Crosscutting Revision Control System

Sagi Ifrah David H. Lorenz
Open University of Israel
Dept. of Mathematics and Computer Science
1 University Rd., P.O.Box 808, Raanana 43107, Israel.
sagiy@cslab.openu.ac.il, lorenz@openu.ac.il

***Abstract*—Large and medium scale software projects often require a source code *revision control (RC)* system. Unfortunately, RC systems do not perform well with obliviousness and quantification found in aspect-oriented code. When classes are oblivious to aspects, so is the RC system, and the crosscutting effect of aspects is not tracked. In this work, we study this problem in the context of using AspectJ (a standard AOP language) with Subversion (a standard RC system). We describe scenarios where the crosscutting effect of aspects combined with the concurrent changes that RC supports can lead to inconsistent states of the code. The work contributes a mechanism that checks-in with the source code versions of crosscutting metadata for tracking the effect of aspects. Another contribution of this work is the implementation of a supporting Eclipse plug-in (named XRC) that extends the JDT, AJDT, and SVN plug-ins for Eclipse to provide *crosscutting revision control (XRC)* for aspect-oriented programming.**

***Keywords*-revision control; version control; aspects**

I. Introduction

From the beginning of *Aspect-Oriented Software Development (AOSD)* [12], [15], the question "[h]ow do I know what aspect affects my code?" [19] has been asked, and partially resolved by enhancements made to the *Integrated Development Environment (IDE)* for supporting *Aspect-Oriented Programming (AOP)* [21].

The *AspectJ Development Toolkit (AJDT)* [7], an Eclipse plug-in for AspectJ [20], for example, 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 soften exceptions. When rolling the mouse cursor over the markers, a *hint message* with addition information is displayed. However, there is no support in AJDT for tracking 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 several classic operations provided by a *Revision Control (RC)* system. Use of RC systems is common in organizations that develop code, and is often integrated with the IDE. Indeed, the Eclipse IDE provides a GUI for common RC 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, soften exceptions) nor JDT markers (breakpoints, tasks, warnings, etc.). Needless to say, there is no support for comparing markers.

A. AOP and Revision Control Systems

Many RC systems follow the tradition of Source Code Control System (SCCS) [25] in managing revisions of programs as (text) documents organized in files (e.g., RCS [26]). They store and display (textual) differences between successive revisions. They also offer facilities for merging parallel revisions by reconciling the differences. However, AOP by nature defies these principles. First, concerns crosscut the file structure. Second, obliviousness [16] leaves certain crosscutting effects undetected in the (textual) display of files and changes. Third, quantification [17] may escape 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 RC systems, inspecting $\Delta(v_1, v_3)$ typically amounts to displaying *diff*(C, C') and *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 *diff*(C, C'). Similarly, thanks to quantification, A' may have an effect on C' that is not visible in *diff*(A, A'). Let *eff*(A, C) denote the "invisible" crosscutting effect that A has on C. Since neither *eff*(A, C) nor *eff*(A', C') are observable in code, the difference between *eff*(A, C) and *eff*(A', C') is, too, not observable. In addition, *eff*(A', C) and *eff*(A, C') may be of interest but are also "invisible."

B. Contribution

This work introduces XRC, which stands for *Crosscutting Revision Control*. XRC is an approach (backed up by a prototyped tool) for better supporting revision control of AOP programs. Intuitively, the core idea of XRC is to persist to the RC system the otherwise transient *eff*(A, C), as *crosscutting metadata (XMD)* associated with C. This, combined with new RC and IDE capabilities for displaying and comparing the XMD, provides crosscutting revision control for AOP programs that is currently lacking.

The general problem of applying RC to XMD may be broken down into four sub-problems: metadata 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 plug-in provides:

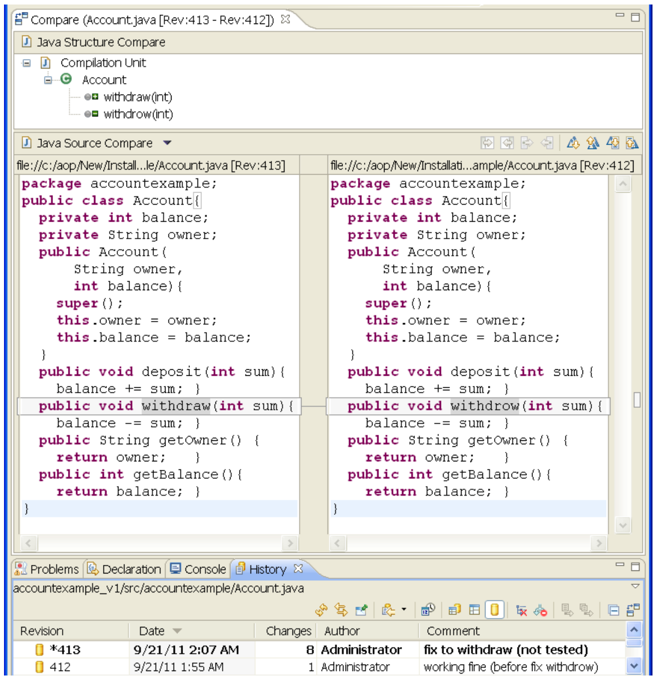
- *Persistent representation of metadata.* The XRC plug-in maintains the XMD and checks it in with the source code files.
- *Diff engine for metadata comparison.* Since the crosscutting effect is often not visible in the code, the plug-in introduces a semantic *diff* that is XMD-aware.
- *Visual means for versioned metadata display.* Markers and hint messages are currently ignored by the RC system and by the IDE compare tool. The XMD plug-in introduces new visual markers and new means to visually display a comparison and present the XMD differences with the compared files.
- *RC–IDE Integration.* A traditional RC system treats the file as the unit of revision. However, XMD by essence crosscuts the file structure. For example, the XMD may change, while the code in the file remains unchanged. This requires adapting the RC–IDE processes for save, build, check-in, display version, compare versions, etc.

The inherent difficulty in bridging RC 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 RC systems are associated with the storing and retrieving of unwoven files and are ignorant of any weaving information.

We provide a solution for AOP-related XMD. However, the solution can be generalized to handle 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 associated with the source code.

II. Problem Illustrated

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").



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; }
}
```

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 1: diff between [Rev:413] and [Rev:412]. No markers are shown despite the existence of an aspect that advises "withdrow" in [Rev:412].

A. Being Unaware of an Aspect (Scenario II-A)

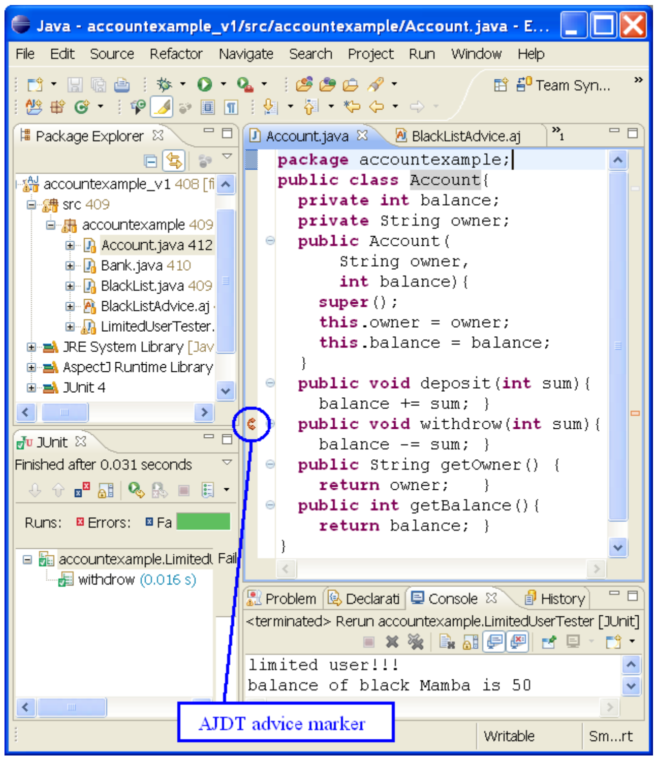
Suppose that another developer notices the typo and in the next revision, $Account'$, corrects the spelling everywhere, except for an occurrence in the code of an aspect (named $BlackListAdvice$), a reasonable oversight when using the *rename* refactoring tool in Eclipse. This oversight will often go unnoticed. On comparing the class source code to its previous versions, *diff*$(Account, Account')$, the only visual difference is the spelling correction. Figure 1 depicts the Compare view, comparing $Account$ with $Account'$. There is no indication 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 or the like, a test might now fail and yield an error (otherwise this bug would be very hard to discover). In Figure 2a, $Account$ 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.

In Figure 2b, $Account$ is shown after correcting the typo:

1) There is no indication of the aspect's effect. Seemingly, everything is fine. Note that also the Project view shows no warnings or errors, since it is perfectly legal for an aspect not to advise any method.
2) JUnit fails, indicated by a red bar. A *withdraw* that should have been rejected was committed successfully.



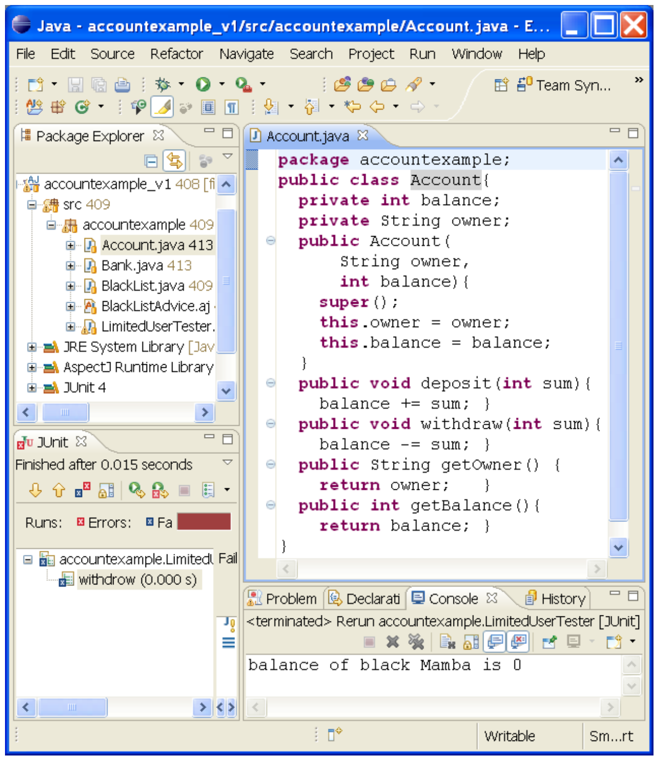
```
package accountexample;
public class Account{
  private int balance;
  private String owner;
  public Account(
      String owner,
      int balance){
    super();
    this.owner = owner;
    this.balance = balance;
  }
  public void deposit(int sum){
    balance += sum; }
  public void withdrow(int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

Finished after 0.031 seconds

accountexample.LimitedUserTester Fail
withdrow (0.016 s)

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

(a) With the typo "withdrow" in the method name.



```
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; }
}
```

Finished after 0.015 seconds

accountexample.LimitedUserTester Fail
withdraw (0.000 s)

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

(b) After correcting the typo in the method name.

Figure 2: Account code displayed in Eclipse with the AJDT plug-in.

```
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 1: LimitedUserTester class after the fix.

However, inspecting the test code (Listing 1) provides no indication as to why the test has failed, or, for that matter, as to why the test passed in the first place. 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 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.

B. Fixing Only the Aspect (Scenario II-B)

Alternatively, suppose that the typo was fixed in the *BlackListAdvice* aspect (but not in the advised classes), and that the aspect was checked in. 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.

A similar scenario could occur in concurrent development. Assume two developers, Sagi and Dave, participate in a project. Sagi works on a Logic domain that is not a crosscutting concern, while Dave is responsible for the Security domain, which is a crosscutting concern. 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. Sagi might then find out that his code is broken. It could be difficult for him to discover the reason for the problem, since he does not have a way to identify aspects that advised the previous version but stopped advising the current version of his class.[1]

C. Consequences

RC systems play an important role in supporting a healthy software development process. Loss of RC support has

[1]Incidentally, this scenario was the motivation for this research.

negative consequences on the entire software development processes.

1) Loss of Code Review: Code inspection [13] or code review [11], [18] refers to the ability to examine the source code, in order to inspect correctness or to investigate failure. However, Scenario II-A illustrates that viewing the previous version of $Account$ (a version that worked) does not reveal the complete behavior of $withdraw$.

2) Loss of Change Control: Before code is checked in, it is considered good practice to have the code reviewed by another programmer ("review before merge" policy [2], [24]). This practice keeps the RC system "safe" with relatively stable code, since all checked-in changes have been reviewed. However, altering the behavior of a class through an aspect occurs without change control on the affected class. When an aspect is checked in, the RC system does not ask for approval, nor does the RC system require to check in the affected classes. These classes thus escape code review and bypass change control.

In Scenario II-A, $Account$ was developed prior to $BlackListAdvice$. When $BlackListAdvice$ was added and checked in, loss of change control occurred from the $Account$ class point of view. In Scenario II-B, after correcting the $BlackListAdvice$ aspect, there was no need to check in the $Account$ class, and therefore loss of change control occurred again.

3) Loss of Code Evolution Tracking: Loss of change control might lead also to loss in evolution tracking. A change to an aspect that advises a class does not require to check in that class. If the class is not checked in, it has no version for that change, which makes evolution especially difficult to track. In Scenario II-B, after the aspect $BlackListAdvice$ ceased to advise $Account$, it was checked in but $Account$ was not. The changes to the woven behavior of $Account$ is thus an evolution that went unnoticed. Note that Figure 1 does not expose the change in the behavior of $Account$.

Even when the aspects that advise the versioned classes are known, the order of the changes might be obscure. For AOP code, the evolution of a class is not necessarily sequential. Without aspects, every change to the class is checked in through consecutive versions. It is clear that the change did not affect previous versions. With aspects, the versions of the classes and aspects are independent and they evolve in parallel. Even with timestamps and version numbers that expose the check-in order, still one cannot determine the actual history.

For example, assume that an aspect A advises a class C and both $\{A, C\}$ were checked in, then evolved to $\{A', C'\}$ and checked in again, and then evolved to $\{C'', A''\}$. Viewing the history, if the check-in order was $A \rightarrow C' \rightarrow A'$, one cannot tell if C' worked with A or with A'. If the check-in order was $A' \rightarrow C' \rightarrow A''$, one cannot tell if C' worked with A' or A''.

Even when "everything" about the aspects that advise the versioned classes is known, the changes to the class might be spread over many versioned aspects and classes. Tracking the changes without tool support is a difficult and tedious task that requires navigating through multiple files back and forth. The Eclipse Compare view, for example, will no longer display all the changes in a single view.

4) Loss of Reversion to a Stable Revision: RC systems help to locate and revert to a previous stable version. However, due to loss of code evolution tracking, it becomes more difficult to find a stable point in the history to revert to.

5) Loss of Team Development Support: RC systems enable several developers to check out and modify the same class, and then merge and resolve conflicts on check-in. However, when different developers work on different aspects that advise the same class, the RC system is unaware of the potential conflicts. The aspects will be checked in with no errors or warnings from the RC system, since they are in different files. Conflicting changes will be discovered during build (in case of a failure), or during deep code review, or at runtime.

III. Solution and Implementation

The main requirement for crosscutting revision control (XRC) is adding revision control support for the crosscutting metadata (XMD). The root of the problem is that the RC system is unaware of the XMD. 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 represent, 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 comparing the XMD of two files (or two versions of the same file).
3) *Visual enhancements* that enable displaying markers (e.g., markers that the Java editor displays for advice) when comparing Java source files and when viewing a previous version.
4) *RC–IDE integration* that, together with the other components, provides a mechanism for saving, comparing, and displaying XMD.

We implemented an XRC plug-in for Eclipse that demonstrates the feasibility of the approach. The XRC plug-in extends JDT and AJDT, and uses the SVN plug-ins for Eclipse to provide the XRC solution.

A. Crosscutting Metadata (XMD)

In Section I 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:

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

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

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

where C ranges over all classes in the project. It should be noted that $\textit{XMD}(C)$ and $\textit{XMD}(A)$ include concrete information, such as the line numbers of specific pieces of advice and effected program elements. For conciseness and for symmetry considerations, we focus in this paper mainly on $\textit{XMD}(C)$. Hereafter, we call the metadata that we want to preserve *XMD information* or simply *XMD*.

1) 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:

$$Map < int,\ List < \textit{IRelationship} >>$$

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.

2) XRC metadata: The XMD that XRC writes to and reads from the RC system 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 RC system repository, and the details of the parent in a declare parents advice.

B. 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 markers is obviously not good enough, because the same icon might 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 flag all the advice as being different from the previous version. A developer will have to inspect each one of them just to conclude that the crosscutting effect stayed the same.

Comparing the pieces of advice sorted according to their line number may still give 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` [4], [5], an enhanced line differencing tool that is capable of tracking text that was moved to another line, thereby distinguishing line additions and deletions from line modifications.

Each line has a list of *IRelationship*, each comprising multiple targets that the piece of advice affects. The XRC *diff* engine compares $\textit{XMD}(C)$ with $\textit{XMD}(C')$ by first "flattening" the data, then computing the differences, and finally "inflating" the result back into a format that Eclipse understands:

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

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

- $flatten$: $\textit{XMD} \rightarrow Set < \textit{IRelationship} >$, denoted $\lfloor . \rfloor$, takes an XMD object, and returns a $Set < \textit{IRelationship} >$ 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.
- $inflate$: $(\textit{XMD}, Set < \textit{IRelationship} >) \rightarrow \textit{XMD}$, denoted $\lceil . \rceil$, takes an XMD object, and a $Set < \textit{IRelationship} >$. 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.

C. Visual Enhancements

1) AJDT Markers and Rulers: AJDT displays the markers in the editor, via a ruler to the left of the source, as shown in Figure 2a. Table I 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 the classes it advises). Since one implies the other, the table reflects a symmetry in both the marker types and their icons. AJDT also provides navigation from the aspect to the class and vice versa via a pop-up menu, by clicking on a marker. AJDT markers are created and updated on build.

2) 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. For this, we extended the Eclipse JDT infrastructure to provide support for rulers in a Compare view. A side benefit of this effort 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 (Table II) to emphasize the differences. This marker changes the background of a marker that has changed, and can thus be visually superimposed over existing markers.

Table I: AJDT Markers.

Marker type on Class	Marker type on Aspect	Subtype	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	implementation	↘	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

Table II: XRC Markers.

Marker type on Class	Subtype	Icon on Class	
Changed advice highlight	N/A	🟨	changedadvice
Changed advice	add	⊕	add2
	remove	⊖	remove2
	modified	◌	modified2f

On a second ruler, named *diff-ruler*, inspired by the obsolete AJDT Crosscutting Comparison view [6] (reviewed in Section V-B1), we mark the nature of the change. Two addition markers, *add* and *remove* (Table II), are used to indicate advice addition and removal, respectively. A fourth marker, named *modified* (Table II), 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 advises 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 *eff*(A', C). Let A'' be a modified version of A' that advises 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. *diff*(C, C') will display the *modified* marker, since A' is unattainable, and it can only hint on A as its predecessor. The *modified* marker helps distinguishing between the situation where the version of the aspect is known and trusted, and the situation where it is unknown and only the previous version of the aspect that has been checked in is known.

D. RC–IDE Integration

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

1) 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 RC system 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.

2) 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

enhances it with the XRC XMD (adding parent details, target revisions, etc.).

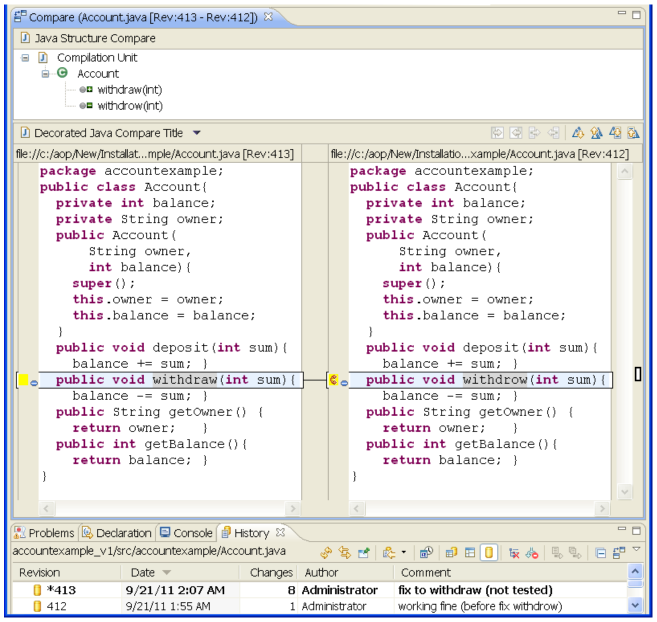
- *Write the XMD.* XRC writes the enhanced XMD as a property of the working copy of the file. The file is then marked as dirty (SVN marks the file as dirty automatically when its properties are modified).

3) Check-in: SVN supports version properties per file. A file might 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. XRC exploits this feature to attach metadata to the file, letting this metadata be versioned with the file.

4) Display a Previous Version: When reading a file from the RC system, XRC reads the XMD from the properties of the file, and displays the markers accordingly.

5) 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, on each side of the Compare view.



file://c:/aop/New/Installat...mple/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/Installatio...xample/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; }
}
```

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: diff between [Rev:413] and [Rev:412] with XRC. In comparison with Figure 1, markers are displayed and compared.

IV. Evaluation

The magnitude of the problem in practical settings can be learned from related studies. Ferrari et al. [14], 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" [14].

To regain crosscutting awareness, XRC enables integrated RC support for crosscutting metadata. 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 [10] 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.

A. Comparing Two Advised Versions

Revisiting the scenarios presented in Section II but this time with the Eclipse plug-in for XRC, comparing the *Account* to its previous version immediately reveals that something is different with the advice (Figure 3). Rolling the mouse cursor over the marker shows a hint message stating that the *BlackListAdvice* aspect ceased to advise.

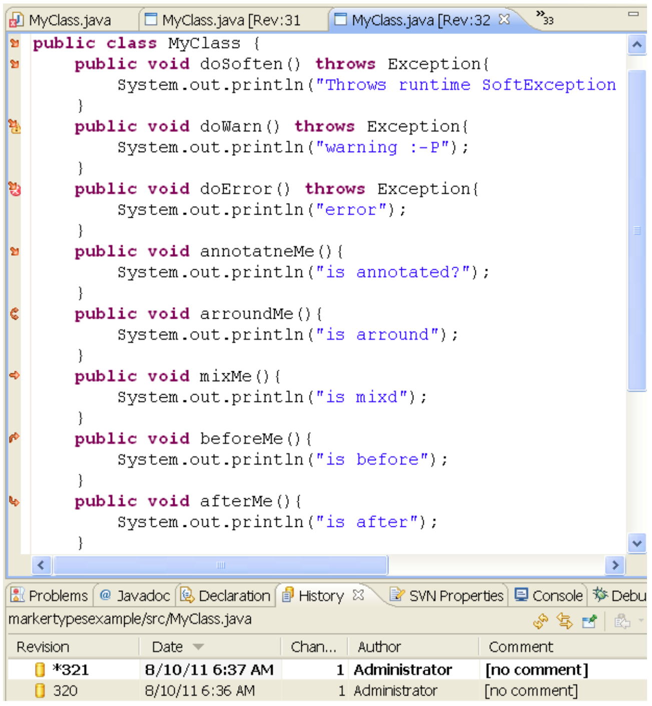
B. Viewing an Advised Revision from SVN History

With the Eclipse plug-in for XRC, markers are displayed also for old versions of *Account*. In Figure 4, the selected title tab confirms 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 have been displayed in the left toolbar.

C. Regaining RC Support for SW Development Processes

1) 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.

2) Regaining Change Control: Repeating Scenario II-B 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



```
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 4: Viewing an old version [Rev:320] of a class retrieves the XMD from SVN and displays the AJDT markers in the left toolbar.

check-in without fixing the bug will distinguish the previous version that worked from the new version that does not work.

3) Regaining Code Evolution Tracking: With XRC, the developer has to check-in the $Account$ class whenever the aspect is modified. This check-in enables tracking code evolution. The *diff* in Figure 3 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.

4) Regaining Reversion to Stable Revision: Checking-in a revision of $Account$ also helps to later identify a stable revision to be checked-out.

5) Regaining Team Development Support: When change control is regained, the RC system will detect and prevent overlapping changes to the XMD of a class, without first resolving all conflicts.

D. Threats to Validity

While the findings in Ferrari et al.'s study [14] indicate a need for a tool like XRC, a user study would be needed to assess the effectiveness of the XRC plug-in as a productivity tool in practice. Analyzing what percentage of the faults reported in the study [14] could supposedly be avoided had XRC been used is also a topic for future work.

The XRC tool was built and tested to work with a specific RC system and IDE, namely: SVN version 1.6.3, Eclipse version 3.7 (Indigo), and AJDT version 2.1.3. However, the approach should be applicable in general, and plug-ins can be implemented for other RC systems or for persisting other sorts of metadata.

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 based 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 is also the default setting in Eclipse). XRC does not require saving the woven classes. For each file, it only persists the source and its relevant XMD, so it is efficient in terms of space requirements. Getting the metadata requires a single read from the RC system. 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.

V. Related Work

A. AspectJ Development Toolkit

The AJDT Eclipse plug-in adds to the IDE new capabilities for visualizing crosscutting effects. It computes the crosscutting metadata and introduces markers that hint about $\mathit{eff}(A, C)$ when editing C or A. Eclipse uses a vertical ruler in the editor in order to display these markers. Additional hint messages are displayed when rolling the mouse cursor over the markers. However, these markers are not saved with the file and thus not checked-in with revisions of C or 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.

AJDT uses advice markers in order to display crosscutting metadata in the current version. We reuse the AJDT markers in order to display the same kind of metadata also for viewing a previous version and for displaying the *diff* of two versions. We added in the Compare view a ruler and designated markers for the purpose of displaying the essence of the change.

B. Obsolete Features of AJDT

Interestingly, early versions of AJDT did include facilities for crosscutting comparison and changes, which were eventually removed.

1) Crosscutting Comparison (Obsolete Feature): A Crosscutting Comparison capability was part of the AJDT 1.2.1 and 1.3 releases. Crosscutting Comparison enabled a developer to take a snapshot of the crosscutting relationships in the project, save it to a file, and then compare the snapshot with the relationships present in a later version of the project. The results of the comparison were displayed in a special designated view [6].

In AJDT 1.6.1, the crosscutting model enhancements and the 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 relied on the model that was removed, and thus abandoned.

In comparison to XRC, the AJDT Crosscutting Comparison obsolete feature had the following disadvantages. First, the Crosscutting Comparison view lacked consistency with the Eclipse Compare view, and did not integrate the display of differences with the Compare view. Second, it managed all the relationships of the project in a single file. Specifically, it did not allow one to save or examine the differences per class or per aspect. Third, it required to save snapshots manually. Fourth, it did not support the enhanced crosscutting model of Eclipse. Eclipse version 3.4 and higher requires AJDT version 1.6.1 or higher, which no longer supports this functionality.

2) Crosscutting Changes (Obsolete Feature): AJDT 1.5 introduced another relevant feature, named *Crosscutting Changes*. With this feature, advice markers were highlighted when the crosscutting effect has changed, such as 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 could be chosen using a drop-down menu on the Crosscutting Comparison view. The possible choices were: to use the last build (of any type), the last full build, or a crosscutting map file in the project. This feature has been abandoned as well, in favor of the crosscutting model enhancements in AJDT 1.6.1.

The Crosscutting Changes feature relied on the Crosscutting Comparison, which is obsolete. In comparison to XRC, it did not support the resolution of versioned classes and aspects from a RC system.

C. Specialized Differencing Tools

Many semantic differencing tools enhance the simple *diff* textual comparison tool for the purpose of tracking software evolution. Some analyze the RC repository to better detect high level structural changes (e.g., UMLDiff [27]), infer systematic changes (e.g., LSdiff [22], [23]), or even recommend adaptive changes for keeping up with the software evolution (e.g., SemDiff [8], [9]). 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.

D. Crosscutting Configuration Management

TOFRA [1] is a tool that addresses the problem of configuration management (CM) in the context of *Crosscutting Frameworks (CFs)* [3]. CFs are aspect-oriented frameworks that handles a single crosscutting concern. One or more CFs may be weaved with the application, and CFs might be reused across different applications. TOFRA's support for version control in CF-based development focuses on managing the dependencies among versioned CFs and versioned applications. In comparison, XRC provides actual RC support for developing AOP applications, integrated with the IDE and the RC system.

VI. 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 software product development, medium or large, requires revision control, the lack of appropriate support is an obstacle that hinders the use of AOP.

This work introduces *crosscutting revision control*—a novel approach and a supporting Eclipse XRC plug-in—that improves the revision control of AOP code. The XRC plug-in for Eclipse 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.

Acknowledgment

We thank Gennady Agranov, Shai Koenig, Oren Mishali, 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.

References

[1] M. M. Arimoto, M. I. Cagnin, and V. V. de 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 Press.

[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 Press.

[3] V. V. de Camargo and P. C. Masiero. A pattern to design crosscutting frameworks. In *Proceedings of the 23rd Annual ACM Symposium on Applied Computing (SAC'08)*, pages 759–764, Fortaleza, Ceara, Brazil, 2008. ACM.

[4] G. Canfora, L. Cerulo, and M. Di Penta. Ldiff: an enhanced line differencing tool. In *Proceedings of the 31st International Conference on Software Engineering (ICSE'09)*, pages 595–598, Vancouver, Canada, May 2009. IEEE Computer Society.

[5] G. Canfora, L. Cerulo, and M. Di Penta. Tracking your changes: A language-independent approach. *IEEE Software*, 26(1):50–57, 2009.

[6] M. Chapman. AOP@Work: New AJDT releases ease AOP development. http://www.ibm.com/developerworks/java/library/j-aopwork9/, Aug. 2005.

[7] A. Clement, A. Colyer, and M. Kersten. Aspect-oriented programming with AJDT. In J. Hannemann, R. Chitchyan, and A. Rashid, editors, *Proceedings of the Workshop on Analysis of Aspect-Oriented Software*, Darmstadt, Germany, July 2003. ECOOP'03.

[8] B. Dagenais and M. P. Robillard. Recommending adaptive changes for framework evolution. In *Proceedings of the 30th International Conference on Software Engineering (ICSE'08)*, pages 481–490, Leipzig, Germany, May 2008. ACM Press.

[9] 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.

[10] A. van Deursen, M. Marin, and L. Moonen. AJHotDraw: A showcase for refactoring to aspects. In *Proceedings of the AOSD'05 Workshop on Linking Aspect Technology and Evolution (LATE'05)*, Chicago, IL, USA, Mar. 2005. AOSD'05, ACM Press.

[11] S. G. Eick, C. R. Loader, M. D. Long, L. G. Votta, and S. A. Vander Wiel. Estimating software fault content before coding. In *Proceedings of the 14th International Conference on Software Engineering (ICSE'92)*, pages 59–65, Melbourne, Australia, June 1992. ACM Press.

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

[13] M. E. Fagan. Design and code inspections to reduce errors in program development. *IBM Systems Journal*, 15(3):182–211, 1976.

[14] 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 *Proceedings of the 32nd International Conference on Software Engineering (ICSE'10)*, pages 65–74, Cape Town, South Africa, May 2010. ACM Press.

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

[16] 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.

[17] R. E. Filman and D. P. Friedman. Aspect-oriented programming is quantification and obliviousness. In Filman et al. [15], pages 21–35.

[18] 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.

[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, *Proceedings of the OOPSLA 2002 Workshop on Tools for Aspect-Oriented Software Development*, Seattle, Washington, 2002.

[20] 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.

[21] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *Proceedings of the 11th European Conference on Object-Oriented Programming (ECOOP'97)*, number 1241 in Lecture Notes in Computer Science, pages 220–242, Jyväskylä, Finland, June 9-13 1997. Springer Verlag.

[22] M. Kim and D. Notkin. Discovering and representing systematic code changes. In *Proceedings of the 31st International Conference on Software Engineering (ICSE'09)*, pages 309–319, Vancouver, Canada, May 2009. IEEE Computer Society.

[23] A. Loh and M. Kim. LSdiff: a program differencing tool to identify systematic structural differences. In *Proceedings of the 32nd International Conference on Software Engineering (ICSE'10)*, pages 263–266, Cape Town, South Africa, May 2010. ACM Press.

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

[25] M. J. Rochkind. The source code control system. *IEEE Transactions on Software Engineering*, SE-1(4):364–470, 1975.

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

[27] 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, Nov. 7-11 2005. ACM Press.