from now on %InstallationKit

2. Make the *Standalone installation:*

   (a) Update Eclipse with the XRC required plug-ins only, and copy it to the folder %InstallationKit/Eclipse

3. Make the *Embedded Installation:*

   (a) Copy the XRC folder from the previous Installation Kit except the plugins folder

   (b) Export and copy the plug-ins jars:

      i. Open the MANIFEST.MF of the xrc plug-in in Eclipse

      ii. Select Overview tab >> exporting >> Export Wizard

      iii. Select the three plug-ins

      iv. Options >> Export source: generate source bundles

      v. Destination >> directory: %InstallationKit/XRC/plugins

   (c) Update in the %InstallationKit/XRC according to the binary jars:

      i. configuration/org.eclipse.equinox.simpleconfigurator/bundlesXRCEntries.info

      ii. artifactsXRCEntries.xml

תוכן העניינים

תקציר

פרוייקט תוכנה בסדר גודל בינוני או גדול דורש לרוב מערכת לניהול גרסאות (RCS). אולם עבור תכנות מונחה היבטים (AOP), תפקודן של מערכות אילו ותמיכתן בתהליך הפיתוח נפגעים משמעותית. מערכות ניהול גרסאות אינן מתמודדות עם "אי־מודעות" (obliviousness) ו"כימות" (quantification) המאפיינים תכנות מונחה היבטים. כפי שמחלקות אינן מודעות להיבטים, כך גם מערכות ניהול הגרסאות. אין בקרה, עקיבה וניהול עבור ההשפעה הצולבת (crosscutting effect) של ההיבטים על מחלקות. בפרט לא ניתן לדעת איזו גרסה של מחלקה הושפעה על ידי אילו גרסאות של אילו היבטים ולהפך. יתר על כן, היעדר המידע על ההשפעות הצולבות, פוגע באפשרות לזהות את ההבדלים בתפקודן של גרסאות שונות של מחלקה, על ידי השוואתן זו לזו. בעיות אילו מועצמות בעקבות תכונת הכימות, המאפשרת להצהרה או עצה יחידה בהיבט להשפיע על מחלקות רבות. בחינת ההבדלים בתפקוד מערכת לניהול גרסאות של תוכנות מונחות היבטים, למול תפקודה עבור מערכות וותיקות, ללא היבטים, חושפת פגיעות נוספות בשירותי המערכת. אי לכך, נפגעים תהליכי הפיתוח התלויים בשירותי המערכת.

עבודה זו לומדת את הבעיה בהקשר של AspectJ (שפת תכנות היבטים פופולרית ביותר), עם Subversion (מערכת ניהול גרסאות נפוצה) ולאחר זיהוי וניתוח הכשלים, מציעה דרך להתאים את המערכת לניהול גרסאות של תוכנות מונחות היבטים. הגישה המוצעת לפתרון מכונה ניהול גרסאות צולבות (XRC). גישה זו מתבססת על העשרה, שמירה, השוואה והצגה של ההשפעות הצולבות. העבודה מציעה ארכיטקטורה גמישה הנסמכת על מודל MVC ליישום הגישה. תרומה נוספת של עבודה זאת היא מימוש הארכיטקטורה באמצעות תוסף בשם XRC, לסביבת הפיתוח המשולבת Eclipse, אשר מרחיב את JDT, AJDT ותוספי SVN עבור Eclipse, על מנת לספק תמיכה בניהול גרסאות של קבצים בעלי השפעה צולבת.

מערכת לניהול גרסאות צולבות

חיבור על מחקר

לשם מילוי חלקי של הדרישות לקבלת התואר

מגיסטר למדעים במדעי־המחשב

שגיא יפרח

המחקר נעשה בהנחיית פרופ' דוד לורנץ

במחלקה למתמטיקה ומדעי־המחשב

האוניברסיטה הפתוחה

הוגש לסנט האו"פ

אלול תשע"ב, רעננה, אוגוסט 2012