Mining Framework Usage Changes from Instantiation Code

Thorsten Schäfer Jan Jonas Mira Mezini
Software Technology Group
Darmstadt University of Technology
{schaefer,jonas,mezini}@st.informatik.tu-darmstadt.de

ABSTRACT

Framework evolution may break existing users, which need to be migrated to the new framework version. This is a tedious and error-prone process that benefits from automation. Existing approaches compare two versions of the framework code in order to find changes caused by refactorings. However, other kinds of changes exist, which are relevant for the migration. In this paper, we propose to mine framework usage change rules from already ported instantiations, the latter being applications build on top of the framework, or test cases maintained by the framework developers. Our evaluation shows that our approach finds usage changes not only caused by refactorings, but also by conceptual changes within the framework. Further, it copes well with some issues that plague tools focusing on finding refactorings such as deprecated program elements or multiple changes applied to a single program element.

Categories and Subject Descriptors

D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement; D.2.13 [**Software Engineering**]: Reusable Software

General Terms

Design

1. INTRODUCTION

As any successful software, frameworks evolve with time. In this process, framework elements (interfaces, classes, methods, fields) may change. If the changed elements are used in existing instantiations (by being implemented, inherited from, instantiated, called, overridden, or accessed), the instantiations may need to be migrated to the new framework version. This is a tedious and error-prone process that benefits from automation.

Several techniques and tools [8, 9, 16, 19, 25, 27] have been developed to discover the refactorings that a software system has undergone during an evolution step by analyzing two subsequent versions of the evolved software system. By applying these approaches to framework code, one can use the produced information to derive guidelines as how to change clients in order to use the new framework version. However, there are two problems with such techniques.

First, only usage changes that are caused by refactorings can be derived. However, as our experiments show (cf. Section 4), a significant number of usage changes (10-34% for the three subject systems) were caused by framework changes that go beyond refactorings. For instance, during the evolution of the framework, concepts were changed or replaced, or the assignment of responsibilities to the building blocks was altered in a way that the previous behavior was not preserved. We refer to this kind of changes as *conceptual changes*. Respective usage changes cannot be derived when using the techniques that focus on discovering refactorings only.

Second, our experiments also show that some specifics of framework evolution complicate the process of discovering framework refactorings and hence the process of deriving usage changes. One the one hand, outdated framework code is often not removed instantly for backward compatibility, but rather documented as deprecated [9]; as the program elements used in the old version are still available in the new version, no refactorings are discovered and hence no usage changes are derived. The analysis is further complicated in situations where a single code element is affected by several changes [25], e.g., a method may be renamed and moved to a different class.

To address these problems, we propose an approach that mines usage change rules by analyzing changes in the way two subsequent versions of framework instantiations use the framework. We use the term *framework instantiation* to refer to any code that uses the framework. This includes framework-based applications as well as test cases; for our purposes, it is not necessary to distinguish between different instantiations.

We argue that information encoded in ported instantiation code is a useful source to understand the changes required for the migration to a new framework version. Our approach in a nutshell is as follows. For each instantiation class, we extract information about how the framework is used in the two versions. By applying an association rule mining algorithm to this data, we find rules of the form "Calls to method `Plugin.shutdown()` in version 1 are replaced by calls to method `Plugin.stop()` in version 2".

Our evaluation shows that the approach exhibits a high

ICSE'08, May 10–18, 2008, Leipzig, Germany.

recall in terms of changes identified by other state-of-the-art tools. The evaluation results confirm our hypothesis that it is beneficial to use instantiation versions for mining usage change rules. First, we show that our approach finds a substantial number of usage changes that are caused by conceptual changes of the framework rather than refactorings. Second, we show that analyzing how instantiations were adapted to a new framework version eases the detection of changes even when outdated program elements are only documented as deprecated or when several changes are applied to a single program element.

The remainder of this paper is structured as follows. Section 2 briefly introduces association rule mining. The approach is presented in Section 3. Section 4 evaluates the approach. Section 5 discusses open issues and Section 6 presents related work. Finally, Section 7 summarizes the paper and discusses future work.

2. BACKGROUND

The approach presented in this paper is based on association rule mining [1], a machine learning technique for finding interesting associations among large set of data items.

Assume that we have a database of *transactions*, where each transaction consists of a set of *items*. The problem of mining association rules is to find all rules `A` $\rightarrow$ `B` that associate one set of items with another set. `A` is referred to as the *antecedent* and `B` as the *consequence* of the rule. Two measures for the interestingness of a rule exist. The *support* specifies how often the items appear together. The *confidence* measures how many transactions that contain all items of the antecedent also contain all items of the consequence. To limit the result to most likely meaningful rules, a *minimum support* and a *minimum confidence* can be specified for the mining process.

For illustration, consider the following market basket transactions:
Transaction 1: [bread, butter, honey]
Transaction 2: [bread, butter, milk]
Transaction 3: [bread, sausages, ketchup]

An association rule mining algorithm with minimum support = 2 and minimum confidence = 0.5 finds two rules:
Rule 1: bread $\rightarrow$ butter; support = 2, confidence = 0.66
Rule 2: butter $\rightarrow$ bread; support = 2, confidence = 1.00

Market basket analysis is only one application of association rule mining. The technique can also be used to analyze software data, e.g., to find programming rules [5, 21] or bugs [18]. The application of association rule mining to software requires to determine (1) the properties of the software to be considered as items, (2) how transactions are defined, and (3) whether additional constraints exist for the rules to generate.

3. MINING USAGE CHANGES

The process of mining usage changes takes two versions of instantiation code as an input. A conceptual overview of the process is depicted in Figure 1[1]. In the first step, information about how the instantiation code uses the framework is extracted, e.g., which framework methods are called, which framework classes are sub-classed, etc. (Section 3.1). Next, transactions are built by combining usage information from the two versions of each instantiation class (Section 3.2). Finally, an association rule mining algorithm is applied to those transactions (Section 3.3).

[1]For simplicity, the instantiation code in Figure 1 consists of a single class.

3.1 Extraction of framework usages

For each usage of the framework found in an instantiation class, a *fact* encoded as `fact_type:program_element` is extracted. For instance, a call to a method `m()` declared in framework class `C` is encoded as `calls:C.m()`. In the following, we discuss the facts that are extracted and illustrate the semantics of the extraction process by the example in Figure 2.

In this example, the framework consists of classes `C1` to `C3` and the interfaces `I1` to `I3`. Of those, only `I2`, `I3`, `C2` and `C3` contribute to the usage interface: they, respectively their elements, are directly used by instantiation classes. `C1` is not directly used by any instantiation class: While `C4` does override method `m()` and call method `n()`, the latter are actually inherited from `C2`. Framework elements that are not used by instantiations are considered internal implementation details of the framework.

The extraction process is designed to be robust against internal restructuring within the framework or within the instantiation hierarchy. For illustration, consider the following two change scenarios. In the first scenario, the framework developers decide to remove the class `C1` in Figure 2 and to push down all its features to `C2`. Such a change does not affect the usage interface of the framework. Hence, it should not affect the set of extracted facts. In the second scenario, the instantiation developer decides to introduce a new class between `C2` and `C4`. Even though the framework class `C2` is now referenced within the new instantiation class, the usage information for `C4` should not change; `C4` uses the framework in a similar way as before.

To make the extraction process robust against internal framework changes, extracted facts always reference the framework type with the nearest distance to the instantiation class in the inheritance chain (gray boxes in Figure 2). For instance, we record the facts `overrides:C2.m()` and `calls:C2.n()` when analyzing `C4`, even though `m()` and `n()` are originally declared in `C1`. Also, only `extends:C2` but not `extends:C1` is extracted when analyzing `C4`, since `C2` is closer to `C4` than `C1` in the inheritance chain.

To make the extraction process robust against internal instantiation changes, the set of facts extracted for an instantiation class also includes facts "inherited" from its supertypes. For illustration, consider the facts extracted for `C5` in Figure 2. The fact `implements:I2` is created as `C5` directly implements the framework interface `I2`. Further, `implements:I3` is created, since `I3` is indirectly implemented (via `I4`). Similarly, the facts `extends:C2`, `overrides:C2.m()`, `calls:C2.n()`, `instantiates:C3` and `calls:C3.o()` are inherited from `C4`.

After having explained the rationales that drive the extraction process in terms of the example in Figure 2, in the following, the rules that guide the extraction of facts are stated in general terms. Given `T` be the instantiation type (class or interface) being analyzed, the following facts are extracted for it, whereby `FT` stands for a framework type:

- `extends:FT` is created if `T` extends `FT` directly.
- `implements:FT`, is created, if the instantiation class `T` either implements the framework interface `FT` directly, or it implements some instantiation interface `T1`, which

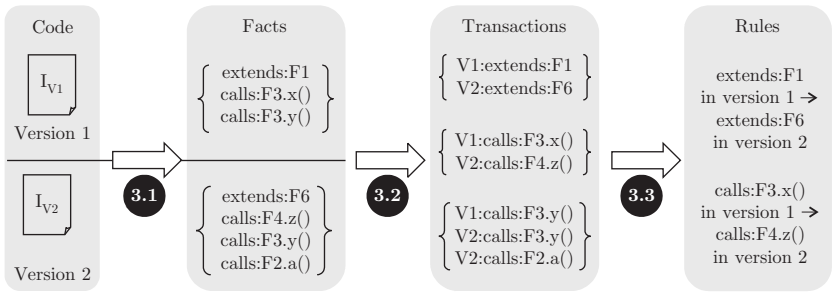

Figure 1: Change detection

in turn extends `FT` directly or indirectly via other instantiation interfaces.

- `overrides:FT.m()` is created whenever the instantiation class `T` overrides or implements a method `m()` declared by the framework. `FT` is the nearest ancestor of `T` in the framework from which `T` inherits an implementation/declaration of `m()`.
- `instantiates:FT` is created when some method of the instantiation class `T` instantiates objects of the framework type `FT`, or of some instantiation type `T1`, whereby `FT` is the most recent framework type that is an ancestor of `T1` in the inheritance hierarchy.
- `calls:FT.m()` is created when the instantiation class `T` contains a call to a method `m()` via some receiver object known to be of the framework type `FT`, or of some instantiation type `T1`, whereby `FT` is the most recent framework type that is an ancestor of `T1` in the inheritance hierarchy.
- `accesses:FT.f` is created when the instantiation class `T` accesses a field `f` of some object known to be of the framework type `FT`, or of some instantiation type `T1`, whereby `FT` is the most recent framework type that is an ancestor of `T1` in the inheritance hierarchy.
- Any fact `f` of any of the above types can also be extracted for `T`, if there is an instantiation type `T1`, `T` extends `T1` directly, and `f` can be extracted when analyzing `T1`. That is, `T` inherits facts from its supertypes, which are part of the instantiation.

Note how the extraction rules for `instantiates`, `calls`, and `accesses` take into consideration polymorphism. For illustration, consider the following code snippet[2]:

```
1 C5 c = new C5();
2 int x = c.f;
3 c.m();
4 c.n();
```

As `C5` is a subtype of the framework class `C2` (cf. Figure 2), the following facts are created: `instantiates:C2` (line 1), `accesses:C2.f` (line 2), `calls:C2.m()` (line 3), and `calls:C2.n()` (line 4).

The result of the extraction process is an abstract representation of the framework usage for both versions of an

[2]This code could be part of the implementation of a method in an instantiation class.

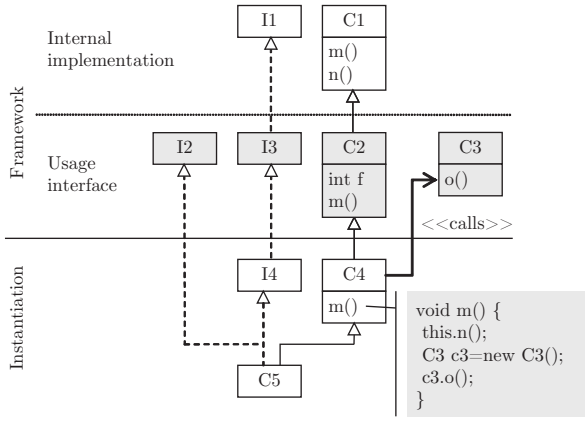

Figure 2: Exemplified class hierarchy

instantiation class. In the remainder of Section 3, we will use the instantiation code in Figure 3[3] rather than the example in Figure 2. While Figure 2 has been conceived sophisticated enough to illustrate the extraction process, it is too elaborated for illustrating the remaining steps of our approach. The example in Figure 3 is conceived simple enough to only illustrate `extends` and `calls` facts; the process is similar for all other facts.

```
1  class C1 extends F1 {
2    void a() { F3.x(); }
3    void b() { F3.x(); }
4    void c() { F3.y(); }
5  }
6
7
8  class C2 extends F2 {
9    void a() { F3.y(); }
10   void b() { F5.a();
11              F5.b(); }
12 }
```

Version 1

```
1  class C1 extends F6 {
2    void a() { F4.z(); }
3    void b() { F4.z(); }
4    void c() { F3.y();
5               F2.a(); }
6  }
7
8  class C2 extends F2 {
9    void a() { F3.y(); }
10   void b() { F5.a2();
11              F5.b2(); }
12 }
```

Version 2

Figure 3: Exemplary instantiation code

Table 1 lists facts that are extracted for each class in the instantiation code. For instance, the first gray row in Table

[3]All classes with names starting with "F" are part of the framework.

1 shows that C1 calls method F3.x() in version 1, while a call to method F4.z() is extracted for version 2. The second gray row indicates that both versions of C2 extend framework class F2.

Class (Line)	Context	Facts V1	Facts V2
C1 (1)	C1	extends:F1	extends:F6
C1 (2)	C1.a()	calls:F3.x()	calls:F4.z()
C1 (3)	C1.b()	calls:F3.x()	calls:F4.z()
C1 (4/5)	C1.c()	calls:F3.y()	calls:F3.y() calls:F2.a()
C2 (8)	C2	extends:F2	extends:F2
C2 (9)	C2.a()	calls:F3.y()	calls:F3.y()
C2 (10/11)	C2.b()	calls:F5.a() calls:F5.b()	calls:F5.a2() calls:F3.b2()

Table 1: Extracted facts

3.2 Creating change transactions

In the next step, we create transactions that serve as input to the data mining algorithm. A straightforward approach is to build one transaction per instantiation class, consisting of the union of all facts from both versions of that class. Such a "one-transaction-per-instantiation-class" mapping has proved successful for mining framework usage rules [5, 21] and for finding bugs [18]. Yet, it leads to many false positives when finding framework usage changes because of the following problems:

P1. Imprecise change information: A "one-transaction-per-instantiation-class" approach ignores details about the structural context of usage facts, which may cause ambiguities when deriving change rules. For illustration, consider the facts for the instantiation class C1 in Table 1. Calls to F3.x() in version 1 disappear in version 2; instead, two new framework methods are called: F4.z() and F2.a(). However, by using class-level transactions, we can not infer whether calls to F3.x() were replaced by calls to F4.z() or by calls to F2.a(). This is because the information about the specific context in which the calls happen (the methods within which they are called) is lost, when all usage facts of a class are mixed together into one transaction.

P2. Unwanted programming rules: With transactions that combine all usage facts of a class into a single unit, the mining algorithm would produce many rules that do not represent usage changes. For illustration, consider the facts for the instantiation class C1 in Table 1. One possible output of the mining process based on an one-transaction-per-instantiation-class approach is the rule "V1:extends:F1 → V2:calls:F2.a()"[4]. This is because the facts involved in the implication occur frequently together, if the transactions are built per class. Such rules often correspond to API contracts and thus are useful to understand the framework [5]; yet, most of them do not represent usage changes.

P3. Too much noise: Obviously, framework usages that did not change between two versions are irrelevant; the respective facts are noise in the data. For instance, given the facts from Table 1, the mining algorithm would produce a rule "V1:calls:F3.y() → V2:calls:F3.y()", which is useless when trying to understand framework usage changes.

[4]A prefix Vx: denotes that a fact is extracted from instantiation code of version x.

To address problems P1-P3, we apply (a) a more fine-grained mapping of facts to transactions, described in items 1 and 2 below, and (b) some filtering described in item 3.

1. Taking structural context into consideration: Our approach to create change transactions takes structural context information into consideration: Instead of considering all framework usages of an instantiation class together, we partition them into several *contexts*, one for the class declaration, and one for each declared field or method.

The basic assumption is that usage changes are typically localized within structural elements of the instantiation code, e.g., usage facts extracted from some method m() of class C in version 1 are not related to usage facts extracted from method n() of the same class in version 2, or to usage facts pertaining to the class declaration.

By creating a transaction per section, we encode more detailed information about the structural context of usage facts. This avoids ambiguous change rules discussed in P1. One may raise the issue that in cases where the instantiation code is heavily restructured, a change may involve facts that are spread over different structural elements, hence, invalidating our assumption. In such cases, we may indeed not be able to map corresponding framework usages of two versions. However, we observed that such restructurings are very rare for the subject systems analyzed in our evaluation; even if they occurred in particular instantiation classes, their effect is often compensated by many other instantiation classes that did not change much.

2. Taking change patterns into consideration: By analyzing migration guides, change logs, and the code of evolving frameworks, we observed that changes typically follow certain *change patterns*, regardless whether they are caused by conceptual changes or refactorings of the framework. For instance, the exchange of the plug-in life-cycle concept in Eclipse required instantiations to override new methods. Similarly, moving field x from class F1 to F2 results in a usage change involving two accesses facts: accesses:F1.x and accesses:F2.x. We derived the change patterns listed in Table 2, which comprise the different kinds of framework usages.

A transaction in our approach combines only facts that belong to the same change pattern. This ensures that only change rules that match one of the four pattern categories can be created. While this significantly decreases the number of false positives, usage changes that do not correspond to one of the defined patterns will not be found. To tackle this problem, the set of change patterns is extensible in our approach.

Pattern	Antecedent	Consequence
1	extends extends implements implements	extends implements extends implements
2	overrides	overrides
3	calls calls accesses accesses	calls accesses accesses calls
4	instantiates	instantiates
5	instantiates calls	calls instantiates

Table 2: Five categories of change patterns

3. Removing unchanged usages: To avoid the generation of rules that only represent the unchanged usage of the framework, we filter facts as follows: if a specific fact, e.g., `V1:calls:F.x()`, is contained in a transaction twice, i.e., one fact for each version, both facts are removed from the transaction. This filtering greatly reduces the number of generated rules, as often large parts of the framework remain stable. Further, those facts would be noise in the data; their removal also hinders the generation of other misleading rules that contain one of the facts.

Algorithm: Putting all pieces together, the transaction generation algorithm performs the following conceptual steps:

- Iterate over all instantiation classes that exist in both version 1 and 2. For each instantiation class:
  - Create an empty transaction for each possible combination of a context and a change pattern. For instance, a class declaration context may only contain facts from change pattern 1. A method context, however, may contain facts from change patterns 2, 3, and 4; hence, three transactions may be created for the same method context.
  - The corresponding facts are added to the empty transactions generated in the previous step.
- Remove facts representing unchanged usages from the transactions created by the previous step.
- Remove all transactions that only contain facts from version 2: no change rules can be learned from them.

Table 3 lists the four transactions (the numbered rows with white background) that are created for the extracted facts from Table 1. The structural context is used to partition the framework usages of a single instantiation class into several transactions (cf. transactions 1-3 for class `C1`). Further, each transaction only contains facts that match one of the change patterns. After filtering facts that appear in both versions (the stroked entries), the gray rows were removed because they only contain facts from version 2.

#	Context	Pattern	Facts V1	Facts V2
1	C1	1	extends:F1	extends:F6
2	C1.a()	3	calls:F3.x()	calls:F4.z()
3	C1.b()	3	calls:F3.x()	calls:F4.z()
	C1.c()	3	~~calls:F3.y()~~	~~calls:F3.y()~~ calls:F2.a()
	C2	1	~~extends:F2~~	~~extends:F2~~
	C2.a()	3	~~calls:F3.y()~~	~~calls:F3.y()~~
4	C2.b()	3	calls:F5.a() calls:F5.b()	calls:F5.a2() calls:F5.b2()

Table 3: Created transactions

3.3 Mining change rules

The created transactions serve as the input to an association rule mining algorithm. The algorithm generates all possible change rule candidates that fulfill three criteria. First, the support of the rule is greater than a threshold *minimum support*. Second, the confidence of the rule is greater than a threshold *minimum confidence*. Third, the antecedent consists of a single fact from version 1 and the consequence consists of a single fact from version 2.

In most applications of data mining techniques, e.g., market basket analysis, one often requires that the association rules have both high support and high confidence: the goal is to find frequently occurring patterns with a high significance. For our purposes, we also accept rules that exhibit low support, because changes may affect a small number of framework users only. Further, we are also interested in rules with a high confidence, but our evaluation shows that the precision is good even with a low minimum confidence.

The third criterion constraints the results to rule candidates that describe changes from version 1 to version 2. Most changes, and in particular refactorings, only affect two program elements. For instance, in case of the refactoring "change signature", the two program elements correspond to the method with the old signature (version 1) and the method with the new signature (version 2). Hence, we only accept rules whose antecedent and consequence are both singletons. Yet, this restriction may result in undetected changes; we will discuss this issue in Section 5.

#	Change rule candidate	Sup.	Conf.
1	V1:extends:F1 → V2:extends:F6	1	1.0
2	V1:calls:F3.x() → V2:calls:F4.z()	2	1.0
3	V1:calls:F5.a() → V2:calls:F5.a2()	1	1.0
4	V1:calls:F5.a() → V2:calls:F5.b2()	1	1.0
5	V1:calls:F5.b() → V2:calls:F5.a2()	1	1.0
6	V1:calls:F5.b() → V2:calls:F5.b2()	1	1.0

Table 4: Mined association rules

The final step of our approach filters some rule candidates if more than one rule with a specific antecedent or consequence exists. We identify the one that most likely represents a valid change; the other rules are considered as misleading and filtered out. Again, the rationale for this filtering is to focus on changes involving the exchange of a single framework usage by a single different single usage.

The decision for the most likely valid change rule is based on the assumption that the identifier of program elements describe their responsibility and thus are relatively stable during evolution. This is particularly true for many refactorings: e.g., when applying the "move method" or "change signature" refactoring, the name of the method remains unchanged.

Similarly to Xing and Stroulia [26], we use a heuristic that measures the textual similarity of a rule's antecedent and consequence based on the Levenshtein distance, i.e., the number of additions, deletions, and substitutions required to transform one string into another string. We modify the distance measure in such that we tokenize the rules and count the edit operations per token, not per character. For instance, the facts `calls:x(String, int)` and `calls:x(int, String)` should only result in one edit operation to reflect the parameter change. Next, for each token in the antecedent, the consequence token with the minimum Levenshtein distance is determined. The distance of a rule corresponds to the sum of the distances of its tokens in relation to the total number of tokens.

Given the example code from Figure 3 and the generated transactions from table 3, the mining algorithm produces six association rules listed in table 4. The rule candidates 3/4 and 5/6 have the same antecedent. Intuitively, we would assess rules 3 and 6 as valid change rules, while the other two rules are misleading (gray rows). This would be also

the result of applying our text-based rule filtering: the Levenshtein distance of the two valid rules is lower than the distance of the misleading rules; rules 4 and 5 would be filtered.

4. EVALUATION

A prototypical implementation of the approach described in Section 3 was evaluated with three subject systems. The evaluation was set up to answer the following questions:
1. How many and which kinds of usage changes are found?
2. How do user-defined thresholds affect the results?

4.1 Prototype

The prototype enables to find usage change rules for Java frameworks. The Eclipse JDT parser is used to create an AST representation of the instantiation code, from which all facts are extracted (step 1 in the process in Figure 1). A custom module generates the transactions (step 2). The apriori algorithm by Borgelt [4] is used to mine change rules (step 3). Finally, the mined rules are filtered in a post-processing step. A minimum support of `2` and a minimum confidence of `33%` are used as defaults; the thresholds are determined empirically by performing a parameter analysis. We did not take detailed performance measures, but the whole algorithm run in less than half an hour for each of the evaluated subject systems.

4.2 Setup

We performed three experiments on three different subject systems. First, our prototype was used to mine usage changes for the Eclipse UI framework [10] during its evolution between versions 2.1.3 and 3.0. The subject of the second experiment was the evolution of JHotDraw [12][5] between versions 5.3 and 5.4. The third experiment was conducted with versions 1.1 and 1.2.4 of Struts [24][6]. Our evaluation setup is appropriate due to the following reasons.

First, an evaluation with three different frameworks enables to judge whether the results gained for one framework are corroborated for the other frameworks.

Second, the subject frameworks range from small to large and are successful open-source frameworks with a large number of instantiations. Hence, mining their evolution changes to support the migration of respective instantiations constitutes a representative use case. The size characteristics of the frameworks are listed in Table 5.

	Size (KLOC)	No. of packages	No. of classes	No. of methods
Eclipse UI 2.1.3	222	105	1,151	10,285
Eclipse UI 3.0	352	192	1,735	15,894
JHotDraw 5.2	17	19	160	1,458
JHotDraw 5.3	27	19	195	2,038
Struts 1.1	114	88	460	5,916
Struts 1.2.4	97	78	469	6,044

Table 5: Size characteristics for the subject systems

Third, the setup allows assessing how the number of the available framework users affects the mining results. The Eclipse experiment covers the situation where ported instantiations are available. In addition to unit test cases, the plug-ins shipped with Eclipse were used; in total, 1622 classes that use the framework functionality were available in two versions, prior and after the migration to the new Eclipse version. On the other hand, the Struts experiment covers the situation where there is no access to ported instantiations; instead, we used test cases of the project as the learning basis (379 classes). For the JUnit experiment we used both, the test cases and the examples shipped with the framework.

Fourth, the systems have also been used for the evaluation of RefactoringCrawler [9], a tool that finds framework refactorings. This enables us to reason about the relation between framework refactorings and usage changes. A direct comparison between our approach and RefactoringCrawler is not possible, as the output of our approach are usage changes, whereas RefactoringCrawler outputs refactorings of program elements and there is no one-to-one mapping between the two sets. For instance, if a framework method is renamed and also moved to another class, there are two refactorings, but only one usage change. On the other hand, if we rename a framework method that is overridden and called by instantiations, a single refactoring corresponds to two usage changes (one for the change of the overridden method and one for the change of the called method). However, by analyzing which usage change rules are caused by refactorings and which of those refactorings are found by RefactoringCrawler, and by analyzing for how many refactorings found by RefactoringCrawler our approach outputs corresponding usage changes, we can get an idea about the strengths and limitations of our approach.

Finally, the authors of this paper are very knowledgeable about the source code of all three frameworks and hence capable to judge whether change rules mined by the algorithm are relevant or whether they are false positives.

4.3 Results

To assess their appropriateness, for each found rule we analyzed whether it is (a) caused by a refactoring performed on the framework, (b) caused by a conceptual change of the framework, or (c) a false positive. For this purpose, we consulted the framework documentation (Javadoc, migration guides, and release notes) and analyzed its source code. Further, we checked whether a refactoring corresponding to a usage change is found by RefactoringCrawler [9].

In the following, we present the quantitative results of our experiments without much comments. An interpretation of the results including the differences between the number of rules found by our approach and the number of refactorings found by RefactoringCrawler is given in Section 5.

An overview of the results of our experiments is given by the gray columns in Table 7. For the three experiments conducted with the default thresholds, a total of 255 correct usage change rules were found. A usage change is correct if it is either caused by refactorings of the framework code (ΣR), or by a conceptual change relevant for the migration according to documentation, migration guides, or our judgment (CC).

Three observations can be derived from Table 7. First, the majority of usage change rules found (193) are caused by refactorings. Second, around a quarter of the found changes that are relevant for the migration (62) are not caused by

[5]A GUI framework for technical and structured graphics.
[6]Struts is a controller framework for web-based applications

refactorings, but e.g., by the introduction of new concepts within the framework. Obviously, such change rules cannot be derived from findings produced by tools like RefactoringCrawler, focusing solely on refactorings. Finally, not all change rules found by our algorithm are correct. In total, 39 false positives (FP) were found in the experiments. However, the precision of our approach was high: 86,7 % of the rules found were correct. Note we can not state a recall measure due to lack of knowledge about *all* valid usage changes. Column FN shows the number of refactorings found by RefactoringCrawler, for which our algorithm was not able to discover the respective usage change rules. These refactorings can be considered as false negatives of our approach.

To give an idea about the quality of the correct change rules proposed by our approach, we discuss representatives of change rules for both, those that are caused by refactorings and those that are not caused by refactorings.

Rules 1 and 2 in Table 6 show examples for usage change rules caused by refactorings. The first change is caused by a "move field" refactoring applied to the Eclipse framework: class `IWorkbenchActionConstants` was split into a generic part and an IDE specific part (class `IDEActionFactory`) and the respective fields were moved into those classes. The second change discovered for the Struts framework is necessary as the functionality to retrieve a user's locale is moved from class `RequestUtils` to class `TagUtils`. Further, the name of the method changed. Hence, two refactorings were performed on the same program element.

Rules 3 and 4 in Table 6 correspond to usage changes that are not caused by refactorings. For instance, in the old version of JHotDraw it was possible to get the title of the current drawing from a `MDI_DrawApplication`. The method has been removed in the newer version; instead, clients need to send a message to the `Drawing` directly. Rule 4 proposes a usage change pattern for instantiation code needed to absorb a change in the Eclipse framework documented in the Javadoc comment as follows: *In Eclipse 3.0, shutdown has been replaced by Plugin.stop(BundleContext context). Implementations of shutdown should be changed to extend stop(BundleContext context) and call super.stop(context) instead of super.shutdown().* While the framework still contains the old hook method to maintain backward compatibility, instantiations should migrate to the new plug-in lifecycle concept.

Columns 2 to 8 in table 7 detail on the change rules that correspond to refactorings: rename class (RC), pull up method (PM), change signature (CS), move method (MM), rename method (RM), replace constructor with factory method (CF), and move field (MF). For each refactoring, the number of respective usage change rules found by our approach is given. This is the first number in each cell. The second number indicates how many of the change rules reference a program element, to which a second refactoring has been applied. The number in brackets shows how many of those usage changes are caused by a refactoring that is found by RefactoringCrawler. The numbers indicate that often the refactorings causing the found usage changes are also found by RefactoringCrawler; but, there are also several instances of usage changes that are caused by refactorings which are not found by RefactoringCrawler. Further, the figures show that there are several changes that concern program elements which were subject to several refactorings.

5. DISCUSSION

In this section we elaborate on three questions:
1. Are the assumptions underlying our approach realistic?
2. What are the strengths and limitations of our approach?
3. What are the threats to validity?

5.1 Assumptions

Three assumptions underly our approach: (A1) Users of the changed framework exist that are ported to the new version; (A2) The instantiation code is stable: Transactions can be only build for program elements that exist in both versions of the instantiations; (A3) Usage changes replace a single program element with another single one. In the following, we reason about the validity of these assumptions.

Assumption (A1) is realistic for large open-source frameworks. There are often early adopters of the changed framework with detailed knowledge of the framework code. For instance, when the Eclipse platform changes, the Java Development Toolkit is one of the first instantiations that is ported. Further, the Struts experiment has shown that our approach performs reasonably well, even if no instantiations are available beyond test cases. The latter are available for a large number of frameworks and they are typically ported to new versions because they are maintained by the framework developers.

To find out whether (A2) is realistic, we investigated the respective versions of the Eclipse instantiations. We discovered that overall 74% of the classes, 67% of the methods, and 56% of the field declarations exist in both versions of the instantiations. A further analysis revealed that in most cases when a program element that existed in version 1 disapeared in version 2, the fully qualified name of the respective element actually changed, e.g., because the program element was renamed. This is a limitation of the prototype, as it uses fully qualified names for mapping program elements between two instantiation versions. To cope with with this issue more advanced techniques for mapping program elements [14] can be used. However, our evaluation showed that we can extract sufficient input data for our approach even using the heuristic based on fully qualified names.

Assumption (A3) was valid for the rules found in our experiments: for rules that have the same antecedent but different consequences, only one of the rules was correct. However, there may be cases were our assumption does not hold. For instance, if `a` is replaced by `a'` and `b`, our approach could find the rule `a` → `a'`, but miss `a` → `b`. To cope with this problem, it is worth investigating a combination of our approach with approaches to detect programming rules, e.g., CodeWeb [21] or FrUiT [5]. These approaches will derive a rule `a'` → `b`, if `a'` and `b` always co-occur in the new instantiation version.

5.2 Strengths and limitations

To assess the strengths and limitations of our approach, we performed the following steps. First, we analyzed those usage changes that are found by our approach for which no respective framework refactorings are found by RefactoringCrawler. Further, we analyzed why our approach misses to detect usage changes for some refactorings found by RefactoringCrawler. Finally, we analyzed under which circumstances our algorithm produces false positives.

We identified several reasons for change rules that can be found by our approach, but for which no refactorings are

#	Change rule
1	`V1:accesses:IWorkbenchActionConstants.REBUILD_PROJECT` → `V2:accesses:IDEActionFactory.REBUILD_PROJECT`
2	`V1:calls:RequestUtils.retrieveUserLocale(PageContext,String)` → `V2:calls:TagUtils.getUserLocale(PageContext,String)`
3	`V1:calls:MDI_DrawApplication.getDrawingTitle()` → `V2:calls:Drawing.getTitle()`
4	`V1:overrides:AbstractUIPlugin.shutdown()` → `V2:overrides:AbstractUIPlugin.stop(BundleContext)`
5	`V1:extends:StatusTextEditor` → `V2:extends:AbstractDecoratedTextEditor`
6	`ImageRegistry.get(String)` → `IconAndMessageDialog.getWarningImage()`

Table 6: Example rules found by our approach

Experiment	RC	PM	CS	MM	RM	CF	MF	ΣR	CC	FP	FN	Precision
Eclipse UI	1/0 (0)	7/0 (7)	22/4 (7)	6/4 (3)	1/0 (0)	0/0 (0)	34/0 (0)	67	34	16	13	86,3 %
Struts	4/0 (4)	1/0 (1)	9/2 (4)	26/4 (12)	10/4 (4)	2/0 (0)	0/0 (0)	47	19	11	20	85,7 %
JHotDraw	1/0 (0)	0/0 (0)	56/3 (18)	3/2 (0)	17/5 (9)	0/0 (0)	7/0 (0)	79	9	12	2	88,0 %
Total	6/0 (4)	8/8 (8)	87/9 (29)	35/10 (15)	28/9 (13)	2/0 (0)	41/0 (0)	193	62	39	35	86,7 %

Legend: RC = Rename Class; PM = Pull up Method; CS = Change Signature; MM = Move Method; RM = Rename Method; CF = Replace Constructor with Factory Method; MF = Move Field; ΣR = Changes caused by Refactorings; CC = Conceptual Change; FP = False Positives; FN = False Negatives

Table 7: Usage change rules found

found by RefactoringCrawler. First, not all usage changes that are relevant for the migration to a new framework version directly correspond to refactorings. For instance, the new plug-in concept of Eclipse led to a change described in rule #4 in Table 6, which is not a refactoring. Second, two refactorings may be applied to a single program element at once. An example is presented in change rule #2 in Table 6. It has been shown that multiple refactorings typically cause problems when analyzing the framework code [25]. Third, tools that rely on code similarity to judge whether a program element in version 1 matches another program element in version 2, such as RefactoringCrawler, have problems with program elements that do not have a body, e.g., fields or interface declarations. For instance, change rule #1 in Table 6 involves moving a field to another class. When usages are not taken into account, the only information that can be used to match the program element is the field's name, which may actually change, too.

Next, we consider refactorings found by RefactoringCrawler, for which we did not find respective usage changes. The reason for such false negatives was that the analyzed instantiations did not use the respective parts of the framework. For instance, 16 refactorings of Struts found by RefactoringCrawler involved signature changes of methods responsible for the validation of the user input. As none of the analyzed instantiations used the validation part of Struts, we did not find usage changes for these refactorings. For such cases, it seems beneficial to combine our analysis with existing techniques that analyze the framework code, as discussed in Section 7.

Finally, we consider the results of analyzing the false positives produced by our algorithm. The source of such false positives are situations, where a program element is used by only few instantiations. For example, consider rule #6 in table 6. Searching for the source of this false positive, we discovered that the fact `V1:calls:ImageRegistry.get(String)` was contained in only two transactions, i.e., it was extremely rarely used by instantiations. Accidentally, both transactions containing this fact also contained the fact `V2:calls:IconAndMessageDialog.getWarningImage()`. As a result, our prototype produced rule #6 in table 6. In other words, if few transactions exist that exhibit a common pattern, a corresponding rule is derived even if it only applies to a specific instantiation and cannot be generalized for all framework users.

The findings that we derive from the results of our experiments can be summarized as follows. Many changes are found by both, our approach and RefactoringCrawler. Analyzing the use of the framework within instantiation code enables to cope with issues of current tools, such as deprecated elements, multiple refactorings applied to the same code elements, changes of body-less elements, and changes that are not caused by refactorings. However, those changes can only be found if instantiation code that uses different parts of the framework exists. Further, our algorithm only outputs change rules; it does not infer the concrete cause of the change as RefactoringCrawler does. We conclude that our approach complements rather than replaces existing work; an investigation of the effectiveness of combining different approaches remains for future work.

5.3 Threats to validity

There are three threats to the validity of our findings. First, although we used frameworks from different domains, written by different developers, they may not be representatives for all kinds of frameworks: All three are implemented in Java as open-source projects and these characteristics are not shared by all frameworks.

Second, a similar threat concerns the selection of instantiations. We used real-world instantiations for the Eclipse study, example instantiations for the JHotDraw study and test cases delivered with the framework for the Struts study. However, we cannot claim that these examples are representative for all kinds of framework instantiations.

Third, the manual investigation to assess the quality of the change rules found – based on the investigation of the API, the release documentation, and the framework source code – is subject to evaluator bias. As a measure against this threat, we make all rules publicly available[7] to enable other researchers to assess the results independently.

[7] `http://www.st.informatik.tu-darmstadt.de/Frameworks`

6. RELATED WORK

Reverse engineering changes occurring during framework evolution has been subject of research for several years. The overall goal has been to make changes explicit in order to (a) ease migration of existing framework instantiations, or (b) understand the evolution of the framework. In the following, we classify related work in two categories.

The first category of related work targets the first goal: facilitating the migration of existing framework users. Early approaches in this category suggest to publish a new version of the framework together with a description of the changes required to adapt existing instantiations. For example, Chow and Notkin [6] introduce a change specification language and a tool that uses change specifications to transform existing framework instantiations. A similar approach is presented by Balaban et al. [3]. To ease the creation of such specifications, Henkel and Diwan [11] propose to record framework refactorings and to enable developers to repeat them on the instantiation code.

Other approaches [13, 17, 20, 23] propose different techniques for specifying, checking, and enforcing explicit specialization interfaces for frameworks. By imposing conformance to the specialization rules upon both instantiations and new versions of the framework, these approaches aim at mitigating the framework evolution problem.

The approaches discussed so far minimize the migration costs for instantiations: The effort is needed only once from the framework developer, while the migration of all instantiations can be performed automatically. The prerequisite for these approaches is that the framework developer uses some special tools and manual techniques in the process of evolving the framework. This requirement is not always feasible since it puts some extra burden on framework developers. Furthermore, for many legacy frameworks for which change descriptions are not available, means to reverse engineer such changes are needed.

Approaches in the second category analyze changes of a software to understand its evolution. They differ primarily in the technique used to find the changes. Malpohl et al. [19] find software elements that have been renamed in their lifecycle using a programming language-aware difference algorithm. Demeyer et al. [8] show that specific refactorings can be found by analyzing the differences of several software metrics. Van Rysselberghe and Demeyer [22] visualize the results of a code clone detection algorithm as dotplots and derive refactorings from visual patterns. Xing and Stroulia [27] present UMLDiff, an algorithm for detecting refactorings that analyzes design-level models of the software.

Further, several approaches enable tracing the evolution of classes, e.g., by detecting renamings, splittings or merges. Antoniol et al. [2] map all classes into a vector space and detect class-level refactorings by vector comparison. Zou and Godfrey [28] present a semi-automatic approach that uses origin analysis. The technique is also used by Kim et al. [16], who present a fully automated approach. Weißgerber and Diehl [25] use version history information to detect refactorings.

In following the goal to analyze changes of a software to understand its evolution, the approaches in the second category assume that changes happen abruptly. In case of frameworks, however, this assumption does not hold. To preserve backward compatibility, outdated code is not removed instantly, but marked deprecated. The coexistence of old and new program elements makes it difficult to find changes by using such algorithms.

RefactoringCrawler [9] addresses this issue for changes that are caused by refactorings. The tool first applies a fast syntactic analysis to detect refactoring candidates, i.e., code elements that are similar to each other. Next, a slower but more accurate semantic analysis is performed to refine the results. Kim et al. [15] presented an algorithm that derives minimal descriptions for higher-level changes by generalizing low-level refactorings. These approaches have in common, that they only use the framework code to detect changes. In contrast, we derive changes describing how to use a new framework version by analyzing the changed usage within ported instantiation code.

Recently, Dagenais and Robillard [7] presented a technique to recommend adaptive changes for clients of an evolving framework by analyzing how the framework adapts to its own changes. It is similar to our approach in such that framework usage changes are analyzed, change rules are created and ranked by their confidence. The main differences are the input data and the kind of changes found. First, in [7] multiple fine-grained change sets extracted from a source code repository are analyzed instead of two course-grained versions as in our approach. This enables to derive usage changes with less noise. Second, the focus of the work presented in [7] is on method call changes while our approach finds different kinds of framework usage changes[8].

7. SUMMARY AND FUTURE WORK

In this paper, we proposed an approach for finding usage changes in evolving frameworks by analyzing instantiation code. This is in contrast to current approaches, which only analyze the framework code. However, often instantiation code exists from which the usage of a new framework version can be learned more directly. The approach uses association rule mining to find patterns describing a changed usage of the framework. We evaluated our approach by mining usage changes of three real-world frameworks. The results show that our approach has a high precision and finds several usage changes that are not found by using refactoring tools.

An area of future work is to investigate in what extent it is worth combining the approach presented in this paper with tools for refactoring finding that analyze the framework code. Such a combination may lead to even better results for the following reasons. First, it would lead to a higher recall, since there are rules that can be derived from refactorings, which are not discovered by our approach and the other way around. Further, if a change rule is detected by both approaches, it is most likely a correct rule: the rules found by both approaches in our experiments were all correct rules. Second, our algorithm does not produce information about the kind of change made to the framework. For instance, the rule `V1:extends:A` → `V2:extends:B` only indicates, that subclasses of `A` in version 1 should subclass `B` in version 2; but, we have no information whether the framework class was renamed, moved, or exchanged by a different implementation. The technique used by other approaches (e.g., [9, 25]) could be used to answer such questions and to get more detailed change descriptions.

[8]According to the authors, it is possible to infer other kinds of framework usage changes, but not as may as supported by our approach.

8. REFERENCES

[1] R. Agrawal, T. Imielinski, and A. Swami. Mining association rules between sets of items in large databases. In *Proceedings of the SIGMOD International Conference on Management of Data*, pages 207–216. ACM Press, 1993.

[2] G. Antoniol, M. D. Penta, and E. Merlo. An automatic approach to identify class evolution discontinuities. In *Proceedings of the International Workshop on Principles of Software Evolution*, pages 31–40. IEEE Computer Society, 2004.

[3] I. Balaban, F. Tip, and R. M. Fuhrer. Refactoring support for class library migration. In *Proceedings of the Conference on Object-oriented Programming, Systems, Languages, and Applications*, pages 265–279. ACM Press, 2005.

[4] C. Borgelt. Software for frequent pattern mining. `http://www.borgelt.net/apriori.html`, 2007.

[5] M. Bruch, T. Schäfer, and M. Mezini. FrUiT: IDE support for framework understanding. In *Proceedings of the OOPSLA Workshop on Eclipse Technology Exchange*, pages 55–59. ACM Press, 2006.

[6] K. Chow and D. Notkin. Semi-automatic update of applications in response to library changes. In *Proceedings of the International Conference on Software Maintenance*, pages 359–368. IEEE Computer Society, 1996.

[7] B. Dagenais and M. P. Robillard. Recommending adaptive changes for framework evolution. In *Proceedings of the International Conference on Software Engineering*. ACM Press, 2008.

[8] S. Demeyer, S. Ducasse, and O. Nierstrasz. Finding refactorings via change metrics. In *Proceedings of the Conference on Object-oriented Programming, Systems, Languages, and Applications*, pages 166–177. ACM Press, 2000.

[9] D. Dig, C. Comertoglu, D. Marinov, and R. Johnson. Automated detection of refactorings in evolving components. In *Proceedings of the European Conference on Object-Oriented Programming*, pages 404–428. Springer, 2006.

[10] Eclipse. `http://www.eclipse.org`, 2007.

[11] J. Henkel and A. Diwan. Catchup!: Capturing and replaying refactorings to support api evolution. In *Proceedings of the International Conference on Software Engineering*, pages 274–283. ACM Press, 2005.

[12] JHotDraw. `http://www.jhotdraw.org/`, 2007.

[13] G. Kiczales and J. Lamping. Issues in the design and specification of class libraries. In *Proceedings of the Conference on Object-oriented Programming, Systems, Languages, and Applications*, pages 435–451. ACM Press, 1992.

[14] M. Kim and D. Notkin. Program element matching for multi-version program analyses. In *Proceedings of the International Workshop on Mining Software Repositories*, pages 58–64. ACM Press, 2006.

[15] M. Kim, D. Notkin, and D. Grossman. Automatic inference of structural changes for matching across program versions. In *Proceedings of the International Conference on Software Engineering*, pages 333–343. IEEE Computer Society, 2007.

[16] S. Kim, K. Pan, and J. E. James Whitehead. When functions change their names: Automatic detection of origin relationships. In *Proceedings of the Working Conference on Reverse Engineering*, pages 143–152. IEEE Computer Society, 2005.

[17] J. Lamping. Typing the specialization interface. In *Proceedings of the Conference on Object-oriented Programming, Systems, Languages, and Applications*, pages 201–214. ACM Press, 1993.

[18] Z. Li and Y. Zhou. PR-Miner: Automatically extracting implicit programming rules and detecting violations in large software code. In *Proceedings of the European Software Engineering Conference held jointly with ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 306–315. ACM Press, 2005.

[19] G. Malpohl, J. J. Hunt, and W. F. Tichy. Renaming detection. *Automated Software Engineering*, 10(2):183–202, 2003.

[20] M. Mezini. Maintaining the consistency of class libraries during their evolution. In *Proceedings of the Conference on Object-oriented Programming, Systems, Languages, and Applications*, pages 1–21. ACM Press, 1997.

[21] A. Michail. Data mining library reuse patterns using generalized association rules. In *Proceedings of the International Conference on Software Engineering*, pages 167–176. ACM Press, 2000.

[22] F. V. Rysselberghe and S. Demeyer. Reconstruction of successful software evolution using clone detection. In *Proceedings of the International Workshop on Principles of Software Evolution*, pages 126–130. IEEE Computer Society, 2003.

[23] P. Steyaert, C. Lucas, K. Mens, and T. D'Hondt. Reuse contracts: managing the evolution of reusable assets. In *Proceedings of the Conference on Object-oriented Programming, Systems, Languages, and Applications*, pages 268–285. ACM Press, 1996.

[24] Struts. `http://struts.apache.org/`, 2007.

[25] P. Weißgerber and S. Diehl. Identifying refactorings from source-code changes. In *Proceedings of the International Conference on Automated Software Engineering*, pages 231–240. IEEE Computer Society, 2006.

[26] Z. Xing and E. Stroulia. UMLDiff: An algorithm for object-oriented design differencing. In *Proceedings of the International Conference on Automated Software Engineering*, pages 54–65. ACM Press, 2005.

[27] Z. Xing and E. Stroulia. Refactoring detection based on UMLDiff change-facts queries. In *Proceedings of the Working Conference on Reverse Engineering*, pages 263–274. IEEE Computer Society, 2006.

[28] L. Zou and M. W. Godfrey. Using origin analysis to detect merging and splitting of source code entities. *IEEE Transactions on Software Engineering*, 31(2):166–181, 2005.