Defect Isolation As Responsibility of the Framework

Automated API Migration from JUnit to JExample

Masterarbeit
der Philosophisch-naturwissenschaftlichen Fakultät
der Universität Bern

vorgelegt von

Lea Hänsenberger

September 2009

Leiter der Arbeit

Prof. Dr. Oscar Nierstrasz

Adrian Kuhn

Institut für Informatik und angewandte Mathematik

Further information about this work and the tools used as well as an online version of this document can be found under the following addresses:

Lea Hänsenberger
lea.haensenberger@gmail.com
http://scg.unibe.ch/wiki/projects/archive/JUnit2JExample

Software Composition Group
University of Bern
Institute of Computer Science and Applied Mathematics
Neubrückstrasse 10
CH-3012 Bern
http://scg.unibe.ch/

Abstract

Unit tests are primarily written as a good practice to support software evolution, *i.e.,* to help developers to identify and fix bugs, to refactor code and to serve as documentation for a unit of software under test. To achieve these benefits, unit tests ideally should cover all possible paths in a program. One unit test usually covers one specific path in one function or method. However, a test method is not necessary an encapsulated, independent entity. Often a test method's coverage is a superset of another test method's coverage set and thus defects are not well isolated, *i.e.,* one defect causes multiple test methods to fail. In this work we present an approach to automatically migrate JUnit test classes to JExample. JExample allows test methods to declare explicit dependencies to other test methods and therefore improves defect isolation. With dynamic analysis we recover the coverage set of each test method and by partially ordering the test methods by means of their coverage sets we derive implicit dependencies between test methods. With program transformation we rewrite the original JUnit test classes as test classes with explicit dependencies between test methods that can be executed with JExample. In a case study on 16 projects we found that 72% of all test methods have latent dependencies to other test methods and that by declaring these dependencies defect isolation (measured as average square of failures per defect) could be improved by a factor of 3.77 or higher.

Acknowledgements

At this point I'd like to express my thanks to all the people that supported me in many different ways during the last year and with their support helped me finishing this work.

Most importantly, my supervisor Adrian Kuhn for always providing me with new ideas, for always helping me when I encountered a problem, and most of all, for all the time he worked with me on this thesis.

Prof. Dr. Oscar Nierstrasz for letting me write my thesis in the Software Composition Group (SCG).

All the members of the SCG for their helpful comments during the weekly meetings.

David Schuler for giving me early access to his code manipulation framework JAVALANCHE and for patiently answering all my questions. Without this help I could not have completed my case study.

My fellow students, especially Dave Erni, Michael Pfeuti, Brigitte Hulliger, Dominique Matter, Thomas Killer, Simon Willi, Camillo Bruni and Florian Gysin, for the interesting discussions and funny moments over lunch and while working in the SCG pool.

Andreas Fischer for proof-reading the final draft of this thesis and providing me with many helpful comments.

Rafael Wampfler for always motivating me to keep working in hard times and to finally finish this work.

My family, especially my parents, without whose support I would not have accomplished all this.

Thank you!

Contents

Chapter 1

Introduction

The key benefit of writing unit tests is that they facilitate changes to a unit of code and that they serve as documentation. When refactoring code at a later date, unit tests help the developer to make sure that the single methods and functions still work correctly. Good unit tests also help a developer to identify and fix new bugs quickly. Moreover, unit tests provide a documentation for the unit under test. What functionality a unit provides and how to use it can usually be learned by browsing the unit tests.

The key question is, what makes a good unit test? In order to be able to identify and fix potential bugs, unit tests must cover all possible paths in a program. Furthermore they need to be well factorized, *i.e.,* every test method covers one specific path in a method or function. For example, when testing the pop method of a stack implementation, one test method tests pop with a full stack, making sure that after pop was performed the stack contains one element less than before. Another test method ensures that a user exception is thrown when popping an element from an empty stack. A fine granularity makes it easier to exactly locate a defect.

Nevertheless, fine grained unit tests do not eliminate some shortcomings current unit testing frameworks such as JUNIT have:

- *defect localization:* Since every test method needs to be an independent artifact, one method of the unit under test might be called in several test methods to, for example, extend the test fixture. Hence, one failing method of the unit under test might cause several failing test methods. This domino effect impedes the locating of the defect.
- *the `setUp` method is not a test method:* In JUNIT, the `setUp` method itself is not a test method. Thus, the `setUp` of a test fixture can not be tested. Additionally, a failing `setUp` method causes all the test methods of a test case to fail, without being marked as failed itself.
- *a growing unit under test:* The fixture creating the instance of the unit under test,

i.e., the `setUp` method in JUNIT, is the only thing all the test methods in a JUNIT test case share. This instance needs to be extended during a test run so the test case can cover all the possible paths. Most probably, multiple test methods need to make the same or similar extensions to the unit under test. This leads to duplicated code in a test case.

Our solution to these problems is to introduce explicit dependencies between test methods. The xUnit family of testing frameworks advises to avoid dependencies between tests because every test method is supposed to be an independent artifact, sharing at most a test fixture creating the unit under test. On the other hand the frameworks JEXAMPLE and TESTNG [2] exist, supporting explicit test dependencies [12, 2]. The advantages and disadvantages of dependencies between tests have received a lot of attention in the literature [5, 8, 15].

As shown by previous research, xUnit tests tend to have implicit dependencies [11]. This suggests that dependencies between tests are inevitable. Furthermore, in xUnit tests the instance of the unit under test is recreated for every test method and thus, every test method has to modify this instance under test so it can test the actual thing to test. For example, the `testRemove()` method in Listing 1.1 has to add an element to the set before it can test the removing of an element. This implies that there are certain implicit dependencies between test methods.

To illustrate such implicit dependencies consider the methods in Listing 1.1. `testAdd()` and `testRemove()` are implemented to run independently of each other. Nevertheless, `testRemove()` implicitly depends on the outcome of `testAdd()`. Both methods must cover `Set`'s `add()` method, hence, an implicit dependency between `testAdd()` and `testRemove()` exists: if `testAdd()` fails `testRemove()` is likely to fail too.

Listing 1.1: Implicit dependency between test methods.

```
@Test
public void testAdd() {
     Set set = new TreeSet();
   set.add("Foo");
   assertEquals(1, set.size());
}

@Test
public void testRemove() {
     Set set = new TreeSet();
   set.add("Foo");
   set.remove("Foo");
   assertEquals(0, set.size());
}
```

Such implicit dependencies are often hard to avoid, as operations for software entities naturally depend on each other [8, 15, 5]. They are not at all uncommon in test code.

In previous work we introduced JEXAMPLE, an extension to JUNIT providing explicit dependencies between test methods [12, 14].

By means of explicit dependencies between test methods, JEXAMPLE solves the previously mentioned problems of conventional JUNIT tests as follows:

- Test methods depending on a failing test method are not executed and marked as *white* (ignored). Thus, only the test method covering the actual cause of the domino effect is marked as *red* (failed). The failing test method points directly to the defect location.
- Creating the instance of the unit under test is done in a normal test method that can contain asserts. The instance of the unit under test is then returned and passed as argument to all the dependent methods.
- An extension to the unit under test is done in one test method. The unit under test is then returned and passed as argument to all the test methods needing this sort of extended unit under test.

Our thesis is that we can automate the migration of original JUNIT *test classes to* JEXAMPLE *test classes, and thus, shift the burden of isolating defects from the developer to the framework.*

By analyzing the coverage sets of the test methods we can automatically recover implicit dependencies between test methods. The recovered dependencies can then be used to automatically migrate the original JUNIT test classes into JEXAMPLE test classes that declare explicit dependencies. We expect the automatically generated JEXAMPLE tests to have better defect isolation. Since test methods depending on a failing test are ignored, ideally only one test fails per defect. Test methods covering the same defect should depend on the failing test method and thus be ignored. Moreover, we expect the JEXAMPLE tests to perform better since the instance of the unit under test is instantiated only once in JEXAMPLE. The growing instance of the unit under test is then passed down from consumers to producers.

To verify our thesis we introduce a procedure to automatically migrate JUNIT 4 tests to JEXAMPLE tests with explicit dependencies. Because most of the open source projects that implement unit tests use JUNIT 3 instead of JUNIT 4 [13], we decided to additionally implement an automatic migration from JUNIT 3 to JUNIT 4.

The migration from JUNIT 4 to JEXAMPLE happens in two steps:

- *Dynamic Analysis.* We run the original JUNIT tests and collect the coverage set of each test method, *i.e.,* all methods that were called during the execution of a test method. The methods are then partially sorted by means of their coverage sets. From this order we extract the implicit dependencies. If a method m_a covers a superset of the method m_b's coverage set, m_a implicitly depends on m_b.
- *Program Transformation.* With the results from the dynamic analysis we transform the original JUNIT tests into JEXAMPLE tests. Former `setUp` and `tearDown` methods (methods annotated with @Before and @After respectively) are migrated to conventional test methods. Based on the results from the dynamic analysis we declare new producer–consumer relationships. If a method m_a implicitly

depends on a method m_b we add an annotation @Given(m_b) to m_a. Furthermore, if possible, we introduce fixture injection and remove statements that both tests in a producer–consumer relationship execute from the body of the consumer.

We present the results of a case study where we compared the JUNIT 4 tests (original or migrated from JUNIT 3) with the automatically migrated JEXAMPLE tests. We compared the two test suites by means of the following criteria:

- *Detected Dependencies:* We plotted the dependency trees and analyzed how the detected dependencies correlate with project characteristics such as number of classes and number of test classes. Furthermore, we investigated correlations among characteristics of the migrated JEXAMPLE tests such as number of connected components, number of single nodes, percentage of nodes being part of a dependency, etc.
- *Performance:* We measured the execution time for the JUNIT 4 tests and the migrated JEXAMPLE tests, once run with the injection policy *rerun* and once run with the injection policy *cloning*.
- *Defect isolation:* By inserting defects into the application code we checked how many tests failed and, in the case of the JEXAMPLE tests, how many were ignored.

In 16 selected projects we found that 72% of all test methods have dependencies on other test methods based on their coverage sets. Also, we found that defect isolation (measured as average square of failures per defect) improved by a factor of 3.77 or higher by declaring explicit dependencies. However, even though for most projects the JEXAMPLE tests had a slightly faster execution time, we could not detect a significant improvement of performance in the matter of execution time.

1.1 Structure

For more information about the state of the art in this area of research, divided into a section about previous work and a section about related work, see Chapter 2. Details on the automatic migration of JUNIT 3 tests to JUNIT 4 tests can be found in Chapter 3. For more details on the automatic migration of JUNIT tests to JEXAMPLE tests see Chapter 4. See Chapter 5 for details on the implementation of the prototype JUNIT2JEXAMPLE that performs the automatic migration from JUNIT 4 to JEXAMPLE. For detailed results of our case study see Chapter 6. The conclusion can be found in Chapter 7.

Chapter 2

State of the Art

2.1 Previous work

In this section we discuss previous work that has been done in this area of research and on which our work is based on.

Gaelli *et al.* introduced the idea of partially ordering failed unit tests by means of their coverage sets in order to focus on the most specific test [11]. When several unit tests fail because of one defect it is often hard to find the actual defect because all tests cover other paths of a program. They therefore suggested to look at the coverage sets of the failing tests and to point to the test with the smallest set of covered methods that is covered by the other failing tests. A test method m_a covers a test method m_b if the set of invoked methods by m_a is a superset of the set of invoked methods by m_b. In their experiments they found that 85% to 95% of the test methods were comparable to other test methods by means of their coverage set. Furthermore, with error seeding they found that in the majority of the cases errors were propagated to all test methods covering them.

Kuhn *et al.* suggested to improve defect localization by using the JUNIT extension JEXAMPLE [14]. In a case study they compared a monolithic white-box test suite for a complex algorithm with three refactorings of the same test, two JUNIT style tests and one JEXAMPLE test. They found that JEXAMPLE tests report five times fewer defect locations and slightly better performance, while having similar maintenance characteristics. Compared to the original JUNIT test however, JEXAMPLE greatly improves maintainability because of the improved factorization of the test methods.

2.1.1 JEXAMPLE in a Nutshell

To facilitate dependent test methods JEXAMPLE extends JUNIT as follows:

- test methods may return values.
- test methods may take arguments.
- test methods may declare dependencies.

As previously mentioned, in JEXAMPLE `setUp` methods are not needed. Any test method m_a may be used as a `setUp` method offering its return value x as test fixture for its dependent methods. JEXAMPLE takes the return value x of m_a and passes it as an argument to all the methods depending on m_a.

The notion of dependent tests is related to the idea of example-driven testing outlined in the work of Gaelli [9], which states that test fixture instances are valuable objects, and hence, to be reused and treated as first-class by the testing framework. Using the terminology of Gaelli we say that each test method consists of an *example*, creating and testing an *example instance* of the unit under test.

JEXAMPLE uses three colors to indicate a test outcome:

green for successfully passed tests.

red to point out sources of failures.

white to indicate follow-up methods that had been skipped due to failing prerequisites.

This color scheme helps to locate defects: only the test method covering the actual cause of the domino effect is marked as *red*, however, dependent test methods will be skipped and thus marked *white*.

Writing Tests with JEXAMPLE

This section illustrates how JEXAMPLE makes dependencies explicit by exercising a sample test case. The unit under test is a simple stack implementation. A conventional JUNIT test case might go as follows:

Listing 2.1: Conventional JUNIT test case.

```
public class StackTest {

   private Stack stack;

   @Before
   public void setup() {
      stack = new Stack();
   }

   @Test
   public void testPush() {
      stack.push("Foo");
      assertFalse(stack.isEmpty());
      assertEquals("Foo", stack.top());
   }

   @Test
   public void testPop() {
      stack.push("Foo");
      Object top = stack.pop();
      assertTrue(stack.isEmpty());
      assertEquals("Foo", top);
   }

   @Test(expected=IllegalStateException.class)
   public void testPopFails() {
      stack.pop();
   }

   @Test
   public void testPushAll() {
      List list = Arrays.asList(new String[] { ... });
      last = list.get(list.size() - 1);
      stack.pushAll(list);
      assertEquals(last, stack.top());
   }

}
```

When running Listing 2.1 `setup()` is executed before every test method and the test fixture is passed as field to the test methods. Considering this we may say that all the test methods depend on `setup()`. In JEXAMPLE `setup()` is a normal test method returning an instance of a Stack.

Listing 2.2: Promote fixture to test with return value.

```
   @Test
   public Stack testEmpty() {
      Stack empty = new Stack();
      assertTrue(empty.isEmpty());
      assertEquals(null, empty.top()));
      return empty;
   }
```

Note, that by setting up the fixture in a test method instead of a `setUp` method, the initialization of the fixture can be tested too. Next, we rewrite `testPush()` and `testPopFails()`. They depend on `testEmpty()` because they need an empty stack as example instance. This is done by annotating them with `@Given`. The Stack returned by `testEmpty()` is passed to the dependent methods as an argument. In order for a dependency to be valid, the type of the returned object and the type of the argument taken by the dependent test method have to be the same.

Listing 2.3: Take another test's result as input value.

```
@Test
@Given("#testEmpty")
public Stack testPush(Stack stack) {
   stack.push("Foo");
   assertFalse(empty.isEmpty());
   assert("Foo", empty.top());
   return stack;
}

@Test(expected= IllegalStateException.class)
@Given("#testEmpty")
public Stack testPopFails(Stack empty) {
   stack.pop();
}
```

When running Listing 2.3 JEXAMPLE runs `testEmpty()` first, as it is the root of the dependency hierarchy. If `testEmpty()` has run successfully `testPush()` and `testPopFails()` are run with the return value from `testEmpty()` as argument (if `testEmpty()` fails, `testPush()` and `testPopFails()` are ignored). In order to have the same state of the stack for both dependent test methods, the return value needs to be cloned or regenerated.

Let's now reconsider Listing 2.1 to find deeper levels of dependencies in order to get a real graph of composed test methods. We find two methods that depend on `testPush()`: `testPop()` cannot be executed without pushing an element on the stack first and `testPushAll()` will probably also fail if pushing a single element fails.

The new implementation of `testPop()` needs to depend on `testPush()`'s return value. This also avoids the duplicate call to `push`.

Listing 2.4: Avoid code duplication using dependencies.

```
@Test
@Given("#testPush")
public Stack testPop(Stack stack) {
   Object top = stack.pop();
   assertEquals(true, empty.isEmpty());
   assertEquals("Foo", top);
   return stack;
}
```

`testPushAll()` needs a list of elements as a second argument. Let's assume we have a class `ListTest` with a method `testAddAll()`. The refactored `testPushAll()` is shown in Listing 2.5. Note that a test may have more than one dependency that can also refer to methods in other test case files.

Listing 2.5: A test may have multiple dependencies.

```
@Test
@Given("#testPush;ListTest.#testAddAll")
public Stack testPushAll(Stack stack, List list) {
   stack.pushAll(list);
   last = list.get(list.size() - 1);
   assertEquals(last, stack.top());
   return stack;
}
```

Figure 2.1 illustrates a sample sequence diagram of the refactored test case: with `Stack's push` method failing, running the test case with JExample will lead to two `white` tests and thus point precisely to the defect location:

1. First, JExample creates a test graph and populates it with the `StackTest` class. The framework processes all methods and their dependencies to initialize the graph of methods. Each method is linked with its dependencies and marked as colorless, i.e. not yet executed.
2. Then the framework orders the graph to run the test methods. It is topologically sorted and the methods that have no dependencies are executed first, then those methods depending on these initial methods, etc.
3. `testEmpty()` has no dependencies and thus is executed first. It is marked as green and its return value is cached by the framework.
4. Next, the framework attempts to run `testPush()`: it first checks if `testEmpty()` is marked *green*, which is the case, and thus catches the cached return value to pass it as an argument to `testPush()`. But the test method fails and is marked as *red*.
5. Now the framework attempts to run `testPop()`: it checks if `testPush()` has passed, which failed, and hence skips the current test method and marks it as *white*.
6. The same holds for `testPushAll()`.

If JExample is not told otherwise it clones a return value before it passes it on to all the dependent test methods. This makes sure that side effects on the fixture don't cause other test methods to fail. JExample can be told to do otherwise by adding the class annotation `@Injection(InjectionPolicy.<POLICY>)` to the test class. The following are the possible injection policies:

CLONE Uses `Object.clone()` to clone the return values provided by the dependencies. If this fails the dependencies are rerun.

DEEP_COPY Uses internal reflection to create a field-by-field deep copy of the return values.

NONE Does not clone any of the return values of the dependencies. This should be used with caution because it might cause test methods to fail if there are side effects on the passed down objects.

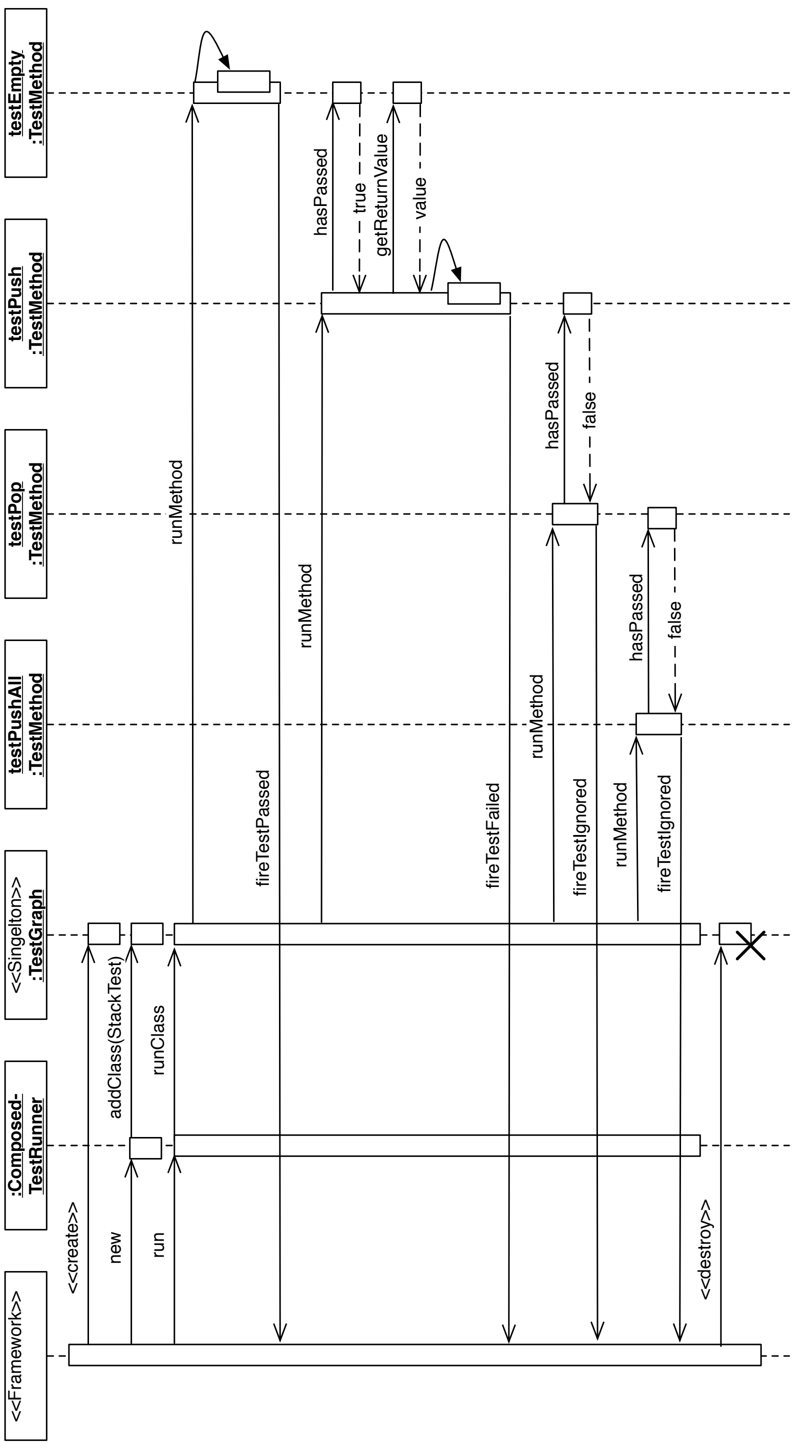

Figure 2.1: Sequence diagram with `Stack's push` method failing

RERUN This policy reruns all of the dependencies to get fresh return values.

2.2 Related work

In this section we discuss related work. This includes work about improvement of defect localization, on how to automatically refactor applications when a framework changes and on automatically factoring fast focused tests based on slow system-wide tests.

2.2.1 Defect localization

In software tests it often happens that one defect causes multiple tests to fail. This makes it hard for developers to find the actual defect. There are, however, some approaches on how to improve defect localization in existing software tests. Most of these approaches are based on unit tests. The approach of test case prioritization, however, is based on non-object oriented test suites.

Smith *et al.* present an approach to improve defect isolation by test suite reduction [20]. As with our (automated migration) approach, they use dynamic analysis to collect the call trace for each test method. They build a dynamic call tree or a context calling tree based on the collected call traces and apply a reduction algorithm to it that generates a modified test suite that is guaranteed to cover all tree paths with (possibly) fewer test cases. During reduction they also apply a prioritizing algorithm to achieve the coverage faster. They report on 45% fewer tests and 85% less execution time in a case study of one project. Their approach guarantees to cover the same production code as the original test suite. In comparison, our approach improves defect isolation while covering the same assertions as the original test suite, a much stronger test suite equivalence criterion.

Xie presents an approach to improve fault detection by augmenting automatically generated unit test suites with oracles [25]. They developed Orstra, a tool that adds oracles to automatically generated test suites. Orstra collects the object states of the class under test during test execution. On these collected object states it invokes public methods with non-void returns of the class under test and collects the actual return values. It then creates assertions to check the return values of these methods against the collected return values. In an experiment with eleven classes and tests automatically generated by JCrasher and Jtest they achieved improvement factors of up to 50 for JCrasher generated tests and improvement factor 6 for Jtest generated tests. These results indicate that object states contain valuable information for unit tests and hence, for future work on our approach we might also take into consideration object states to refine the detection of dependencies.

Rothermel *et al.* suggest to apply test case prioritization techniques to increase the effectiveness at fault detection, *i.e.,* how quickly faults are detected within the testing process [18]. They describe several techniques for test case prioritization based on test execution information. These techniques include i) ordering test cases based on their total coverage of code components, ii) ordering test cases based on their coverage of code components not previously covered and iii) ordering test cases based on their estimated ability to reveal faults in the code components that they cover. Experiments showed that each of the prioritization techniques could improve the rate of fault detection of test suites, but that there was also much room for improvements.

2.2.2 Framework refactoring

Framework-based applications often use an old version of a framework because the costs to refactor an application to use a new version of a framework with a new API are usually quite high. Below can be found some approaches and proceedings to automate the refactoring of applications to use the new version of a framework.

Tansey *et al.* discuss an algorithm to automatically refactor applications using an old framework using naming conventions to use a new framework using annotations instead [21]. The motivating example is the change from JUNIT 3 to JUNIT 4. The transformation consists of three phases. First the framework developer creates a *representative example* of a class using the framework, i.e. the old and the new version of that class. Second, provided with the representative example, the algorithm compares the two class versions at the level of classes, methods and statements and generates a set of generalized transformation rules. In addition to the representative example the framework developer should provide the algorithm with the respective upgrade pattern. The authors identified three possible patterns:

Bottom-Up: Restructurings are applied based on the level of granularity in the encapsulation hierarchy (level) itself and its enclosing levels.

Top-Down: Restructurings are applied based on the level itself and its contained levels.

Identity: This pattern applies restructurings only on the basis of the level itself, assuming that all annotations in the old version have a one-to-one mapping in the new version.

Finally application developers parametrize the program transformation engine with the generated rules. The engine then automatically refactors the application.

Roock *et al.* propose *refactoring tags* to help developers migrating applications to a new framework version with a changed API [17]. Often frameworks are not refactored because this causes a lot of effort for migrating framework dependent applications. However this does not correspond to the idea of extreme programming. Refactoring tags support XP for developing frameworks. They represent modifications done to the framework. If e.g. a class is renamed the framework developer adds the `@Past` tag stating the old class signature to the javadoc of the renamed class. The tags are then interpreted by migration tools. Those tools support application developers when migrating their applications.

Tourwé *et al.* introduce the idea of *metapatterns* as automated support for framework-based software evolution [22]. Framework-based software evolution includes the instantiation of a framework for new applications as well as migrating an existing application to an upgraded framework. Metapatterns define the design of the framework by means of hot spots (specific places where a framework can/should be extended with application specific code). Changes to a framework are defined by means of metapattern transformations. A set of possible conflicts when specific transformations are applied in parallel is defined as well. Possible conflicts are detected automatically when changing a specific version of a framework. This approach has been implemented in the SOUL logic meta programming environment [24], a research prototype.

Dagenais *et al.* discuss an approach to automatically find recommendations for adaptions of clients of evolving frameworks [4]. The targeted changes are not refactorings but non-trivial evolutions of a framework like e.g. moving or removing entire methods or classes. These recommendations are found by analyzing the adaptions of the framework to its own changes. Similar adaptions are then recommended to the client's developers. They developed SemDiff, a tool that analyzes the evolution of method calls in a framework (e.g. a call to a method m_a was replaced by calls to the methods m_b and m_c). Using SemDiff developers get recommendations on how to replace calls to deprecated or deleted methods.

Dig *et al.* introduce an approach to detect refactorings between two versions of evolving components [6]. First they do a syntactic analysis to detect refactoring candidates. It is based on Shingles encoding [3], a technique to find similar fragments in text files. The algorithm applies Shingles to the source files to find similar fragments of source code after possible repartitionings of the files. Shingles are computed for methods, classes and packages. Second, a semantic analysis is applied to the candidate pairs to find out whether they are indeed the result of refactorings. This analysis first detects refactorings by applying seven strategies: RenamePackage, RenameClass, RenameMethod, PullUpMethod, PushDownMethod, MoveMethod and ChangeMethodSignature. All detected refactorings are stored in the refactoring log. Afterwards, the likelihood of refactorings is computed based on references among the entities in each of the versions of the component. The algorithm was implemented in the eclipse plugin RefactoringCrawler.

Visser discusses strategies for rule-based program transformation systems [23]. After applying rules to an application the semantics of a program should still be the same. Rules do not change the visible behavior of an application but other aspects like optimizations. In our approach we also identified a set of rules to transform JUnit tests to JExample. The transformed JExample tests still cover the same assertions as the original JUnit tests. Hence, the visible behavior of the tests is the same. However, the transformed tests are optimized in the matter of improved defect isolation.

2.2.3 Test factoring

Unit tests are supposed to be small isolated artifacts that test different paths of an application. When changing an existing application developers can only run the according tests instead of all of them and therefore, get a fast feedback on the changes they made. But, what if an application only has slow system-wide tests?

Saff *et al.* present a prototype to automatically factor fast focused unit tests based on slow system-wide tests. They suggest the use of *mock objects* to factor the tests. The program is partitioned in two parts T and E where T is the code under test and E is the "environment" with which T interacts. Every object in E is replaced with a mock object. The mock checks whether the calls from T are as expected and simulates the behavior of E in response. Like this the factored tests can isolate bugs in T from bugs in E. Furthermore, if E is expensive or slow, they can improve the test performance. Mock objects are created via a dynamic, capture-replay technique. During the capture state the original system-wide test is run and a transcript of the interaction between T and E is written. When replaying the test T is executed as usual. However, whenever T would have interacted with E no computation is done but the value recorded during the capture state is used.

Chapter 3

Automatically Migrating JUNIT 3 to JUNIT 4

In this section we very shortly explain why we decided to implement an automatic migration from JUNIT 3 to JUNIT 4 beside the actual migration from JUNIT to JEXAMPLE.

Then we report on the differences of the two versions of the framework and how the characteristics of JUNIT 3 can be converted to the new characteristics of JUNIT 4.

In order to find projects for our case study we searched the code search database SOURCERER [16, 1] for open source java projects with unit tests. We found that of the 2,848 projects in the database a total of 1,160 projects are covered with test suites, i.e. only 40%. The usage of the different testing frameworks is as follows: 85% of the projects use version 3 of JUNIT, only 10% use version 4 of JUNIT and 5% use TESTNG [13].

Because of these many projects using JUNIT 3 and only very few using JUNIT 4 for their unit tests we decided to implement an automatic migration from JUNIT 3 to JUNIT 4, called JUNIT3TO4. JUNIT 3 is mainly based on naming conventions whereas JUNIT 4 is entirely based on annotations. The migration thus basically consists of statically analyzing the JUNIT 3 tests and matching JUNIT 3's naming conventions to JUNIT 4's annotations.

In JUNIT 3 every class that contains test methods has to extend `junit.framework.TestCase`. Also test suites, *i.e.,* classes containing the `suite` method where test classes that constitute the test suite are added to a `Test` object, extend `junit.framework.TestCase` or `junit.framework.TestSuite`.

Otherwise, JUNIT 3 works with naming conventions. A method is only a test method if it is `public`, its return type is `void`, it takes no arguments and its name starts with "test". setUp and tearDown methods have to be protected with return type void, no arguments and the exact name setUp and tearDown respectively. See Listing 3.1 for a simple JUNIT 3 test class.

Listing 3.1: A simple JUNIT 3 test class

```
public class StackTest extends TestCase {

        Stack stack;

        protected void setUp(){
              stack = new Stack();
        }

        public void testEmpty(){
              assertTrue(stack.isEmpty());
        }

        protected void tearDown(){
              stack = null;
        }
}
```

Unlike JUNIT 3, JUNIT 4 is fully based on annotations. That is, test classes don't have to extend a certain class anymore and naming conventions disappeared. On the contrary, test methods don't even need to be declared in a special test class anymore. A test method is a test method as soon as it is annotated with @Test. setUp and tearDown methods are annotated with @Before and @After respectively. Additionally, test, setUp and tearDown methods have to be public and they have to have return type void. Thanks to the annotations test, setUp and tearDown methods can be specified in any class in the application. When running that class as a JUNIT test case only the accordingly annotated methods are executed. When running it as a java class the JUNIT specific methods are ignored.

Migrating JUNIT 3 test classes to JUNIT 4 test classes thus merely consists of removing unnecessary things, like the extension of a certain class, and doing a one to one transformation of a naming convention to its corresponding annotation. Hence, we can achieve the migration by traversing the abstract syntax tree (AST) and statically analyzing and changing the original JUNIT 3 test class.

- The extension of junit.framework.TestCase is removed. As consequence calls to super in constructors and methods have to be removed as well if TestCase is a direct superclass of the original JUNIT 3 test class.
- Methods being public, having return type void and a name starting with "test" are annotated with @Test.
- Methods being protected, having return type void and the name "setUp" are made public and annotated with @Before.
- Methods being protected, having return type void and the name "tearDown" are made public and annotated with @After.

In the end, all import statements pointing to JUNIT 3 classes are removed and the necessary JUNIT 4 imports are inserted into the newly generated test class.

See Listing 3.2 for the migrated JUNIT 4 test class equivalent to the previously given JUNIT 3 example class.

Listing 3.2: The automatically migrated JUNIT 4 test class

```java
public class StackTest {

	Stack stack;

	@Before
	public void setUp(){
		stack = new Stack();
	}

	@Test
	public void testEmpty(){
		assertTrue(stack.isEmpty());
	}

	@After
	public void tearDown(){
		stack = null;
	}
}
```

The described tool, however, cannot transform all sorts of test suites to JUNIT 4. In JUNIT 3 one can create test suites only containing a subset of the test methods of one or multiple test classes. This is also the only way for a developer to define the order in which the test methods are supposed to be executed. However, JUNIT 4 does not support the creation of this sort of test suite. In JUNIT 4 only whole test classes can be combined to a test suite, therefore, only this kind of test suite can be automatically transformed to JUNIT 4 by our tool.

We use an internal API of the Java Compiler [7] to do static analysis and program transformation. We programmatically use the Java Compiler to parse and analyze the java files of the test classes that are to be converted. The resulting `com.sun.source.tree.CompilationUnitTree` is copied so we can apply changes without modifying the original test class. Afterwards we cast the `CompilationUnitTree` to `com.sun.tools.javac.tree.JCTree` which we then can visit component by component. The visited components are statically analyzed and, if necessary, modified. *E.g.,* we visit every method declaration and if it is a test method we add the `@Test` annotation to it. We then cast the modified `JCTree` to a `JTree$JCCompilationUnit` and write the string representation of the `JCCompilationUnit` to a new java file in the same package as the original test class.

See Section A.1 for a quick start guide on where to find and how to use JUNIT3TO4.

Chapter 4

From JUnit to JExample

In this chapter we report on the migration of JUnit tests to JExample tests. This chapter is a conceptual explanation of our approach without any implementation details. For more details about the actual implementation of our tool to automatically migrate JUnit tests to JExample see Chapter 5.

The migration consists of a dynamic analysis to detect dependencies between test methods and program transformation to declare the detected dependencies.

The migration consists of the following briefly summarized steps. First, we declare the `setUp` method as a dependency for every test method in a test case. Next, with dynamic analysis we try to find implicit dependencies between test methods by means of their coverage sets. Then we merge the dependency trees from steps one and two. Next, we want to remove as many dependencies to the `setUp` method as possible. This can be done if a method indirectly depends on the `setUp` method via another dependency and if it only depends on test methods that have no side effects on the test fixture. Finally, we do a fixture injection if a test case only has one field that serves as test fixture.

Let's consider the JUnit test case with one `setUp` and two test methods in Listing 4.1 as an example to illustrate the detailed explanation of the steps of the migration below.

Listing 4.1: JUNIT Test Case

```
private List<String> list;

@Before
public void setUp(){
  list = new ArrayList<String>();
}

@Test
public void testSize(){
  assertEquals(0, list.size());
  list.add("Foo");
  assertEquals(1, list.size());
}

@Test
public void testAdd(){
  assertEquals(0, list.size());
  list.add("Foo");
  assertEquals(1, list.size());
  assertEquals("Foo", list.get(0));
}
```

4.1 Setup Method as Dependency

A dependency that already exists in conventional JUNIT tests is the dependency of every test method on the `setUp` method. When running a JUNIT test case the `setUp` method is executed before every test method execution. Usually the instance under test is set up in that method.

In JEXAMPLE the notion of a `setUp` method does not exist anymore. Any test method can be used as test fixture to set up the instance under test (to simplify matters, however, we'll still call it `setUp` method).

The first step in the migration from JUNIT to JEXAMPLE is thus to make the `setUp` method a conventional test method and to declare it as an explicit dependency for every test method. Hence, we add the `@Given` annotation pointing to the `setUp` method to every test method.

Listing 4.2: Setup Method as Dependency

```
private List<String> list;

@Test
public void setUp(){
  list = new ArrayList<String>();
}

@Test
@Given("#setUp")
public void testSize(){
  assertEquals(0, list.size());
  list.add("Foo");
  assertEquals(1, list.size());
}

@Test
@Given("#setUp")
public void testAdd(){
  assertEquals(0, list.size());
  list.add("Foo");
  assertEquals(1, list.size());
  assertEquals("Foo", list.get(0));
}
```

The only difference between running the test in JUNIT or running it in JEXAMPLE is the number of times the `setUp` method is executed. In JUNIT the `setUp` method is executed before each test method, in our example this means two times. In JEXAMPLE however, the `setUp` method is executed once. After the execution of the `setUp` method the state of the test class instance is cached by the framework. If later a test method depending on the `setUp` method is executed, instead of rerunning the `setUp` method JEXAMPLE gets the state of the test class instance cached after the execution of the `setUp` method and executes the test method on this test case instance.

4.2 Dependency Detection

Next, we want to detect latent dependencies between test methods.

We assume that a test method m_a should depend on a test method m_b, if m_a covers at least the same code as m_b. The execution of method m_a can be skipped, if the framework already knows that m_b fails.

Gaelli *et al.* have shown that a partial order of test methods by means of *coverage sets*, *i.e.,* the set of methods invoked by a test method, helps developers to locate a defect by pointing out the test method with the most specific debugging context [10].

Thus, we propose to migrate JUNIT tests to JEXAMPLE by recovering the coverage set of each test method with dynamic analysis, *i.e.,* running the tests, recording the coverage set of each test and partially ordering the recorded coverage sets. Based on the partially ordered coverage sets we recover coverage dependencies between test methods. If the coverage set of a JUNIT method m_a is found to be a superset of the coverage set of a JUNIT method m_b, then m_a is migrated to a JEXAMPLE method with an `@Given` annotation pointing to m_b.

However, we do not declare every dependency. If a dependency is found to be transitive, *i.e.,* a method m_a depends on m_b and m_c, and m_b also depends on m_c, the transitive dependency of m_a on m_c is not declared because we know that m_a covers m_b and m_b covers m_c.

Considering our example the coverage set of `testSize()` is {`size(), add()`}, the one of `testAdd()` {`size(), add(), get()`}. `testSize()`'s coverage set is thus a subset of `testAdd()`'s coverage set. `testAdd()` should therefore declare an explicit dependency to `testSize()`, as shown in Listing 4.3.

Listing 4.3: Dependencies between Test Methods

```
private List<String> list;

@Test
public void setUp(){
  list = new ArrayList<String>();
}

@Test
public void testSize(){
  assertEquals(0, list.size());
  list.add("Foo");
  assertEquals(1, list.size());
}

@Test
@Given("#testSize")
public void testAdd(){
  assertEquals(0, list.size());
  list.add("Foo");
  assertEquals(1, list.size());
  assertEquals("Foo", list.get(0));
}
```

4.3 Merging the Trees

We now have two dependency trees. One where every method depends on the `setUp` method. And one where every method points to its coverage dependency, detected by recording and analyzing the coverage set of each test method.

As a next step we want to merge these two trees. This means that every test method's `@Given` annotation now declares as first dependency the `setUp` method and as second, third, etc. dependency the test methods that were detected as dependencies during dynamic analysis.

Listing 4.4: Merged Dependency Trees

```
private List<String> list;

@Test
public void setUp(){
  list = new ArrayList<String>();
}

@Test
@Given("#setUp")
public void testSize(){
  assertEquals(0, list.size());
  list.add("Foo");
  assertEquals(1, list.size());
}

@Test
@Given("#setUp;#testSize")
public void testAdd(){
  assertEquals(0, list.size());
  list.add("Foo");
  assertEquals(1, list.size());
  assertEquals("Foo", list.get(0));
}
```

Like that we make sure that not only the `setUp` method is executed before the execution of a test method but also the test methods recovered as dependencies of that test method during dynamic analysis. If a test method fails, JEXAMPLE knows that it can skip all of its dependent methods since they are covering the same code, and therefore also the same defect, as the dependency and hence, will fail too.

Of course a latent dependency for a test method might not have been detected. Then this test method only depends on the `setUp` method.

4.4 Side Effects on Instances of Units Under Test and Test Class Instances

Before we continue with the next steps of the migration we first need to point out some difficulties that arise when making the test fixture a conventional test method and declaring dependencies to other test methods.

When executing a JUNIT test declaring a `setUp` method, *i.e.*, a test fixture, this `setUp` method is executed before the execution of every test method. Usually, an instance of the unit under test (instance under test) is created in the `setUp` method and assigned to a field of the test class so the particular test method can access, modify and test it.

As mentioned earlier, in JEXAMPLE the notion of a `setUp` method does not exist anymore. Any test method can generate a test fixture (to simplify matters we'll call it `setUp` method nevertheless) by either returning the instance under test which is then passed on to all the dependent test methods by the framework or by assigning the instance under test to a field of the test class, like it is usually done in JUNIT tests[1].

Let's consider a situation where a test method transitively depends on the `setUp` method, *i.e.*, it depends on a test method that depends on the `setUp` method. In the example from Listing 4.1 that would look as follows.

Listing 4.5: Transitive dependency on the `setUp` method

```
private List<String> list;

@Test
public void setUp(){
  list = new ArrayList<String>();
}

@Test
@Given("setUp")
public void testSize(){
  assertEquals(0, list.size());
  list.add("Foo");
  assertEquals(1, list.size());
}

@Test
@Given("testSize")
public void testAdd(){
  assertEquals(0, list.size());
  list.add("Foo");
  assertEquals(1, list.size());
  assertEquals("Foo", list.get(0));
}
```

The method `testSize()` will run fine since it directly depends on the `setUp` method and therefore the instance under test is an empty list, as expected by `testSize()`. `testAdd()` however, will fail. `testSize()` is its first and only dependency, hence, `testAdd()` is executed on the instance of the test class cached after the execution of `testSize()`. Since `testSize()` has side effects on the instance under test (it adds an element to the list), the instance under test will not be an empty list when `testAdd()` is executed and `testAdd()` fails.

The framework gets the instance of the test class cached after the execution of the first dependency. To solve the problem mentioned above, we can thus declare a direct dependency of `testAdd()` on the `setUp` method, additional to the one on `testSize()`. Like `testSize()`, `testAdd()` is then executed on the instance of the

[1]Possible return values of test methods and the state of the test class instance after executing a test method are cached by JEXAMPLE. If a test method depending on the `setUp` method, for instance, is about to be executed, the framework gets the instance of the test class cached after the execution of the `setUp` method and executes the test method on that instance. If the test method declares multiple dependencies, the framework gets the instance of the test class after the execution of the first dependency. Furthermore, if the `setUp` method returns the instance under test and the depending test method takes an argument matching the type of the instance under test, the framework gets the cached return value of the `setUp` method and passes it as parameter to the invocation of the test method. Again, if the test method declares multiple dependencies but takes only one argument, the return value of the first dependency is passed to the invocation of the test method.

test class cached after the execution of the `setUp` method that has an empty list as instance under test. `testAdd()` will now run fine.

The same solution also works fine if the instance under test is returned by the dependency methods and taken as argument by the dependent methods instead of being a field of the test class instance.

When automatically transforming a JUNIT test to JEXAMPLE we therefore need to make sure that test methods whose dependencies have side effects on the instance under test declare the `setUp` method as first dependency.

4.5 Removing Setup Dependencies

Once merged the two dependency trees we will have some transitive dependencies on the `setUp` method. If a method m_a depends on the `setUp` method and a method m_b, and m_b also depends on the `setUp` method, m_a might not need to depend on the `setUp` method directly. However, we can only remove the transitive dependency from m_a to the `setUp` method if m_b has no side effects on the instance of the unit under test (instance under test). Otherwise, it is very likely that the unit under test is in an invalid state for test method m_a to pass (see Section 4.4 for further details on side effects on the instance under test and the test class instance respectively).

Considering our example we have a list as unit under test. `testSize()` adds an element to that list to check whether the size of the list changes. The list contains one element after `testSize()` was executed. Since in JUNIT every test method "depends" on the `setUp` method, which in this case initializes an empty list, `testAdd()` expects the instance under test to be an empty list when it is executed. However, if we remove the dependency of `testAdd()` on the `setUp` method `testSize()` is executed before `testAdd()` and the list will already contain an element when `testAdd()` is run and `testAdd()` will fail.

Thus, a direct dependency on the `setUp` method can only be removed if we are sure that all other test methods a method depends on, directly or indirectly, have no side effects on the instance under test. In our example from Listing 4.4 we can therefore not remove the direct dependency of `testAdd()` on the `setUp` method.

4.6 Fixture Injection

Test methods in JEXAMPLE can pass an instance under test from one test method to another. A test method m_b can return a value and the JEXAMPLE framework caches this return value. If later the framework is about to execute a test method m_a that depends on m_b and takes the instance under test as a parameter, the cached return value of m_b

is injected as an argument to the method invocation of m_a. As such, in JEXAMPLE a test method may provide the fixture for its dependent methods.

As a last step in the migration from JUNIT to JEXAMPLE we also want to add fixture injection to our current test case. At the moment this is only possible if the test case has exactly one field that might be the instance under test, i.e. the test case declares exactly one non-constant field. We then assume that this field is the instance under test.

Once the test case is found to be fixture injection-compatible we transform the field to be a local variable in the the `setUp` method. We transform the `setUp` method to return this local instance under test. Then we migrate all test methods to take the instance under test as parameter and to return it as well. See Listing 4.6 for the according changes in our example.

Listing 4.6: Fixture Injection

```
@Test
public List<String> setUp(){
  List<String> list = new ArrayList<String>();
  return list;
}

@Test
@Given("#setUp")
public List<String> testSize(List<String> list){
  assertEquals(0, list.size());
  list.add("Foo");
  assertEquals(1, list.size());
  return list;
}

@Test
@Given("#setUp;#testSize")
public List<String> testAdd(List<String> list){
  assertEquals(0, list.size());
  list.add("Foo");
  assertEquals(1, list.size());
  assertEquals("Foo", list.get(0));
  return list;
}
```

Often it is not enough to only transform the test methods. In some test classes helper methods are implemented that access the instance under test. Since we transformed the instance under test to be a local variable that is passed from test method to test method we also have to adapt the helper methods accessing the instance under test. We have to add a method parameter, corresponding to the instance under test, to the helper methods and to the according method invocations we have to add the instance under test as an argument.

If not removed earlier we can now check again whether we can remove transitive dependencies to the `setUp` method. Therefore we have a closer look at the bodies of a method and its dependent methods. If the first n statements of a dependent method m_a exactly match all the statements of its dependency m_b, *i.e.*, the statements' string representations match, these n statements can be removed from m_a's body (return statements inserted when adding fixture injection to a test are ignored). Then, the transitive dependency to the `setUp` method can be removed as well since side effects on the instance under test in m_b are now expected by m_a. In our case study we had no case where we could remove duplicated statements based on their string representation (see Section 4.7 for a discussion about alternative approaches).

In our example the first three lines of `testAdd()` exactly match the body of `testSize()`, ignoring the return statement. As shown in Listing 4.7 we can remove these three statements from `testAdd()` because we know that they are already covered in `testSize()` and that we get the changed fixture from `testSize()`. We can now also remove the direct dependency to the `setUp` method because `testAdd()` now expects the list to already contain an element.

Listing 4.7: Method Body Transformation

```
@Test
public List<String> setUp(){
  List<String> list = new ArrayList<String>();
  return list;
}

@Test
@Given("#setUp")
public List<String> testSize(List<String> list){
  assertEquals(0, list.size());
  list.add("Foo");
  assertEquals(1, list.size());
  return list;
}

@Test
@Given("#testSize")
public List<String> testAdd(List<String> list){
  assertEquals("Foo", list.get(0));
  return list;
}
```

Our example now looks like we expect a JEXAMPLE test case to look like or rather like we expect a JEXAMPLE test case to be written like.

4.7 Future Work

As mentioned before we could not remove duplicated code by comparing the string representations of the method bodies in the projects of our case study.

As a future work one might consider using object states in a test execution, as introduced by Xie *et al.* [26, 25], to have information about the state of the instance under test after the execution of a test method additionally to the coverage sets of the test methods. With this information we might be able to recover additional dependencies. E.g. if we have two test methods with overlapping coverage sets that cannot be partially ordered definitely, we might be able to determine a dependency direction with the information about the object state of the instance under test. We might even detect dependencies that we are not able to detect at all by means of the coverage sets.

With the additional information about object states in a test execution we might also be able to remove duplicated code in producer–consumer relationships. As mentioned before, we were not able to remove duplicated code in our case study when comparing the string representation of the method bodies. With the information about the object states we could do a more sophisticated comparison of the single statements of the method bodies. Instead of the string representation of two statements we might compare the abstract syntax trees (ASTs) of the statements. When comparing two instances of an object, that are *e.g.,* passed to an assertion, we might compare the object states to determine whether two statements are equivalent or not.

We might also use an approach similar to the one presented by Saff *et al.* [19] to automatically divide long test methods into smaller, more specific ones. This could further improve defect localization because every test method then covers a very specific part of a program.

Chapter 5

JUnit2JExample, a Prototype to Migrate JUnit 4 tests to JExample

In this chapter we report on our prototype JUnit2JExample that automatically migrates JUnit 4 test classes to JExample test classes that declare explicit dependencies between test methods. Compared to Chapter 4 where we conceptually explained the steps of the migration, in this chapter we talk about what technologies we use to do dynamic analysis and program transformation. Furthermore, we comment on certain design decision we have taken.

The prototype consists of two main parts: i) the dynamic analysis part where information about the tests to transform is gathered and possible dependencies between tests are detected, and ii) the program transformation part where the JUnit 4 tests are actually transformed to JExample tests based on the information gathered during dynamic analysis.

5.1 Dynamic Analysis

For the dynamic analysis we execute the test cases specified by the developer. Every class that should be instrumented according to the developer's specification (often developers might not want to instrument classes of external libraries) is instrumented before it is loaded the first time. Then the test cases are run with JUnit and by means of the instrumentation runtime information is collected. If there are failing tests the transformation is aborted. If all tests finish successfully possible dependencies between test methods are searched based on the gathered runtime information. See Figure 5.1

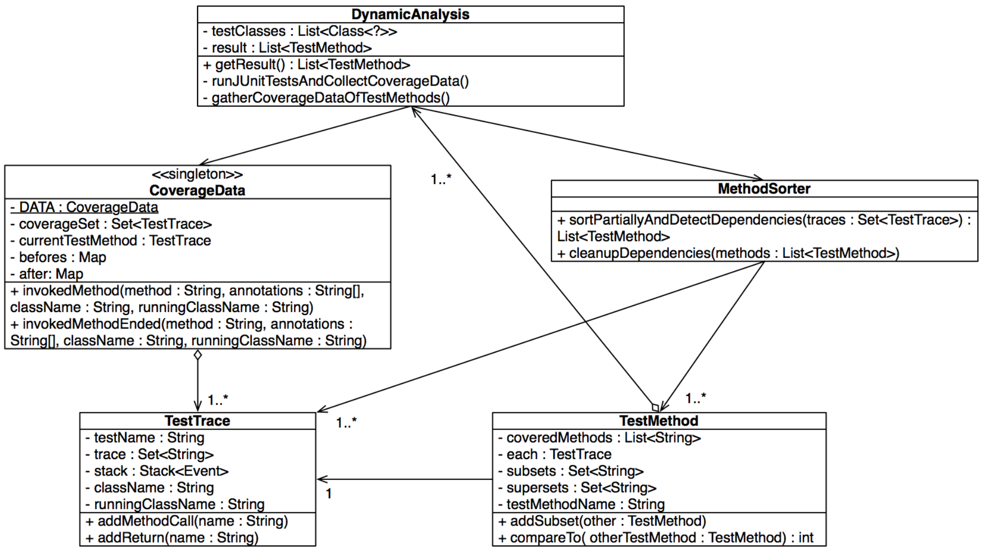
DynamicAnalysis
- testClasses : List<Class<?>>
- result : List<TestMethod>
+ getResult() : List<TestMethod>
- runJUnitTestsAndCollectCoverageData()
- gatherCoverageDataOfTestMethods()
<<singleton>>
CoverageData
- DATA : CoverageData
- coverageSet : Set<TestTrace>
- currentTestMethod : TestTrace
- befores : Map
- after: Map
+ invokedMethod(method : String, annotations : String[], className : String, runningClassName : String)
+ invokedMethodEnded(method : String, annotations : String[], className : String, runningClassName : String)
MethodSorter
+ sortPartiallyAndDetectDependencies(traces : Set<TestTrace>) : List<TestMethod>
+ cleanupDependencies(methods : List<TestMethod>)
1..*
1..*
1..*
1..*
1
TestTrace
- testName : String
- trace : Set<String>
- stack : Stack<Event>
- className : String
- runningClassName : String
+ addMethodCall(name : String)
+ addReturn(name : String)
TestMethod
- coveredMethods : List<String>
- each : TestTrace
- subsets : Set<String>
- supersets : Set<String>
- testMethodName : String
+ addSubset(other : TestMethod)
+ compareTo(otherTestMethod : TestMethod) : int

Figure 5.1: Class Diagram Dynamic Analysis

for an overview of the classes relevant for the dynamic analysis. In the following the different steps are explained more detailed.

5.1.1 Instrumentation

For the instrumentation we decided to use a java agent. A java agent is a pluggable library that intercepts the class loading process and in our case manipulates the bytecode of every class that should be instrumented before it is loaded the first time. The bytecode manipulation is done with JAVASSIST [1]. In every method and constructor we insert a statement before and after the original body. Both statements are method calls to the single instance of `CoverageData`. In the statement before the original body the method `CoverageData.invokedMethod(String, String[], String, String)` is called. After the original body, or rather before every return of the method, the call to `CoverageData.invokedMethodEnded(String, String[], String, String)` is inserted. The passed parameters for both method calls are the fully qualified method name, an array with all the annotations of this method, the name of the declaring class and the name of the currently running class.

[1] a bytecode manipulation framework, http://www.csg.is.titech.ac.jp/~chiba/javassist/

5.1.2 Running the Tests

During the execution of the test cases to be transformed runtime information is collected in the `CoverageData` instance by means of the statements inserted by the instrumentation. For every test method a new `TestTrace` instance is created. In this `TestTrace` we record the instrumented methods that are called during the execution of the current test method. We call the set of all the invoked methods per test method the *coverage set* of this test method.

To make sure that every method terminates at the right time we keep track of the started methods with a stack of `Event`s in the `TestTrace`. If the method name passed from `CoverageData.invokedMethodEnded(String, String[], String, String)` to `TestTrace.addReturn(String)` does not match the current top element of the stack an error is thrown and the transformation is aborted.

If there are errors running the test cases the transformation is aborted.

5.1.3 Detecting Dependencies

First a `TestMethod` instance is created for every `TestTrace` collected by `CoverageData`. For each `TestMethod` all super- and subsets are saved, i.e. `TestMethod`s whose coverage set is a super- or subset respectively of the current test method's coverage set. Then the `TestMethod`s are sorted by means of their coverage sets.

This sorted list of `TestMethod`s is the result of the dynamic analysis. It is passed to the program transformation and used to transform the JUNIT tests into JEXAMPLE tests with explicit dependencies.

5.2 Program Transformation

During program transformation we do a step by step transformation of the original JUNIT 4 test cases to JEXAMPLE test cases. See Figure 5.2 for the important classes composing the program transformation.

First we collect all the declaring classes from the `TestMethod`s in the result list. Then we use the `javax.tools.JavaCompiler` to get the compilation units with the types of these declaring classes.

As a next step we create the `TransformationModel`. In this model we save the superclass, if there is one, the declared fields and the declared methods for each class. For every method we save all the used fields including fields that are declared in a superclass. Once the `TransformationModel` is built we can start transforming the classes. The model makes sure that, if a class has a superclass, this superclass is transformed first.

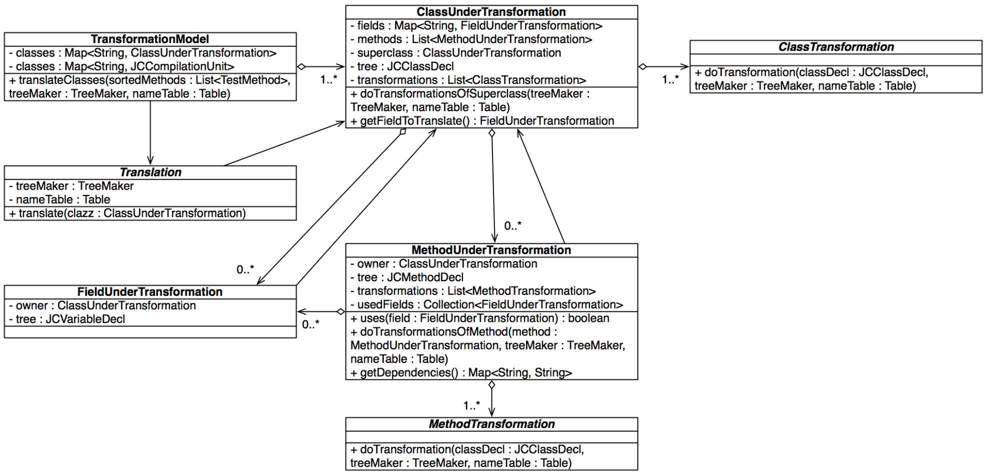
TransformationModel
- classes : Map<String, ClassUnderTransformation>
- classes : Map<String, JCCompilationUnit>
+ translateClasses(sortedMethods : List<TestMethod>, treeMaker : TreeMaker, nameTable : Table)
1..*
ClassUnderTransformation
- fields : Map<String, FieldUnderTransformation>
- methods : List<MethodUnderTransformation>
- superclass : ClassUnderTransformation
- tree : JCClassDecl
- transformations : List<ClassTransformation>
+ doTransformationsOfSuperclass(treeMaker : TreeMaker, nameTable : Table)
+ getFieldToTranslate() : FieldUnderTransformation
1..*
ClassTransformation
+ doTransformation(classDecl : JCClassDecl, treeMaker : TreeMaker, nameTable : Table)
Translation
- treeMaker : TreeMaker
- nameTable : Table
+ translate(clazz : ClassUnderTransformation)
0..*
0..*
FieldUnderTransformation
- owner : ClassUnderTransformation
- tree : JCVariableDecl
0..*
MethodUnderTransformation
- owner : ClassUnderTransformation
- tree : JCMethodDecl
- transformations : List<MethodTransformation>
- usedFields : Collection<FieldUnderTransformation>
+ uses(field : FieldUnderTransformation) : boolean
+ doTransformationsOfMethod(method : MethodUnderTransformation, treeMaker : TreeMaker, nameTable : Table)
+ getDependencies() : Map<String, String>
1..*
MethodTransformation
+ doTransformation(classDecl : JCClassDecl, treeMaker : TreeMaker, nameTable : Table)

Figure 5.2: Class Diagram Program Transformation

When transforming a class all the single transformations are recorded in the appropriate object, either in the object representing the class (`ClassUnderTransformation`) or in the representing a method (`MethodUnderTransformation`).

After transforming the superclasses the transformations applied to the superclasses are also applied to the current class. Only then is the step by step transformation applied to the class to find additional transformations that are needed. With this approach we make sure that if, *e.g.*, a field is used for fixture injection in the superclass it is also used for fixture injection in all the subclasses. If a method m_a overrides a method m_b and m_b is migrated to take the former field as parameter and to return it to its dependent methods, the same transformations have to be applied to m_a in order for it to still override m_b.

The transformation steps can be divided into two types. Those which are based on the slightly different API of JEXAMPLE compared to JUNIT 4 and the ones based on the result of the dynamic analysis. The former include

- the translation of import declarations,
- appending ”JExample” to the class name so the original java file is not overwritten,
- appending ”JExample” to the extended class if this class was transformed as well,
- and adding the annotations `@RunWith(JExample.class)` and `@Injection(InjectionPolicy.DEEP_COPY)`.

`@Injection(InjectionPolicy.DEEP_COPY)` forces the cloning of the return value of producers before they are passed on to the consumers and the cloning of the test class instance with all its fields. If we cannot transform a test class to use fixture injection this behavior makes sure that possible side effects on the test class instance caused by other test methods do not influence the test outcome of a certain test method (see Section 4.4 for more details about side effects on test class instances).

The transformations based on the result of the dynamic analysis are

- the transformation of `setUp` and `tearDown` methods,
- adding the `@Given` annotation with the dependencies to the test methods,
- removing statements in consumer methods that are also executed in their producer,
- transforming a field to a fixture that is injected,
- and at last the organization of the dependencies according to the parameters of the test method, *i.e.,* the order the producer's return types has to match the order of the types of the parameters declared by the dependent method.

In a first step the transformation of `setUp` and `tearDown` methods consists of removing the `@Before` resp. `@After` annotation and adding the `@Test` annotation instead. Later, the `tearDown` method will have to declare dependencies to all leaf methods of the dependency tree.

For the addition of the `@Given` annotation we need the names of the producer methods. By getting the subsets of the `TestMethod` instance corresponding to the current test method we get all the test methods whose coverage sets are subsets of the current test method's coverage set and hence, are coverage dependencies. If there are no subsets the `setUp` methods are inserted as dependencies. If there are no `setUp` methods either, the method has no dependencies and therefore no `@Given` annotation.

`tearDown` methods should be executed at the end of a test case. Hence, we define all test methods without any supersets, i.e. test methods that no other test method depends on, as dependencies. This might not always work. If a `setUp` method prepares external resources, *e.g.,* it opens a database connection, for every execution of the `setUp` method there should be an execution of the `tearDown` method that closes that connection. In our current approach this database connection is cloned if there are multiple methods depending on the `setUp` method. But, the `tearDown` method will only close one of these connections. In our case study, however, we never encountered a case where our approach did not work.

If a `tearDown` method is implemented In JUNIT, this `tearDown` method is executed after the execution of every test method. Hence,

When the dependencies are set we check whether we can remove redundant statements at the beginning of consumer methods (dependent methods). The statements of the consumer have to cover all statements of the producer (dependency method). Additionally, there must not be any creation of a local variable in these statements, otherwise, we might have accesses to undefined variables after removing the statements from the consumer's body. If these conditions hold the statements already executed by the producer can be removed from the consumer's body.

The next step is to use a field for fixture injection. The condition for a field to be used for fixture injection is that the field has to be the only transformable field in this class, i.e. the only field that is not a constant. The reason for this is simple. We can only return one object per method and if there are multiple transformable fields we don't know which one is the fixture that should be passed on among the methods. However, if there is only one transformable field, we assume that this is the instance of the unit under test and that we can use it for fixture injection.

If there is a field we can use for fixture injection we first have to check whether this field was initialized in the field declaration. If this is the case and there is at least one `setUp` method we move the initialization to the `setUp` method. The field is now declared and initialized as local variable in the `setUp` method. If there is no `setUp` method we cannot transform the field.

The next step is to transform all the methods accessing the former field to take the former field as a parameter. This is done for all methods but the `setUp` methods, *i.e.,* test methods and helper methods. Since `setUp` methods are always run first and should never be called by another method they cannot get the field as a parameter.

Test and `setUp` methods additionally get a statement returning the now local variable, however, only if they take a corresponding parameter (test methods) or have a local variable declaration initializing the former field. (`setUp` methods).

As a last step of the field transformation, invocations of methods using the former field have to be changed to pass the former field as an argument to the methods.

The last transformation is the organization of the dependencies. For test methods taking parameters the dependencies are ordered in a way that the return types of the producers correspond to the types of the parameters taken by the consumer. If they do not match, JEXAMPLE will throw a corresponding error when running the tests.

5.3 Limitations

The current implementation of JUNIT2JEXAMPLE still has some limitations. Due to these limitations not every java project can be transformed without manual modification of the test classes. The first of these limitations can even impede the migration of whole projects.

JUNIT2JEXAMPLE cannot handle concurrent threads during dynamic analysis. When recording the coverage sets JUNIT2JEXAMPLE pushes every method whose execution started on a stack. When the return of a method is recorded it checks whether the returning method is on top of the stack and aborts the migration if this is not the case. With multiple threads this is not necessarily the case. Due to this issue we had to ignore some projects that we would have used for our case study otherwise (Chapter 6).

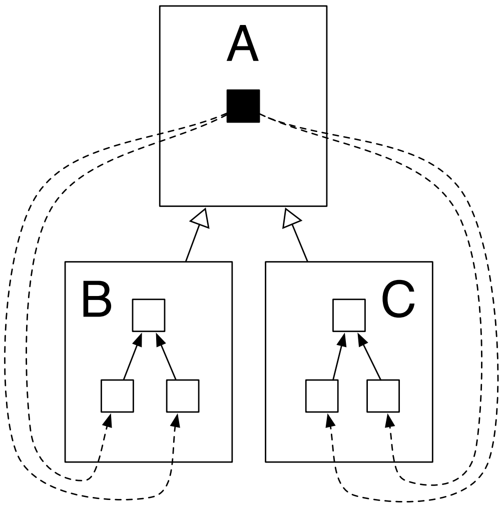

Figure 5.3: Inheritance in tests with the `tearDown` method in the root class. (A, B and C: test classes, black square: `tearDown` method, white squares: test methods)

Another limitation is that it cannot handle `tearDown` methods that are declared in a superclass. Often `tearDown` methods are implemented in the root class of a test hierarchy. In JEXAMPLE, however, the `tearDown` method is supposed to be the leaf of the whole generated dependency graph since it is the last method to be executed. See an example scenario in Figure 5.3. If the `tearDown` method is implemented in the root class of a hierarchy it would have to depend on all the leaf methods in all the subclasses (dashed arrows). When running only test class C with these dependencies JEXAMPLE would also run the entire test class B because in order for the `tearDown` method to be run all of its dependencies have to be run. For this reason JUNIT2JEXAMPLE does not insert dependencies to methods in subclasses. However, if the `tearDown` method has no dependency at all (since we do not insert dependencies to methods in subclasses) we cannot control at which point in the execution of the test class it is run. If it is run in the middle of a test class it might, *e.g.,* delete a temporary file that is still needed in other test methods.

The solution to this problem at the moment is to push down the `tearDown` method to the leaf classes of a hierarchy by hand. Like this it will only depend on the leaf methods of the class it is declared in and you can be sure that it is run as the last test method.

Chapter 6

Case Study

As mentioned earlier, our thesis is that we can shift the burden of defect isolation from the developer to the framework by automatically recovering implicit dependencies between test methods and declare them explicitly.

By means of dynamic analysis we can record the coverage sets of all test methods and by means of a partial order of these coverage sets we can recover implicit dependencies between test methods. The recovered dependencies can then be used to automatically migrate the original JUNIT test classes to JEXAMPLE test classes that declare the recovered dependencies explicitly.

We expect the automatically generated JEXAMPLE tests to have better defect isolation. Since test methods depending on a failing test are ignored, ideally only one test fails per defect. Test methods covering the same defect should depend on the failing test method and thus be ignored.

Moreover, we expect the JEXAMPLE tests to perform better in the matter of execution time since the unit under test is instantiated only once in JEXAMPLE. The evolving instance of the unit under test is cached by the framework and passed down from consumers to producers. JUNIT on the other side initializes the instance of the unit under test for each test method.

In order to further investigate our thesis, we selected a sample of 16 projects for closer inspection. With JUNIT2JEXAMPLE we automatically transformed the original JUNIT tests to JEXAMPLE tests and compared them to each other by means of the following criteria.

- *Detected Dependencies.* We plotted the dependency graph of the automatically migrated JEXAMPLE tests and extracted information about the connected components, length of the dependency paths, fan ins and outs of the nodes, number of single nodes and percentage of nodes being part of a producer–consumer relationship. We also looked at correlations among these characteristics and between these characteristics and project properties such as number of classes and number of test classes.

- *Performance.* We measured and compared the execution time for each project's JUnit 4 tests and JExample tests. We executed the JExample tests with two different injection policies, once with the rerun policy and once with the clone policy. (See Subsection 2.1.1 for details about the different injection policies.)
- *Defect Isolation.* With error seeding we inserted defects into the application code. Per defect we counted the number of failed test methods. In the case of the JExample tests we also counted the number of ignored test methods. We then compared the number of tests failing in JUnit with the number of tests failing in JExample.

6.0.1 Project Selection

We looked at the code search database Sourcerer [16, 1] to find projects suitable for our case study.

In a first step we considered suitable all those projects that are covered by test suites. Of those we picked out projects with different ratios of test classes compared to the total number of classes. See Figure 6.1 for the ratios of test classes compared to the total number of classes for all the projects with at least one test class. The square markers indicate the 16 finally selected projects [13].

The next criterion was whether the projects could be installed and compiled in a reasonable time. Unfortunately, a lot of projects are shipped without the referenced libraries. If there is no maven or ant file to get the libraries and build the project, collecting all the right libraries in the right version takes a considerable amount of time.

The last and nearly most important criterion was whether the unit tests were green, *i.e.,* all test methods terminated successfully. We did not consider projects with any failing tests.

In the beginning we only looked at projects with JUnit 4 tests. But since only 10% of the projects covered by test suites had JUnit 4 tests we decided to additionally implement an automatic JUnit 3 to JUnit 4 conversion (see Chapter 3 for details) and to also consider projects with JUnit 3 tests.

In the final selection we also have three projects where the original unit tests were written for TestNG [2]. TestNG is also based on annotations and also allows developers to declare dependencies between test methods. However, in these three projects no dependencies were declared. Therefore, we could easily replace the annotations with the according JUnit 4 annotations by hand to get JUnit 4 tests.

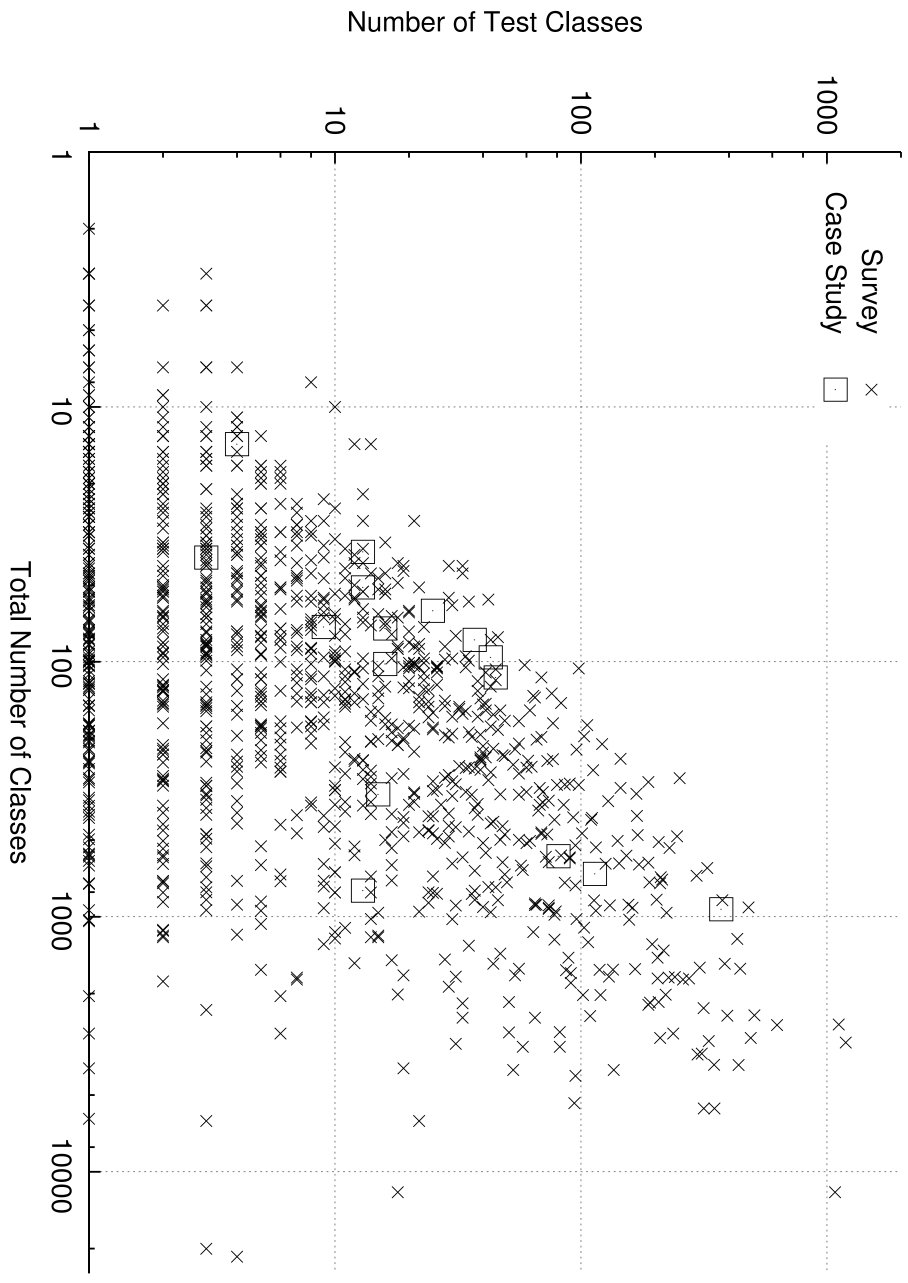

Figure 6.1: Distribution of test classes compared to total project size (both axes are logarithmic, square markers indicate those projects selected for closer inspection through dynamic analysis).

Data cleansing

Having found 16 projects that all match the criteria mentioned above, for some projects, we still had to make some modifications to the source code of the tests in order to get a satisfying result, *i.e.*, compiling and successfully terminating JEXAMPLE tests, from the API migration from JUNIT to JEXAMPLE.

One of the most common problems was the use of inheritance in the test classes. When inspecting the projects in SOURCERER we found that inheritance between test classes is commonly used [13]. By default all JUNIT 3 test classes use a 1 level inheritance since they all have to extend `junit.framework.TestCase`. But still one third of all projects use 2 levels of inheritance of which another third uses 3 levels of inheritance and so on. The inheritance level increases beyond 6 levels for some projects.

Inheritance in unit tests can impose several problems for the automatic transformation from JUNIT to JEXAMPLE.

A problem we encountered with inheritance is the existence of classes in the middle of a test class hierarchy that are not really test classes, *i.e.*, classes that do not contain any `setUp`, `tearDown` or test method. In the step of the dynamic analysis we save the execution trace for each test method and the declaring and the running class for each `setUp`, `tearDown` and test method. Later in the program transformation we transform all the classes that were recorded during the dynamic analysis. Hence, classes that do not contain a `setUp`, `tearDown` or test method are not transformed. These classes might however contain helper methods that need to be transformed as well if the instance variable of a superclass is used for fixture injection (see Section 4.6).

This problem can be solved by overriding a `setUp` or test method that just calls itself on the superclass in the according class. In the projects of the case study we've always overridden the `setUp` method if this problem occurred.

Due to a limitation of the implementation of JUNIT2JEXAMPLE we had problems to correctly transform test hierarchies that declared a `tearDown` method in the root class of the hierarchy. Such problems, however, could be solved easily by hand (see Section 5.3 for more details).

Another problem was that often helper or dummy classes were declared in the same compilation unit as the test class. JUNIT2JEXAMPLE creates a new compilation unit for the migrated test class in the same package as the original compilation unit. This results in compilation errors because we then have multiple classes with the same name in one package.

For the moment such helper or dummy classes are not renamed automatically, we did it by hand after the transformation was finished. In a future version this could be done automatically.

Also, JUNIT2JEXAMPLE cannot handle concurrency at the moment. Due to that limitation we had to ignore several, otherwise fitting, projects (see Section 5.3 for more details).

We encountered some more project specific problems that we solved by hand. In one project some test classes implemented a listener interface that was broken after the transformation because helper methods declared by the interface were transformed. Since there were only a few classes implementing that interface we just didn't transform them.

Finally, in one project we had to ignore one test class that wouldn't run successfully anymore after the transformation to JEXAMPLE. Another test class we ignored because it accessed an URL with a restricted number of accesses per day.

Despite the encountered difficulties, of a total of 836 test classes we could migrate 98% to JEXAMPLE without manual intervention.

6.0.2 Results

Detected Dependencies

We used dynamic analysis to recover the latent dependencies between test methods. A latent dependency is given if a test method covers a superset of another test method [10]. Overlapping coverage sets impact defect localization: a single defect might cause multiple failing tests if the defect is covered by multiple test methods.

For each project, we recovered the dependency graph of the test methods. A dependency between method m_a and method m_b is given, if method m_a covers a superset of m_b's coverage set. We found latent dependencies between 72% of all test methods. Table 6.1 lists the detected dependencies. For each project we list the total number of nodes (*i.e.,* test methods), the number of singular nodes, and the number of connected components (*i.e.,* clusters of nodes with dependencies). For the connected components we list maximum and median of number of nodes, length of longest dependency path, and largest dependency fan in/out.

The projects in the sample fall into two groups. There are 6 projects with fewer than 30% connected nodes (*i.e.,* test methods with latent dependencies) and thus well isolated test methods. However, 10 projects showed a high ratio of latent dependencies with more than 70% connected nodes, which might impact defect localization. We found only small correlation between the ratio of connected nodes and project size ($r = 0.35$).

We found a large correlation between the longest dependency path (Len$_{max}$) and ratio of connected nodes (Rel) ($r = 0.81$), as well as between Len$_{max}$ and the largest fan in/out (Fan$_{max}$) ($r = 0.74$). This means that dependency trees grow in length as well as width the more nodes they contain. The correlation with the longest dependency path indicates that there are *Chained Tests* [15], *i.e.,* chains of test methods with each method covering an extended fixture of the same unit under test. The correlation with the fan in/out is an artifact of JUNIT's `setUp` method idiom: typically the node with the largest fan is either a former `setUp` or `tearDown` method. Often in these cases we

Name	NOCl	NOTCl	NCC	NN	NS	Rel	$\text{CC}_{max/med}$	$\text{Len}_{max/med}$	$\text{Fan}_{max/med}$
choco-cp*	679	114	81	785	153	81%	63 / 1	4 / 0	61 / 0
choco-kernel*	331	15	12	68	8	88%	15 / 3	4 / 2	13 / 1
choco-old_version	578	81	62	426	114	73%	19 / 1	5 / 0	17 / 0
dcm4che-cda	51	13	0	13	13	0%	1 / 1	0 / 0	0 / 0
dcm4che-core	74	16	3	37	30	19%	3 / 1	1 / 0	2 / 0
dcm4che-hp	39	3	1	11	9	18%	2 / 1	1 / 0	1 / 0
dcm4che-imageio	14	4	4	13	4	69%	3 / 2	2 / 1	1 / 1
findbugs	790	13	10	72	11	85%	17 / 1	3 / 0	16 / 0
jfreechart	933	372	273	1986	529	73%	33 / 1	5 / 0	31 / 0
maven-artifact	73	9	5	63	6	92%	35 / 2	8 / 1	33 / 1
maven-artifact-manager	37	13	4	24	0	100%	15 / 3	3 / 2	13 / 2
maven-model	82	37	7	150	125	17%	4 / 1	2 / 0	3 / 0
maven-project	96	43	35	266	24	91%	44 / 3	5 / 2	40 / 1
miscela*	102	16	13	153	2	99%	50 / 6	6 / 2	49 / 5
uncommons-math*	63	25	6	143	104	27%	12 / 1	2 / 0	10 / 0
watchmaker*	115	45	5	116	96	17%	8 / 1	1 / 0	7 / 0
Total	4057	819	521	4325	1227	72%			

Table 6.1: Results of coverage analysis. (NOCl: number of classes, NOTCl: number of test classes, NCC: number of connected components, NN: number of nodes, NS: number of single nodes, Rel: percentage of nodes with dependencies, CC: size of connected component, Len: length of dependency path, Fan: size of fan in or fan out; *: projects with JUnit 4 or TestNG tests)

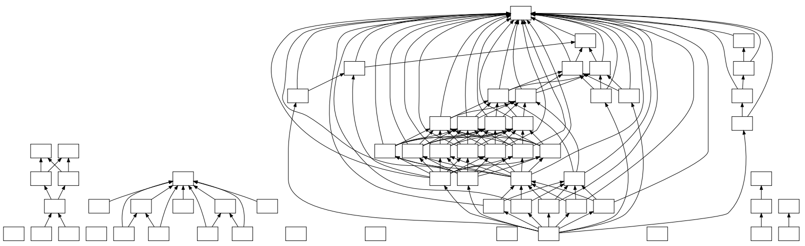

Figure 6.2: Dependency graph of the automatically migrated `maven-artifact` project; the large connected component consists of 35 test methods.

had not been able to remove transient dependencies (and thus limit the fan of these method) because of side-effects introduced by instance variables. We expect to see smaller fan in/out counts in test suites written first hand for JExample compared to these migrated test suites.

Figure 6.2 shows the dependency graph of the `maven-artifact` project, including a connected component with large depth and width. The graph shows five connected components and five single nodes, *i.e.,* methods without dependencies. The largest connected component consists of 35 nodes and the largest fan in of a node is 33. Looking at the largest connected component in the graph we see all methods but one pointing to the one on top. This is the former `setUp` method. The method not pointing to the former `setUp` method is the former `tearDown` method.

In all projects, the number of connected components (single nodes are not counted as connected components) is smaller than the number of test classes. This can happen for two reasons: 1) the dependencies go beyond class boundaries. In this case either the single test classes are not isolated, *i.e.,* multiple test classes test the same unit under test, or the test classes use inheritance and therefore dependencies to methods in the superclass are very probable; 2) there are test classes containing only methods with no dependencies. Some of the bigger projects use inheritance in their test classes. And, in big as well as in small projects, we find test classes that contain only methods with no dependencies.

We found that only a small correlation exists between the number of connected components (NCC) and the resulting dependency tree ($r = 0.11$), and the number of single nodes (NS) and the resulting dependency tree ($r = -0.06$). NCC and NS therefore have no influence on how many dependencies are found, and therefore no influence on the size of the connected components, the length of the dependency sequences and the size of the fan ins and outs.

We found that between the number of classes (NOCl) and the largest connected component the correlation is small ($r = 0.40$). Furthermore, there is also a small correlation between NOCl and the median size of the connected components ($r = -0.28$). So, if a project has a lot of classes, this does not necessarily mean that there are big test classes with a great number of implicit dependencies that result in larger connected components. Nor does it mean that there are more dependencies beyond test class borders than in smaller projects. Hence, we can say that also in projects with a great number of classes the test classes are well isolated.

Performance

Table 6.2 lists the execution time for the two test implementations whereas the JEXAMPLE tests are executed twice, once with the cloning and once with the rerun policy. The execution time is measured with the UNIX *time* command on an Intel Core 2 Duo, 2.4 GHz, 4GB RAM, Sun JDK 1.6.0, JUnit 4.3.1.

Additionally, Table 6.2 shows how much time was spent with cloning when running the JEXAMPLE tests with the cloning policy.

We observe that for the majority of the projects one of the JEXAMPLE test runs was the fastest. However, when looking at the time the automatically transformed JEXAMPLE tests are not significantly faster than the original JUNIT tests.

When running the JEXAMPLE tests with the rerun injection policy we expected about the same execution time as the execution time of the JUNIT tests. In JUNIT the unit under test is newly instantiated for every test method. With the rerun injection policy JEXAMPLE also reruns every dependency of the test method that is about to be executed.

However, we assumed that the JEXAMPLE tests executed with the clone injection policy will run faster because the unit under test in JEXAMPLE is instantiated once and is then cached by the framework. The cached instance of the unit under test is cloned before a test method reuses it. Since the tests we looked at often used large test fixtures, cloning the test class instance or the instance of the unit under test needed about the same time

Name	JU	Re	Cl	CT	Re/JU	Cl/JU	CT/JU	Fastest
choco-cp	267.86	336.01	390.79	84.55	125.44%	145.89%	31.56%	junit
choco-kernel	0.69	0.67	1.24	0.59	97.10%	179.71%	85.92%	jexample
choco-old_version	73.89	72.19	75.25	0.37	97.70%	101.84%	0.50%	jexample
dcm4che-cda	0.34	0.4	0.41	0.0009	117.65%	120.59%	0.26%	junit
dcm4che-core	3.42	3.59	3.28	0.0034	104.97%	95.91%	0.10%	jexample
dcm4che-hp	1.12	1.13	1.2	0.0015	100.89%	107.14%	0.13%	junit
dcm4che-imageio	5.62	7.01	5.54	0.0030	124.73%	98.58%	0.05%	jexample
findbugs	0.61	0.52	0.6	0.06	85.25%	98.36%	10.15%	jexample
jfreechart	21.22	21.02	21.49	0.37	99.06%	101.27%	1.74%	jexample
maven-artifact	2.16	2.32	1.68	0.36	107.41%	77.78%	16.63%	jexample
maven-artifact-manager	2.27	2.72	2.43	0.48	119.82%	107.05%	21.24%	junit
maven-model	0.71	0.48	0.5	0.0022	67.61%	70.42%	0.31%	jexample
maven-project	9.01	12.94	9.69	1.55	143.62%	107.55%	17.24%	junit
miscela	2.98	3.07	2.4	0.54	103.02%	80.54%	18.27%	jexample
uncommon-math	3.24	2.7	2.77	0.01	83.33%	85.49%	0.20%	jexample
watchmaker	1.87	1.69	1.75	0.01	90.37%	93.58%	0.46%	jexample

Table 6.2: Execution time (JU: execution time of JUNIT tests, Re:execution time of JEXAMPLE tests run with rerun injection policy ,Cl: execution time of JEXAMPLE tests run with deep clone injection policy, CT: time spent cloning when running JEXAMPLE tests with deep clone injection policy)

as newly generating it. Hence, the automatically generated JEXAMPLE tests had no significantly faster execution time.

Defect Isolation

We automatically inserted defects into the application code of the projects under study to get information about the defect isolation of the tests. A defect is well isolated if, after the insertion of the defect, only few tests fail. In the best case only one test method fails.

Table 6.3 lists the results of the error seeding. For each project we measured the percentage of defect isolation for both the original JUNIT tests and the migrated JEXAMPLE tests. 100% defect isolation would mean that for every inserted defect only one test method fails. Additionally, we computed the average number of failures per defect and the maximum number of failures over all defects for the JUNIT and JEXAMPLE tests of each project. The table also shows the square of the average number of failures because this number gives more weight to large numbers.

When running the JEXAMPLE tests there were still multiple failing test methods per defect for some defects. Since in JEXAMPLE methods depending on a failing method are ignored, a defect should only cause one test method to fail. All other test methods covering the same defect should depend on the failing test and thus be ignored. The fact that there is no 100% defect isolation in the migrated JEXAMPLE tests indicates that we did not recover all dependencies in a test case by solely looking at the partial order of the coverage sets of the test methods. We lack *e.g.,* the latent dependencies when two coverage sets intersect but are not nested. In this case we don't know which test method should be the producer and which the consumer.

Name	1-Rel	DI_U	DI_E	avg_U	avg_E	avg^2_U	avg^2_E	max_U	max_E
dcm4che-cda	100%	34%	33%	3.82	3.82	29.29	29.30	13	13
dcm4che-core	81%	35%	37%	3.67	2.92	22.82	13.60	15	11
dcm4che-hp	82%	28%	28%	2.99	2.99	11.32	11.26	7	7
maven-artifact	8%	34%	44%	9.22	2.59	219.27	10.72	56	10
maven-artifact-manager	0%	65%	56%	1.72	1.83	5.71	5.08	14	9
maven-model	83%	72%	78%	1.43	1.25	2.86	1.82	5	3
uncommons-math	73%	28%	34%	6.74	5.74	139.55	106.54	41	39

Table 6.3: Defect isolation results of error seeding. (1-Rel: percentage of nodes not being part of dependencies, DI: percentage of defects that cause only one failure, avg: average failures per defect, avg^2: average failures squared per defect, max: maximum failures; U: JUnit, E: JExample)

We found that there is only a small correlation between the percentage of methods not being part of dependencies and the percentage of defect isolation ($r = -0.23$ for JUnit and $r = -0.24$ for JExample). That is, the dependencies we recovered by partially ordering the coverage sets do not significantly improve defect isolation in the JExample tests. Also we cannot say that the fewer dependencies are found the better dependency isolation is in the original JUnit tests.

We found a medium correlation between 1-Rel and the ratio of avg_U and avg_E ($r = -0.58$). Hence, we can say that for half of the projects the more methods without dependencies were found the smaller is the difference between avg_U and avg_E, that is, for half of the projects the recovered and introduced dependencies improve the defect isolation in the JExample tests.

We measured the change of defect isolation from JUnit to JExample as ratio of the average square of failures per defect in the JUnit tests and the JExample tests repsectively (avg^2_U and avg^2_E in Table 6.3). We used the average square of failures per defect to emphasize substantial changes in the defect isolation.

Over all projects defect isolation in the JExample tests improved by a factor of 3.77 (the average of the ratio of avg^2_U and avg^2_E).

Chapter 7

Conclusion

In this work we presented the thesis that we can shift the burden of isolating defects in unit tests from developer to framework by automatically recovering and declaring dependencies between test methods. By gathering and partially ordering the coverage sets of test methods we expected to be able to automatically recover latent dependencies between test methods and to automatically rewrite the original JUnit 4 tests as JExample tests that explicitly declare the dependencies recovered during dynamic analysis.

Our assumption that latent dependencies can be recovered by looking at the coverage sets of test methods was based on previous research by Gaelli *et al.* [11]. They propose that most tests either cover a superset of another test method's coverage or a subset of another test method, concluding that most test methods implicitly depend on other test methods. With independent test methods sharing only a test fixture (as the xUnit family of testing frameworks advises to write tests), every test method has to modify the instance of the unit under test in order to assert certain facts about it. This unavoidably leads to overlapping coverage sets of test methods.

We thus expected to be able to recover latent dependencies between test methods by partially ordering the test methods by means of their coverage sets, gathered with dynamic analysis. By rewriting the original JUnit 4 tests as JExample tests with explicitly declared dependencies between test methods we expected to get tests with an improved defect isolation. In JExample test methods that depend on a failing test are ignored because the framework assumes that they will fail as well. In the transformed tests we expect test methods covering a common subset to depend on each other. Hence, a defect should only cause one test to fail since all tests covering the same defect depend on the failing test.

We also expected the tests to perform better in the matter of execution time. Compared to JUnit, in JExample the unit under test is instantiated once, when the first test method of a test class is executed. The instance of the unit under test is cached and reused by the framework whereas JUnit creates a new instance for every test method execution.

We verified that thesis by implementing a tool, JUnit2JExample, for the automatic migration from JUnit 4 tests to JExample. However, other than we expected, almost 90% of the open source java project we looked at used JUnit 3 instead of JUnit 4, the new version of the framework [13]. Therefore, we decided to additionally implement a tool, JUnit3To4, to automatically migrate JUnit 3 tests to JUnit 4 tests. JUnit3To4 uses static analysis of the AST to analyze the JUnit 3 tests and transform them to JUnit 4 tests.

For a case study we selected a sample of 16 open source projects who's tests we automatically transformed to JExample tests. We found that dynamic analysis of the 16 projects revealed 72% latent dependencies, despite the fact that test methods are supposed to be independent artifacts. This confirms the conclusion of Gaelli *et al.* that most test methods implicitly depend on other test methods.

Comparing the two test suites by means of the criteria of execution time and defect isolation we could mostly verify our thesis. Even though most of the test suites ran slightly faster with JExample, we could not find a significant improvement of performance in the matter of execution time. In terms of defect isolation (measured as average square of failures per defect) however, we found that it improved by factor of 3.77 or higher.

This leads to the conclusion that the guideline of the xUnit family of testing frameworks to avoid dependencies is not necessarily realizable. The fact that in JUnit tests the instance of the unit under test is recreated for every test method and thus, every test method has to modify this instance under test so it can then test the actual thing to test already implied this conclusion that can now be confirmed with numbers.

We take the results presented in this paper as an indication that further exploration of the design space of test frameworks is worthwhile.

Appendix A

Quick Start

A.1 JUNIT3TO4

In the following you find a short tutorial on how to install and use JUNIT3TO4 to migrate JUNIT 3 tests to JUNIT 4 tests on the command line and in eclipse.

Download the JUNIT3TO4 jar from `http://scg.unibe.ch/download/jexample`.

Arguments:

- `--input, -i`: The path to the java file of the test class to be converted or the path to the folder containing test classes to be converted.
- `--output, -o`: The path to the directory where the transformed classes should be written to.
- `-r`: If the specified input path points to a directory and if this flag is set test classes to be transformed in this directory are collected recursively. If the flag is not set only test classes directly residing in the directory specified as input are transformed.

A.1.1 Usage on the Command line

```
java -cp path/to/junit3to4.jar:your_class_path ch.unibe.junit3to4.Converter \
      --input <input>\
      --output <output>\
      [-r]
```

A.1.2 Usage in Eclipse

1. Add the downloaded jar to the classpath of the project the test classes to be transformed belong to.
2. Open the run configurations dialog and add a new run configuration for a java application.
3. In the main tab: select as project the project the test classes to be transformed belong to and
4. enter ch.unibe.junit3to4.Converter as main class.
5. In the arguments tab: add `--input <input> --output <output> [-r]` to the program arguments.
6. Run the created run configuration.

A.2 JUNIT2JEXAMPLE

In the following you find a short tutorial on how to install and use JUNIT2JEXAMPLE to migrate JUNIT 4 tests to JEXAMPLE tests on the command line and in eclipse.

Download the JUNIT2JEXAMPLE jar from
`http://scg.unibe.ch/download/jexample`.

Arguments:

- `--testClass, -t`: The fully qualified class name of the test class to be migrated or the package prefix for all test classes to be migrated.
- `--sources, -s`: The path to the location of the source files of the test classes to be migrated.
- `--output, -o`: The path to the directory where the transformed classes should be written to. If this argument is not set the migrated test classes are written to the directory specified in –sources.
- `-r`: If the specified testClass is a package prefix and if this flag is set all test classes with this package prefix are migrated. If the flag is not set only test classes with exactly that package name are migrated.
- `-javaagent`: The path to the agent jar file. If the JUNIT2JEXAMPLE jar is in the classpath this is just "agent.jar". As argument to the agent jar you have to pass the package prefixes that are to be instrumented. Multiple packages are separated with a colon. E.g., `-javaagent:agent.jar=pkg1:pkg2`.

A.2.1 Usage on the Command line

```
java -javaagent:agent.jar=<package>\
      -cp path/to/junit2jexample.jar:your_class_path\
      ch.unibe.junit2jexample.main.JUnit2JExample \
      --testClass <testClass>\
      --sources <sources>\
      --output <output>\
      [-r]
```

A.2.2 Usage in Eclipse

1. Add the downloaded jar to the classpath of the project the test classes to be transformed belong to.
2. Open the run configurations dialog and add a new run configuration for a java application.
3. In the main tab: select as project the project the test classes to be transformed belong to and
4. enter ch.unibe.junit2jexample.main.JUnit2JExample as main class.
5. In the arguments tab: add `--testClass <testClass> --sources <sources> --output <output> [-r]` to the program arguments.
6. In the arguments tab: add `-javaagent:agent.jar=<package>` to the VM arguments.
7. Run the created run configuration.

List of Figures

Listings

List of Tables

Bibliography

[1] Sushil Bajracharya, Joel Ossher, and Cristina Lopes. Sourcerer: An internet-scale software repository. *Search-Driven Development-Users, Infrastructure, Tools and Evaluation, ICSE Workshop on*, 0:1–4, 2009.

[2] Cédric Beust and Hani Suleiman. *Next Generation Java Testing: TestNG and Advanced Concepts*. Addison-Wesley Professional, 2007.

[3] Andrei Z. Broder. On the resemblance and containment of documents. In *In Compression and Complexity of Sequences (SEQUENCES'97)*, pages 21–29. IEEE Computer Society, 1997.

[4] Barthélémy Dagenais and Martin P. Robillard. Recommending adaptive changes for framework evolution. In *ICSE '08: Proceedings of the 30th international conference on Software engineering*, pages 481–490, New York, NY, USA, 2008. ACM.

[5] Arie van Deursen, Leon Moonen, Alex van den Bergh, and Gerard Kok. Refactoring test code. In M. Marchesi, editor, *Proceedings of the 2nd International Conference on Extreme Programming and Flexible Processes (XP2001)*, pages 92–95. University of Cagliari, 2001.

[6] Danny Dig, Can Comertoglu, Darko Marinov, and Ralph Johnson. Automated detection of refactorings in evolving components. In *ECOOP*, pages 404–428, 2006.

[7] David Erni. The hacker's guide to Javac. Bachelor's thesis, supplementary documentation, University of Bern, August 2008.

[8] M. Fewster and D. Graham. *Software Test Automation*, chapter 7: Building maintainable tests. ACM Press, 1999.

[9] Markus Gaelli. *Modeling Examples to Test and Understand Software*. PhD thesis, University of Bern, November 2006.

[10] Markus Gaelli, Michele Lanza, Oscar Nierstrasz, and Roel Wuyts. Ordering broken unit tests for focused debugging. In *20th International Conference on Software Maintenance (ICSM 2004)*, pages 114–123, 2004.

[11] Markus Gaelli, Oscar Nierstrasz, and Stéphane Ducasse. One-method commands: Linking methods and their tests. In *OOPSLA Workshop on Revival of Dynamic Languages*, October 2004.

[12] Lea Haensenberger. JExample. Bachelor's project, University of Bern, March 2008.

[13] Adrian Kuhn, Lea Hänsenberger, and Oscar Nierstrasz. Shifting the burden of unit test isolation from the developer to the framework. Under submission to ICSE 2010.

[14] Adrian Kuhn, Bart Van Rompaey, Lea Haensenberger, Oscar Nierstrasz, Serge Demeyer, Markus Gaelli, and Koenraad Van Leemput. JExample: Exploiting dependencies between tests to improve defect localization. In P. Abrahamsson, editor, *Extreme Programming and Agile Processes in Software Engineering, 9th International Conference, XP 2008*, Lecture Notes in Computer Science, pages 73–82. Springer, 2008.

[15] Gerard Meszaros. *XUnit Test Patterns – Refactoring Test Code*. Addison Wesley, June 2007.

[16] Joel Ossher, Sushil Bajracharya, Erik Linstead, Pierre Baldi, and Cristina Lopes. Sourcererdb: An aggregated repository of statically analyzed and cross-linked open source java projects. *Mining Software Repositories, International Workshop on*, 0:183–186, 2009.

[17] Stefan Roock and Andreas Havenstein. Refactoring tags for automatic refactoring of framework-dependent applications. In *Proceedings of Extreme Programming Conference'02*, pages 182–185, 2002.

[18] Gregg Rothermel, Roland J. Untch, and Chengyun Chu. Prioritizing test cases for regression testing. *IEEE Trans. Softw. Eng.*, 27(10):929–948, 2001.

[19] David Saff, Shay Artzi, Jeff H. Perkins, and Michael D. Ernst. Automatic test factoring for Java. In *ASE 2005: Proceedings of the 20th Annual International Conference on Automated Software Engineering*, pages 114–123, Long Beach, CA, USA, November 9–11, 2005.

[20] Adam M. Smith, Joshua Geiger, Gregory M. Kapfhammer, and Mary L. Soffa. Test suite reduction and prioritization with call trees. In *ASE '07: Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, pages 539–540, New York, NY, USA, 2007. ACM.

[21] Wesley Tansey and Eli Tilevich. Annotation refactoring: inferring upgrade transformations for legacy applications. In *OOPSLA '08: Proceedings of the 23rd ACM SIGPLAN conference on Object-oriented programming systems languages and applications*, pages 295–312, New York, NY, USA, 2008. ACM.

[22] Tom Tourwé and Tom 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.

[23] Eelco Visser. A survey of strategies in rule-based program transformation systems. *Journal of Symbolic Computation*, 40(1):831–873, 2005. Special issue on Reduction Strategies in Rewriting and Programming.

[24] Roel Wuyts. *A Logic Meta-Programming Approach to Support the Co-Evolution of Object-Oriented Design and Implementation*. PhD thesis, Vrije Universiteit Brussel, 2001.

[25] Tao Xie. Augmenting automatically generated unit-test suites with regression oracle checking. In *ECOOP 2006: Object-Oriented Programming*, pages 380–403, 2006.

[26] Tao Xie, Evan Martin, and Hai Yuan. Automatic extraction of abstract-object-state machines from unit-test executions. In *ICSE '06: Proceedings of the 28th international conference on Software engineering*, pages 835–838, New York, NY, USA, 2006. ACM.