SemDiff: Analysis and Recommendation Support for API Evolution

Barthélémy Dagenais and Martin P. Robillard

School of Computer Science
McGill University
Montréal, QC, Canada
E-mail: {bart,martin}@cs.mcgill.ca

Abstract

As a framework evolves, changes in its Application Programming Interface (API) can break client programs that extend the framework. Repairing a client program can be a challenging task because developers need to understand the context surrounding the API change. This paper describes SemDiff, a tool that recommends replacements for framework methods that were accessed by a client program and deleted during the evolution of the framework. SemDiff recommends replacements for non-trivial changes undiscovered by other change-detection techniques and also enables developers to look at the context of the changes that led to the deletion of a framework method.

1. Introduction

When writing software applications, developers often rely on frameworks to reuse common features and speed up development. As a framework evolves, developers need to adapt their applications to the changes made in the framework's Application Programming Interface (API). For example, let us assume that a developer is writing a program that extends the Eclipse JDT framework, version 3.2.[1] When version 3.3 of the framework is released, the developer tries to compile the program with the new version, but the Java compiler returns an error indicating that one of the framework classes used by the program does not exist: the class has been removed in the new version of the framework. To fix this error, the developer needs to find a *replacement* for the missing class.

Searching for a replacement is often a challenging task that does not provide significant added-value (i.e., the developer is trying to *repair* the application and is not adding new features nor improving its quality). To mitigate such situations, we created SemDiff, a tool that recommends replacements for framework elements that were accessed by a client program and deleted as part of the framework's evolution [4]. Our recommendations are produced by analyzing how the framework was adapted to its own changes and by recommending similar adaptations to client programs. Specifically, SemDiff analyzes the evolution of method calls in the framework (e.g., a call to method `m1` was replaced by a call to methods `m2` and `m3`). A developer can then use SemDiff to obtain a recommendation on how to replace a call to a deprecated or deleted method. An empirical study we previously performed on the Eclipse JDT framework and three client programs showed that SemDiff could recommend relevant method replacements with a high precision, and detected changes typically undiscovered by other change-detection techniques.

Change-detection techniques such as UMLDiff have been proposed to help developers who are using an API that is no longer backward compatible [10]. Usually, these approaches track equivalent framework elements (e.g., functions, classes) across multiple versions by computing a fingerprint for each element. Although the comparison of similar elements can detect simple changes such as a refactoring (e.g., renaming a method) with a high precision, this strategy generally misses certain types of complex changes: (1) software elements that are heavily modified between two versions produce fingerprints that are too different to be comparable, (2) software elements that have been split or merged are ignored by techniques that look for one-to-one changes, and (3) software elements that are deleted and replaced by elements imported from an external codebase are not detected. Because SemDiff looks for the adaptation instead of the identity of changed elements, it is not affected by such non-trivial changes.

Although the first prototype of SemDiff could analyze industrial-sized systems such as Eclipse, only the recommender was accessible to the end-user. After a major reengineering effort, we developed a version of SemDiff[2] that exposes the underlying analysis framework to the user and improves the recommender user interface.

[1] www.eclipse.org

[2] SemDiff is available at www.cs.mcgill.cs/∼swevo/semdiff

ICSE'09, May 16–24, 2009, Vancouver, Canada
978-1-4244-3452-7/09/$25.00 © 2009 IEEE

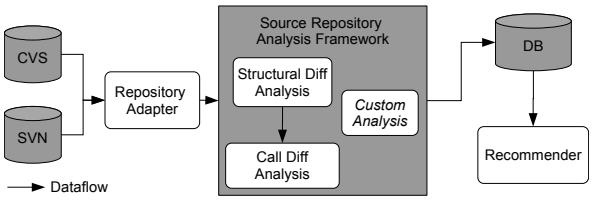

Figure 1. SemDiff Overview

SemDiff's analysis framework mines the *software repository* containing the source code history of a framework to study the evolution of method calls. As it is the case with most approaches that mine software repositories, SemDiff takes as input the coarse-grained data provided by a repository (i.e., the files that were changed) and performs multiple analyses to find fine-grained changes (e.g., method call changes). Notable examples of repository analyses include the detection of refactorings [9], the location of cross-cutting concerns [2], and the automatic classification of fine-grained changes [5, 10]. Future repository analyses could benefit from SemDiff's infrastructure.

In this paper, we present a description of SemDiff and review its two main contributions: a recommender enabling software engineers to adapt their client programs to the evolution of a framework, and a repository analysis framework enabling users to easily analyze the evolution of software products.

2. SemDiff

SemDiff is a client-server application implemented as a set of Eclipse plug-ins. Figure 1 provides an overview of SemDiff's architecture. The server component, which we exposed in the new version of SemDiff, is described in Section 2.2 and consists of the repository adapter that connects to a source version control system such as CVS, the source repository analysis framework, and a database that stores the results of the analyses. The client component, described in Section 2.1, can access the server remotely and consists of the recommender.

For SemDiff, the main unit of change is a transaction, i.e., a set of files that were committed together at some point in the evolution of a software product. Transactions are retrieved by the source repository analysis framework and used by the recommender. In SemDiff, the Transactions View (see Figure 3) provides the user with a convenient way to browse and review the following details of transactions: the transaction date, the username of the developer who committed the transaction, the comment written by the developer, the files that were changed,[3] the Java elements (methods, types, fields) that were changed, and the method calls that were added or removed. This information can be used by the recommender or by the user to analyze the evolution of the software application.

[3] In the remainder of this paper, we consider a "change" to be any addition, deletion or modification of an element (e.g., file, Java method, etc.)

```
public class Client {

    public ContentProvider createContentProvider() {
        return new ListContentProvider();
    }
```

Figure 2. Broken method call due to framework evolution

2.1. Recommending Adaptive Changes

When the developer mentioned in the introduction tries to compile the program with the most recent version of the Eclipse JDT framework, the Java compiler returns an error indicating that the application class `Client` is calling the missing framework constructor `ListContentProvider()` (Figure 2): this constructor and its class were removed in the most recent version of the framework.

To fix this error, the developer can use SemDiff to identify a replacement for the constructor: the developer right-clicks on the constructor call in the editor and selects the menu option "SemDiff / Get Recommendations".

Once SemDiff has computed a list of potential replacements, it opens the Call Recommendations View, which displays a ranked list of recommended calls. For example, in Figure 4, the first recommendation of SemDiff is to replace a call to the constructor `ListContentProvider()` (first line prefixed by REQ) by a call to the constructor `ArrayContentProvider()` (second line prefixed by REC). The fully qualified name of the recommended method indicates that the recommendation comes from an external library, a change typically undetected by other approaches. The developer can also right-click on the recommendation to navigate to the Transactions View and see the transaction where the change occurred (Figure 3). This view shows the *context* of the change that led to the removal or deprecation of the framework method. For example, the commit comment, "Use `ArrayContentProvider` instead of `ListContentProvider`", confirms that SemDiff's recommendation is valid. The developer can then use the recommendation and make the appropriate changes to the client program. In a previous study, we found that the top three recommendations made by SemDiff could fix 89% of the evolution errors in three client programs [4].

Recommender Implementation. To compute the set of method replacements, SemDiff searches the framework's history to find the transactions where calls to the broken constructor `ListContentProvider()` were removed. SemDiff then suggests calls that were added in these transactions. The complete description of the algorithm, and the heuristics that SemDiff uses to handle non-trivial changes are available in a previous publication [4].

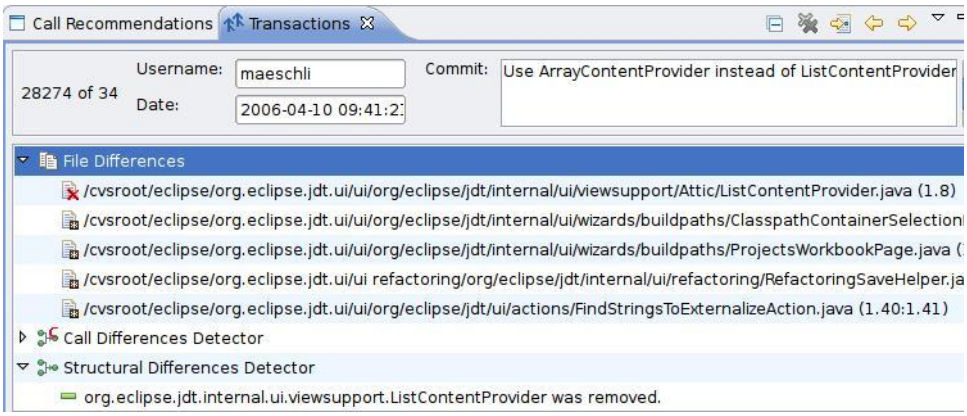

Figure 3. Transactions View showing the details of a single transaction for Eclipse JDT

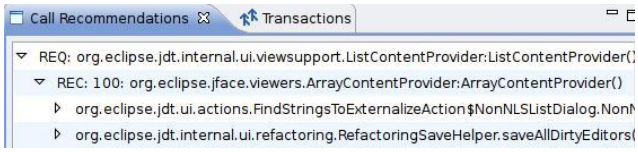

Figure 4. The Recommendations View

The Call Recommendations View shows how SemDiff computed the recommendation. For example, Figure 4 shows that a call to `ListContentProvider()` was replaced by a call to `ArrayContentProvider()` in several framework methods (e.g., `saveAllDirtyEditors()`). If SemDiff recommends multiple method calls, the developer can double-click on each recommendation to open the compare editor, which will show two versions of a file in the framework where a call was removed and replaced by the recommendation.

2.2. Source Repository Analysis

To get recommendations on how to replace a call to `ListContentProvider()`, SemDiff must first retrieve the transactions from the Eclipse JDT source repository.

Repository. SemDiff can retrieve transactions from CVS and Subversion source repositories. The user has to enter the following connection information in an input dialog: connection string, username, password, module paths.

Repository Analysis. Once SemDiff has downloaded the files changed in each transaction, a list dialog asks the user to choose which analyses to perform: these analyses find fine-grained changes from the coarse-grained data provided by the repository. Two analyses are provided by default, StructDiff and CallDiff. StructDiff indicates the fields, methods and types that were changed in each transaction. CallDiff, which depends on StructDiff, indicates the method calls that were added or removed in each class. Users can also create their own analyses through the use of an Eclipse extension point and these analyses are automatically added to the list dialog.

Storage. The results of the analyses are stored in a database. The user can choose to either rely on an internal (HSQLDB[4]) or an external (PostgreSQL[5]) database: the former is automatically installed and configured by SemDiff, while the latter usually provides better performance for large repositories containing thousands of transactions.

User analysis. SemDiff offers many ways to browse and analyze transactions. The Transactions View is the preferred tool to manually review the details of an individual transaction and look at the context of a change. For example, Figure 3 shows which files, Java elements, and calls were impacted by the removal of the `ListContentProvider` class. For each item displayed in the Transactions View, the user can either navigate to the version of a file before and after the change or open the *compare editor* that shows the two versions of the file side-by-side and that highlights the main differences. When the user selects a fine-grained element (e.g., a Java method), the editor automatically scrolls down to focus on the element.

Systematic analysis of transactions can be performed through three main techniques. First, log files can be generated containing a textual representation of the changes in each transaction. These log files were used in two previous studies on recommending elements that should be investigated during a maintenance task [7, 8]. Second, users can access the object model of transactions maintained by SemDiff by writing an Eclipse plug-in (like SemDiff's recommender) or by executing a Groovy script, a Java-like dynamic language.[6] Finally, users can connect to the database containing the transaction data and perform SQL queries.

Remote access. Users may access transactions data stored on another machine by entering the connection string of a remote database in SemDiff. This enables users to delegate the execution of transaction analysis to more powerful computers: retrieving and analyzing transactions from large repositories is CPU and memory intensive. For example, with two GB of heap space, it took 13 hours to process the

[4] hsqldb.org
[5] www.postgresql.org
[6] groovy.codehaus.org

Eclipse JDT repository (∼5 seconds per transaction) on a Pentium D 3.2 Ghz.

Implementation. To compute transactions, SemDiff automatically retrieves from the repository a log describing the files that were changed. From this log, SemDiff downloads all the versions of each file that was ever created in the project and associates each version to a transaction. Although Subversion repositories explicitly keep track of the files that were changed together, CVS repositories do not group the files together and the log must be processed using a common technique to recover the transactions [11] (SemDiff hides this processing phase to the user).

SemDiff computes structural differences by comparing the abstract syntax tree of the source files between each version. To compute call differences, SemDiff uses Partial Program Analysis (PPA) to recover the method signature of the calls [3].

Because SemDiff only stores the files that were changed at each transaction (as opposed to the entire program), it is generally not possible to compile those files because their dependencies are missing. PPA enables us to perform static analysis on these partial programs by inferring types and resolving syntactic ambiguities.

3. Related Work

Fluri et al. created ChangeDistiller, a tool that transforms source code changes into abstract syntax tree edits [5]. These tree edits are then used to classify the source code changes according to a taxonomy of changes. ChangeCommander, an extension to ChangeDistiller, recommends the addition of guard conditions before method calls and can complement the recommendations made by SemDiff [6].

eRose is a tool that recommends files that should be changed together during a maintenance task [11]. eRose also mines CVS repositories to produce its recommendations, but it does not address the problem of adapting client programs to the new version of a framework.

Xing and Stroulia proposed UMLDiff, an approach that provides very precise analysis of the changes that happened between two major revisions of a software application [10]. UMLDiff also provides a classification of changes and, like SemDiff, can detect refactorings. JDiff is a tool that builds and compares the control flow graph of two versions of a Java program to model the difference in their behavior [1]. As opposed to SemDiff though, UMLDiff and JDiff require the complete versions of the software application (hence their ability to precisely determine the full signature of method calls), do not work with software repositories, and cannot detect complex changes such as program elements that are replaced by external elements. A more complete review of the related work is available in our previous publication [4].

4. Conclusion

We presented SemDiff, a toolset integrated within Eclipse that recommends to developers how to adapt their client programs to the evolution of a framework. SemDiff also enables users to easily analyze the evolution of a software application. Ultimately, we hope that tools like SemDiff will encourage early adoption of frameworks that change often and that developers will spend more time on changes that have higher added-value than adaptive changes.

Acknowledgments

The authors are grateful to Ekwa Duala-Ekoko and Tristan Ratchford for theirs valuable comments on this paper. This work was supported by NSERC and FQRNT.

References

[1] T. Apiwattanapong, A. Orso, and M. J. Harrold. JDiff: A differencing technique and tool for object-oriented programs. *Automated Software Engineering*, 14(1):3–36, 2007.

[2] S. Breu and T. Zimmermann. Mining aspects from version history. In *Proc. of the 21st Int'l Conf. on Automated Software Engineering*, pages 221–230, 2006.

[3] B. Dagenais and L. Hendren. Enabling static analysis for partial Java programs. In *Proc. of the 23rd Conf. on Object Oriented Programming Systems and Applications*, pages 313–328, 2008.

[4] B. Dagenais and M. P. Robillard. Recommending adaptive changes for framework evolution. In *Proc. of the 30th Int'l Conf. on Software engineering*, pages 481–490, 2008.

[5] B. Fluri, M. Wuersch, M. Pinzger, and H. Gall. Change distilling: Tree differencing for fine-grained source code change extraction. *IEEE Transactions on Software Engineering*, 33(11):725–743, 2007.

[6] B. Fluri, J. Zuberbüehler, and H. Gall. Recommending method invocation context changes. In *Int'l Workshop on Recommendation Systems for Software Engineering, FSE Workshop*, 2008.

[7] M. P. Robillard and B. Dagenais. Retrieving task-related clusters from change history. In *Proc. of the 2008 15th Working Conf. on Reverse Engineering*, pages 17–26, 2008.

[8] M. P. Robillard and P. Manggala. Reusing program investigation knowledge for code understanding. In *Proc. of the 2008 The 16th Int'l Conf. on Program Comprehension*, pages 202–211, 2008.

[9] P. Weissgerber and S. Diehl. Identifying refactorings from source-code changes. In *Proc. of the 21st Int'l Conf. on Automated Software Engineering*, pages 231–240, 2006.

[10] Z. Xing and E. Stroulia. Understanding the evolution and co-evolution of classes in object-oriented systems. *Int'l Journal of Software Engineering and Knowledge Engineering*, 16(1):23–51, 2006.

[11] T. Zimmermann, A. Zeller, P. Weißgerber, and S. Diehl. Mining version histories to guide software changes. *IEEE Transactions on Software Engineering*, 31(6):429–445, 2005.