Annotation Refactoring:
Inferring Upgrade Transformations for Legacy Applications

Wesley Tansey Eli Tilevich

Department of Computer Science
Virginia Tech, Blacksburg, VA 24061, USA
{tansey,tilevich}@cs.vt.edu

Abstract

Since annotations were added to the Java language, many frameworks have moved to using annotated Plain Old Java Objects (POJOs) in their newest releases. Legacy applications are thus forced to undergo extensive restructuring in order to migrate from old framework versions to new versions based on annotations (*Version Lock-in*). Additionally, because annotations are embedded in the application code, changing between framework vendors may also entail large-scale manual changes (*Vendor Lock-in*).

This paper presents a novel refactoring approach that effectively solves these two problems. Our approach infers a concise set of semantics-preserving transformation rules from two versions of a single class. Unlike prior approaches that detect only simple structural refactorings, our algorithm can infer general composite refactorings and is more than 97% accurate on average. We demonstrate the effectiveness of our approach by automatically upgrading more than 80K lines of the unit testing code of four open-source Java applications to use the latest version of the popular JUnit testing framework.

Categories and Subject Descriptors D.2.3 [*Coding Tools and Techniques*]: Object-Oriented programming; D.2.6 [*Programming Environments*]: Integrated environments; D.2.7 [*Distribution, Maintenance, and Enhancement*]; D.3.3 [*Language Constructs and Features*]: Frameworks, Patterns

General Terms Languages, Experimentation

Keywords Refactoring, Upgrading, Frameworks, Metadata, Java, Annotations, Eclipse, JUnit

OOPSLA'08, October 19–23, 2008, Nashville, Tennessee, USA.

1. Introduction

By providing reusable designs and a predefined architecture, frameworks enable developers to streamline the software construction process and have consequently become a mainstay of object-oriented software development. One design decision that has to be made when creating a framework is how application and framework objects will interact with each other. Traditionally, frameworks have employed type and naming conventions to which the programmer must adhere. However, since metadata support has been added to modern object-oriented languages (e.g., Java Annotations and .NET Attributes), frameworks have increasingly moved to using language-supported metadata facilities. Being embedded within the source code next to the program elements they describe, annotations provide declarative information more robustly and concisely. As a result, many existing frameworks have switched from using type and naming conventions to using annotated Plain Old Java Objects (POJOs) in their latest releases.

Despite the benefits of annotation-based frameworks, several major drawbacks hinder their adoption and use. Legacy applications that were developed using older framework versions based on type and naming requirements must be upgraded to the annotation-based versions. This upgrade often requires hundreds or even thousands of tedious changes to source files scattered throughout the codebase. These changes are in fact refactorings, as they preserve the semantics of the application in the presence of a new framework.

While these refactorings are intuitively obvious to the programmer, automating them is not trivial. A text-based find-and-replace approach is not sufficient, as the required changes cannot be correctly detected with a regular expression search. Furthermore, existing inference algorithms cannot detect them automatically, and writing an automated refactoring tool by hand can be time-consuming and error-prone. These complications often force a manual refactoring in order to upgrade an application. The benefits of upgrading to the latest annotation-based version of a framework may therefore not be worth the programming effort re-

quired to apply these refactorings manually, resulting in an anti-pattern that we call *Version Lock-in*.

As noted in a recent article [31] describing metadata-based enterprise frameworks, annotations present an additional challenge. Due to the tight coupling between annotations and source code, switching between different framework vendors can be prohibitively expensive for large software projects. Such a high re-engineering cost results in the phenomenon known as the *Vendor Lock-In* anti-pattern [6].

This paper presents a novel refactoring approach that solves both the Version and Vendor Lock-In problems outlined above. Our approach has three phases: first, the framework developer creates representative examples of a class before and after transitioning; second, our algorithm infers generalized transformation rules from the given examples; finally, application developers use the inferred rules to parameterize our program transformation engine, which automatically refactors their legacy applications.

We validate our approach by inferring refactorings for transitioning between three different unit testing frameworks (JUnit 3 [3], JUnit 4, and TestNG [4]), as well as three different persistence frameworks (Java Serialization, Java Data Objects (JDO) [34], and Java Persistence API (JPA) [15]). With only five minor refinements to the inferred unit testing transformation rules, we automatically upgraded more than 80K lines of testing code in JHotDraw, JFreeChart, JBoss Drools, and Apache Ant from JUnit 3 to JUnit 4.

This paper makes the following novel contributions:

- *Annotation Refactoring*– a new class of refactorings that replaces the type and naming requirements of an old framework version or annotation requirements of a different framework with the annotation requirements of a target framework.
- An approach to removing the Version and Vendor Lock-in anti-patterns for annotation-based frameworks.
- A differencing algorithm that accurately infers general transformation rules from two versions of a single example.

The rest of this paper is structured as follows. Section 2 motivates our work by presenting a real-world example. Section 3 gives an overview of our approach. Section 4 describes our inference algorithm formally. Section 5 presents the results of the case studies we have conducted. Section 6 explains why existing approaches are insufficient. Section 7 outlines future work directions, and Section 8 summarizes our contributions.

2. Motivating Example

Whenever a widely-used framework undergoes a major version upgrade (i.e., changing the structural requirements for application classes), framework developers or other domain experts commonly release an upgrade guide. A typical presentation strategy followed by such guides is to show an example legacy class and its corresponding upgraded version. With respect to annotation refactorings, recent framework upgrades that have led to the creation of such guides include Enterprise JavaBeans (EJB) version 2 to 3 [27], Hibernate annotations to the Java Persistence API (JPA) [40], and JUnit version 3 to 4 [36].

JUnit 3

```
import junit.framework.*;

public class ATest extends TestCase
{

  //called before every test
  protected void setUp() {}


  //called after every test
  protected void tearDown() {}


  //tests Foo
  public void testFoo() {}


  //tests Bar
  public void testBar() {}
}
```

JUnit 4

```
import org.junit.*;

public class Atest
{
  //called before every test
  @Before
  public void setUp() {}

  //called after every test
  @After
  public void tearDown() {}

  //tests Foo
  @Test
  public void testFoo() {}

  //tests Bar
  @Test
  public void testBar() {}
}
```

Figure 1. Comparisons of a JUnit test case in version 3 and 4.

To illustrate the challenges associated with annotation refactorings, consider upgrading a JUnit application from version 3 to 4. Figures 1 and 2 show two example legacy classes and their corresponding upgraded versions, derived from the upgrade guide [36]. Application classes that use JUnit fall into two categories: test cases and test suites. A test case class contains a set of methods for testing a piece of application functionality. A test suite class groups related test cases, so that the framework can invoke them as a single unit. Applications using JUnit 3 must adhere to type and naming requirements to designate test cases and suites. However, JUnit 4 switched to using an annotated POJO paradigm to express the same functionality. Table 1 summarizes the differences between the application requirements of the two versions.

While the refactorings required to upgrade a JUnit application may be straightforward to the programmer, no existing refactoring tool can make these changes automatically.

JUnit 3

```
//gets a collection of two
//test cases to run
public class AllTests
{
  public static Test suite() {
    TestSuite suite =
      new TestSuite(ATest.class);
    suite.addTestSuite(BTest.class);
    return suite;
  }
}
```

JUnit 4

```
@RunWith(Suite.class)
@SuiteClasses({ATest.class,
               BTest.class})
public class AllTests
{
  //no suite method required
}
```

Figure 2. Comparison of a JUnit test suite in version 3 and 4.

Application Element	JUnit 3	JUnit 4
Test case class	Extend `TestCase`	None (POJO)
Initialization method	Override `setUp`	`@Before`
Destruction method	Override `tearDown`	`@After`
Test method	Name starts with "test"	`@Test`
Test suite class	Provides `suite` method	`@RunWith` `@SuiteClasses`

Table 1. A summary of the requirements for application elements using JUnit versions 3 and 4.

Therefore, if the framework developer wishes to provide automated upgrade support, she often has no choice but to create a refactoring tool by hand. Such a task may require a significant investment by the framework developer, as she must first gain proficiency in a refactoring library API (e.g., the Java Development Toolkit (JDT) [20]) or a domain-specific language (e.g., Spoon [28]). After becoming familiar with a library or a DSL, she must then use it to write the refactoring tool from scratch. As a result, framework developers typically opt to provide backwards compatibility support rather than automated upgrade tools. Although providing backwards compatibility allows legacy applications to use a newer version of a framework, it does not allow them to take advantage of newly-introduced framework features.

Thus, an approach to automatically generating refactoring tools capable of upgrading legacy applications has great potential benefit. Next we provide an overview of our approach and show how it can be used to automatically generate a refactoring tool for upgrading JUnit applications.

3. Approach Overview

Our approach is based on the wide availability of upgrade guides containing examples of legacy classes and their upgraded versions. If the human developer is expected to infer general rules for upgrading their legacy applications using these guides, then the examples are likely to contain a level of detail that one could leverage to automate the process. Our approach automatically extracts this knowledge, creating specialized refactoring tools that can upgrade legacy applications. Our Eclipse Plug-in called Rosemari (**R**ule-**O**riented **S**oftware **E**nhancement and **M**aintenance through **A**utomated **R**efactoring and **I**nferencing) concretely implements our approach.

Creating an annotation refactoring tool involves three steps. First, the framework developer selects two versions of a class, one before and one after upgrading, that we call *representative examples*. The developer then picks a predefined specialization of our inference algorithm, called an *upgrade pattern*. Finally, the generated rules can be downloaded by application developers and subsequently used to parameterize our transformation engine, which will then automatically refactor their legacy applications. Next we detail the main steps of our approach, using the previously presented JUnit example for demonstration.

3.1 Representative Examples

A representative example is a class or interface, such as those commonly included in framework tutorials, that uses framework features that differ between versions or vendors (e.g., Figures 1 and 2 can serve as representative examples for upgrading JUnit). A representative example using an older framework is called a *prior example*. A representative example using a newer framework is called a *posterior example*. The framework developer provides a prior example and its corresponding posterior example, which may differ in the following ways:

1. Super type changes
2. Method signature changes
3. Field type changes
4. Annotations added or removed
5. Annotation argument added or removed
6. Imports added or removed
7. Statement added or removed

The prior and posterior examples are compared at different levels of granularity in the encapsulation hierarchy, which we call *levels* for short. Specifically, examples are compared at class, method, and statement levels. Differences between levels are used to detect the restructurings required to upgrade the prior example into the posterior example. However, detecting restructurings is only one facet of creating a refactoring. The issue of when to apply a restructuring to a level is not clear without additional knowledge about how the application is being upgraded.

```
public class D extends C
{

    public void foo() {
        System.out.println();
    }
}
```

```
public class D extends C
{
    @Ann
    public void foo() {
        System.out.println();
    }
}
```

Figure 5. An example showing the ambiguity inherent in inferring refactorings between two arbitrary representative examples.

For instance, consider the simple pair of prior and posterior examples in Figure 5. While it is obvious that there is a method-level difference that requires adding the `@Ann` annotation to `foo`, the refactoring rule to be inferred is unclear. One possible rule may be that a method named "foo" in any subclass of `C` should be annotated with `@Ann`. However, another rule could be that any method which calls

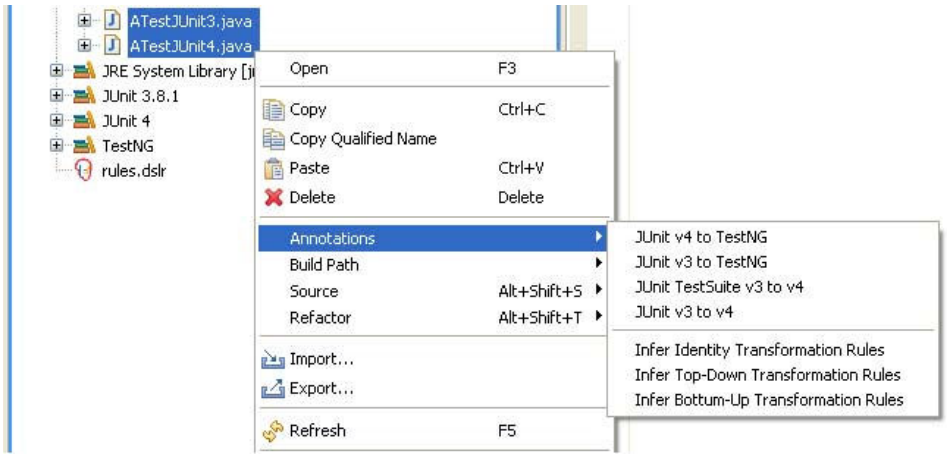

Figure 3. The Rosemari context menu. The top set of menu items are available refactorings; the bottom set of menu items are available inference patterns.

`System.out.println` should be annotated with `@Ann`. To disambiguate when a restructuring should be applied in a refactoring, we follow a pattern-based approach described next.

3.2 Upgrade Patterns

When upgrading from a type and naming convention-based framework version to a newer annotation-based version, or when switching between different annotation-based frameworks, refactoring rules typically follow common patterns, which we call *upgrade patterns*. One reason why these patterns occur is that evolving a framework to use annotations is normally driven by the desire to improve the software engineering quality of the framework (e.g., looser coupling between application and framework classes). Another reason is that switching between different annotation-based frameworks often requires using a different vocabulary to describe essentially the same functionality.

Therefore, inferring the refactorings between two representative examples depends on the upgrade pattern followed. Next we describe three such common patterns we have identified from our experiences. While we have found that these patterns successfully capture the refactorings for many upgrade scenarios, other patterns can be plugged into our system as needed.

3.2.1 Bottom-Up

This upgrade pattern applies restructurings on the basis of the level itself and its enclosing levels. For example, refactoring the methods in a JUnit 3 test case follows a bottom-up pattern. Specifically, the `test`, `setUp` and `tearDown` method restructurings are only relevant if their enclosing class is a `TestCase`.

3.2.2 Top-Down

In contrast to the bottom-up pattern, a top-down pattern applies restructurings on the basis of the level itself and its contained levels. For example, refactoring test suite classes in JUnit 3 follows a top-down pattern. Specifically, the class is annotated with the `@RunWith(Suite.class)` annotation on the basis of containing a `suite` method.

3.2.3 Identity

This upgrade pattern applies restructurings only on the basis of the level itself, assuming that all annotations in the first representative example have a one-to-one mapping in the second representative example. For instance, the TestNG framework uses `@BeforeMethod`, which is identical in its functionality to the `@Before` annotation in JUnit 4.

The developer can choose an appropriate upgrade pattern by observing the purpose of a given annotation. If adding an annotation removes tight coupling between the *enclosing* elements of a level (e.g., a class enclosing a method) and the framework, then it is likely a bottom-up pattern. If, however, the annotation describes the *contained* elements of a level (e.g., a method contained in a class), then it is likely a top-down pattern. If the annotation simply changes how the level is expressed, then it is likely an identity pattern.

3.3 Transformation Rules

Inferred refactorings are represented as a collection of first-order *when-then* transformation rules,[1] expressed using a Domain Specific Language (DSL) we developed. A transformation rule is composed of restructurings and constraints on their application. The *when* portion of a rule defines the constraints under which to apply the restructurings defined in the *then* portion of the rule.

While the generated rules possess a high degree of precision, some rules may require manual refinement. Since the developer is not expected to write the transformation rules from scratch but rather fine-tune the generated rules, we

[1] We use the JBoss Drools engine [30].

```
rule "#1) Transform classes matching ATest"
  no-loop
  salience 10
    when
      $class : Application Class
        - Visibility is public
        - Superclass is "junit.framework.TestCase"
    then
      Remove superclass from $class
      Update $class
end
rule "#3) Transform all methods matching tearDown"
  no-loop
  salience 20
  when
    $class : Application Class
      - Visibility is public
      - Superclass is "junit.framework.TestCase"
    $method : Application Method
      - Name matches "tearDown"
      - Declaring class is $class
      - Not annotated with "After"
      - Visibility is protected
      - Scope level is member
      - Return type is "void"
      - Has 0 parameters
  then
    Add annotation "After" to $method
    Set access level of $method to public
    Update $class
    Update $method
end
rule "#5) Manage imports"
  no-loop
  salience 0
  when
    $file : Application File
      - Does not import "org.junit.After"
      - Does not import "org.junit.Before"
      - Does not import "org.junit.Test"
      - Does not import "org.junit.Assert.*"
      - Does import "junit.framework.TestCase"
  then
    Add import "org.junit.After" to $file
    Add import "org.junit.Before" to $file
    Add import "org.junit.Test" to $file
    Add static import "org.junit.Assert.*" to $file
    Remove import "junit.framework.TestCase" from $file
    Update $file
end
```

```
rule "#2) Transform all methods matching setUp"
  no-loop
  salience 20
  when
    $class : Application Class
      - Visibility is public
      - Superclass is "junit.framework.TestCase"
    $method : Application Method
      - Name matches "setUp"
      - Declaring class is $class
      - Not annotated with "Before"
      - Visibility is protected
      - Scope level is member
      - Return type is "void"
      - Has 0 parameters
  then
    Add annotation "Before" to $method
    Set access level of $method to public
    Update $class
    Update $method
end
rule "#4) Transform all methods matching test.*"
  no-loop
  salience 20
  when
    $class : Application Class
      - Visibility is public
      - Superclass is "junit.framework.TestCase"
    $method : Application Method
      - Name matches "test.*"
      - Declaring class is $class
      - Not annotated with "Test"
      - Visibility is public
      - Scope level is member
      - Return type is "void"
      - Has 0 parameters
  then
    Add annotation "Test" to $method
    Update $class
    Update $method
end
```

Figure 4. The generated transformation rules to refactor a JUnit 3 test case class to use JUnit 4.

chose to trade conciseness for readability and ease of understanding in designing our DSL. Having a collection of generic natural-language statements that can be parameterized with concrete values, the developer can easily refine complex rules to better express the desired refactorings.

Given the JUnit test case representative examples in Figure 1 and the Bottom-Up upgrade pattern, Rosemari generates five transformation rules, shown in Figure 4. Rules 2-4 originally require that the target class directly extends `TestCase`. However, these transformations are valid for any class that extends `TestCase`, directly or indirectly. Therefore, the developer can change this constraint in each of the three rules by refining it to its more general version: *Superclass is a "junit.framework.TestCase"*. Additionally, the *Visibility is protected* constraint in rules 2 and 3 can easily be generalized by changing the constraint to *Visibility is at least protected*.

The Rosemari plug-in provides a searchable collection of supported natural language statements that the developer can reference when refining transformation rules. Once refined and named appropriately (e.g., *JUnit v3 to v4*), the refactoring can be applied by selecting a JUnit 3 test case source file and choosing the refactoring from the context menu, as shown in Figure 3. The source file will then be automatically upgraded to JUnit 4.

Next we present our algorithm for inferring annotation refactoring rules.

4. Inference Algorithm

Our algorithm accepts two representative examples and infers a generalized set of transformation rules that com-

EJB 2

```
public class HelloWorldBean
             implements SessionBean
{
  public void ejbActivate() {}


  public void ejbRemove() {}
}
```

EJB 3

```
@Stateful
public class HelloWorldBean
{
  @PostActivate
  public void ejbActivate() {}

  @Remove
  public void ejbRemove() {}
}
```

Figure 6. A simple Enterprise Java Bean example in EJB 2 and 3.

pose a refactoring. We present our algorithm as a collection of seven smaller set manipulation algorithms configurable through a user-defined partial order on the levels of a representative example. To further illustrate how our algorithm works, we show how each component algorithm contributes to learning the transformation rules for a simple Enterprise JavaBeans (EJB) upgrade scenario. Figure 6 shows the two EJB representative examples, derived from the Oracle guide [27] on migrating applications from EJB 2 to 3.[2]

4.1 Decomposing Representative Examples

In order for our algorithm to calculate transformation rules, each representative example must first be decomposed into a set of levels, $P = \{L_1, L_2, \ldots, L_n\}$. A *level* is a set, $L = \{e_1, e_2, \ldots, e_m\}$, of signature elements (i.e., tokens in a program element's signature) at a particular point in the encapsulation hierarchy. For example, the method level contains a set of annotations, a visibility identifier, a scope identifier, a return type, a set of parameters, and a set of exceptions. Figure 7 shows a full list of the supported levels and their definitions.

As discussed in Section 3.2, inferring the correct transformation rules given a pair of representative examples is not possible without additional information about the refactoring. The Rosemari plug-in implementing our algorithm uses the concept of upgrade patterns to simplify this process. However, upgrade patterns are merely wrappers for specifying a partial order over the set of levels in a representative example. Figure 8 shows the partial orders corresponding to each of the three previously discussed upgrade patterns. Figure 9 shows the level sets for the EJB example, which matches to a Bottom-Up pattern.

4.2 Component Inference Algorithms

The foundation of our algorithm is to construct sets of independent components and then merge them hierarchically. This section presents the five algorithms that compose these independent components.

[2] Fully upgrading EJB applications requires static analysis of both Java source code and XML metadata files and is thus outside the scope of this paper. The rules learned in this example are therefore only a subset of the total required to refactor an entire EJB application.

4.2.1 LevelRestructurings

Our algorithm starts by first calculating the restructurings for each level. A *restructuring* is a simple program transformation function that adds, replaces, or removes a signature element in a level. We call a restructuring a *positive restructuring* if it adds or replaces an element. A restructuring is a *negative restructuring* if it removes an element. Figure 10 presents the LevelRestructurings algorithm for discovering a level's restructurings.

The Prune method (line 6) removes unnecessary elements from $NEGELS$. This is an implementation addition to reduce the number of negative restructurings generated, making the final transformation rules more concise and readable. For instance, changing the visibility of a level is done by replacing the current visibility with the new visibility (a positive restructuring) rather than first removing the current visibility (a negative restructuring) and then adding the new visibility (a positive restructuring).

For the EJB example, LevelRestructurings is called three times (i.e., once for each level). For the *HelloWorldBean* level, $POSELS_{HelloWorldBean} = \{A^{Stateful}\}$, and thus a positive restructuring of *Add annotation "Stateful"* is generated. Similar positive restructurings are generated for the *ejbActivate* and *ejbRemove* levels. However, only the *HelloWorldBean* level produces a non-null value for removed elements, as $NEGELS_{HelloWorldBean} = \{I^{SessionBean}\}$, resulting in a negative restructuring of *Remove interface "SessionBean."*

Lemma 1. Let $L = \{e_1, e_2, ..., e_m\}$ be a prior level, and let $L' = \{e_1, e_2, ..., e_n\}$ be a posterior level, such that L' is the restructured version of L. $LevelRestructurings(L, L')$ returns a set of restructurings, $R = \{r_1, r_2, ..., r_k\}$ representing exactly every required positive and negative restructuring to transform L to L'.

Proof. If there is an element, $e_u \in L'$ such that $e_u \notin L$, then a positive restructuring $r_i^+ = PositiveRestructuring(e_u)$ is required to transform L to L'. Thus, $POSELS = L' - L$ (line 2) will contain e_u, and adding a positive restructuring to R for every element in $POSELS$ (lines 3-4) guarantees that r_i^+ will be in R. If a positive restructuring, $r_j^+ = PositiveRestructuring(e_v)$ is not required to transform L to L', then either $e_v \in L$ or $e_v \notin L'$. If $e_v \in L$, then $L' - L$ will remove e_v and it will not be added to $POSELS$; likewise, if $e_v \notin L'$ then $L' - L$ and consequently $POSELS$ will not contain it. If $POSELS$ does not contain e_v, then r_j^+ will not be created in line 4 and $r_j^+ \notin R$. Therefore, R contains every positive restructuring and no more. An analogous and opposite argument applies to negative restructurings. □

4.2.2 LevelConstraints

Our algorithm next computes a set of constraints defining when the discovered restructurings should be applied to a level. We call a constraint a *positive constraint* when it re-

$$
\begin{array}{ll}
L_{Class} & = \{Annotations, Visibility, SuperClass, SuperInterfaces\} \\
L_{Iface} & = \{Annotations, Visibility, SuperInterfaces\} \\
L_{Field} & = \{Annotations, Visibility, Scope, Type\} \\
L_{Meth} & = \{Annotations, Visibility, Scope, \\
 & \quad ReturnType, Parameters, Exceptions\} \\
L_{Cons} & = \{Annotations, Visibility, Scope, Parameters, Exceptions\} \\
L_{MethInvoke} & = \{Name, Arguments\} \\
L_{ConsInvoke} & = \{Type, Arguments\} \\
L_{Arg} & = \{Expression\}
\end{array}
$$

Figure 7. Levels of a representative example.

Bottom-Up
$L_{Arg} <_L \{L_{ConsInvoke}, L_{MethInvoke}\} <_L \{L_{Cons}, L_{Meth}, L_{Field}\} <_L \{L_{Iface}, L_{Class}\}$

Top-Down
$L_{Arg} <_L \{L_{ConsInvoke}, L_{MethInvoke}\} <_L \{L_{Iface}, L_{Class}\} <_L \{L_{Cons}, L_{Meth}, L_{Field}\}$

Identity
$\{L_{Iface}, L_{Class}, L_{Cons}, L_{Meth}, L_{Field}, L_{ConsInvoke}, L_{MethInvoke}, L_{Arg}\}$

Figure 8. Upgrade patterns as partial orders on P.

$$
\begin{array}{l}
EJB2_{HelloWorldBean} = \{V^{public}, I^{SessionBean}\} \\
EJB3_{HelloWorldBean} = \{A^{Stateful}, V^{public}\} \\
\\
EJB2_{ejbActivate} = \{V^{public}, S^{member}, R^{void}\} \\
EJB3_{ejbActivate} = \{A^{PostActivate}, V^{public}, S^{member}, R^{void}\} \\
\\
EJB2_{ejbRemove} = \{V^{public}, S^{member}, R^{void}\} \\
EJB3_{ejbRemove} = \{A^{Remove}, V^{public}, S^{member}, R^{void}\}
\end{array}
$$

Figure 9. The level decomposition of the EJB example.

quires that an element be present. A constraint is a *negative constraint* when it requires that an element not be present. Negative constraints are necessary to ensure the generated transformation rules are not applied unnecessarily (e.g., to levels that have already been transformed by hand or a previous upgrading session). Figure 11 presents the LevelConstraints algorithm for discovering a level's constraints.

For the *HelloWorldBean* level, lines 2-3 create two positive constraints, *Visibility is public* and *Directly implements SessionBean*; lines 5-6 then add one negative constraint, *Not annotated with "Stateful"*. For both method levels, positive constraints requiring public visibility, member scope (i.e., non-static scope), and a return type of void are added along with negative constraints requiring *ejbActivate* and *ejbRemove* to not be annotated with *"PostActivate"* and *"Remove"*, respectively. Our implementation also handles special cases such as when a method does not contain any parameters, as with both EJB method levels which receive a negative constraint requiring that there are no parameters in the method signature.

Lemma 2. Let $L = \{e_1, e_2, ..., e_m\}$ be a prior level, and let $L' = \{e_1, e_2, ..., e_n\}$ be a posterior level, such that L' is the restructured version of L. $LevelConstraints(L, L')$ returns a set of constraints, $S = \{s_1, s_2, ..., s_k\}$ representing exactly every required positive and negative constraint to match L.

Proof. If there is an element, $e_u \in L$, it must have an associated positive constraint, $s_i^+ = PositiveConstraint(e_u) \in S$. All elements in L are iterated over and their corresponding positive constraints are added to S in lines 2-3. If a positive constraint $s_j^+ = PositiveConstraint(e_v)$ is not required to match L, then $e_v \notin L$ and s_j^+ is never added to S. Thus, S contains exactly every positive constraint. If and only if there is an element $e_w \in L'$ such that $e_w \notin L$, it must have an associated negative con-

ALGORITHM: Level Restructurings
INPUT: Two levels $L = \{e_1, e_2, \ldots, e_m\}, L' = \{e_1, e_2, \ldots, e_n\}$ where L' is the restructured version of L.
OUTPUT: A set $R = \{r_1, r_2, \ldots, r_p\}$ of restructurings.

1. $R \leftarrow \emptyset$
2. $POSELS \leftarrow L' - L$
3. For $i \leftarrow 1$ *to* $|POSELS|$
4. $\quad R \leftarrow R \bigcup$ PositiveRestructuring$(POSELS[i])$
5. $NEGELS \leftarrow L - L'$
6. Prune$(NEGELS)$ // remove unnecessary elements
7. For $j \leftarrow 1$ *to* $|NEGELS|$
8. $\quad R \leftarrow R \bigcup$ NegativeRestructuring$(NEGELS[j])$
9. return R

Figure 10. The $LevelRestructurings$ algorithm for calculating the required set of restructurings to transform between two levels.

ALGORITHM: Level Constraints
INPUT: Two sets $L = \{e_1, e_2, \ldots, e_m\}, L' = \{e_1, e_2, \ldots, e_n\}$ of signature elements where L' is the restructured version of L.
OUTPUT: A set $S = \{s_1, s_2, \ldots, s_q\}$ of context-free constraints.

1. $S \leftarrow \emptyset$
2. For $i \leftarrow 1$ *to* $|L|$
3. $\quad S \leftarrow S \bigcup$ PositiveConstraint$(L[i])$
4. $NEGELS \leftarrow L' - L$
5. For $j \leftarrow 1$ *to* $|NEGELS|$
6. $\quad S \leftarrow S \bigcup$ NegativeConstraint$(NEGELS[j])$
7. return S

Figure 11. The $LevelConstraints$ algorithm for calculating the context-free set of constraints for a level.

straint, $s_k^+ = PositiveConstraint(e_w) \in S$, since according to lemma 1 it will generate a positive restructuring in $LevelRestructurings$. $NEGELS$ will therefore contain e_w (line 4) and add a negative constraint, s_k^+, for exactly all elements satisfying the aforementioned constraint (lines 5-6). □

4.2.3 NamingConvention

For method and field levels, our algorithm adds a name matching constraint intended to capture naming conventions. Figure 12 shows the $NamingConvention$ algorithm which takes a set, $N = \{n_1, n_2, \ldots, n_m\}$, of fields or methods with identical signatures in both the prior and posterior examples and returns a generalized regular expression matching all the names in N. The algorithm first tokenizes a name based on the Java naming convention of uppercase delimiters, then calculates the tokens which match identically. If no naming convention is found, a literal expression matching only N is returned. As this is the case for both EJB method levels, each has an exact naming constraint added.

ALGORITHM: Naming Convention
INPUT: A set $N = \{n_1, n_2, \ldots, n_m\}$ of member names
OUTPUT: A regular expression naming convention

1. $TOKENS \leftarrow$Tokenize(n_1)
2. For $i \leftarrow 2$ *to* m
3. $\quad TOKENS \leftarrow TOKENS \bigcap$ Tokenize(n_i)
4. $\quad$ If $TOKENS = \emptyset$
5. $\quad\quad$ return LiteralRegex(N)
6. return GeneralizedRegex$(TOKENS)$

Figure 12. The $NamingConvention$ algorithm for generalizing a set of names to a naming convention.

4.2.4 ContextConstraints

Although a level is context-free, program transformations are often not context-free operations but are rather based on some framework-dependent notion of context. To accommodate these scenarios, we introduce the notion of *context constraints*. A context constraint, q for level L_1 is a level constraint, $s \in L_2$, such that $L_1 <_L L_2$ where $<_L$ is a user-defined partial order on the set of levels, P. Figure 13 shows the ContextConstraints algorithm for discovering all context constraints for a set of prior levels. It should be noted that the algorithm present is only a semantically-equivalent version of the implementation, as in practice caching data structures can be used to eliminate the internal loop (lines 4-6).

Since our EJB example follows a Bottom-Up pattern, all method levels are defined as dependent on their enclosing class level. Thus, ContextConstraints adds the additional requirements that annotating a method named *ejbActivate* or *ejbRemove* with `@PostActivate` or `@Remove`, respectively, is only correct if the enclosing class has public visibility and implements `SessionBean`.

ALGORITHM: Context Constraints
INPUT: A set $P = \{L_1, L_2, \ldots, L_n\}$ of levels.
A set $S_{all} = \{S_1, S_2, \ldots, S_n\}$ of sets of constraints such that $\forall S_i \in S_{all}, \forall L_i \in P$, S_i is the set of level constraints for L_i.
A partial order $<_L$ on P.
OUTPUT: A set $Q_{all} = \{Q_1, Q_2, \ldots, Q_n\}$ of context constraints such that $\forall Q_i \in Q_{all}, Q_i$ is the set of constraints defining the context of L_i.

1. $Q_{all} \leftarrow \emptyset$
2. For $i \leftarrow 1$ *to* n
3. $\quad Q_i \leftarrow \emptyset$
4. $\quad$ For $j \leftarrow 1$ *to* n
5. $\quad\quad$ If $P[i] <_L P[j]$
6. $\quad\quad\quad Q_i \leftarrow Qi \bigcup S_{all}[j]$
7. $\quad Q_{all} \leftarrow Q_{all} \bigcup \{Q_i\}$
8. return Q_{all}

Figure 13. The $ContextConstraints$ algorithm for calculating the additional constraints for every level.

Lemma 3. Let $P = \{L_1, L_2, ..., L_n\}$ be a set of prior levels, let $<_L$ be a partial order on P, and let $S_{all} = \{S_1, S_2, ..., S_n\}$ be a set of sets of constraints, such that $\forall S_i \in S_{all}, \forall L_i \in P$, S_i is the set of context-free constraints for L_i. $ContextConstraints(P, S_{all}, <_L)$ returns an ordered set of sets of context constraints, $Q_{all} = \{Q_1, Q_2, ..., Q_n\}$ such that $\forall Q_i \in Q_{all}$, Q_i is exactly the set of contexts constraints for L_i.

Proof. It is necessary to first show that $\forall Q_i \in Q_{all}$, Q_i is *a* set of context constraints for L_i, then we show that Q_i is the *exact* set of context constraints as specified by $<_L$. $ContextConstraints$ creates a new set of context constraints for every level in P (line 3) and adds the set to Q_{all} (line 7); thus, $|Q_{all}| = |P|$, and this completes the first half of the proof. For every Q_i, $ContextConstraints$ iterates over all elements in P and adds a set of level constraints to Q_i if and only if their corresponding level satisfies $<_L$; thus, Q_i is exactly the set of context constraints for L_i, and this completes the second half of the proof. □

4.2.5 SalienceValues

Context-dependent transformations hinge on the assumption that the context of the level they are transforming will not be invalidated before the level has been transformed. This assumption could not be guaranteed if the rules were executed arbitrarily. For instance, the method-level transformations for the EJB example require that the target method is defined in a class implementing `SessionBean`. If the class-level transformations are arbitrarily executed before the method-level transformations, the interface will be removed and the method-level transformations will fail. To overcome this limitation, our algorithm calculates an execution order for each level, called a salience value. Rules with higher salience values will be executed prior to rules with lower salience values (ties are broken arbitrarily). Figure 14 shows the $SalienceValues$ algorithm for calculating salience values for a set of prior levels. The algorithm is analogous to a sorting algorithm parameterized by the partial order on L.

ALGORITHM: Salience Values
INPUT: A set $P = \{L_1, L_2, \ldots, L_n\}$ of levels
A partial order $<_L$ on L.
OUTPUT: A set $Y_{all} = \{y_1, y_2, \ldots, y_n\}$ of salience values, $\forall\ y_i \in Y_{all}$, y_i is the salience of L_i.

1. $Y_{all} \leftarrow \emptyset$
2. For $u \leftarrow 1$ *to* n
3. $\quad y_u \leftarrow 0$
4. $\quad$ For $v \leftarrow 1$ *to* n
5. $\quad\quad$ If $P[v]\ <_L\ P[u]$
6. $\quad\quad\quad y_u \leftarrow y_u + 10$
7. $\quad Y_{all} \leftarrow Y_{all} \bigcup \{y_u\}$
8. return Y_{all}

Figure 14. The $SalienceValues$ algorithm for calculating the order of execution for every level.

4.3 Merging Algorithms

Once the necessary independent components have been inferred, two merging algorithms are invoked. The first merging algorithm, $LevelTransformations$, combines the independent components for a level into a set of transformations for that level. The second merging algorithm, $ProgramTransformations$, combines the level transformations into a complete set of program transformations.

4.3.1 LevelTransformations

The $LevelTransformations$ algorithm takes the results of the algorithms described in the previous section for a single level, and combines them into a set of generalized transformation rules. Figure 15 shows the $LevelTransformations$ algorithm. Until now, we have focused on structural inference at or above the level of method headers. We have deliberately not detailed how we infer annotation attribute values, which can be any valid Java expression and as such require a deeper level of inference than that of headers. Thus, inference of attribute values is delayed until the $LevelTransformations$ algorithm (lines 4-10), when more complete program information is known. For every argument to the attribute, $LevelTransformations$ checks if a matching expression exists in the prior representative example (Lines 6-7). If it does not, the algorithm generates a literal transformation which adds the expression to the attribute (line 8). If the expression does exist, however, it generates a generalized transformation based on the context of the expression. In both cases, the salience of attribute transformations must be slightly lower than that of the rest of the level transformations, since the annotation itself must be added first before an attribute can be added.

4.3.2 ProgramTransformations

The final algorithm, $ProgramTransformations$, takes two decomposed representative examples, P and P', as well as a set of expressions, X, in the prior representative example, and returns the set, T_{all}, of inferred program transformations. First, $ProgramTransformations$ builds the sets of fundamental components, S_{all}, R_{all}, Q_{all}, and Y_{all}, for every level (lines 1-11). Then, it builds the set of context constraints (if any) for every expression (lines 14-16). Finally, $ProgramTransformations$ iterates over all levels and adds each set of level transformations to T_{all} (lines 17-19). Figure 17 shows the rules output by the algorithm to transform the EJB example used through this section.

ALGORITHM: Program Transformations
INPUT: A set $P = \{L_1, L_2, \ldots, L_n\}$ of prior levels.
A set $P' = \{L'_1, L'_2, \ldots, L'_n\}$) of posterior levels.
A set $X = \{x_1, x_2, \ldots, x_m\}$ of expression nodes in an AST.
A partial order $<_L$ on L.
OUTPUT: A set $T_{all} = \{T_1, T_2, \ldots, T_q\}$ of transformations for this program.

1. $S_{all} \leftarrow \emptyset$
2. $R_{all} \leftarrow \emptyset$
3. For $i \leftarrow 1$ *to* n
4. $\quad L_1 \leftarrow P[i]$
5. $\quad L'_1 \leftarrow P'[i]$
6. $\quad S_i \leftarrow$ LevelConstraints(L_i, L'_i)
7. $\quad R_i \leftarrow$ LevelRestructurings(L_i, L'_i)
8. $\quad S_{all} \leftarrow S_{all} \bigcup \{S_i\}$
9. $\quad R_{all} \leftarrow R_{all} \bigcup \{R_i\}$
10. $Q_{all} \leftarrow$ ContextConstraints$(S_{all}, <_L)$
11. $Y_{all} \leftarrow$ SalienceValues$(P[0])$
12. $T_{all} \leftarrow \emptyset$
13. $Q_X \leftarrow \emptyset$
14. For $i \leftarrow 1$ *to* m
15. $\quad v \leftarrow V[i]$
16. $\quad Q_X \leftarrow Q_X \bigcup \{Q_{all}.v\}$
17. For $i \leftarrow 1$ *to* n
18. $\quad T_i \leftarrow$ LevelTransformations$(S_{all}[i], Q_{all}[i], R_{all}[i], X, Q_X)$
19. $\quad T_{all} \leftarrow T_{all} \bigcup T_i$
20. return T_{all}

Figure 16. The $ProgramTransformations$ algorithm for inferring a set of transformation rules from two representative examples.

5. Evaluation

We evaluated our approach on two criteria. First, we measured the accuracy of our inference algorithm by applying it to seven different refactoring scenarios. Second, we demonstrated the effectiveness of the automatically inferred refactorings by upgrading the testing portion of four well-known, open-source projects. The results of our evaluation show that our approach produces highly-accurate refactorings which, with few minor refinements, can be automatically-applied to large-scale applications, effectively solving the Vendor and Version Lock-in anti-patterns for applications that use annotation-based frameworks.

5.1 Inferred Refactorings

For our approach to be viable in a realistic setting, it has to infer refactorings with a high degree of accuracy. To assess the accuracy of our inferencing algorithm, we manu-

ALGORITHM: Level Transformations
INPUT: A set $S = \{s_1, s_2, \ldots, s_k\}$ of context-free constraints.
A set $Q = \{q_1, q_2, \ldots, q_m\}$ of context constraints.
A set $R = \{r_1, r_2, \ldots, r_n\}$ of restructurings.
A set $X = \{x_1, x_2, \ldots, x_u\}$ of expression nodes.
A set $Q_X = \{Q_1, Q_2, \ldots, Q_u\}$ of sets of context constraints for the expression nodes.
A salience score Y for this level.
OUTPUT: A set $T_{level} = \{t_1, t_2, \ldots, t_p\}$ of transformations for this level.

1. $T_{level} \leftarrow \emptyset$
2. For $i \leftarrow 1$ *to* n
3. $\quad r_i \leftarrow R[i]$
4. $\quad$ If $TypeOf(r_1) =$ AttributeAddition and $|r_i.args| > 0$
5. $\quad\quad$ For $j \leftarrow 1$ *to* $|r_i.args|$
6. $\quad\quad\quad X_a \leftarrow X \bigcap r_i.args[j]$
7. $\quad\quad\quad$ If $X_a = \emptyset$
8. $\quad\quad\quad\quad T_{level} \leftarrow T_{level} \bigcup \{$LiteralTransformation$(r_i.args[j], Y - 1)\}$
9. $\quad\quad\quad$ Else
10. $\quad\quad\quad\quad T_{level} \leftarrow T_{level} \bigcup \{$GeneralTransformation$(r_i.args[j], Q_a, Y - 1)\}$
11. $\quad T_{level} \leftarrow T_{level} \bigcup \{$GeneralTransformation$(r_i, S \bigcup Q, Y)\}$
12. return T_{level}

Figure 15. The $LevelTransformations$ algorithm for generating a set of transformation rules for a level.

ally evaluated each inferred refactoring in four categories: constraints, restructurings, rule execution order, and manual refinements required. The metrics used to evaluate each category are defined below. To help explain the metrics, we give examples from the transformation rules listed in Figure 4.

5.1.1 Constraint Metrics

For constraints, we measured the number of correct, excessive, erroneous, and missing constraints, defined as follows:

- *Correct.* A correct constraint accurately captures a single requirement of a transformation rule. For example, *Name matches "test.*"* correctly requires the name of any test method to start with the `test` prefix.
- *Excessive.* An excessive constraint unnecessarily limits the scope of a transformation rule. For example, *Visibility is protected* limits the applicability of rules 2 and 3 to only `protected` methods, even though `public` methods should be captured as well.
- *Erroneous.* An erroneous constraint captures requirements that are incorrect for a specific transformation rule. For example, *Has 1 parameter* would be erroneous for identifying a `setUp` method in rule 2.
- *Missing.* A missing constraint is a necessary requirement not present in a transformation rule. For example, if *Return type is "void"* were not present in rule 2, it would be a missing constraint.

5.1.2 Restructuring Metrics

For restructurings, we measured the number of correct, erroneous, and missing restructurings as follows:

- *Correct.* A correct restructuring performs a required atomic transformation in a transformation rule. For example, *Set access level of $method to public* in rule 2 correctly changes the visibility of a protected `setUp` method.
- *Erroneous.* An erroneous restructuring performs an atomic transformation that invalidates a transformation rule. For example, *Set scope level of $method to static* would be erroneous for rule 2 because the method should maintain its member scope.
- *Missing.* A missing restructuring is an atomic transformation not present in a transformation rule. For example, if *Add annotation "Before" to $method* were not present in rule 2, the composite refactoring would be incomplete.

```
rule "#1) Transform classes matching HelloWorldBean"
  no-loop
  salience 0
    when
      $class : Application Class
        - Visibility is public
        - Implements "javax.ejb.SessionBean"
    then
      Remove interface from $class
      Update $class
end
rule "#2) Transform all methods matching ejbActivate"
  no-loop
  salience 10
  when
    $class : Application Class
      - Visibility is public
      - Implements "javax.ejb.SessionBean"
    $method : Application Method
      - Name matches "ejbActivate"
      - Declaring class is $class
      - Not annotated with "javax.ejb.PostActivate"
      - Visibility is public
      - Scope level is member
      - Return type is "void"
      - Has 0 parameters
  then
    Add annotation "javax.ejb.PostActivate" to $method
    Update $class
    Update $method
end
rule "#3) Transform all methods matching ejbRemove"
  no-loop
  salience 10
  when
    $class : Application Class
      - Visibility is public
      - Implements "javax.ejb.SessionBean"
    $method : Application Method
      - Name matches "ejbRemove"
      - Declaring class is $class
      - Not annotated with "javax.ejb.Remove"
      - Visibility is public
      - Scope level is member
      - Return type is "void"
      - Has 0 parameters
  then
    Add annotation "javax.ejb.Remove" to $method
    Update $class
    Update $method
end
```

Figure 17. The rules learned to transform the EJB example from version 2 to version 3.

5.1.3 Rule Order Metrics

For transformation rules, we measured the number of correct and erroneous rule execution orders (i.e., salience values) as follows:

- *Correct.* A correctly ordered transformation rule does not invalidate other rules when executed. For example, executing rule 2 before rule 1 will not preclude rule 1 from executing.
- *Erroneous.* An erroneously ordered transformation rule invalidates another rule when executed. For example, executing rule 1 before rule 2 would invalidate rule 2 by prematurely removing the super class from the target test case.

5.1.4 Manual Refinement Metrics

To measure the manual effort required by a developer, we counted the number of minor and major refinements needed for each refactoring as follows:

- *Minor.* A minor or small refinement is a required change to a single constraint, restructuring, or rule execution order. For example, changing *Visibility is protected* to *Visibility is at least protected* in rule 2 is a minor refinement.
- *Major.* A major or large refinement is a required addition or removal of an entire rule. For example, if rule 1 were not inferred, a major refinement would be needed to add it by hand.

Scenario	Upgrade Pattern
JUnit 3 test cases to JUnit 4	Bottom-Up
JUnit 3 test suites to JUnit 4	Top-Down
JUnit 3 test cases to TestNG	Bottom-Up
JUnit 4 test cases to TestNG	Identity
Serializable classes to JDO	Bottom-Up
Serializable classes to JPA	Bottom-Up
JDO classes to JPA	Identity

Table 2. The seven different upgrading scenarios and their corresponding upgrade patterns.

The above criteria were used to measure the accuracy of the inferred refactorings for seven scenarios, shown in Table 2 with their corresponding upgrade patterns. The first two refactoring scenarios focus on upgrading from JUnit 3 to JUnit 4, as has been discussed throughout the paper. The third and fourth scenarios focus on switching unit testing framework vendors from JUnit to TestNG [4]. The TestNG framework was developed as a next generation testing framework extending beyond unit testing to support regression, integration, and functional testing. Recognizing that many existing applications have implemented their testing functionality using JUnit, TestNG provides a hand-written automatic upgrade utility in their Eclipse plug-in. However, as of the latest version of TestNG, this upgrade utility has several software defects when upgrading JUnit 4 test cases, such as not properly removing JUnit annotations, inserting deprecated TestNG annotations, and incorrectly matching test method names. For this scenario, we used a representative example of a TestNG test case that was identical to the JUnit 4 test case example in Figure 1, but with the corresponding TestNG annotations.

The remaining three scenarios focus on upgrading applications to use different enterprise *orthogonal persistence* frameworks. In the first of these three scenarios, a `Serializable` class must be upgraded to use Apache's Java Data Objects (JDO) [34] annotations. In the second scenario, the same `Serializable` class must be upgraded to use the standardized J2EE Java Persistence API (JPA) [15]

Scenario		Constraints				Restructurings			Rules		Refinements	
Target	Upgrade	C	M	X	E	C	M	E	C	E	S	L
Test Cases	JUnit 3 to 4	29	0	5	0	11	0	0	5	0	5	0
	JUnit 3 to TestNG	28	0	5	0	10	0	0	5	0	5	0
	JUnit 4 to TestNG	25	0	0	0	11	0	0	5	0	0	0
Test Suites	JUnit 3 to 4	52	0	0	0	11	0	0	6	0	0	0
Persistence	Serializable to JDO	78	0	0	0	14	0	0	8	1	1	0
	Serializable to JPA	78	0	0	0	14	0	0	8	1	1	0
	JDO to JPA	67	0	0	0	24	0	0	8	1	1	0
Total		357	0	10	0	95	0	0	43	3	13	0

Table 3. The accuracy of the inferred rules for the seven different upgrading scenarios. C=Correct; M=Missing; X=Excessive; E=Erroneous; S=Small(Minor); L=Large(Major).

annotations. In the third scenario, a class marked with JDO annotations must be transitioned to use JPA annotations.[3]

Table 3 shows the accuracy of the inferred refactorings. As proved in Lemmas 1-3, our algorithm does not miss any required constraints or restructurings, nor does it infer any erroneous ones. Similarly, no major refactoring refinements are required by the developer in any of the seven scenarios.

The inference algorithm generated the same five excessive constraints in both of the JUnit 3 test case upgrading scenarios. Two of these excessive constraints unnecessarily limit the applicability of the transformation rules for the `setUp` and `tearDown` methods to only protected visibility, even though such methods can be public. The remaining three excessive constraints unnecessarily require the declaring class of `setUp`, `tearDown`, and `test` methods to directly extend `TestCase`, even though an indirect extension is valid. These two situations arise due to the small sample size used by our approach (i.e., only two representative examples). Thus, while the inferred JUnit 3 test case refactorings are correct for the given examples in both scenarios, five minor refinements are necessary to make the refactorings general enough to fully capture all valid test cases.

For the persistence scenarios, each generated refactoring requires one transformation rule reordering. The inferred rules find a naming convention for both the primary database key and persisted elements in a class, however the latter is a more general version of the former (i.e., both conventions will match the primary database key). Thus, to ensure that the primary database key is annotated before regularly persisted elements, it must be given a slightly higher precedence in the execution order, resulting in one required minor refinement for each of the persistence refactorings.

The accuracy metrics presented above show that the rules automatically inferred by our algorithm have a high degree of accuracy. On average, 97% of the total constraints, restructurings, and rule orderings require no refinement by the developer, with five out of seven refactoring scenarios requiring at most one minor change. The accuracy of the inferred refactorings has an important practical significance. Unlike the hand-written upgrade utility provided by TestNG, the JUnit 4 to TestNG refactorings inferred by our approach accurately upgrade all JUnit 4 test cases without requiring any manual refinement.

[3] For the three orthogonal persistence scenarios, we have selected a commonly-used subset of functionality of JDO and JPA specifications. The technical report version of this paper [38] contains representative examples for these persistence frameworks.

5.2 Case Studies

To demonstrate the effectiveness of a refined refactoring, we have upgraded the JUnit 3 test cases of four open-source, real-world applications to JUnit 4. Table 4 presents the total number of lines of testing code, `TestCase` classes, `test` methods, `setUp` methods, and `tearDown` methods upgraded in each application. Overall, we have successfully upgraded more than 80K lines of Java source code, eliminating the need to perform this refactoring by hand.

6. Related Work

Our approach relies on automated inference of program transformation and structural program differences. While these are broad and extensive research areas, to the best of our knowledge, none of the existing techniques in either of these areas are sufficient to automatically upgrade applications that use annotation-based frameworks.

6.1 Technique Classification

In a comprehensive survey, Visser [41] presents a taxonomy of program transformation systems. This taxonomy divides program transformations into two broad categories: translation and rephrasing. Translation involves transforming a program from one language to another, whereas rephrasing is concerned with program-improving transformations within the same language. Refactoring is a special subclass of rephrasing that improves the design of a program while maintaining its functionality. Renovation brings a program up to date with changed requirements, and migration ports a program from one language to another.

Application	Lines of Code	Test Cases	Tests	`setUp` Methods	`tearDown` Methods
JHotDraw 7.0.9	378	2	42	2	2
JBoss Drools 4.0.3	16,942	101	453	38	11
Apache Ant 1.7.0	21,969	251	1,714	187	99
JFreeChart 1.0.8	41,056	318	1,739	39	1
Total	80,345	672	3,948	266	113

Table 4. Upgrade statistics for the four real-world case studies.

Our approach entails changing the application code to use a different framework. The program's semantics is preserved–the program does the same thing but using a different framework, thus classifying our approach as a refactoring. However, traditional refactoring techniques do not capture large scale changes such as the use of a different framework. Upgrading an application that uses a framework based on subtyping and naming requirements to an annotation-based framework can also be considered a renovation. Additionally, transforming legacy applications written in a language without annotations to a language with annotations is migration, as the different versions of a language (e.g., Java 1.4 vs. Java 1.5) can be considered as two different languages. Since semantic equality is the main goal of our transformations, we consider our approach a refactoring.

6.2 Program Transformation Systems

Our technique relies on rule-based program transformations to implement the automatically-inferred refactorings. Multiple program transformation systems have appeared in the research literature. A representative of a state-of-the-art general program transformation system is Stratego/XT [42], which enables a variety of transformations from both the translation and rephrasing categories. Another example of a multi-language transformation system is DMS®[2], which focuses on scalability and efficiency.

Several other transformation systems target a single language. JaTS [7] provides a Java-like syntax for specifying program transformations in a manner similar to macros. Inject/J [22] enables program transformations at a level higher than that of an abstract syntax tree (AST) by providing a meta-model that can be manipulated via a domain-specific scripting language. TXL [10] uses a first order functional programming model, allowing explicit programmer control over several phases of the parsing and rewriting process. iXJ [5] aims at providing a visual language to enable interactive program transformations. The Smalltalk Refactoring Browser [32], a key example of a successful application of program transformation in a commercial setting, uses an extended Smalltalk syntax to specify AST pattern trees. The Arcum framework [35] uses declarative pattern matching and substitution to specify crosscutting design idioms.

While these systems are extremely powerful tools for implementing program transformations, they do not provide support for automatically inferring transformation rules, as required by our approach. However, our algorithm is general enough that it could be used to infer transformation rules in any of the above systems that support metadata transformations.

6.3 Program Differencing

To infer the necessary set of transformations, our technique requires the ability to calculate differences between two program versions. Program differencing is an active research area and several differencing algorithms have been proposed recently.

Dmitriev describes a make utility for Java [19] that leverages program change history to selectively recompile dependent source files. UMLDiff [43, 45] detects structural differences between two successive version of a program and accurately models the design evolution of the system. DSMDiff [26] identifies differences between domain-specific models. Kim et al. [24] use a string similarity measure to infer structural changes at or above the level of a method header, represented as first-order relational logic rules. Since none of these techniques extend beyond the method header level, they cannot be leveraged to detect upgrade patterns at the required level of granularity.

The Breakaway tool [13] helps determine the detailed correspondences between two classes through the visualization of similarities between two ASTs. The Change Distilling algorithm [21] uses an optimized version of a tree differencing algorithm for hierarchically structured data [8] to extract fine-grained source code changes. The JDiff [1] algorithm uses an augmented representation of a control-flow graph to identify changes in the behaviors between two versions of an object-oriented program. Since none of these algorithms can generalize the inferred differences, they cannot be leveraged to infer generalized refactoring rules.

6.4 API Evolution

Transitioning a legacy application from a convention-based to an annotation-based framework is closely-tied to the problem of API evolution, which has been a highly-active area of recent research. Explicit documentation (e.g., change annotations [9], refactoring tags [33], metapatterns [39], and deprecation inlining [29]) has been proposed as a means of facilitating evolution of framework dependent applications.

More recent approaches aim at automating the inference and application of refactorings. CatchUp! [23] records refactorings done by framework developers and provides facilities for replaying them on the client to update application code. Extension rules [11, 12] enable generalization transformations that add variability and flexibility into the class structure of a framework, thereby ensuring consistency with client applications. RefactoringCrawler [16] combines syntactic and semantic analyses to detect refactorings in evolving components. RefacLib [37] follows a similar approach, but replaces semantic analysis with various analysis heuristics. MolhadoRef [17] is a software configuration management system that reduces merge conflicts and facilitates program evolution comprehension by tracking refactorings and being aware of program entities. ReBA [18] generates compatibility layers that ensure binary compatibility between new library APIs and old clients, similarly to a binary adaptation layer in [14] that adapts legacy binaries for new framework releases. Diff-CatchUp [44] leverages design differences inferred by the UMLDiff algorithm described above to apply a set of heuristics to suggest API replacements in response to compilation errors.

While these approaches have all been very effective for their target domains, they differ from our approach. Specifically, they only support simple refactorings such as *Change Signature*, and they do not combine and generalize these refactorings, as required for upgrading applications that use annotation-based frameworks.

6.5 Programming by Demonstration

Inferring a set of rules from a pair of examples bears similarity to programming by demonstration [25]. For a system to be classified as "programming by demonstration," it must meet two criteria. First, the programmer must create the application via the same commands or process that would be used to perform the task manually. Second, the programmer must write the program by giving an example of the desired behavior. Since Rosemari enables programmers to use the standard Eclipse interface to input representative examples, it could be classified as a *refactoring by demonstration* system.

7. Future Work

Currently, our inferencing algorithm supports only Java 5 annotations. In the future, we plan to extend this work to support the upcoming Java 7 annotations that enable annotating a broader set of program elements. However, our structural differencing algorithm will need to be extended to handle annotated local variables in method bodies, possibly requiring static analysis.

In addition, many frameworks that used XML-based metadata rather than type and naming requirements in their previous versions have since transitioned to annotations. We plan to extend our approach to automatically infer refactorings for legacy applications that use XML-based metadata. Such an extension could potentially be enabled by extending the notion of a prior representative example to include XML snippets, and subsequently incorporating XML analysis into our algorithm.

Finally, finding a more unified approach to inferring refactoring rules has great potential benefits. This may be realized either by replacing the current pattern-based approach with a more sophisticated analysis, introducing a step for inputting domain-specific knowledge, or adding algorithmic support for multiple representative examples. These enhancements could eliminate the need for a developer to decide on which upgrade pattern to use, flattening the learning curve for using our system and increasing the possibility of widespread adoption.

8. Conclusions

This paper presented Annotation Refactoring, an approach to solving the Vendor and Version lock-in problems associated with annotation-based frameworks. Our approach is based on the wide availability of upgrade guides containing examples of legacy classes and their upgraded versions. Leveraging these examples, our tool enables framework developers to generate refactoring utilities capable of automatically upgrading legacy applications. As demonstrated by our case studies, the inferred refactorings are highly-accurate and can effectively upgrade large-scale, real-world applications.

Acknowledgments

The authors would like to thank Taweesup "Term" Apiwattanapong, Godmar Back, Patrick Yaner, Dave Zook, and the anonymous reviewers for their useful comments that helped improve this paper. This research was supported by the Department of Computer Science at Virginia Tech.

References

[1] T. Apiwattanapong, A. Orso, and M. J. Harrold. A differencing algorithm for object-oriented programs. In *Automated Software Engineering, 2004. Proceedings. 19^{th} International Conference on*, pages 2–13, 2004.

[2] I. D. Baxter, C. Pidgeon, and M. Mehlich. DMS: Program transformations for practical scalable software evolution. In *ICSE '04: Proceeding of the 26^{th} International Conference on Software Engineering*, pages 625–634, Los Alamitos, CA, USA, 2004. IEEE Computer Society.

[3] K. Beck and E. Gamma. Test Infected: Programmers love writing tests. *Java Report*, 3(7):37–50, 1998.

[4] C. Beust and H. Suleiman. *Next Generation Java Testing: TestNG and Advanced Concepts*. Addison-Wesley Professional, 2007.

[5] M. Boshernitsan, S. L. Graham, and M. A. Hearst. Aligning development tools with the way programmers think about code changes. In *CHI '07: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 567–576, New York, NY, USA, 2007. ACM.

[6] W. Brown, R. Malveau, H. McCormick III, and T. Mowbray. *AntiPatterns: refactoring software, architectures, and projects in crisis*. John Wiley & Sons, Inc. New York, NY, USA, 1998.

[7] F. Castor and P. Borba. A language for specifying Java transformations. In *V Brazilian Symposium on Programming Languages*, pages 236–251, 2001.

[8] S. S. Chawathe, A. Rajaraman, H. Garcia-Molina, and J. Widom. Change detection in hierarchically structured information. In *SIGMOD '96: Proceedings of the 1996 ACM SIGMOD international conference on Management of data*, pages 493–504, New York, NY, USA, 1996. ACM.

[9] K. Chow and D. Notkin. Semi-automatic update of applications in response to library changes. In *ICSM '96: Proceedings of the 1996 International Conference on Software Maintenance*, page 359, Washington, DC, USA, 1996. IEEE Computer Society.

[10] J. Cordy. The TXL source transformation language. *Science of Computer Programming*, 61(3):190–210, August 2006.

[11] M. Cortés, M. Fontoura, and C. Lucena. Using refactoring and unification rules to assist framework evolution. *SIGSOFT Softw. Eng. Notes*, 28(6):1–5, 2003.

[12] M. Cortés, M. Fontoura, and C. Lucena. A Rule-based Approach to Framework Evolution. *Journal of Object Technology (JOT)*, 5(1), jan-feb 2006.

[13] R. Cottrell, J. J. C. Chang, R. J. Walker, and J. Denzinger. Determining detailed structural correspondence for generalization tasks. In *ESEC-FSE '07: Proceedings of the the* 6^{th} *ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 165–174, New York, NY, USA, 2007. ACM.

[14] I. Şavga and M. Rudolf. Refactoring-based support for binary compatibility in evolving frameworks. In *GPCE '07: Proceedings of the* 6^{th} *International Conference on Generative Programming and Component Engineering*, pages 175–184, New York, NY, USA, 2007. ACM.

[15] L. DeMichiel and M. Keith. JSR 220: Enterprise JavaBeans 3.0, 2008. `http://jcp.org/aboutJava/communityprocess/final/jsr220/index.html`.

[16] D. Dig, C. Comertoglu, D. Marinov, and R. Johnson. Automated detection of refactorings in evolving components. In *ECOOP*, pages 404–428, 2006.

[17] D. Dig, K. Manzoor, R. Johnson, and T. N. Nguyen. Refactoring-aware configuration management for object-oriented programs. In *ICSE '07: Proceedings of the* 29^{th} *International Conference on Software Engineering*, pages 427–436, Washington, DC, USA, 2007. IEEE Computer Society.

[18] D. Dig, S. Negara, V. Mohindra, and R. Johnson. ReBA: refactoring-aware binary adaptation of evolving libraries. In *ICSE '08: Proceedings of the 30th international conference on Software engineering*, pages 441–450, New York, NY, USA, 2008. ACM.

[19] M. Dmitriev. Language-specific make technology for the Java programming language. *SIGPLAN Not.*, 37(11):373–385, 2002.

[20] Eclipse Foundation. Eclipse Java development tools, March 2008. `http://www.eclipse.org/jdt`.

[21] B. Fluri, M. Wuersch, M. Pinzger, and H. Gall. Change distilling: Tree differencing for fine-grained source code change extraction. *IEEE Trans. Softw. Eng.*, 33(11):725–743, 2007.

[22] T. Genssler and V. Kuttruff. Source-to-source transformation in the large. In *Modular Programming Languages*, pages 254–265. Springer-Verlag, 2003.

[23] J. Henkel and A. Diwan. CatchUp!: capturing and replaying refactorings to support API evolution. In *ICSE '05: Proceedings of the* 27^{th} *International Conference on Software Engineering*, pages 274–283, New York, NY, USA, 2005. ACM.

[24] M. Kim, D. Notkin, and D. Grossman. Automatic inference of structural changes for matching across program versions. In *The* 29^{th} *International Conference on Software Engineering (ICSE'07)*, pages 333–343, 2007.

[25] H. Lieberman. *Your Wish is My Command: Programming By Example*. Morgan Kaufmann, 2001.

[26] Y. Lin, J. Gray, and F. Jouault. DSMDiff: A differentiation tool for domain-specific models. *European Journal of Information Systems*, 16:349–361, 2007.

[27] D. Panda, D. Clarke, and M. Schincariol. EJB 3.0 migration. Technical report, Oracle, October 2005.

[28] R. Pawlak, C. Noguera, and N. Petitprez. Spoon: Program analysis and transformation in Java. Technical report, INRIA Research Report, 2006.

[29] J. H. Perkins. Automatically generating refactorings to support API evolution. In *PASTE '05: Proceedings of the* 6^{th} *ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, pages 111–114, New York, NY, USA, 2005. ACM.

[30] M. Proctor, M. Neale, P. Lin, and M. Frandsen. Drools Documentation. Technical report, JBoss Inc., 2006.

[31] C. Richardson. Untangling enterprise Java. *Queue*, 4(5):36–44, 2006.

[32] D. Roberts and J. Brant. Tools for making impossible changes - experiences with a tool for transforming large Smalltalk programs. *Software, IEE Proceedings* -, 151(2):49–56, 2004. 1462-5970.

[33] S. Roock and A. Havenstein. Refactoring tags for automatic refactoring of framework dependent applications. In *Proc. Int'l Conf. eXtreme Programming and Flexible Processes in Software Engineering (XP)*, 2002.

[34] C. Russell. Java Data Objects 2.1, June 2007. `http://db.apache.org/jdo/specifications.html`.

[35] M. Shonle, W. G. Griswold, and S. Lerner. Beyond refactoring: a framework for modular maintenance of crosscutting design idioms. In *ESEC-FSE '07: Proceedings of the the* 6^{th} *ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 175–184, 2007.

[36] R. Stuckert. JUnit reloaded, December 2006. `http://today.java.net/pub/a/today/2006/12/07/junit-reloaded.html`.

[37] K. Taneja, D. Dig, and T. Xie. Automated detection of API refactorings in libraries. In *ASE '07: Proceedings of the* 22^{nd} *IEEE/ACM International Conference on Automated Software Engineering*. IEEE Computer Society, 2007.

[38] W. Tansey and E. Tilevich. Refactoring object-oriented applications for metadata-based frameworks. Technical report, Virginia Tech, January 2008.

[39] T. Tourwé and T. Mens. Automated support for framework-based software evolution. In *ICSM '03: Proceedings of the International Conference on Software Maintenance*, page 148, Washington, DC, USA, 2003. IEEE Computer Society.

[40] D. Vines and K. Sutter. Migrating legacy Hibernate applications to OpenJPA and EJB 3.0, August 2007. `http://www.ibm.com/developerworks/websphere/techjournal/0708_vines/0708_vines.html`.

[41] E. Visser. A survey of strategies in program transformation systems. *Electronic Notes in Theoretical Computer Science*, 57, 2001.

[42] E. Visser. Program transformation with Stratego/XT: Rules, strategies, tools, and systems in StrategoXT-0.9. In *Domain-Specific Program Generation*, volume 3016 of *Lecture Notes in Computer Science*, pages 216–238. Spinger-Verlag, June 2004.

[43] Z. Xing and E. Stroulia. UMLDiff: an algorithm for object-oriented design differencing. In *ASE '05: Proceedings of the* 20^{th} *IEEE/ACM International Conference on Automated Software Engineering*, pages 54–65, 2005.

[44] Z. Xing and E. Stroulia. API-evolution support with Diff-CatchUp. *IEEE Trans. Softw. Eng.*, 33(12):818–836, 2007.

[45] Z. Xing and E. Stroulia. Differencing logical UML models. *Automated Software Engineering*, 14(2):215–259, 2007.