ABSTRACT
In the course of a framework’s evolution, changes ranging from a simple refactoring to a complete rearchitecture can break client programs. Finding suitable replacements for framework elements that were accessed by a client program and deleted as part of the framework’s evolution can be a challenging task. We present a recommendation system, SemDiff, that suggests adaptations to client programs by analyzing how a framework adapts to its own changes. In a study of the evolution of the Eclipse JDT framework and three client programs, our approach recommended relevant adaptive changes with a high level of precision, and detected non-trivial changes typically undiscovered by current refactoring detection techniques.

Categories and Subject Descriptors: D.2.7 [Software Engineering]: Distribution, Maintenance, and Enhancement

General Terms: Documentation, Experimentation

1. INTRODUCTION
Application frameworks support large-scale reuse and free developers from low-value programming tasks. By developing clients that integrate with the framework code, developers are able to customize and enhance the framework to suit their specific needs. However, as the framework evolves, changes ranging from a simple refactoring to a complete rearchitecture can break client programs. To lower the cost of adapting client programs to changes in the framework, framework developers rely on a variety of techniques such as automatically capturing and documenting some of their changes [7, 10], providing migration paths [3], or deprecating existing methods and indicating new replacements. Unfortunately, current tools cannot capture changes more complex than refactorings, and manually documenting a framework’s evolution is not always cost-effective, especially for fast-evolving frameworks. Although framework users are encouraged to only use public Application Programming Interfaces (API) as they usually provide a long-term contract of stability, developers often use internal and undocumented parts of frameworks for a variety of good reasons, such as accessing functionality that goes beyond the ones available in the public interfaces [1].

A recent study of API evolution found that more than 80% of API-breaking changes were caused by refactorings, and concluded that techniques aiming at documenting or detecting refactorings were desirable [6]. The authors of the study also mentioned that “Application developers will have to carry only a small fraction [less than 20%] of the remaining changes. These are changes that require human expertise” [6, p.105]. To detect the largest portion of API-breaking changes, i.e., refactorings, several approaches have been proposed [5, 9, 13, 14, 20, 21].

Although refactoring detection techniques partially automate the tedious task of identifying and repairing small changes such as a renamed method, refactorings tend to be minor changes easily identified through a manual inspection. Indeed, refactorings usually involve only one change dimension: name or location. For example, if a method is no longer accessible in the new version of a framework, a developer can often simply perform a lexical search (“grep”) to find similarly named methods (name dimension) or can look in the same module to find potential replacements (location dimension). However, it will generally be harder to repair a client program if the framework went through major modifications that led to non-trivial changes (e.g., a composition of simple refactorings).

To help developers repair client programs that are affected by the non-trivial evolution of a framework, we propose an approach to recommend adaptive changes, a form of maintenance aiming at adjusting a software to comply with its technological environment [15]. Our idea is to automatically analyze how the framework was adapted to its own changes, and to recommend similar adaptations. Basically, if a method m1 is removed from the framework code, we can identify all of the callers of m1 within the framework and analyze how they were adapted to the removal of m1.

We implemented this approach for Java in a client-server application called SemDiff. The SemDiff server component is responsible for analyzing the source code repository of a framework and for inferring high-level changes such as method additions and deletions. The client component takes as input calls to methods that no longer exist in a framework and produces recommendations in the form of recommended replacement methods, accompanied by a confidence value.

We evaluated the effectiveness and the need for our approach by using SemDiff in an historical study of one framework in which we recommended adaptive changes for three
broken client programs. We then compared our results with the recommendations of a typical refactoring detection tool. Our study showed that our approach provided a relevant functionality replacement for 89% of the broken methods, detected non-trivial changes that were more complex than refactorings, and could recommend methods from external libraries that replaced a framework’s functionality, a change that is typically not detected by other techniques.

The contributions of this paper include (1) a technique to automatically recommend adaptive changes in the face of non-trivial framework evolution, (2) the architecture of a complete system to track a framework’s evolution and infer non-trivial changes, and (3) an historical study on the redesign of a framework component that occurred in Eclipse, providing evidence for the effectiveness for our system.

In the remainder of this paper, we present a sample scenario that illustrates the difficulties associated with non-trivial framework evolution (Section 2). We then describe the principles and implementation details underlying our approach (Section 3) and present a study on the evolution of the Eclipse Java Development Tool (JDT) framework (Section 4). We conclude with an overview of the related work (Section 5).

2. Sample Scenario

Let us now consider the concrete case of a developer who decides to reuse internal classes of the Eclipse framework in a client program. In Eclipse, classes that are in a package containing the word internal are, by convention, not part of the supported API. It is generally understood that internal classes can change from one version of the framework to the other and that no documentation (e.g., javadoc) or migration path is provided.

One of the classes the developer considers for reuse, org.eclipse.jdt.internal.corext.util.TypeInfo, is contained in the org.eclipse.jdt.ui plug-in in Eclipse release 3.2. This class, along with several others, such as TypeInfoFactory, TypeInfoUtil and OpenTypeHistory, provides a lightweight API for searching and displaying Java type information (e.g., a type name, package name, or access modifiers).

When Eclipse 3.3 is released, the developer loads the client project in the development environment, which automatically tries to build the developer’s program against the new version of the framework. At this point, the developer discovers that the compiler generates multiple compilation errors, because the TypeInfo class and its methods are no longer accessible in Eclipse 3.3. The developer then starts exploring the source code of the new version of Eclipse in the hopes of finding a suitable replacement for these missing methods, searching for a class with a name similar to TypeInfo. Seeing a few classes named TypeInfo, the developer quickly realizes that they are defined in external libraries and that no similarly-named classes provide the required functionality. The developer then moves on to see if the missing class is in the same package but under a different name. Again, no classes are found with the same functionality. Moreover, the TypeInfoFactory and TypeInfoUtil classes also have disappeared from this package.

Having ruled out a simple refactoring, the developer then looks at other classes that used to depend on TypeInfo and sees that some of them now refer to the org.eclipse.jdt.

[1] 1www.eclipse.org
Figure 2: The SemDiff Recommendations View (bottom) displays the recommendations to replace the Type-InfoFactory.create() method and the Compare Editor (top) shows how the framework was adapted to its own changes.

Figure 3: Using method call changes we want to find a suitable replacement for m1, we first find all of the methods where a call to m1 was deleted (e.g., methods caller1 and caller2 in Figure 3). Then, we gather all calls that were added in these methods as part of the the same change set (e.g., m2 and m3). Since we expect that methods might be adapted along with additional changes, we sort added calls by a confidence metric to provide a ranking of the potential replacements. Assuming that we only look for change sets made prior to an implicit target version v, we define the confidence value of a method n that replaces a call to method m with the following equations:

\[
\text{Rem}(m) := \{ x \mid x \text{ is a method that removed a call to } m \}
\]

\[
\text{Add}(m) := \{ x \mid x \text{ is a method that added a call to } m \}
\]

\[
\text{Callees}(m) := \{ x \mid m \text{ calls } x \}
\]

\[
\text{Callers}(m) := \{ x \mid x \text{ calls } m \}
\]

\[
\text{Potential}(m) := \bigcup_{x \in \text{Rem}(m)} \text{Callees}(x)
\]

\[
\text{Support}(m, n) := |\text{Rem}(m) \cap \text{Add}(n)|
\]

\[
\text{Confidence}(m, n) := \frac{\text{Support}(m, n)}{\text{Max}(\bigcup_{c \in \text{Potential}(m)} \text{Support}(m, c))}
\]

Figure 4: Support and confidence of a replacement

Figure 4 shows an example of each of the above definitions in the context of requesting a replacement for method m. The confidence metric of a recommendation n to replace a method m is the ratio of the recommendation’s support to the maximum support for all potential recommendations. The confidence metric is therefore a normalized value with a range of [0, 1] that is used to compare the relevance of the recommendations. For example, if o is the recommendation with the highest support, 2, and recommendation n has half of this support, 1, n will have a confidence of 50% (0.5). Because the support is bounded by the number of callers of the removed method (|Rem(m)| ≤ |Callers(m)|), we hypothesize that the framework will have a sufficient amount of calls to its own methods so relevant recommendations can be discriminated from irrelevant ones. Section 4.2 shows how this hypothesis held in practice.

When looking for a replacement for a method m, we search for the methods that removed a call to m and not all callers of m. Moreover, we do not care if method m still exists. This enables us to get recommendations (1) for methods...
that are deprecated but still accessed by a subset of the framework or (2) for methods that are replaced before they are actually deleted. This is one difference with previous approaches that use the addition and deletion of methods as the basis for detecting changes and refactoring [13, 14, 20, 21].

**Change chains.** It is possible that during the course of a framework’s evolution, a method is replaced several times, i.e., it is part of a change chain. As illustrated by Figure 5, a method might be renamed once in one version and renamed a second time in another version. Additionally, since we study the evolution of a framework at the change set level, it is probable that we will come across small changes that were never accessible to client programs (e.g., a method name was misspelled and corrected in the next change set, a developer reverted to the old version of a class, etc.).

To account for these situations, we must slightly modify the strategy defined above to detect whether a method is part of a change chain. Indeed, we do not want to recommend a method that changed subsequently and that is no longer accessible or relevant. One solution would be to automatically check if the recommended call exists in the framework version used by the client program. Unfortunately, this is not an adequate solution if the recommended method is deprecated or if the method is no longer used by the framework but was not removed. We thus rely on a different heuristic to determine whether a recommendation is part of a change chain. If we find that some methods calling the recommended method removed a call to the recommended method later, we conclude that our recommendation is probably in a change chain and the initial recommendation might still be valid. If we find that all methods calling the recommended method removed the call to the recommended method, we conclude that this recommendation is part of a change chain and we discard the recommendation because it is no longer relevant.

Once we have identified a recommendation as being part of a change chain, we reapply the call difference analysis described above to find a more relevant recommendation. This is illustrated in Figure 5 where our system would recommend to replace a call to m1 by a call to m3.

**Caller stability.** Because our strategy only relies on the outgoing call relationship, it is sensitive to the stability of callers throughout the framework’s evolution. For example, Figure 6 shows a situation where both the requested method, m1, and the caller, caller1, are deleted in the same change set. In this situation, the caller cannot be used to find a replacement for the requested method. To cope with this issue, we first need to find a replacement for the deleted caller and then, we can recommend the methods that are called by the caller replacement minus the methods that were previously called by the deleted caller. Figure 6 illustrates the case where caller1 was replaced by caller2 and caller2 replaced a call to m1 by a call to m2.

When finding a replacement for a caller method, SemDiff can generate multiple recommendations (e.g., the method was splitted, there are truly multiple relevant replacements, false positives, etc.). To reduce the impact of false positives, we first remove from the potential caller replacements all recommendations that have a confidence value below a certain threshold (0.6) and then, we perform the call difference analysis on each of the remaining recommendations. We chose a threshold of 0.6 because it appeared to offer the best results during initial prototyping of the approach.

**Spurious call removal.** When finding a replacement for a method m1, SemDiff looks for change sets where a call to m1 was removed because this is typically where the framework will be adapted to a change concerning m1. It is possible though that a call to m1 is removed in one place and added in another place in the same change set (e.g., the caller was refactored, the caller was made more cohesive and the call to m1 was moved elsewhere, etc.). In these cases, the framework is not being adapting to the loss of m1 since it is still calling it elsewhere. To make sure our analysis does not take into account these spurious call removals, SemDiff will ignore all change sets where the requested method call (e.g., m1) was removed from one caller and added in another caller.

**Complexity.** The main factors affecting the computational complexity of SemDiff’s algorithm are the number of methods that removed a call to the queried method, the number of different added calls, the maximum change chain length, and the maximum length of unstable callers. A complete description of SemDiff’s algorithmic complexity is not possible due to space constraints.

**Viewing recommendations.** The recommendations produced by SemDiff are presented to the user in the Eclipse development environment and take the form of a list of methods, ranked by their confidence value. The user can also double-click on a recommendation to open the Eclipse compare editor with one example where the recommended method replaced the broken call. This allows the user to understand why this particular recommendation was provided and see how the framework was adapted to this change.

**Current limitations.** A number of factors constrain the applicability of our approach. The most important limitation is that the framework cannot issue recommendations for root methods, i.e., methods that are never called within the framework. Classes that are called back by other libraries or protected methods that are called by a parent class not in the framework will hinder the ability of our approach to make adaptive change recommendations. One solution to the problem of root methods is to include in our analysis example programs that depend on the framework and that were adapted to its various versions. In practice, however, we expect changes to root methods to be significant, and as such to be more likely to be documented.

Although our approach can make multiple recommendations per request, it does not group the recommendations together. For example, a call to method m1 might have been
3.2 Analyzing source code history

To provide adaptive change recommendations, SemDiff must first analyze a framework's source code repository by (1) retrieving the files and change data for each version of the framework, (2) running several analyses to infer high-level changes such as structural and method call differences, and (3) storing those high-level changes in a database for future use by our recommendation system.

Source Repositories. Currently, SemDiff provides adapters to retrieve information from CVS and Subversion (SVN) repositories. SVN has the concept of change set embedded in its protocol, which makes it easy to retrieve and group files that were changed together. On the other hand, CVS does not group file changes and some preprocessing of the repository log file is needed to infer change sets. We employed a technique previously used to mine CVS repositories to retrieve the change sets [20, 23]: we group all log entries that occurred within a certain time window (300 seconds) and that shared the same user and log message. This concept of time window is essential to capture transactions that could span across multiple seconds [22].

The merging of branches is another issue that arises when analyzing software repositories. This operation is not explicitly documented by neither CVS nor SVN. Detecting merges is important because we do not want to analyze the same change twice: one in the branched version and one in the merged version. To detect merges, we employed a simple heuristic used in previous work on source repository mining [20, 23]: we ignored change sets involving more than 40 files.

Change Analysis. For each change set, SemDiff can run custom analyses to infer high level information such as the removal or addition of methods from the raw line difference data provided by the repository. This high level information is then used by our recommender. Because we only retrieve the files that were added, removed or modified in

// method m1 in Bar.java, version 1.1
private void m1 {
    System.out.println();
}

// method m1 in Bar.java, version 1.2
private void m1 {
    System.out.println(Math.random());
}

Figure 7: Outgoing call differences

replaced by a call to methods m2 and m3. Our approach will present those two calls as two separate recommendations with, possibly, a different confidence value. To help a user identify relationship between recommendations, SemDiff displays the code where the changes happened: automatically inferring those relationships remains an area for future work.

Finally, adaptive changes proposed by SemDiff might not be semantically equivalent to features that need to be replaced, and blindly applying the recommended changes without proper testing could lead to serious flaws. Other research projects currently aim at generating test cases to ensure that the semantic of a program is preserved when applying a refactoring [4]: it is possible that our approach could directly benefit from such developments.

3.3 Partial Program Analysis

With Java, most parsers and static analysis programs fail to reconstruct the complete type hierarchy if they only receive as input a subset of the program source code without the dependencies and the rest of the program (in the form of source or binary). A few parsers, like the one provided by the Eclipse Java Development Tool framework, can construct abstract syntax trees (AST) from a subset of the program source code, but the information they provide is incomplete. For example, in Figure 8, the Eclipse parser would be able to recognize that in method doSomething(), there is a call to a method named x at line 7, but it would not indicate its target, an object of type Y, because it is an unknown class. To enable the analysis of partial programs,
we modified the Soot Java static analysis framework [19] and the Java source parser it uses, Polyglot [16].

Our implementation of Partial Program Analysis first replaces every occurrence of unknown types by a placeholder type: UNKNOWN. Then, it tries to conservatively infer the actual type of the placeholder types by analyzing how the various unknown types are used. For example, our analyzer would infer that the following methods are called in doSomething(Y): Y.x(), Util.method2(Y,double), Foo.doThis(Z), Util.method3(), and UNKNOWN.method4().

We configured our partial program analyzer so that the inferred type will always be in the hierarchy of the formal (defined) type, i.e., it will be a subtype, a supertype or the same type as the formal type. For example, our analyzer infers that the method Util.method2() on line 9 returns an object of type Z, even if this method might be defined to return a subtype of Z. It follows that the inference result, Z, is not equals to the formal type, but it is in its hierarchy. Inferring a hierarchy-related type is still more precise than inferring that the return type is UNKNOWN or Object.

The recommender also needs to take into account that the type information of a call might be incomplete. In the worst case, it might be impossible to know the target and the parameter types of a call. This is the case of method m1 in the following example, if that we do not have the definition of myObj's type:

```
myObj.myMethod().m1(myObj.myOtherMethod())
```

Polymorphism can also be an issue. In the next example, the calls at line 2 and 3 refers to the same method, but our partial program analysis would treat them as two different calls, ArrayList.add(Object) and ArrayList.add(String), if we do not have the definition of ArrayList.

```
1: List list = new ArrayList();
2: list.add(new Object());
3: list.add(new String());
```

This lack of accurate type information can be a serious problem for common method names, such as add and remove. Indeed, if we want to find a replacement for a method add(Object) that is no longer accessible, we cannot search for all methods that removed a call to a method named add with one parameter: we would probably retrieve a lot of false positives coming from other irrelevant classes that defined a similar method. Currently, we try three matching criteria, starting from the strictest one, until we can find calls. We first try to find methods that removed a call sharing the same name, number of parameters and target type as the call we want to replace. If we do not find such methods, we then try to find calls that share the same name, number of parameters and parameter types. Finally, if this still does not return any results, we then try to find calls only by their name and number of parameters. The complete details of our partial program analysis algorithms can be found in a separate report.5

4. EVALUATION

The main strategy underlying SemDiff relies on a number of hypotheses we made on framework evolution. We designed a study to assess the validity of these hypotheses and to evaluate the effectiveness of our approach. This study helped us answer the following questions:

1. Can SemDiff recommend to a client program adaptive changes that replace a functionality deleted during a framework’s evolution?
2. Is the confidence value good enough to discriminate relevant recommendations from false positives?
3. Can SemDiff detect changes more complex than refactorings?

4.1 Study Design

To answer the above questions, we performed an historical study of one framework and three client programs. We used SemDiff to adapt an old version of a client program to the new version of the framework. We then compared our adaptation recommendations to the historical (real) adaptation of the client program. To evaluate the complexity of the changes that occurred during the framework’s evolution, we also used a refactoring detection tool to analyze the framework’s history and provide recommendations to client programs.

In summary, for each client program, we selected two versions (c1,c2): one that was using an old version of the framework (f1) and one that was using the most recent version (f2). We then tried to compile the c1 version of each client program with the f2 version of the framework. For each method call in the client program that could not be resolved or that was deprecated (as determined through warnings based on the use of the @deprecated javadoc tag), we used the SemDiff recommender and a refactoring detection tool to see if we could find a suitable replacement for the broken method call. We then analyzed the c2 version of the client program to see if the recommended methods were called.

Target systems. We chose the Eclipse Java Development Tool (JDT) platform as the framework to analyze in our study. This framework is large enough to provide evidence that our approach scales, its source history is publicly available, it is actively maintained and has a large ecosystem of client programs. We chose to study two modules of this framework, the org.eclipse.jdt.core and org.eclipse.jdt.ui plug-ins from version 3.1 to 3.3. These plug-ins are mainly responsible for the Java compiler and Java editor in the Eclipse development environment and, in our experience, client programs that depend on JDT always depend on at least one of those two plug-ins. From release 3.1 to release 3.3, the jdt.core and jdt.ui plug-ins grew respectively from 222 to 261 kLOC and from 256 to 311 kLOC. We chose to study those plug-ins across three major revisions (3.1, 3.2 and 3.3) to increase the odds of finding non-trivial changes and change chains.

Finding suitable client programs was an harder task. We needed client programs that (1) depended on JDT, (2) had been adapted to the two versions of the framework we studied (3.1 and 3.3), and (3) replaced a functionality that disappeared during the framework’s evolution by a functionality provided by the last release of the framework. We ran into several cases where the last condition was not met. For example, the AspectJ Development Tool6 client program copied entire classes from JDT release 3.1 into its own code base instead of calling a new JDT functionality. Another JDT client program, the Eclipse Modeling Framework,7 replaced a deprecated framework functionality with its own

6www.eclipse.org/ajdt/
7www.eclipse.org/modeling/emf/
SemDiff. We used SemDiff to analyze the source history of the Eclipse framework. SemDiff processed 10127 change sets for the two jdt plug-ins in the Eclipse CVS repository from January 2005 to July 2007, as Eclipse 3.1 and 3.3 were respectively released on June 27 2005 and June 25 2007. Because work on Eclipse 3.2 might have begun before the release of Eclipse 3.1 (e.g., in a branch), we started to study the framework in January 2005 instead of June 2005.

Once we analyzed the framework’s source history, we tried to compile the first version of the client programs with Eclipse 3.3. For each call to a framework method that was deprecated or that could not be resolved by the compiler, we ran the SemDiff recommender and noted its recommendations. We then looked at the version of the client program that had been adapted to Eclipse 3.3: if the client program called one of the top three recommendations for each broken call, we considered it to be a relevant recommendation.

RefactoringCrawler. We also used a typical refactoring detection tool to discriminate non-trivial changes that occurred during the framework’s evolution from simple refactorings. We chose RefactoringCrawler [5] as it was easy to use, configurable, readily available and representative of several refactoring detection techniques; a more detailed comparison of such techniques is given in Section 5. In essence, RefactoringCrawler takes two complete versions of a project as input and gives a list of refactoring pairs (e.g., method m1 was renamed to method m2).

Following the tool author’s recommendations,8 we configured RefactoringCrawler to raise the number of detected refactorings at the expense of a higher number of false positives by lowering several threshold values. Indeed, we did not want to assess the accuracy of the tool, but use it as a baseline to differentiate refactorings from non-trivial changes.

In addition to the jdt.core and jdt.ui plugins, we added the jdt.ui.tests and jdt.ui.tests.refactoring to the set of plug-ins analyzed by RefactoringCrawler to increase the incoming calls to the jdt.ui plug-in, which was required by this approach to increase the odds of detecting refactorings. We combined in one result set the detected refactorings from the following three version pairs: 3.1 to 3.2, 3.1 to 3.3 and 3.2 to 3.3.

We then followed the same procedure as we did for SemDiff: for each broken call in the first version of a client program, we tried to find a refactoring involving the called method. If we found such a refactoring and the refactored element was used by the second version of the client program, we considered that the refactoring detection tool succeeded in providing a relevant adaptive change and that this change was a refactoring.

4.2 Results

Table 2 shows the results of our study. For each client program, we list the number of compilation errors related to the JDT framework (Errors), the number of errors within the scope of our approach (Scope), the number of errors that could be fixed based on the top three SemDiff recommendations (SemDiff) and the number of errors that could be fixed based on the refactorings detected by RefactoringCrawler (RC). The number of errors represents all deprecated accesses and all compilation errors (e.g., import statement referring to an unknown class, unknown parameter type when declaring a method, unknown method call, etc.). For example, in Mylyn, there were 13 errors related to the JDT framework of which 8 were within the scope of our approach. SemDiff provided relevant recommendations for 8 of them while RefactoringCrawler detected no refactoring relevant to these errors.

We consider an error to be within the scope of our approach if the type of the error is in the input domain of SemDiff (or RefactoringCrawler). Because SemDiff takes as input a method and gives as output a list of methods, we only considered unresolved and deprecated method calls to be within the scope of our approach. Errors such as unknown import statements cannot be provided as input to SemDiff so we did not try to fix them. Even if method recommendations can be indirectly used to fix these kinds of errors, there was not always an objective way to measure the success of our recommendations.

The two numbers in the Scope column for jdt.debug.ui represent two interpretations of the scope of our approach. Although there were 19 errors that are applicable to our approach, five could not be validated by following our experimental methodology because the client program replaced the missing functionality by its own implementation instead of using methods in the new version of the framework. In this case, even if our approach (or RefactoringCrawler) provided the correct recommendations, we would not be able to assert this fact using the client’s history as an oracle. We thus include 14 as the number of adaptation problems for which there is an objectively verifiable solution.

The execution of the repository analysis framework took 16 hours on a Pentium D 3.2 Ghz with 2 Gb of RAM and running Ubuntu Server 7.04. This analysis needs only to be performed once before a user can make requests in different disconnected sessions. On average, each request took 1 second to complete. Running the three analysis with RefactoringCrawler took 13 hours.

---

8D. Dig. Personal communication, 25 August 2007
Relevant recommendations. SemDiff found relevant recommendations for 89% of the problematic calls in the client programs. For example, in Mylyn, SemDiff suggested two relevant replacements with a confidence value of 1.0 for `TypeInfo.Factory.create(...)`, which returned an instance of the `TypeInfo` class presented in Section 2. Those two replacements were a call to the `JavaSearch.TypeNameMatch` constructor and a call to the method `SearchEngine.createTypeNameMatch(...)`, both returning a subclass of `TypeNameMatch`.

Again in Mylyn, SemDiff was able to correctly recommend a call to `OpenTypeHistory.remove(TypeNameMatch)` which replaced a call to `OpenTypeHistory.remove(TypeInfo)`. Although this change might appear to be a simple refactoring, it was not detected by RefactoringCrawler as such. The detection of this change also indicates that the heuristics introduced to cope with the inevitable inaccuracy of Partial Program Analysis succeeded to find a replacement to a method with a common name.

Finally, in `jdt.debug.ui`, SemDiff was not able to provide the correct recommendation for four methods. These are protected methods defined in the class `TypeSelectionDialog2` and accessed solely by this class. Because `TypeSelectionDialog2` was deprecated but not removed in Eclipse 3.3, no call to those four methods were removed and SemDiff could not generate any recommendation. As an alternative strategy, we asked SemDiff to find a replacement for a public method of `TypeSelectionDialog2`, and this time, the recommender suggested a method from a new class that was used by `jdt.debug.ui`. Indeed, the framework was adapted to the depreciation of a public method, but not to the depreciation of the protected ones.

Confidence value. In most cases, the confidence value was necessary to discriminate relevant replacements from false positives because SemDiff produced an average of 7.1 recommendations per request. In average, there were 1.6 recommendations with a confidence value of 1.0 per request. The support of the relevant recommendations had an average of 2.2 methods. Because this low support was enough to distinguish relevant recommendations from bad ones, we consider this to be evidence that our approach can work only by analyzing the code of the framework itself and does not require a set of examples using the framework.

Non-trivial changes. Although RefactoringCrawler found 319 refactorings between JDT releases 3.1 and 3.3, only one of them was related to errors in the client programs we studied (the six errors reported in Table 2). This observation provides evidence that our approach works in the face of non-trivial changes. For example, in Mylyn, the suggestion to replace a factory method involving the `TypeInfo` class with methods related to `TypeNameMatch` was far from trivial: as illustrated by Figure 9, this change spanned across the two `jdt` plug-ins and was part of a change chain.

Another interesting recommendation was provided for the JBoss IDE: SemDiff recommended to replace the constructor of `ListContentProvider` with the constructor of `ArrayContentProvider`. Although the former is located in the `jdt.ui` plug-in, the latter is located in the `org.eclipse.ui` plugin, which was not even analyzed by SemDiff. This shows that SemDiff can provide recommendations when a framework feature is replaced by an external functionality. Being able to track changes that are outside the analyzed framework could also enable us to recommend adaptive changes related to a framework, but only by analyzing a subset of its client programs. This would make our approach usable even if the framework’s source code and source history were not publicly available.

Finally, to detect non-trivial changes, SemDiff used the three heuristics presented in Section 3.1 several times when recommending adaptive changes to the three client programs: change chains were detected in 6% of the requests, caller replacements had to be found in 6% of the requests and change sets with spurious calls were removed 67% of the time.

Summary. SemDiff was able to recommend relevant adaptive changes in the face of framework evolution in 89% of the time. The fact that RefactoringCrawler was only able to detect a small subset of the changes that broke the client programs indicates that a developer would probably have struggled to find a suitable replacement for most of the broken calls. Arguably, even if a developer had found a replacement, the low cost of SemDiff on the client side (each request took an average of 1 second) makes our approach more effective in most cases. On the server side, SemDiff took less than 6 seconds to process each change set, which is typically faster than the compilation of the framework or the execution of test suites in a continuous build environment. SemDiff could then be integrated with such environment without affecting the development process.

4.3 Threats to Validity

The external validity of this study is limited by the fact that we only studied the evolution of the Eclipse JDT framework and it might not be representative of the code and evolution patterns of other frameworks. Multiple factors such as change set granularity, method cohesion, and programming idioms vary between software projects and can affect our approach. For example, the two first factors will introduce noise while some programming idioms such as using long call chains (e.g., `a1().a2().a3().a4()`) are likely to decrease the precision of our call difference analysis. Still, the impact of these factors on the results is mitigated by the strategies we devised to prevent them (e.g., confidence value). Moreover, because 21 developers contributed to JDT, we can reasonably assume that the results we obtained were not related to a particular developer profile.

To evaluate the relevance of our recommendations, we analyzed the evolution of client programs. Since we used historical data, we can only speculate on why the methods we recommended were called by the client programs. We cannot assess how the developers would have used our recommendations, although such an assessment would form a natural next step in the evaluation of our approach.

Finally, by using client programs and a refactoring detection tool, we tried to limit the need for personal judgment when assessing the relevance of a recommendation and the complexity of a change. The choice of client programs is
still subject to investigator bias, but this risk is mitigated by the fact that the investigators had no control and were not involved in the evolution of the selected client programs.

5. RELATED WORK

Supporting framework evolution is an active research area and various techniques have been proposed with this goal.

Migration Path. A number of approaches have been proposed to document framework changes and allow client programs to be automatically repaired. For example, transformation rules [3] can accompany a framework to indicate how calls to functions in the old framework version should be modified in order to work with the new version. CatchUp! [10] is a tool integrated in the Eclipse development environment that captures refactorings explicitly applied (i.e., using Eclipse refactoring tools) by developers. The captured refactorings can then be replayed in a client program to repair broken method calls. Explicit support for saving and replaying refactorings was introduced in Eclipse 3.3 in the form of Refactoring Scripts. As opposed to transformation rules, our approach is fully automated and does not require the framework authors to explicitly describe how the client program should adapt to the framework. Our approach also captures more changes (e.g., non refactorings, imported functionality) than the second technique as it is not limited to explicitly applied refactorings. Moreover, author of the client program does not need to search through a list of refactorings to find the one that is relevant to a broken method.

Change Detection. Most refactoring detection techniques refer to Origin Analysis [9] as the basis of their approach. Origin Analysis is a semi-automated technique that aims at tracing back to the source of a program element in a previous version of the program to detect changes such as splitting and merging. Similarly, if a refactoring detection technique finds that a method \( m \) in version \( n+1 \) is the origin of method \( f \) in version \( n \), it will conclude that method \( e \) was refactored to method \( f \).

To identify the origin of a program element, refactoring detection techniques use a variety of strategies based on program element characteristics (e.g., name, location, outgoing and incoming references, textual similarity) to assess the similarity of two elements across two versions. If the similarity of the program elements is beyond a certain threshold, those techniques conclude that one is a refactored variant of the other.

More precisely, techniques such as UMLDiff [21] analyze code relationship similarity (e.g., calls, hierarchy, accesses, etc.) between complete versions of a program to detect high level changes such as refactorings. RefactoringCrawler [5] is a similar but more lightweight alternative to this approach that also analyzes some code relationships and uses Single, a custom syntactic similarity analysis, on two complete versions of a program to detect refactorings. A later implementation of Origin Analysis [14] fully automated the approach for Java, but also reduced the number of change types that the technique detected to consider program elements renaming and move. Mining software repositories [23] is another technique that can be used to track changes: by analyzing a program’s evolution and detecting code clone patterns (textual similarity), researchers were able to detect refactorings [20]. Other researchers also used repository mining to predict the likelihood of a class to be refactored in the next two months [17] or to classify fine-grained changes that occurred inside method bodies [8]. Finally, M. Kim et al. proposed a technique for detecting change patterns by leveraging the similarity of program element names [13]. Table 3 shows a comparison between the characteristics that are used to assess the similarity of program elements and the types of programs these various approaches require...

Typically, techniques based on Origin Analysis will not be able to detect some or all of the following three kinds of changes: first, changes that disfigure a significant part of the program structure such as a rearchitecture will throw out most refactoring detection techniques because the level of change in a program element’s characteristics will be too high to safely conclude that two program elements are similar. Second, small changes performed on small methods will also hinder the accuracy of most refactoring detection techniques: for instance, if one line of code is changed in a two-lines method, some technique will conclude that 50% of the method changed and that the two are not similar. Third, changes involving code external to the program under study will be missed. For example, if a feature that was originally in the program is now imported from an external program (e.g., a library offers a better version of the feature), techniques based on origin analysis will not be able to capture the change because the destination is not in the analyzed program.

As shown with the study presented in Section 4, our approach does not suffer from these three limitations because we do not try to assess the similarity of methods that changed: we only analyze what happens to methods that were referring to the changed methods. Still, techniques based on origin analysis can give a certain degree of confidence about the origin of an element throughout the evolution of a program. This confidence is required by activities demanding detailed traceability information, such as bug tracking.

Finally, another approach using associative data mining on framework usage changes (such as method call changes) also outperformed an origin analysis-like technique and could even recommend more change patterns than SemDiff (e.g., a field access should be replaced by a method call) [18]. As opposed to SemDiff, this approach only compares two full versions, which can introduce more false-positives in the results and it also requires stable callers.

Framework Usage. Another family of approaches could potentially be used to support framework evolution. Strathcona [11] and FRUIT [2] are systems that mine a set of framework usage examples and recommend program elements of potential interest for framework users based on the local programming context. For example, if a developer is using class \( C \) and calling methods \( m_1 \) and \( m_2 \) from a certain framework, framework usage tools will typically recommend other program elements that are used along those classes and methods in the mined examples.

We could potentially use framework usage tools to recommend adaptive changes by mining usage examples of the new framework version and running the tools on each method in the client program that has a broken method call. The recommendations would probably contain the correct replacements. One of the issues with these approaches is that the data mining techniques they use usually need a fair number of usage examples in order to produce accurate results: unfortunately, it takes some time before an adequate number of usage examples becomes available when a new framework
Acknowledgments

The authors are grateful to Danny Dig for his suggestions on using RefactoringCrawler and thank Ekwa Duala-Ekoko for his valuable comments on the paper. This project was supported by the NSERC.

7. REFERENCES


