Extending Project Lombok to improve JUnit tests

Master's Thesis

Joep Weijers

Extending Project Lombok to improve JUnit tests

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

Joep Weijers
born in Vught, the Netherlands

Software Engineering Research Group
Department of Software Technology
Faculty EEMCS, Delft University of Technology
Delft, the Netherlands
`www.ewi.tudelft.nl`

TOPdesk
Martinus Nijhofflaan 2
Delft, the Netherlands
`www.topdesk.com`

Extending Project Lombok to improve JUnit tests

Author:	Joep Weijers
Student id:	1308432
Email:	`j.weijers-1@student.tudelft.nl`

Abstract

In this thesis, we look at unit and integration test suites and look for ways to improve the quality of the tests. The first step in this research is the development of the JUnitCategorizer tool to distinguish between unit tests and integration tests. JUnitCategorizer determines the actual class under test using a new heuristic and ultimately decides whether the test method is a unit or integration test. We show that JUnitCategorizer correctly determines the class under test with an accuracy of over 90%. Our analysis also shows an accuracy of 95.8% on correctly distinguishing unit and integration tests. We applied JUnitCategorizer on several open and closed source projects to obtain a classification of the test suites based on their ratio of integration tests.

The second part of this research looks at applicable methods to increase the quality of the tests, for example elimination of boiler-plate code and detection (and possibly generation) of missing tests. Using the classification of test suites, we show that the aforementioned test problems occur in both unit as integration tests. We then propose a new tool called Java Test Assistant (JTA), which can generate boiler-plate tests and some hard tests. An experiment was conducted to assess JTA and showed promising results. Code coverage increased and several of our generated tests fail on the current code base because the implementations are not entirely correct.

Preface

Spending nine months full time on a single research project is unlike any other project in my academic career. The available time allows a much deeper exploration of the subject, but on the other hand requires discipline to stay focused and to keep making progress. I'd like to thank several people who have helped me bring my master's thesis to a successful end.

First off, I'd like to thank my supervisors for all their support. Roel Spilker, supervisor from TOPdesk, for aiding me with his great in-depth Java knowledge and great comments and Andy Zaidman, supervisor from the TU Delft, for giving valuable feedback on my drafts and his continuous availability to answer questions.

I want to thank TOPdesk for being a great employer and facilitating the wonderful option of combining a research project with getting to know how development works in practice. Development team Blue deserves a special mention for the warm welcome I got in the team. And an even more special mention to Bart Enkelaar; without all his stories about how awesome TOPdesk is, I might not have considered TOPdesk as potential for an internship.

I also want to thank family, friends and colleagues for asking "what is it exactly that you do for your thesis?", requiring me to come up with comprehensible answers for people with very different backgrounds.

Joep Weijers
Delft, the Netherlands
August 15, 2012

Contents

List of Figures

Chapter 1

Introduction

In order to assess quality of software, it is submitted to various tests. E.g. high-level acceptance tests to see if the software performs the correct functions, and fine-grained tests to ensure sub-systems behave as specified. It is an important step in software development and there are various ways to accomplish this. In this research, we look at unit and integration test suites and look for ways to improve the quality of the tests. The first step in this research is the development of a tool to distinguish between unit tests and integration tests. With that division in hand, we will look at applicable methods to increase the quality of the tests, for example elimination of boiler-plate code and detection (and possibly generation) of missing tests. We will then propose an extension to Project Lombok, a compiler plug-in which removes boiler-plate code from Java programs. An experiment is conducted to assess this extension.

1.1 Terminology

The most important distinction to be made in this research is between *unit testing* and *integration testing*. Binder defines unit testing as “testing the smallest execution unit” and integration testing as “exercising interfaces among units to demonstrate that the units are collectively operable” [1]. But what should be considered as a unit? Is it a sub-system, a class, or a single method? We prefer classes and methods over entire sub-systems, because they are more fine-grained than sub-systems and thus errors can be detected at a lower level. This argument also implies preferring methods over classes, however many authors consider classes as the units under test. Pezzè and Young note that “treating an individual method as a unit is not practical, because [. . .] the effect of a method is often visible only through its effect on other methods” [2]. While this is true, the functionality of a class is defined by its public methods. So in order to test a class, we still need to test each of its public methods. Therefore, we will consider *methods* as the units under test. The class these *methods under test (MUT)* belong to, will be referred to as *class under test (CUT)*, following the terminology Binder uses [1].

How can we distinguish between unit and integration tests? According to Koskela [3], a good unit test abides several rules:

- A good test is atomic.
- A good test is isolated.
- A good test is order independent.
- A good test should run fast (typically in the order of milliseconds).
- A good test shouldn't require manual set-up.

The atomicity criterion is vague, i.e. what is atomic? The author means that a good test should test only a small, focused part of the system, so in our case: a good test tests a single method. The test isolation criterion says that a test must not be influenced by external factors, like other tests. To show this is a desirable property, take for example the case where a certain test B depends on a field set by a preceding test A. If A fails, the field is not set and thus B will also fail. When a programmer sees that both A and B failed, it is not clear for him if A is the only problem or if B also has a problem. As Beck puts it: "If I had one test broken, I wanted one problem. If I had two test broken, I wanted two problems" [4]. Test order independence directly follows from test isolation, because if a test is completely isolated from the other tests, it does not depend on the order of the tests.

In order to find defects early, the programmer must test his classes often. This can be inhibited by test suites taking a long time to run or requiring a complex action before starting a test suite. Hence the two rules that tests should run fast and should not require manual set-up. Ideally, we want the entire unit test suite to only take some seconds. The JUnit testing framework handles the set-up of tests and can be integrated into an Integrated Development Environment like Eclipse so running a test suite is just a press of a button. Note that not all fast tests are unit tests; integration tests may be fast too.

Feathers [5] suggested a set of concrete rules to indicate practical cases where a test is no longer working on a single unit and should be considered an integration test. A test is not a unit test when:

- It talks to the database.
- It communicates across the network.
- It touches the file system (e.g read from or write to files through the code).
- It can't run at the same time as any of your other unit tests.
- You have to do special things to your environment (such as editing configuration files) to run it.

Excluding these cases from unit tests advocates the implementation and use of *mock objects* to replace the real database, network, and file system adapters. A *mock object* is a "substitute implementation to emulate or instrument other domain code. It should be simpler than the real code, not duplicate its implementation, and allow you to set up private state to aid in testing" [6]. The benefit of this approach is that unit tests are isolated from influences that not necessarily are problems with the unit itself, e.g. network time-outs due to unavailability of the network, slow database connections or file system errors. We do want to detect these problems, but they are at the integration level. Note that the list is not exhaustive; we can also add system dependent values like the current time or the machine platform and architecture to the list. Generally, when in a test a mock object is replaced by the real implementation of that object, we speak of an integration test.

JUnit is a testing harness for the Java programming language. Using JUnit, programmers write their test cases in Java and can easily run and re-run an entire test suite, inspecting the test results in between different runs. Due to its widespread use in Java applications, JUnit will be the testing harness we apply this research on.

A *test command (TC)* is a test method, annotated in JUnit with the `@Test` annotation, which contains one or more *assertions*. An assertion is a method that evaluates an expression and throws an exception if the assertion fails. For each public method of a class, developers write one or more test commands. Test commands that share a common setup, known as *fixture*, are grouped together in a *test class*. This usually, but not necessarily, leads to one test class per CUT. A *JUnit test suite* consists of several test classes.

One of the Java applications that will be subject of this research is *TOPdesk*, a Service Management tool which has been in development for over a decade with varying levels of code consistency. This has resulted in a giant organically grown code-base, including all problems that come with working with legacy code of varying code quality.

Boiler-plate code are sections of code that have to be included in many places with little or no alteration. The code often is "not directly related to the functionality a component is supposed to implement" [7]. In those cases we can consider it a burden to write. Due to its repetitive nature, it suffers from problems identical to code duplication, such as being difficult to maintain.

The tool we will use and extend in this research is *Project Lombok*. Project Lombok is a compiler plug-in that aims to remove boiler-plate code from Java programs. This is accomplished by using declarative programming, mainly by the use of annotations. For example, if the developer adds `@Data` to a class, getter and setter methods will be created for all fields, as well as the methods `equals(Object o)`, `hashCode()` and `toString()`. Since the boiler-plate code is omitted, only relevant code remains. Since error-prone `equals(Object o)` and `hashCode()` implementations are generated, using Project Lombok should result in improved program behavior.

1.2 Research Questions

The goal of this research is to improve the quality of tests by creating and applying an extension of Project Lombok. The main test subject will be *TOPdesk*.

First we will write a tool called JUnitCategorizer to review several test suites, including the current TOPdesk test suite, to determine the applicability of a generalization of JUnit tests using Project Lombok. We want to know where Project Lombok is best applicable, either in unit tests, integration tests, or both. Since both unit and integration tests can be written in a single testing harness, it has become common practice in JUnit test suites to use one single suite for both unit and integration tests. Our first objective is to separate those. Additionally, two separate test suites, one for pure unit tests and one for integration tests, are beneficial, because they can be utilized in a two-step testing approach. Using this approach, a developer first runs the unit test suite and if it passes, runs the integration test suite. This ensures that the integration test suite, with essential tests to check the interoperability of classes, uses correctly behaving units. So far, to the best of our knowledge, there are no automatic tools to split this single large and possibly slow test suite to one suite for unit tests and one suite for integration tests. The development of such a tool could lead to a more efficient method of testing, because the integration tests are not run if the unit tests fail. This saves time since not all tests are run, but, more importantly, this lets the developer focus on the failing tests at the unit level.

The second objective is the analysis of a sample of the tests to determine which boiler-plate code can be removed by Project Lombok, e.g. many similar (cloned) tests. Additionally, we would like to determine common errors in test cases, such as trivial tests a tester typically omits. If Project Lombok can alleviate these problems, untested code will become tested, which is an increase in code coverage. To accomplish these objectives, we will use tools that provide the required insight, e.g. CCFinderX to detect code clones.

The next step is to extend Project Lombok to include annotations or other constructs to ease the implementation of unit tests. This extension will be applied to the TOPdesk test suite and the results will be analyzed. We expect that these constructs decrease the number of lines of test code significantly as well as increase the quality of the tests.

The research questions investigated in this thesis are:

- How can we distinguish unit and integration tests in JUnit?
- Which types of tests are best suitable to extend Project Lombok to?
- Which code can be considered as boiler-plate in tests based on JUnit?
- Which common test errors could be alleviated by using the extension of Project Lombok (e.g. trivial tests a tester typically omits)?
- What does the extension of Project Lombok add to existing tooling?
- Does the extension of Project Lombok increase the quality of unit testing, compared to manually writing unit test code?

These questions will be treated in the following chapters. Chapter 2 will explain how we distinguish unit and integration tests and will discuss the implementation and analysis of our tool. We locate and analyze possible extensions of Project Lombok in chapter 3. Chapter 4 will discuss the implementation of these extensions and will analyze them. Scientific work related to the subject discussed in this thesis will be mentioned in chapter 5 and we will conclude in chapter 6.

Chapter 2

Distinguishing unit and integration tests with JUnitCategorizer

In this chapter we explain how we distinguish unit and integration tests. We will discuss how we implemented the distinction in our tool called JUnitCategorizer. We then present an analysis of JUnitCategorizer, where we assess how reliably it can determine classes under test and how accurate the detection of unit and integration tests is.

2.1 Distinguishing unit and integration tests

In our definitions, we stated that when a mock object is replaced by the real implementation of that object, we speak of an integration test. Conversely, if a test only uses the object under test and mock objects, it is likely a unit test. Likely, but not definitely, because the test might still be order dependent or slow. If we can determine all objects called from a test method, determine the class under test and identify mock objects, we can make an educated guess if the test is unit or integration.

2.2 Detecting the called classes

Our objective of registering which objects are called is achieved by a technique called *bytecode instrumentation*. Java programs are compiled to *bytecode*, machine readable instructions in a `*.class` file that the Java Virtual Machine can execute. This bytecode can be changed after compilation, for example using a bytecode manipulation library like ASM. The change can be static, i.e. replacing the `*.class` files, or dynamic, i.e. changing a class at the moment it is loaded. We want to inject a method call to a *Collector* class in every method of every class. So, as soon as any method of a certain class `SomeClass` is called, it will signal the collector and tell that class `SomeClass` is called. This is a type of *dynamic analysis*, i.e. the analysis is performed when the program is running, either through the testing framework, or an actual run.

2.2.1 Instrumenting class files

Our initial approach tried the static transformation in which the actual `*.class` files are replaced. We extended the open source Cobertura code coverage tool[1], because it uses static instrumentation and is designed to work with JUnit. Cobertura uses the ASM library and was easily extended to call a modified collector to register which classes are called. The next step was to register the called classes per test method, since Cobertura registers the entire program run at once. To this end, we added a `public static void nextTestMethod(String className, String methodName)` method.

The major drawback of using Cobertura is its static method of instrumenting. Overwriting `*.class` files by their instrumented counterparts is aimed at code that you wrote and compile yourself. It is much less suited to instrument and replace the core Java class files. For the completeness of the analysis, we also want to register called core Java system classes, for example to register if classes from the `java.io` package are used to access files during a test. A solution was found by using bytecode instrumentation on the fly.

2.2.2 Instrumentation on the fly

If a class is used for the first time in the Java Virtual Machine, its class file will be loaded by a `ClassLoader` from the hard drive into the memory of the computer. This also holds for core Java classes, so if we can transform the bytecode at the moment it is loaded into memory, we can also instrument those classes. This technique, also known as *bytecode weaving*, can be achieved in two ways: writing a custom ClassLoader, or using a *Java agent*. A Java agent is a `jar` file containing an *agent class* which contains some specific methods to support instrumentation. The Java agent is specifically designed for bytecode instrumentation and is therefore preferred over writing a custom ClassLoader.

One catch is that classes that are already loaded before the Java agent is operational, like `java.lang.ClassLoader` and `java.util.Locale`, are not instrumented by the Java agent. However, since Java 6, they can be instrumented by forcing a retransformation, so we are able to register calls to all core Java classes by requiring a Java 6 JVM.

2.2.3 Bytecode transformations

We have specified a number of bytecode transformations with ASM to insert different calls to the collector.

We add a call to `Collector.touchedMethod(String className, String methodName)` to every method. This is the method that collects all called classes. Listing 2.2 is the Java representation of the change in bytecode when the transformer is applied to the example method in listing 2.1.

[1] http://cobertura.sourceforge.net

```java
public class ACertainClass {
  public void someMethod() {
    // Method content
    ...
  }
}
```

Listing 2.1: *Example method*

```java
public class ACertainClass {
  public void someMethod() {
    Collector.touchedMethod(
        "aCertainClass",
        "someMethod");
    // Method content
    ...
  }
}
```

Listing 2.2: *Method instrumented*

In order to exclude some classes from instrumentation, we created an empty `DoNotInstrument` interface. If a class implements the `DoNotInstrument` interface, then the transformer will not instrument the class. The most important example of a class that should not be instrumented is the collector. We enter an infinite loop if `touchedMethod` calls itself to signal it has been called, resulting in a program crash.

We need to add a call to `Collector.nextTestMethod(String className, String methodName, String methodType)` to certain methods related to JUnit tests. There are five methodTypes we want to register, each identified by their respective JUnit annotation:

- `Test` indicates a test method and informs the collector to register all incoming called methods for this test method.
- `BeforeClass` indicates a method that is run once, namely at the initialization of the test class. Typically used to setup a fixture for a certain test class.
- `Before` indicates a method that is called before every test method, for example the setup of a fixture.
- `After` indicates a method that runs after every test method, for example the teardown of a fixture.
- `AfterClass` indicates a method that is run once, after all test methods of the test class have been run.

Classes called in `BeforeClass`, `Before`, `After` and `AfterClass` are considered to be needed by the test method and will therefore be added to the list of called methods of these test methods. Since JUnitCategorizer depends on the JUnit annotations, we require the unit tests to be written in JUnit version 4 and up.

We also included a `Collector.finishedTestMethod(String className, String methodName, String methodType)`, to prevent the collection of classes called in between two test methods, e.g. several JUnit internal methods. We also use this method to detect the end of an `AfterClass` method, so we can append all classes called in that `AfterClass` method to all test methods in the same test class. Listing 2.4 shows how an example test method as in listing 2.3 is transformed. This transformation is identical for methods annotated with `BeforeClass`, `Before`, `After` and `AfterClass`.

```java
public class ACertainTest {
  @Test
  public void testSomeMethod() {
    // Test method content
    ...
  }
}
```

Listing 2.3: Example test method

```java
public class ACertainTest {
  @Test
  public void testSomeMethod() {
    try {
      Collector.nextTestMethod(
          "aCertainTest",
          "testSomeMethod",
          "Test");
      // Test method content
      ...
    }
    finally {
      Collector.finishedTestMethod(
          "aCertainTest",
          "testSomeMethod",
          "Test");
    }
  }
}
```

Listing 2.4: Test method instrumented

Note that our implementation with a single collector requires that the test methods are run sequentially. Tests that run concurrently will not register called classes correctly.

2.2.4 Filtering

We do not wish to list all called classes, because usage of certain classes does not render a test an integration test. We allow a test method to use many Java core classes, for example `java.lang.String`. JUnitCategorizer provides three methods of filtering the list of called classes: a blacklist, a suppressor and a whitelist.

Blacklist

The first way to limit the registration of called classes, is a blacklist. The blacklist is a list of methods whose classes we only want to collect if that particular method is called. For example, `java.util.Locale` has a `getDefault()` method that relies on the current system settings. If a test uses this method, it potentially is system dependent and therefore not a unit test. However, all other operations on `java.util.Locale` are not dependent on the current system settings, so we adopted the following practice: if we don't list a blacklisted class in the list of called classes, we know that only safe methods are used. If it is listed, then we know it potentially is an integration test.

Suppressor

There are some methods of which we don't care what happens in that method. For example, `java.lang.ClassLoader` has a method `loadClass` which loads the bytecode of a class into memory. If the loaded class is in a `jar` file, it uses several classes from the `java.io`

```java
public class ACertainClass {
  public void
      someSuppressedMethod() {
    // Suppressed content
    ...
  }
}
```

Listing 2.5: Example suppressed method

```java
public class ACertainClass {
  public void aSuppressedMethod() {
    try {
      Suppressor.startSuppress();
      // Suppressed content
      ...
    }
    finally {
      Suppressor.stopSuppress();
    }
  }
}
```

Listing 2.6: Suppressed method instrumented

and `java.util` packages. We don't wish to record this inner working of `loadClass` in the current test method, since loading a class does not determine the difference between a unit or integration test.

The solution we introduced is called a *Suppressor*. The bytecode transformer surrounds the suppressed method with method calls to `Suppressor.startSuppress` and `Suppressor.stopSuppress`. As long as `Suppressor.isSuppressed` returns true, we will not collect called classes in `touchedMethod`. Listings 2.5 and 2.6 show the result of the bytecode transformation.

Our implementation uses an XML file to specify per class the names of methods to suppress. This ignores method overloading, so suppressing the `println` method in `java.io.PrintStream`, will suppress all the ten different `println` methods.

Whitelist

We needed a third method of filtering that operates on the list of called classes. While a good number of unwanted classes are kept out of the list by the suppressor and the blacklist, classes from used frameworks, especially JUnit, and several core Java classes show up on the called classes list. To filter these, we could use the blacklist, but it requires specifying all methods in a class. We therefore opted to introduce a whitelist: an XML file which defines classes that are allowed to be used by the test methods. The whitelist provides easy whitelisting of entire packages, single classes or classes that contain a specific string in their name. This last option was added to allow all `Exception` classes, both Java core as custom exceptions.

2.2.5 What classes to blacklist, suppress or whitelist?

The following types of classes each have reasons why they could be blacklisted, suppressed or whitelisted, in order to improve the accuracy of JUnitCategorizer.

Locale sensitive classes

Several Java core classes contain locale sensitive methods, i.e. they accept a `Locale` as parameter for internationalization purposes, or refer to the system default locale if it is not specified. As mentioned before, if a test depends on the default system locale, it is an integration test. To detect the usage of the default locale in locale sensitive classes, we register calls to the `getDefault` method in `java.util.Locale`.

This is not the ideal solution, since it falsely reports `testCompareDefaultLocale` in listing 2.7 as an integration test. The outcome of the test does not depend on the default locale (as in `testCompareNonDefaultLocale`), since we only compare the defaults. Another valid unit test in listing 2.7 is `testSetGetDefaultLocale`, where we set a default locale before getting it, thus we are not depending on the original system default locale.

A correct way to solve these problems would be to implement a kind of taint analysis, which verifies that either a default locale is never compared to a specific locale, which is unrelated to the current system settings, or ensures that a default locale has been set prior to the comparison. Due to time constraints, this solution has not been implemented in JUnitCategorizer. As a results we might mark a number of unit tests as integration test, which is preferred over erroneously marking integration tests as unit tests.

```java
public class TheLocale {
  public static boolean checkLocale(Locale aLocale) {
    String country = Locale.getDefault().getCountry();
    return country.equals(aLocale.getCountry());
  }
}

public class TheLocaleTest {
  @Test
  public void testCompareDefaultLocale() {
    Locale defaultLocale = Locale.getDefault();
    assertTrue(TheLocale.checkLocale(defaultLocale));
  }

  @Test
  public void testCompareNonDefaultLocale() {
    Locale nonDefaultLocale = Locale.US;
    assertFalse(TheLocale.checkLocale(nonDefaultLocale));
  }

  @Test
  public void testSetGetDefaultLocale() {
    Locale.setDefault(Locale.US);
    Locale defaultLocale = Locale.getDefault();
    assertTrue(defaultLocale.getCountry().equals("US"));
  }
}
```

Listing 2.7: *Problematic tests using* `Locale`

Time dependent classes

There are also several classes that depend on the system's time settings. For example, `java.util.TimeZone` has `getDefault` and `getDefaultRef` methods to get the system's current timezone. We added these methods to the blacklist, so if `TimeZone` shows up in the list of called classes, then either `getDefault` or `getDefaultRef` have been called, and the test might be depending on the current system setting.

Another important class is `java.util.Date`, which has an empty constructor which constructs a `Date` object for the current system time. As with `Locale` and `TimeZone`, if the default system value is compared in a test to a value that is unrelated to the current system, the test is not a unit test. Listing 2.8 illustrates a simple example: `testDateInFuture` is not a unit test, whereas `testDateRunningInFuture` is. Unfortunately, apart from being unable to detect the comparison of a system default value against fixed value, we can not blacklist the `Date` constructor, since it is used for both creating a `Date` object with the current system time as one with a fixed value.

```java
public class DateTest {
  @Test
  public void testDateInFuture() {
    Date now = new Date();
    // Future is 29-06-2012 12:00:00
    Date future = new Date(1340989200000L);
    assertTrue(now.before(future));
  }

  @Test
  public void testDateRunningInFuture() {
    Date now = new Date();
    Date future = new Date(now.getTime() + 1000);
    assertTrue(now.before(future));
  }
}
```

Listing 2.8: *Problematic tests using* `Date`

A solution would be to blacklist `currentTimeMillis` in `java.lang.System`, which provides the current system time. This would also solve a deeper problem, namely the collection of all tests that are using the current system time in one way or another. Unfortunately, this is not possible, because `currentTimeMillis` is a `native` method, i.e. it has no method body to add the call to the collector in. The correct way of instrumenting a native method requires the method to be wrapped in a non native method and adding the call to the collector in the new non native method. Java does not allow the addition of methods or fields into the `System` class, so we can't wrap `currentTimeMillis` and therefore we can not instrument it.

Another solution for registering the correct `Date` constructor would be to extend the blacklist to include method signatures, i.e. the precise specification of return type and argument of a method. Since this does not solve the collection of tests using the current system time and because we can not detect whether only default values are compared, we did not implement this.

Framework classes

Most software depends on external libraries or frameworks that provide part of the implementation. Some frameworks used in TOPdesk are Apache Commons, Google Commons, JFreeChart and SLF4J. Classes from these frameworks appear often in the list of called classes. We add these packages to the whitelist for TOPdesk, since we assume that these frameworks are properly tested themselves and put them in the same league as the Java standard library.

Additionally, the whitelist depends on the program under test. For example, if we look at the JUnit test suite, whitelisting all JUnit classes will have a negative impact on the results.

2.2.6 Detecting the class under test

We use a relatively simple algorithm for detecting the Class Under Test (CUT), based on the assumption that a test class has a single CUT. We award points for hints that a certain called class is the CUT. Class `SomeClass` is possibly the CUT if:

- it is used in a test, i.e. only classes that are registered as called in the methods of this test class may be the CUT.
- there are tests that only exercise `SomeClass`, i.e. they have exactly one called class, namely `SomeClass`.
- inner classes of `SomeClass` are used, e.g. `SomeClass$SomeInnerClass`.
- the name of the test class is `SomeClassTest`, `SomeClassTests` or `TestSomeClass`.
- the test class is in the same (sub)package as `SomeClass`.

The exact number of points awarded, or weight, for each criterion is specified in the analysis section. The class that received the most points is the most likely class under test. If there are multiple classes with the same highest score, we are probably looking at an integration test class and we determine that the class under test is “unknown”.

2.2.7 Detecting mock objects

We distinguish two ways of creating mock objects: using a mocking framework like EasyMock and Mockito or creating mock objects manually. The first way allows for an easy way to not collect classes that are being mocked. We suppress all relevant methods from the mock framework used. Mockito additionally required to allow classes that contain “$EnhancerByMockitoWithCGLIB$” in their class name, since their mocked classes show up as for example `$java.util.List$$EnhancerByMockitoWithCGLIB$$[someHexValue]`.

It is much harder to detect manually created mock objects. The best method we currently provide to not collect these mock objects as called classes, is to use “Mock” in the class name as naming convention, and then whitelist classes that contain “Mock” in their name.

2.2.8 Unit or integration?

The final decision of JUnitCategorizer whether a test method is unit or integration depends on the amount of called classes that remains after the whitelisting and filtering of the class under test. If the test method has no called classes, we can conclude it is a unit test. If it used other classes than a possible class under test, then we conclude it is an integration test.

2.3 Detecting the Class Under Test

As mentioned in section 2.2.6, we use an algorithm based on weights for five parameters. In this section we will determine and analyze the optimal weights for the parameters, such that we detect a high number of CUTs correctly.

The five parameters used in the algorithm are: called class, called single class, called inner class, matching file name and matching package. We wish to detect a high number of CUTs correctly, because if we can not determine the CUT, it will not be removed from the list of called classes for that test class. This results in methods erroneously being marked as integration tests, while they only exercise the CUT. To get an optimal set of parameters, we performed an experiment to determine the optimal weights.

The test set we used in the experiment contains 588 test classes in seven projects and has been compiled from test suites included in TOPdesk, Maven, JFreeChart, and Apache Commons. We manually determined the CUT in every test class for comparison to the output of JUnitCategorizer.

2.3.1 Determining the possible values for the parameters

Suppose we try a certain amount of values for each parameter and call this amount x. The values we use will be the integers in the interval $[0, x-1]$. This results in x^5 combinations of the five parameters. We try all these combinations on every test class of our test set, to determine which combination has the highest fitness, i.e. which combination yields the most correctly detected CUTs in most of the test suites. But we first need to determine a suitable value of x, i.e. one that both returns fit combinations, as well as returns results in a reasonable amount of time.

We create *box plots*, also known as box-and-whisker plots [8], for different values of x. A box plot is a visualization of data on the vertical axis with a box containing three horizontal lines showing the values of the median, lower quartile and upper quartile of the data. The other two horizontal lines, known as whiskers, mark a distance of 1.5 times the *interquartile range (IQR)* from the median. The IQR is the difference between the upper and lower quartile. If the maximum or minimum value of the data lie within 1.5 times the IQR, then the corresponding whisker marks that value instead of 1.5 times the IQR.

The box plots for interval lengths up to and including 10 are depicted in figure 2.1. From these box plots we can conclude that the larger the interval length, the more combinations

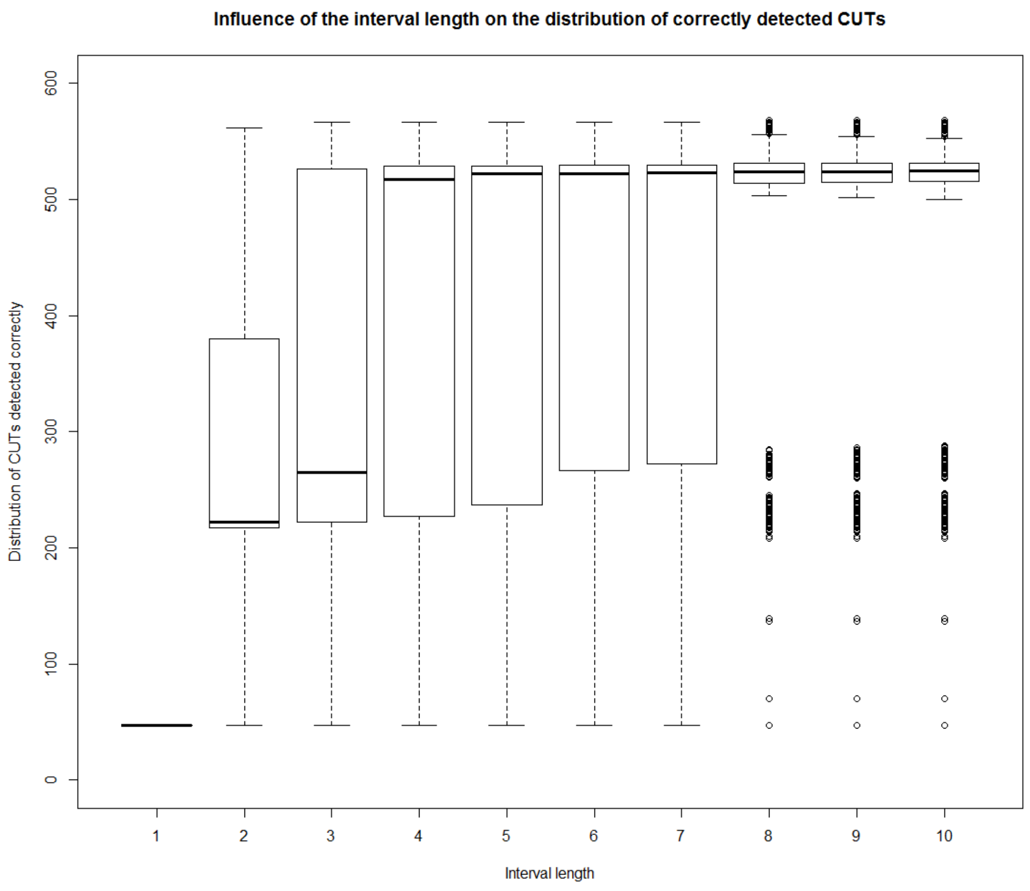

Figure 2.1: *Influence of the length of the interval of parameters x on the distribution of correctly detected CUTs. The interval corresponding to interval length x is [0, $x-1$]*

of parameters yield more correctly detected CUTs. Any interval length larger than seven ensures that a large majority of combinations correctly detect at least 500 CUTs. This indicates that when certain parameters are set to seven or higher, their combinations detect a high number of CUTs. We conclude that an interval length of eight is a suitable choice.

2.3.2 Parameter dependence on the number of methods in a test class

We expect that the points for some of the parameters should relate to the number of test methods in a test class. For example, if the name of the test class is either `SomeClassTest`, `SomeClassTests` or `TestSomeClass`, we award the points for a matching file name once per test class. The points awarded for classes and inner classes called in the test methods are awarded per test method. This results in the effect of the one time bonus of a matching file name or matching package being diminished if there are many test methods. This problem might also apply, to a lesser extent, to test methods testing a single class in the case of having one method testing a single class and many other test methods that test multiple classes. To

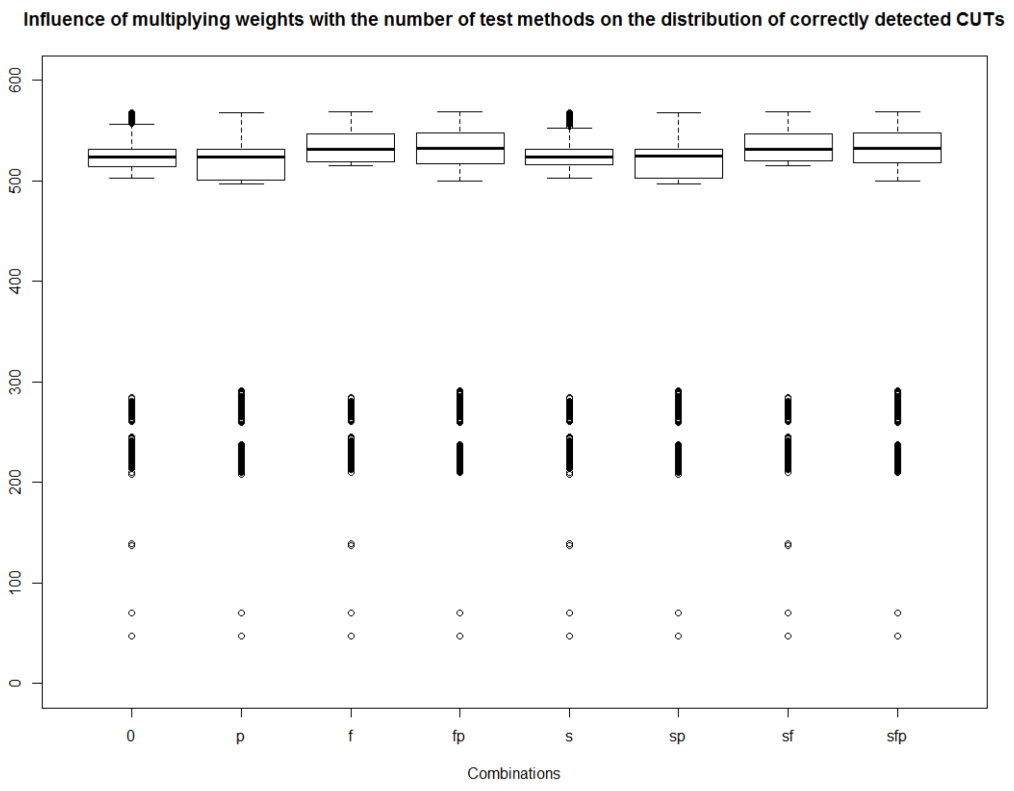

Figure 2.2: *Influence of multiplying weights with the number of test methods on the distribution of correctly detected CUTs. 0 indicates that none of the weights are multiplied, s, f, p indicate that the weight of respectively a single called class, a matching file name, or the weight of a matching package is multiplied.*

counteract this effect, we can multiply the number of points by the number of methods in a test class.

We conducted the experiment for all eight possible combinations of multiplying the number of test methods in a test class to a single called class, matching file name and matching package. We encoded these combinations as follows: 0 indicates that none of the weights are multiplied, s, f, p indicate that the weight of respectively a single called class, a matching file name, or the weight of a matching package is multiplied. The parameters can be combined, so for example fp means that both the weight of a matching filename as the weight of a matching package and sfp means all parameters are multiplied.

The resulting box plots can be found in figure 2.2. Multiplying the parameter for a single called class has no significant influence, since there is no significant difference between 0, p, f, fp and s, sp, sf, sfp, respectively. Multiplying the matching package parameter has a negative influence, as shown by p having its lower quartile at less correctly detected CUTs than the lower quartile of 0.

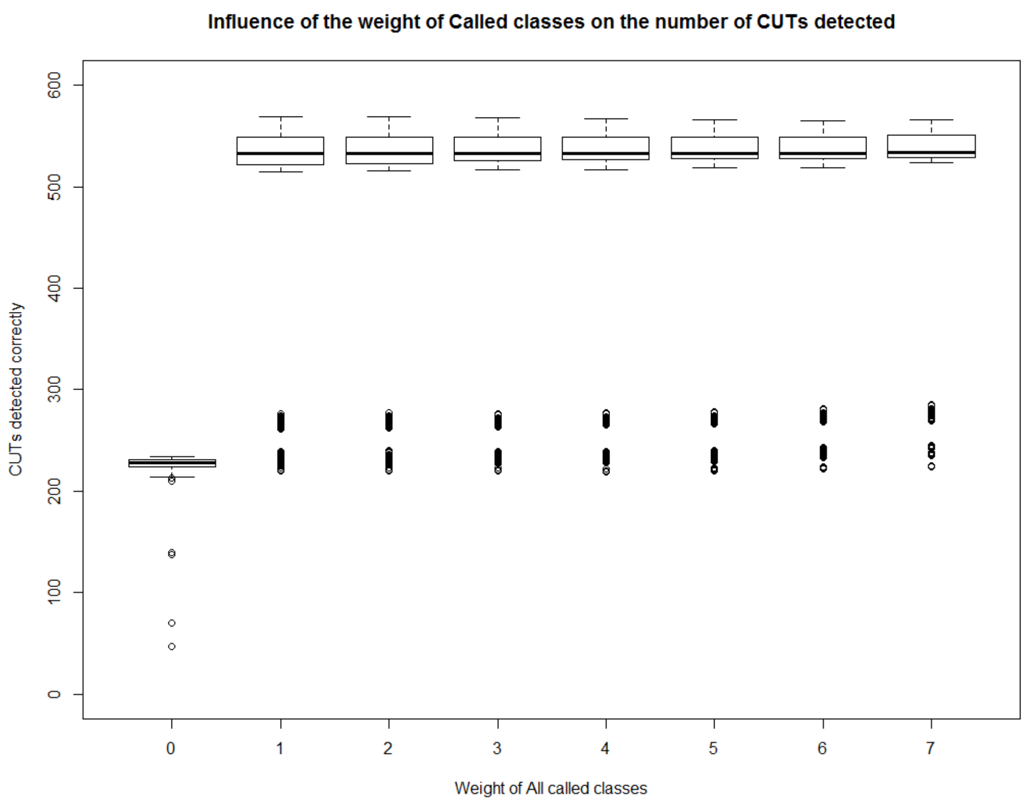

Figure 2.3: *Influence of the weight of called classes w_a on the number of CUTs detected.*

Multiplying the parameter for the matching file name (cases f and sf) results in a higher mean and quartiles compared to not multiplying (0 and s). This effect carries over to fp and sfp, where the negative influence of multiplying the matching package is decreased, but still is visible. We therefore settle for the benefit of only multiplying the weight of a matching file name.

2.3.3 Determining a fit combination

With the possible values of the parameters set and the multiplier applied to the matching file name parameter, we start the search for a fit combination. We could just search for the set of combinations that yield the most correctly detected CUTs and randomly select a combination from that set. This does not give us much insight into the data set, so we apply some exploratory data analysis.

We first try to set two parameters, so the search space reduces from five dimensions to three dimensions. The best candidate is the weight that is assigned to all called classes w_a. Figure 2.3 shows that the higher the parameter is set, the more CUTs are correctly detected. We therefore set this value $w_a = 7$.

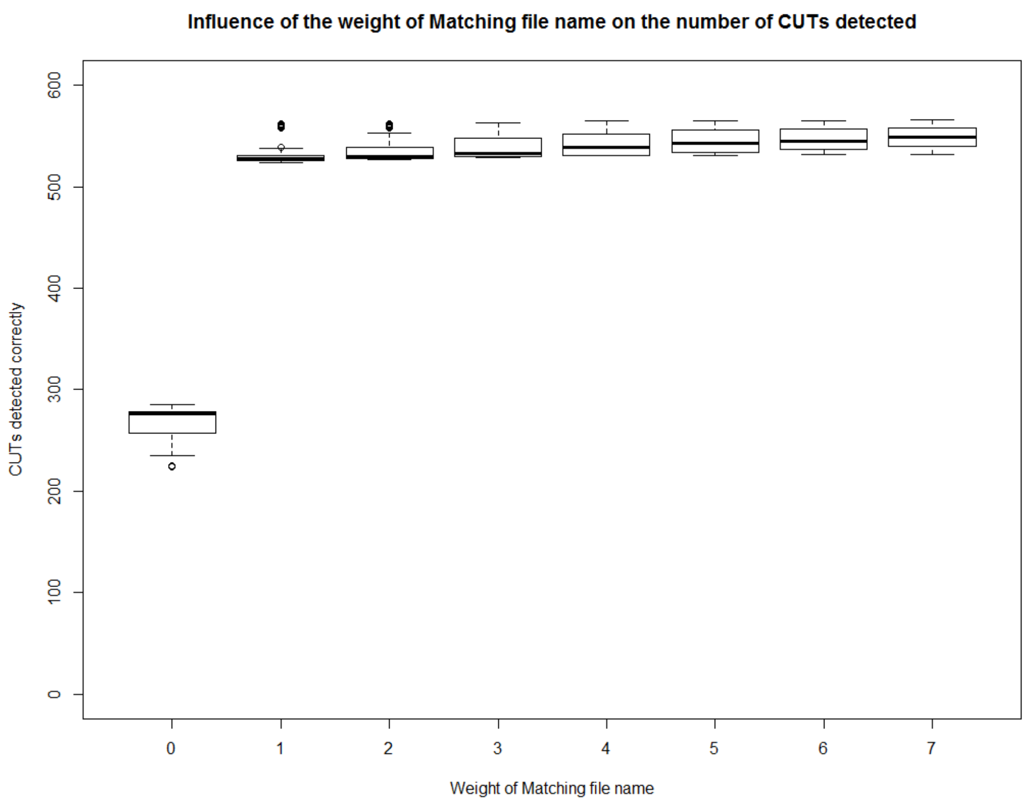

Figure 2.4: *Influence of the weight of matching file name w_f on the number of CUTs detected, with $w_a = 7$.*

Next we consider the weight of the file name w_f. Also for w_f holds: the higher it is set, the more CUTs we detect correctly, as shown in figure 2.4. We set it to the maximum value of 7. Note that applying the CUT detection algorithm with the weight of a file name set to 0 is equivalent to applying the algorithm on a test set that does not adhere to the naming conventions we assume in the algorithm. So the distribution on $w_f = 0$ gives an indication of how the CUT detection algorithm performs when our naming conventions are not followed.

The third parameter we inspect is the weight of called inner classes w_i. Figures 2.5 and 2.6 show how it interacts with the weights of single class and matching package, provided that $w_f = 7$ and $w_a = 7$. In both figures, the number of correctly detected CUTs increases if w_i goes to zero, so we set $w_i = 0$.

That leaves the two weights for matching package w_p and single class w_s, which we plotted in figure 2.7. With all the points very closely to, and many exactly on, the plane at 565 correctly detected CUTs, we can conclude that if $w_a = 7$, $w_f = 7$ and $w_i = 0$, then the values for w_p and w_s are of no significant influence. This means that we can assign $w_p = 0$ and $w_s = 0$, showing that three of the five cases we identified as being hints for the CUT are irrelevant and can be left out of the algorithm.

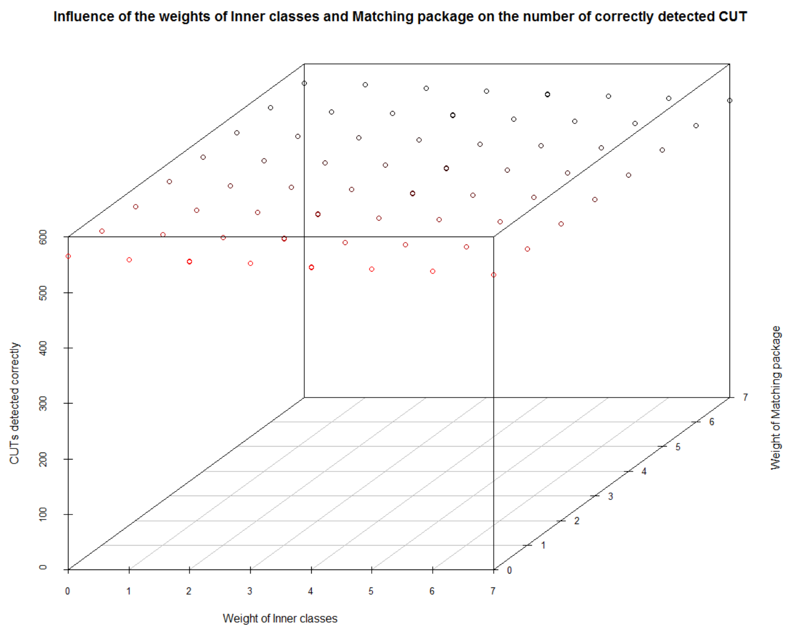

Figure 2.5: *Influence of the weight of inner classes w_i and the matching package w_p on the number of CUTs detected, with $w_a = 7$ and $w_f = 7$.*

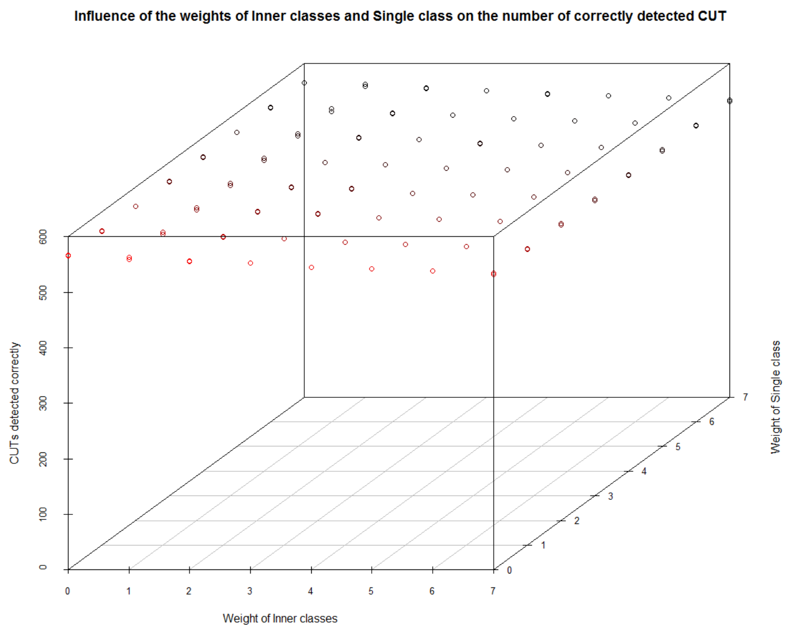

Figure 2.6: *Influence of the weight of inner classes w_i and the single class w_s on the number of CUTs detected, with $w_a = 7$ and $w_f = 7$.*

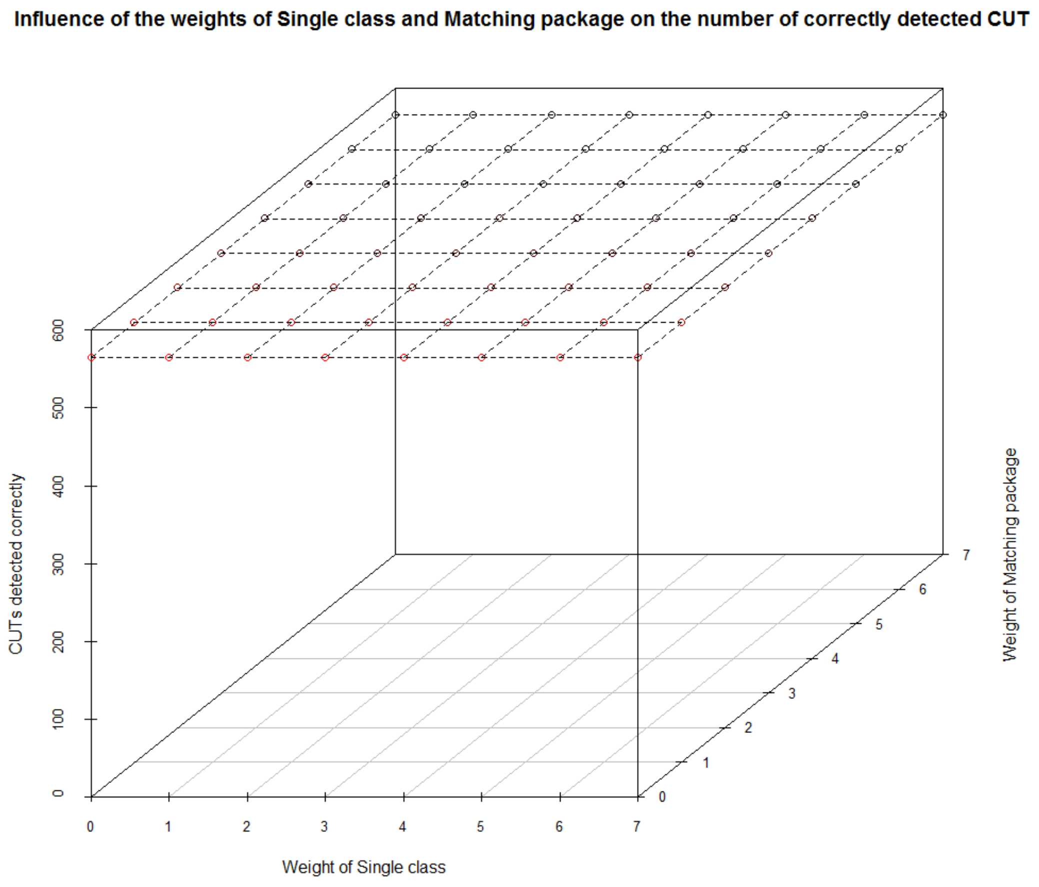

Figure 2.7: *Influence of the weight of matching package w_p and the single class w_p on the number of CUTs detected, with $w_a = 7$, $w_f = 7$ and $w_i = 0$. The plane is drawn at CUTs detected correctly = 565.*

2.3.4 Validating the chosen combination

To validate the combination of parameters we selected, we state an hypothesis and inspected it against the accumulated results of three more projects. The projects, TOPdesk's mangomodels and Maven's model builder and settings builder, contain 31 test classes and, as with the original test set, we determined all classes under test.

Our hypothesis H_0 is that the combination $w_a = 7$, $w_f = 7$, $w_i = 0$, $w_p = 0$ and $w_s = 0$ is a fit combination. More specifically, we state that this combination is able to determine at least 90% of the CUTs correctly. Our test statistic T in this case is the percentage of correctly detected tests. In the previous results, we correctly detected 96% of the CUTs: 565 correctly detected CUTs out of 588 test classes, but we allow for a margin of error, hence the more pessimistic expectation. The expectation of T is $E[T] = 0.9 * N$, where N is the number of test classes.

Our alternative hypothesis H_1 is that the combination is unfit and detects few CUTs correctly. Test statistic T can take values from 0 to 31, where any value between $0.9 * 31 = 28$ and 31 is a sign that our hypothesis H_0 is correct. Values lower than 28 are grounds to reject H_0.

Applying JUnitCategorizer with parameters $w_a = 7$, $w_f = 7$, $w_i = 0$, $w_p = 0$ and $w_s = 0$ results in 29 correctly detected CUTs. This is above our set threshold of 28, so we do not reject the hypothesis H_0.

2.3.5 Incorrectly detected CUTs

We manually inspected the incorrectly detected CUT, determined why our tool did not correctly detect the CUT, and listed the reasons in table 2.1. The most important reason why JUnitCategorizer is unable to detect a CUT, is that the test class only contains integration tests. An integration test uses at least two different classes. If all test methods are integration, then there are several potential CUTs. In these cases there is not one single CUT, but JUnitCategorizer usually understands which class the programmer intended to be the CUT.

There are two special cases of integration test classes where JUnitCategorizer could not determine the intended CUT. The first case is that the inner class of the CUT would be used as a tie breaker. For example, a test uses `SomeClass`, `AnotherClass` and `SomeClass$ItsInnerClass`. We would expect `SomeClass` to be the CUT. In all combinations for the optimal CUT detection, the number of points awarded for inner classes is set to zero, so JUnitCategorizer can not decide on the CUT in these cases.

The second case is the "non matching file name" reason, which indicates that the file name would have been used as a tie breaker, but it did not match the expected form of `SomeClassTest`, `SomeClassTests` or `TestSomeClass`. This occurs several times in Apache Commons Collections, because there are two test classes that test a class, for example `DualTreeBidiMap` is tested by two test classes, namely `TestDualTreeBidiMap` and `TestDualTreeBidiMap2`. We can't add `TestSomeClass2` to the expected forms of test class name, because it introduces ambiguity. We can not say if `TestSomeClass2` is the second test class of `SomeClass` or if it tests `SomeClass2`.

Reason	Number of CUTs
Inner class influence	8
Non matching file name	6
Integration test	4
CUT is whitelisted	4
Found CUT in abstract class	1
Total	23

Table 2.1: *Incorrectly detected Classes Under Test grouped by reason of incorrectness.*

2.4 Detecting unit and integration tests

With the optimal parameters set for the CUT detection algorithm, we applied JUnitCategorizer on several Java test suites to see how well it performs the task of distinguishing unit tests from integration tests. We will measure four metrics on a set of test suites using three different combinations of whitelists and compare the results.

To determine how well JUnitCategorizer distinguishes between unit and integration tests, we manually classified tests from several test suites as either unit or integration. The suites we analyzed are from the TOPdesk code base: `application`, `application-utils` and `core`. We also created a `JUnitCategorizer.examples` test suite containing many border cases that are known to be troublesome for our tool, e.g. locale sensitive and time dependent classes.

There are several metrics to assess our tool. The ones we will use depend on the number of true and false positives and true and false negatives. A True Positive (TP) is a correctly detected unit test and a True Negative (TN) is a correctly detected integration test. A False Positive (FP) occurs when JUnitCategorizer detected a unit test while expecting an integration test, and a False Negative (FN) is detecting an integration test while expecting a unit test.

These values are used in the following metrics that we will use to assess JUnitCategorizer: *accuracy*, *precision*, *specificity* and *recall*. Accuracy is the fraction of correct classifications, precision is the fraction of correctly detected unit tests from all detected unit tests, specificity is the fraction of correctly detected integration tests from all integration tests and recall is the fraction of expected unit tests. They are defined as $accuracy = TP + TN/(TP + FP + TN + FN)$, $precision = TP/(TP + FP)$, $specificity = TN/(TN + FN)$ and $recall = TP/(TP + FN)$ [9].

Table 2.2 lists the results of applying JUnitCategorizer to the various test suites and we calculated the metrics in table 2.3. For all metrics hold: the closer to one, the better the performance in that metric is. The test was run without all Java core classes on the whitelist, but a handpicked set of safe Java core classes, e.g. the `java.io` package is not whitelisted, but `java.util.HashSet` is.

JUnitCategorizer yields the correct result (accuracy) on 87% of the hard test cases in JUnitCategorizer examples, which are designed to indicate flaws in the detection. It is less accurate on TOPdesk's application-utils and core with an accuracy of only 80% and 78.8%, respectively. These projects have a lot of false negatives,which occur almost exclusively in tests that exercise date dependent and locale sensitive code, so `java.util.Locale`, `java.util.TimeZone` and `java.util.Date` show up in the list of called classes. Combining this fact with the low number of true negatives results in a very low specificity in application-utils. In other words, if JUnitCategorizer detects an integration test in application-utils, it is usually wrong.

		Detected		Expected	
Project name	Total	Unit	Integr.	Unit	Integr.
JUnitCategorizer examples	54	17	37	24	30
TOPdesk application	9	3	6	3	6
TOPdesk application-utils	95	72	23	91	4
TOPdesk core	297	120	177	210	87

Table 2.2: *Number of detected unit and integration tests for different Java projects, without the Java core packages on the whitelist.*

Project name	TP	FP	TN	FN	Acc	Pre	Spe	Rec
JUnitCategorizer examples	17	0	30	7	0.87	1.0	0.81	0.71
TOPdesk application	3	0	6	0	1	1	1	1
TOPdesk application-utils	72	0	4	19	0.80	1.0	0.17	0.79
TOPdesk core	118	2	116	61	0.788	0.983	0.655	0.659

Table 2.3: *Performance on different Java projects, without the Java core packages on the whitelist. TP are the True Positives, FP are the False Positives, TN are the True Negatives, FN are the False Negatives, Acc is Accuracy, Pre is Precision, Spe is Specificity and Rec is Recall.*

Whitelisting the date dependent and locale sensitive classes will reduce the number of false negatives, but might increase the number of false positives. In general, we rather have false negatives than false positives, because a slow integration test incorrectly being marked as unit test will slow down the entire unit test suite. A false negative means that a unit test is marked as integration test, which might slightly reduce the traceability of errors. The speed impact of a false positive is much larger than the traceability impact of a false negative, since the speed impact will show every test run. The traceability will only be an issue if an integration test fails. However, in this particular case of whitelisting date dependent and locale sensitive classes, the induced false positives are generated by fast core Java classes, so the speed impact is minimal.

Tables 2.4 and 2.5 shows the results of running the tool with the `java.util` package allowed. In this case, we give programmers using the default locales and timezones the benefit of the doubt, expecting that they have used them in such a way that the tests are not system dependent. Almost all metrics improve, for example the accuracy on TOPdesk's application-utils and core test suites rises to 99% and 96.3% of the tests correctly detected, respectively. However, as expected, we do get nine more false positives in core, resulting in a lower precision (94.2% now versus 98.3% before). JUnitCategorizer detects the same number of tests in JunitCategorizer examples correctly, but it finds two more false positives, which also results in a lower precision, but higher specificity rate.

We also applied our tool on the test suites with the `java`, `sun` and `com.sun` packages whitelisted and report the results in tables 2.6 and 2.7. Whitelisting these packages means that we register system specific tests, like file and network access, as unit instead of integration. Ignoring system specific restrictions in unit tests allows the focus on test case

		Detected		Expected	
Project name	Total	Unit	Integr.	Unit	Integr.
JUnitCategorizer examples	54	21	33	24	30
TOPdesk application	9	3	6	3	6
TOPdesk application-utils	95	90	5	91	4
TOPdesk core	297	190	107	210	87

Table 2.4: *Number of detected unit and integration tests for different Java projects, with the* `java.util` *packages on the whitelist.*

Project name	TP	FP	TN	FN	Acc	Pre	Spe	Rec
JUnitCategorizer examples	19	2	28	5	0.87	0.91	0.85	0.79
TOPdesk application	3	0	6	0	1	1	1	1
TOPdesk application-utils	90	0	4	1	0.99	1.0	0.80	0.99
TOPdesk core	179	11	107	0	0.963	0.942	1.00	1.00

Table 2.5: *Performance on different Java projects, with the* `java.util` *packages on the whitelist. TP are the True Positives, FP are the False Positives, TN are the True Negatives, FN are the False Negatives, Acc is Accuracy, Pre is Precision, Spe is Specificity and Rec is Recall.*

		Detected		Expected	
Project name	Total	Unit	Integr.	Unit	Integr.
JUnitCategorizer examples	54	24	30	24	30
TOPdesk application	9	6	3	3	6
TOPdesk application-utils	95	90	5	91	4
TOPdesk core	297	190	107	210	87

Table 2.6: *Number of detected unit and integration tests for different Java projects, with the Java core packages on the whitelist.*

Project name	TP	FP	TN	FN	Acc	Pre	Spe	Rec
JUnitCategorizer examples	19	5	25	5	0.82	0.79	0.83	0.79
TOPdesk application	3	3	3	0	0.7	0.5	1	1
TOPdesk application-utils	90	0	4	1	0.99	1.0	0.80	0.99
TOPdesk core	179	11	107	0	0.963	0.942	1.00	1.00

Table 2.7: *Performance on different Java projects, with the Java core packages on the whitelist. TP are the True Positives, FP are the False Positives, TN are the True Negatives, FN are the False Negatives, Acc is Accuracy, Pre is Precision, Spe is Specificity and Rec is Recall.*

		Conservative		`util` allowed		Core allowed	
Project name	Total	Unit	Integr.	Unit	Integr.	Unit	Integr.
Apache Commons Collections	1005	232	773	232	773	242	763
Apache Commons Digester	200	17	183	17	183	18	182
JFreeChart	2209	194	2015	236	1973	290	1919
Maven core	237	15	222	15	222	15	222
Maven model	148	96	52	96	52	96	52
TOPdesk application	9	3	6	3	6	6	3
TOPdesk application-utils	95	72	23	90	5	90	5
TOPdesk core	297	120	177	190	107	190	107

Table 2.8: *Number of detected unit and integration tests for different Java projects. The "conservative" approach has only handpicked Java core classes on the whitelist, "`util` allowed" has the entire `java.util` package on the whitelist and "core allowed" has all Java core packages on the whitelist.*

atomicity. The metrics on TOPdesk's application-utils and core test suites have exactly the same value as with only the `java.util` package on the whitelist. The accuracy and precision on the JUnitCategorizer examples and TOPdesk application test suites drops, in the latter case because it uses `java.io.File`, which is now whitelisted.

JUnitCategorizer produces the best results if we allow the entire `java.util` package and thereby do not try to minimize the number of false positives. It has an accuracy of 95.8% (436 out of 455) over all the test cases we used in the analysis. The most conservative approach, where we do not depend on the programmer correctly using default locales, timezones and dates, has an accuracy of 80.4% (366 out of 455) over all the test cases. Allowing all Java core classes correctly distinguishes between unit and integration tests in 94.5% (430 out of 455) of the cases, but has the most false positives.

We expect that if JUnitCategorizer would use dynamic taint analysis as described in section 2.2.5, it will match or improve on the best results currently, while still using the conservative approach. This is an interesting research question for future work.

2.5 Application on open source projects

We implemented JUnitCategorizer to distinguish between unit and integration tests, so we can inspect which type is the best candidate for boiler-plate removal. Besides the TOPdesk code base, we selected test suites from several open source projects for inspection. To get the division between unit and integration tests, we applied JUnitCategorizer on them. This will also provide some insights into how unit and integration tests are distributed in those test suites. Table 2.8 lists the results of applying the tool with the same whitelist configurations as the three accuracy tests.

It is noteworthy that none of the JUnit test suites consists solely of what we define as unit tests. To get a more clear insight into the distribution of unit or integration tests, we

Project name	Total	Unit	Integr.	% Integr.	Classification
Apache Commons Collections	1005	232	773	76.9	mostly integration
Apache Commons Digester	200	17	183	91.5	purely integration
JFreeChart	2209	236	1973	89.3	purely integration
Maven core	237	15	222	93.7	purely integration
Maven model	148	96	52	35.1	mostly unit
TOPdesk application	9	3	6	66.7	mostly integration
TOPdesk application-utils	95	90	5	5.2	purely unit
TOPdesk core	297	190	107	36.0	mostly unit

Table 2.9: *Classification of test suites on the percentage of tests detected as integration.*

classify test suites by the percentage of tests that are detected as integration. We use four classifications: purely unit (0 - 15% integration), mostly unit (15 - 50%), mostly integration (50 - 85%) and purely integration (85 - 100%). This is a somewhat arbitrary scale, but it is only used to get an indication of the percentage of integration tests. The classifications, based on the results with `java.util` whitelisted, are listed in table 2.9.

Table 2.9 shows that only three test suites are classified as unit tests and the other five consist mainly of integration tests. A quick look at the different test suites shows that these classifications match the test suites. For example, Maven core really is an integration test suite, testing the interactions between the different Maven components, whereas the Maven model package consists of simple classes describing the conceptual model of Maven, thus requiring simple tests that are mostly unit tests. The Apache Commons Digester tests several aspects and extensions of an XML digester. Almost every test integrates an aspect or extension, implemented as a Java class, with the main `Digester` class, so those tests use at least two classes and are integration.

2.6 Conclusion

A shown in the previous sections, JUnitCategorizer is able to detect a CUT reliably. Apart from some hard border cases it can also detect the difference between a unit and integration test very well. Our application on several Java projects has given us the division between unit and integration tests needed for the remainder of this thesis. It additionally provided insight into the types of test suites, showing that most test suites consist mainly of integration tests.

Chapter 3

Finding extension points for Project Lombok in developer testing

In this chapter we will look at possible extensions of Project Lombok. We first determine which type of tests to focus on: unit tests, integration tests or both. With that focus set, we will search for boiler-plate code and common testing errors and propose solutions to remove the boiler-plate and alleviate test errors.

3.1 Focus

In the previous section, we classified several test suites into five categories in table 2.9: purely unit test, mostly unit tests, mixed, mostly integration tests and purely integration tests. We would like to establish which type of tests is the best candidate for boilerplate removal or common test error alleviation. We try to track down boilerplate code by looking at duplication of code. We will apply the mutation testing technique to detect common testing errors. Additionally, we will also look at the code coverage metric.

3.1.1 Duplication

We detect duplicate pieces of code using the CCFinderX tool as introduced by Kamiya et al. [10]. Duplicate code, or code clones, might resemble boiler-plate code, so a large percentage of code clones contains more opportunities for boiler-plate.

There are several algorithms for detecting code clones, for example comparing code line-by-line or even by comparing (small groups of) characters. CCFinderX uses token based duplication detection, where the groups of characters that are compared are sensible combinations for individual programming languages. Compared to character or line based duplication detection, token based duplication detection is much less susceptible to insignificant textual differences, like whitespace. A parser is used to transform the stream of characters into tokens. However, since every programming languages has different

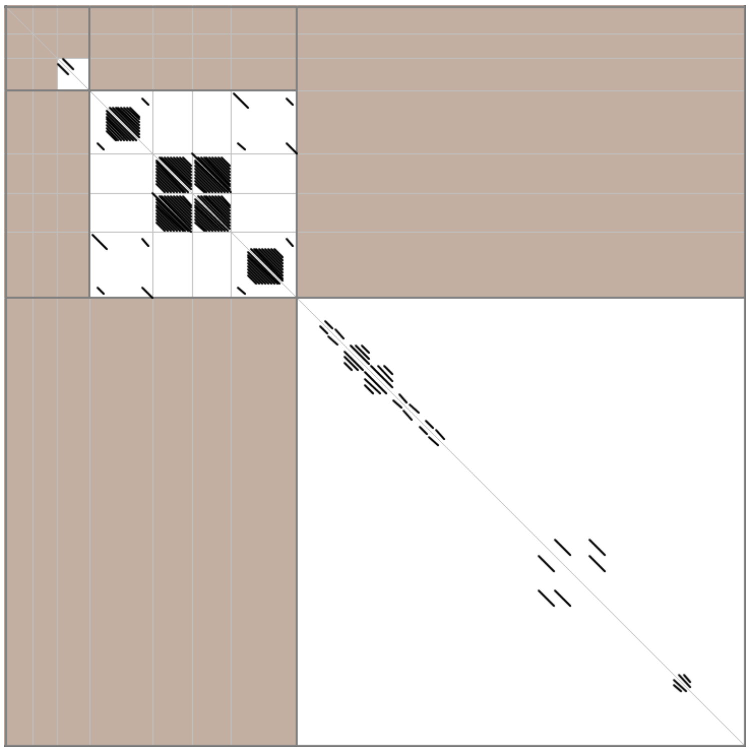

Figure 3.1: Scatter plot showing the duplicate lines of code in TOPdesk's application-utils test suite.

syntax, CCFinderX requires a parser for each programming language. Luckily, CCFinderX includes parsers for several programming languages, including Java.

For this experiment we look at three line-based metrics: *SLOC*, *CLOC* and *CVRL*. SLOC is the count of lines, excluding lines without any valid tokens. The tokens are specific to CCFinderX, for example the definitions of simple getter and setter methods in a Java source file will be entirely neglected by CCFinderX. CLOC is the count of lines including at least one token of a code fragment of a code clone. CVRL is the ratio of the lines including a token of a code fragment of a code clone. $CVRL = CLOC/SLOC$.[1]

CCFinderX also outputs a scatter plot which indicates where the code clones are located, for example figure 3.1, which is the scatter plot of the duplications in TOPdesk's application-utils test suite. The source code lines are positioned along both the x and y axes. If two lines of code are duplications of each other, a point is plotted at their coordinates. The diagonal, where every line of source code is compared to itself, is colored gray. The boxes with thick gray borders indicate packages and the smaller boxes with light gray borders represent classes. Figure 3.1 shows us that the fifth and sixth class are almost fully duplicated between themselves. Additionally the large class in the bottom right corner has some occurrences of duplication within itself.

[1]http://www.ccfinder.net

Note that we are interested in duplications in the test code, not the code under test, so we applied CCFinderX only on the folder containing the test code. In all cases, the folder containing the tests is different from folder containing the code under test.

3.1.2 Mutation testing

Mutation testing is a technique where the code of a program is mutated before a test run to see if this mutation causes tests to fail. If the tests fail then the mutation is said to be killed; if the tests pass, in spite of running with modified code, then we say the mutation has survived. The percentage of mutations killed can then be used as a measure of test adequacy. We however, will apply mutation testing to determine whether mutators typically survive in unit or integration tests. Additionally, mutators that typically survive might indicate a (common) testing error, so we will also keep the mutation test results in mind when looking for common testing errors.

As indicated by an extensive literature survey [11], mutation testing has been a field of research for several decades, tracing back to the seventies. Mutation testing for Java started in 2001 with Jester [12] which led to several other approaches, like MuJava [13], Jumble [14] and Javalanche [15]. The mutation testing tool we use is called PIT[2], whose earliest versions were based on Jumble. PIT was preferred over the other mutation testing tools, because it is in active development, has a reasonable amount of documentation and has support for many build systems, including Maven, so it was easy to apply to our projects under test.

We used all the default mutators that PIT provides[3]:

- Conditionals Boundary Mutator, e.g. change $>$ to $\geq$ and vice versa.
- Negate Conditionals Mutator, e.g. change != to == and $>$ to $\leq$.
- Math Mutator, e.g. change + to - and vice versa.
- Increments Mutator, e.g. change *i*++ to i$--$.
- Invert Negatives Mutator, e.g. change *-i* to *i*.
- Return Values Mutator, e.g. always return 0 for integers.
- Void Method Calls Mutator, remove methods that return void.

3.1.3 Code coverage

Initial runs with mutation testing showed that many mutations are not killed because the code they mutate is not covered by a test. We therefore decided to also get an indication of how much code is covered by the test suites. Because we only need an indication, we opt for the relative simple metric line coverage, i.e. the ratio of lines of code covered by a test suite to the total number of lines of code. A quick search in a survey of coverage tools for Java [16] showed that Cobertura is a feasible option. Combining this fact with our previous experience with Cobertura, we decided to use it to calculate the line coverage.

[2]http://www.pitest.org

[3]http://pitest.org/quickstart/mutators/

Project name	Classification	CVRL	Mutation	Coverage
Apache Commons Collections	mostly integration	0.3761	n.a.	0.8088
Apache Commons Digester	purely integration	0.3612	0.470	0.7133
JFreeChart	purely integration	0.54866	0.2992	0.55908
Maven core	purely integration	0.2033	0.0152	0.1314
Maven model	mostly unit	1.00	0.01047	0.0227
TOPdesk application	mostly integration	0.12	0.034	0.0739
TOPdesk application-utils	purely unit	0.257	0.522	0.613
TOPdesk core	mostly unit	0.529	0.447	0.4346

Table 3.1: *Result of several metrics for test suites and their classifications. CVRL is the ratio of the lines including a token of a code fragment of a code clone. Mutation indicates the ratio of mutations that were killed. Coverage indicates the ratio of code covered by the tests.*

Project name	Classification	DUT	DIT
TOPdesk application	mostly integration	0	2
TOPdesk application-utils	purely unit	48	1
TOPdesk core	mostly unit	11	23

Table 3.2: *Duplication in the TOPdesk test suites. DUT indicates the number of duplications in unit tests, DIT indicates the number of duplications in integration tests.*

3.1.4 Experimental results

We applied CCFinderX, PIT and Cobertura to the same set of test suites we used in the previous chapter. Table 3.1 lists the results. We could not determine a mutation ratio for Apache Commons Collections because PIT would not run properly on the test suite.

We first inspected the very high CVRL value of 1.00 for the Maven model test suite, because this value means that all tests are in some way duplicated. Closer inspection shows that indeed all the test classes are identical, differing only in the class they test as illustrated in listings 3.1 and 3.2. These listings also show that only the `hashCode()`, `equals(Object o)` and `toString()` methods are tested, whereas the classes under test contain many more methods. This explains the very low coverage ratio of roughly 2%. A test suite with a low coverage rating is likely to also have a low mutation rating, because a mutation in code that is not covered by tests will survive.

The results in table 3.1 do not clearly point out a relation between the classification of the test suite and a large number of clones or a low ratio of killed mutations. We therefore inspected the TOPdesk test suites at the test method level to get a more fine-grained insight. The results of this inspection are reported in table 3.2, and show that duplication occurs in both unit and integration tests.

There is no apparent relation between the problems we sought for and the distinction between unit and integration test, so we will focus our efforts to remove boiler-plate code and alleviate test errors on both unit and integration tests.

```java
public class ActivationFileTest extends TestCase {
    @org.junit.Test
    public void testHashCodeNullSafe() {
        new ActivationFile().hashCode();
    }

    @org.junit.Test
    public void testEqualsNullSafe() {
        assertFalse( new ActivationFile().equals( null ) );
        new ActivationFile().equals( new ActivationFile() );
    }

    @org.junit.Test
    public void testEqualsIdentity() {
        ActivationFile thing = new ActivationFile();
        assertTrue( thing.equals( thing ) );
    }

    @org.junit.Test
    public void testToStringNullSafe() {
        assertNotNull( new ActivationFile().toString() );
    }
}
```

Listing 3.1: *Test class for ActivationFile in Maven model*

```java
public class ActivationOSTest extends TestCase {
    @org.junit.Test
    public void testHashCodeNullSafe() {
        new ActivationOS().hashCode();
    }

    @org.junit.Test
    public void testEqualsNullSafe() {
        assertFalse( new ActivationOS().equals( null ) );
        new ActivationOS().equals( new ActivationOS() );
    }

    @org.junit.Test
    public void testEqualsIdentity() {
        ActivationOS thing = new ActivationOS();
        assertTrue( thing.equals( thing ) );
    }

    @org.junit.Test
    public void testToStringNullSafe() {
        assertNotNull( new ActivationOS().toString() );
    }
}
```

Listing 3.2: *Test class for ActivationOS in Maven model*

3.2 Detected problems

We manually inspected the Maven model and the TOPdesk application, application-utils and core projects to see which code can be marked as boiler-plate. Additionally, we inspected the code to see if we could find some common testing errors, that might be alleviated in the same way we will try to remove boiler-plate code.

3.2.1 Duplication in tests

The most obvious boiler-plate code is duplicated code. We split these into two categories: inter test class duplication and intra test class duplication. Inter test class duplication is duplication that occurs between different test classes, e.g. common tests for `hashCode()` and `equals(Object o)` applied on each class under test. A good example of inter test class duplication is the entire Maven model test suite, which is the same set of tests duplicated over and over again for every model class.

Intra class duplication is duplication that occurs inside a single test class. Listing 3.3 shows a typical example of code duplication in a test class from the TOPdesk core project. There are seven test methods like the two listed here, testing various inputs to the `DateParser` class.

```
@Test
public void testDayFirstLongWithDashes() {
  Parser parser = new DateParser(new SimpleDateFormat("dd-MM-yy"));
  assertEquals(getDate(2009, Month.JAN, 1), parser.parse("1-1-2009"));
  assertEquals(getDate(2010, Month.DEC, 1), parser.parse("1-12-2010"));
  assertEquals(getDate(2011, Month.JAN, 12), parser.parse("12-1-2011"));
}

@Test
public void testDayFirstLongWithoutDashes() {
  Parser parser = new DateParser(new SimpleDateFormat("dd-MM-yy"));
  assertEquals(getDate(2009, Month.JAN, 1), parser.parse("1␣1␣2009"));
  assertEquals(getDate(2010, Month.DEC, 1), parser.parse("1␣12␣2010"));
  assertEquals(getDate(2011, Month.JAN, 12), parser.parse("12␣1␣2011"));
}
```

Listing 3.3: *Test class for DateParser in TOPdesk core.*

The TOPdesk projects have very little inter test class duplication, only the four test classes testing `StringShortener` contain duplications between them, as can be seen in figures 3.1, 3.3 and 3.4. The Maven model project exclusively has inter class duplication, as can be seen in figure 3.2. This indicates that typical inter test class duplication is concerned with testing similar functionality between different classes under test, concretely the `hashCode()`, `equals(Object o)` and `toString()` methods.

In the TOPdesk projects the majority of detected duplications are intra test class. In fact, if the four test classes testing `StringShortener` would have been put in a single

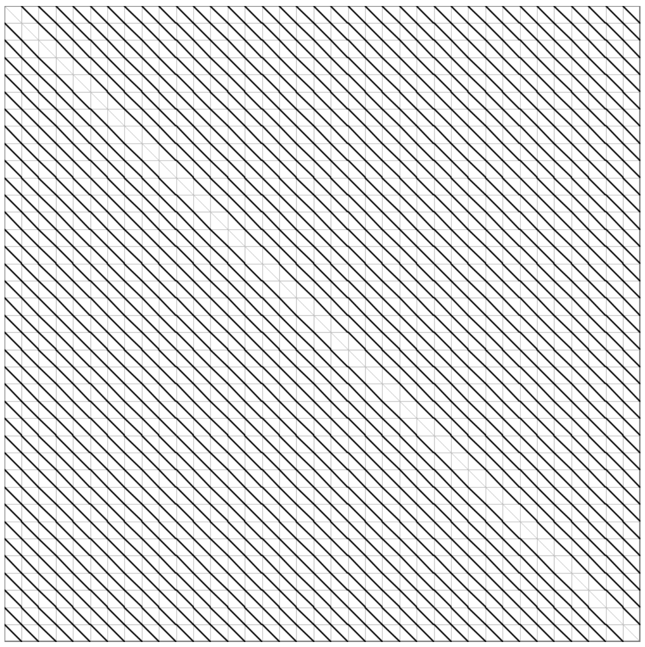

Figure 3.2: *Scatter plot showing the duplicate lines of code in Maven's model test suite.*

`StringShortenerTest` class, all duplication would be intra class. We found that typical intra test class duplication consists of checking various inputs to methods, in the same way as happens in listing 3.3.

3.2.2 No coverage

The biggest and most easily detected problem with tests is the lack thereof. Test code coverage is the typical metric to show how much and which code is actually touched by tests, but it will hardly say anything on the quality of the tests. The biggest gain from code coverage is that it indicates how much code is not tested. Ideally, we want a code coverage of 100% with quality tests, e.g. tests that are not written to just touch the code once just to get it covered, but tests that extensively test the code to verify its correct operation in multiple cases.

A metric that can give an indication about the quality of tests is the *assertion density*, i.e. the amount of assertions per line of code. There is a negative correlation between the *fault density*, i.e. the amount of faults per line of code, and assertion density. Thus, code with a high assertion density implies a low fault density [17].

The "coverage column" in table 3.1 shows that for example the Maven model project has a very low coverage (2%) and the TOPdesk projects having varying coverages, 7% for applications, 61% for application-utils and 43% for core. As mentioned before, the Maven model test suite tests only the `hashCode()`, `equals(Object o)` and `toString()` methods. The classes under tests primarily consist of so called model classes, e.g. classes

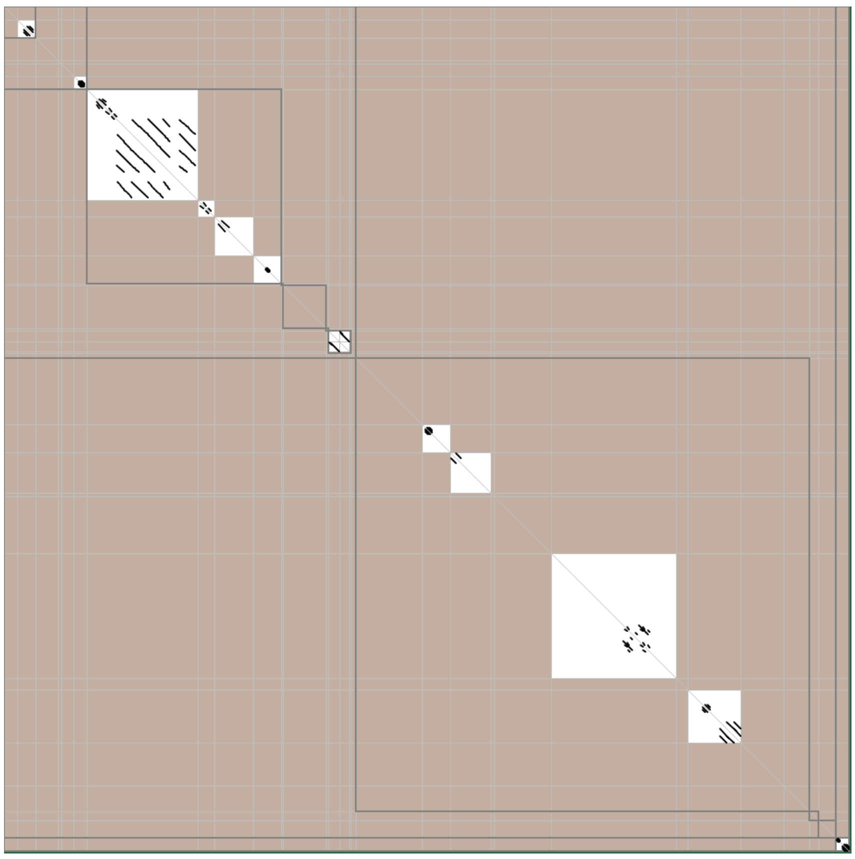

Figure 3.3: *Scatter plot showing the duplicate lines of code in TOPdesk's core test suite.*

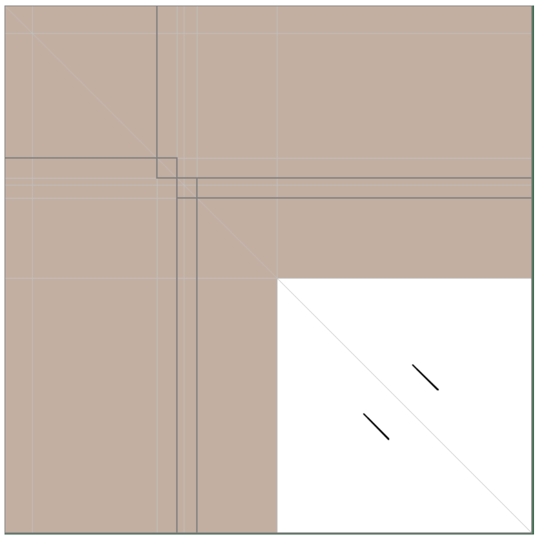

Figure 3.4: *Scatter plot showing the duplicate lines of code in TOPdesk's application test suite.*

that represent the data underlying the object [18]. A model class primarily contains methods to set and get the fields of the class. A large improvement in code coverage can be made for the Maven model test suite by adding tests for getting and setting fields in the classes under test.

The TOPdesk application test suite tests only a few classes. To be precise, all code coverage is located in eight out of 201 classes in the package. Even though those 201 classes include interfaces and abstract classes which can not be tested, it is still a very low number of classes under test. We inspected the untested classes and many of them have methods to get the values of their fields and some override the `hashCode()`, `equals(Object o)` and `toString()` methods. Testing these methods would greatly improve coverage in the application test suite.

The TOPdesk application-util test suite achieves a reasonable level of coverage, with five out of 14 classes untested. These untested classes are small utility classes with methods to manipulate data. To increase coverage in this project, we need to know exactly what every method is supposed to do and verify it with different inputs.

The TOPdesk core project consists of a mix of model classes and utility classes. To increase coverage in this package, one needs to apply a combination of the previously mentioned methods.

3.2.3 Corner cases

When conditional statements are used, decisions in the code will be made based on the input. When testing conditional statements, it is impossible to test all the inputs, so we need to select an interesting set of inputs, such that we can verify that all conditional statements are executed as we expect them to execute. Interesting inputs are inputs around the corner cases, mostly used to verify if a programmer correctly used the $<$, $\leq$, $>$, $\geq$, != or == operators. Mutation testing is an effective way to be notified about possibly missed corner cases, since it will mutate the operator and expect the current tests to fail. A low rate of killed mutations may be an indication that the corner cases are not adequately tested.

3.2.4 Equals and HashCode

In addition to easily being duplicated, the tests for the `equals(Object o)` and `hashCode()` methods are often not complete. The `equals(Object o)` and `hashCode()` methods have strict contracts, as can be read in the Java documentation[4]:

> The equals method implements an equivalence relation on non-null object references:
>
> - It is reflexive: for any non-null reference value x, x.equals(x) should return true.

[4]http://docs.oracle.com/javase/7/docs/api/java/lang/Object.html

- It is symmetric: for any non-null reference values x and y, x.equals(y) should return true if and only if y.equals(x) returns true.
- It is transitive: for any non-null reference values x, y, and z, if x.equals(y) returns true and y.equals(z) returns true, then x.equals(z) should return true.
- It is consistent: for any non-null reference values x and y, multiple invocations of x.equals(y) consistently return true or consistently return false, provided no information used in equals comparisons on the objects is modified.
- For any non-null reference value x, x.equals(null) should return false.

The general contract of hashCode is:

- Whenever it is invoked on the same object more than once during an execution of a Java application, the hashCode method must consistently return the same integer, provided no information used in equals comparisons on the object is modified. This integer need not remain consistent from one execution of an application to another execution of the same application.
- If two objects are equal according to the equals(Object) method, then calling the hashCode method on each of the two objects must produce the same integer result.
- It is not required that if two objects are unequal according to the equals(java.lang.Object) method, then calling the hashCode method on each of the two objects must produce distinct integer results. However, the programmer should be aware that producing distinct integer results for unequal objects may improve the performance of hash tables.

It is adamant that this contract is adhered to, because a bug in the `equals(Object o)` or `hashCode()` might show up in unexpected places. For example, take a certain object *o* with a bug that causes the equals method to not be reflexive, i.e. `x.equals(x)` returns `false`. Suppose we add this object to a `Set` *s*, e.g. `s.add(o)`. Now if we ask *s* if it contains *o*, i.e. `s.contains(o)`, it will return `false`, even though we just added it. The `contains(Object o)` method returns `true` if and only if this set contains an element *e* such that `(o == null ? e == null : o.equals(e))`[5]. Since `o.equals(o)` returns `false`, the `contains(Object o)` method will also return `false`. This simple example shows why `equals(Object o)` and `hashCode()` should be tested thoroughly.

[5]http://docs.oracle.com/javase/7/docs/api/java/util/Set.html#contains(java.lang.Object)

3.2.5 An (unmodifiable) view or copy of collections

A problem that sometimes arises in practice, but can not be detected using our methods described here, occurs when working with members of Java's `Collections` framework or subclasses thereof. Since it is a potential improvement to tests, we do consider it for an extension to Project Lombok.

A field f in a class can be of type `Collection`. How it will be set and used, is up to the programmer. If the programmer decides to store a reference to an existing `Collection` x, we call f a *view* of x. The property of a view is that operations on the underlying `Collection`, in this case x, reflect in the view f and vice versa.

Besides a view, f can also contain a *copy* of x. Operations on a copy do not reflect on the original collection and vice versa. If we, for example, add an item i to copy f, the original `Collection` x will not contain i.

In the `Collections` class, Java provides methods for creating unmodifiable collection types. These `UnmodifiableCollections` throw an `UnsupportedOperationException` if any modifier method is called on the `UnmodifiableCollection`, e.g. `add(Object o)` or `remove(Object o)`. An `UnmodifiableCollection` is a view of the underlying `Collection`, so it can be changed by editing the underlying collection. If f is an `UnmodifiableCollection` with underlying collection x, we say that f is an *unmodifiable view* of x. An *unmodifiable copy* of x can be created by setting f to be an `UnmodifiableCollection` with a copy of x as underlying collection. As one might expect, an unmodifiable copy f can neither be changed directly nor indirectly, since changing x does not reflect in f. We say that f is *immutable*, as defined by the Java documentation[6].

It does not always matter how the `Collection` is used, but in those cases it does, programmers tend to not explicitly test nor document whether they actually use a *view*, an *unmodifiable view*, a *copy* or an *unmodifiable copy*.

3.3 Solutions

In this section, we propose solutions for the detected problems. Some solutions are suitable as extensions of Project Lombok, while others refer to already existing testing tools.

3.3.1 Duplication in tests

The inter test class duplication we ran into in the Maven model package can be reduced by automatically generating tests for the `hashCode()`, `equals(Object o)` and `toString()` methods. We will create an extension to Project Lombok that can be used in a test class to generate test methods for these three methods. That will greatly reduce the duplication in at least the Maven model project, but will also reduce the size of the test code in all projects.

[6]http://docs.oracle.com/javase/7/docs/technotes/guides/collections/overview.html

Intra test class duplication can be reduced by being able to separate the inputs and their expected outputs from the code that actually tests an implementation. The long list of near identical assertions can then be turned into a collection of inputs and expected outputs and a single verifying test method. This concept is called *Parameterized Unit Tests* [19] and JUnit 4 has built-in support for it. The JUnit implementation is adequate and would solve a lot of the observed intra test class duplication, so we will not further be concerned by intra test class duplication.

3.3.2 No coverage

The ideal way of testing would be to not having to write tests at all, but have automated test case generation that will test your implementation. The biggest challenge for those generators is to determine what the expected output of applying a method on some input will be. This is very difficult, if not impossible, so even automated test case generators need a programmer to determine what the expected output is. An example of this research is JMLUnit, which generates unit tests based on annotations in the Java Modeling Language (JML) [20]. Examples of tools that request user input for generated test cases are Parasoft Jtest or CodePro AnalytiX [21].

However, we do propose the automated generation of tests for methods that get and set fields, by providing an implementation with a single default value. This single default value is needed, because we cannot make assumptions about the fields under test. If the setter or getter method can accept this default value, then the automatically generated test can be used to verify the correctness of the setter and getter method partially. If the getter or setter cannot accept a default value, then the programmer will have to provide his own tests. Note that the generated tests are not adequate, since they will only apply a single value, so a programmer should provide tests for different inputs (like corner cases) either way. The generator's goal is to help programmers to find simple bugs with getter and setter methods in boiler-plate free manner.

3.3.3 Corner cases

To test corner cases, a programmer needs to specify exactly what inputs should result in which output. By stating these specifications in JML, unit tests can automatically be generated. Parasoft Jtest or CodePro AnalytiX will try to find corner cases automatically and will then ask the programmer what the output should be. These tools are very mature and suitable for testing corner cases, so we will not consider corner cases for an extension of Project Lombok.

3.3.4 Equals and HashCode

Testing `equals(Object o)` and `hashCode()` extensively can generate a lot of boiler-plate and is quite hard to get correct. A former TOPdesk employee created the `EqualsVerifier`[7],

[7]http://code.google.com/p/equalsverifier/

a tool that extensively inspects the `equals(Object o)` and `hashCode()` methods of the class it is passed. This tool has several configuration possibilities, but also provides assumption free verification, where only the class under test is passed. We will use this assumption free version in our generated tests for the `equals(Object o)` and `hashCode()` methods.

3.3.5 An (unmodifiable) view or copy of collections

Verifying whether a field containing a `Collection` is a view, unmodifiable view, copy or unmodifiable copy requires extensive testing. To be thorough, a programmer should test all modifying methods of `Collection` to ensure that the correct behavior is exhibited. This results in a lot of boiler-plate code to test a single field, so this is an ideal opportunity for an extension to Project Lombok.

3.4 Conclusion

In this chapter, we determined that boiler-plate code and common testing problems occur in both unit as integration tests. We detected the following problems: duplicate code in tests, tests inadequately covering code, corner cases not fully tested, incomplete tests of the `equals(Object o)` and `hashCode()` methods and untested behavior of collections. We proposed solutions for all problems and will design extensions to Project Lombok to reduce the duplicate code between test classes, increase code coverage, generate correct tests of the `equals(Object o)` and `hashCode()` methods and verify the behavior of collections.

Chapter 4

Extending Project Lombok with Java Test Assistant

In this chapter we will explain the implementation of *Java Test Assistant* (JTA) in detail in section 4.1. We set up an experiment to determine how JTA can help developers. The experiment is explained in section 4.2 and the results are presented in section 4.3.

4.1 Implementation

The goal of our implementation is to create a tool that can achieve the goals we specified in section 3.3, i.e. lower the duplication in test code, increase code coverage, correctly test the `equals(Object o)` and `hashCode()` methods and generate tests to verify if a field containing a `Collection` is a view, unmodifiable view, copy or unmodifiable copy.

The original vision we had for Java Test Assistant (JTA), was based on a tight IDE integration, similar to what Project Lombok offers with its annotations. For example, in Project Lombok you can set the `@Data` annotation on a certain class `SomeClass` and it will generate the `equals(Object o)`, `hashCode()` and `toString()` methods, as well as getter methods for all fields and setter methods for all non-final fields. These generated methods will not appear in the source file `SomeClass.java`, but they are available to the programmer. Project Lombok can be integrated with the Eclipse IDE, in which case the generated methods will be available in the outline of `SomeClass` and through the autocomplete function.

Unfortunately, Project Lombok performs complex interactions with the IDE, using internal Java classes to achieve this functionality. As a result, the information JTA needs about the classes under test is not available, rendering a direct integration into Project Lombok infeasible.

To preserve Project Lombok's declarative style, we opted to create an annotation named `TestThisInstance`, which must be placed on an instance field of the class under test

```java
public class SomeClassTest {
  @TestThisInstance(getters={"*"}, setters={"someField"})
  SomeClass instance;

  // Developer written test methods for SomeClass here
}
```

***Listing 4.1:** Example usage of JTA's `TestThisInstance` annotation.*

in a test class, as shown in listing 4.1. Since version 6, Java provides an Application Programming Interface for writing an annotation processor. An annotation processor works at compile time and basically takes the annotation and the context in which the annotation was placed and provides these to the developer.

We want to create a new test class containing generated test methods, so we take the annotation and the context, inspect the annotated instance, generate test methods and write it to a new Java source file. This new Java source file then needs to be compiled and included in the test phase; a task that can easily automated in build systems like Ant and Maven, resulting in an unobtrusive, one time setup, manner of generating new tests.

An annotation can contain data, as is demonstrated in listing 4.1 by the `getters` and `setters` assignments. We use this feature to specify which functionalities should be applied in the annotation processor, e.g. which test should we generate and which should we skip. The functionalities included in the annotation processor are our implementations of the solutions from section 3.3 and are listed in the following subsections.

4.1.1 Assumption free tests

As mentioned in the previous chapter, we want to automatically generate tests without making assumptions on input or output values of methods. The most important feature of JTA is the ability to generate default instances of Java objects for testing. We create such default instances using the following heuristic:

- Java primitives will be initialized with the value 1 or true.
- `List`, `Map`, `Set` and `Collection` interfaces will be initialized as `ArrayList`, `HashMap`, `HashSet` and `HashSet` respectively.
- For enumerations, return their first `EnumConstant`.
- For arrays, return an array of length 1 containing the default value of its component.
- For Objects, try to instantiate it with the constructor that requires no arguments. If this fails, try all constructors with default instances for arguments.
- If all above fails and we are not instantiating a class under test, use `Mockito` to create a mock and return it.
- If all above fails and we are not instantiating a class under test, return `null`.

We impose a requirement on the default instance of a class under test: it may not be a mock or `null`, because testing on a mock or `null` has no use.

```java
public class SomeClass {
  int someInt; // Java initializes this field as 0
  ...
  public void setSomeInt(int i) {
    // Setter not implemented
  }
  public String getSomeInt() {
    return someInt; // always returns 0
  }
}

public class SomeClassTest {
  SomeClass instance = new SomeClass();
  @Test public void badTest() { // Passes
    int test = 0;
    instance.setSomeInt(test);
    assertEquals(test, instance.getSomeInt());
  }
  @Test public void betterTest() { // Fails
    int test = 1;
    instance.setSomeInt(test);
    assertEquals(test, instance.getSomeInt());
  }
}
```

Listing 4.2: *Example showing the need for default instances other than Java's initial values for fields*

We explicitly chose to not use the default values that Java uses to initialize fields, i.e. false for booleans, 0 for all other primitives and `null` for reference types[1]. Testing a value for a field with which it also is initialized, is ineffective, as demonstrated in listing 4.2. The `badTest` incorrectly assumes the test value was set with the setter method and will pass. The `betterTest` will fail, indicating something is wrong with the getter or setter method, since it will expect 1 but the actual result is 0.

The only assumption we make with this heuristic is that for every test using a default instance, its method under test can accept such an instance as input. For example, if we test a set method for a field of type integer, the set method should accept the default instance, i.e. 1, as input value or our approach will not work.

4.1.2 Duplication in test

With this feature we want to address the test methods testing `toString()`, `equals(Object o)` and `hashCode()`. We noticed that these tests are often duplicated or omitted. A valid reason for omission is that the class under test does not override any of the three methods, but instead relies on the default implementation in `Object`.

[1]http://docs.oracle.com/javase/specs/jls/se7/html/jls-4.html#jls-4.12.5

The Java documentation mentions the following about the `toString()` method: "In general, the toString method returns a string that "textually represents" this object. The result should be a concise but informative representation that is easy for a person to read. It is recommended that all subclasses override this method."[2]. A computer cannot simply detect whether a representation is concise, informative and easy to read for a person. There are however three easy to detect cases that are definitely not informative, namely the `null` value, an empty string and the default value, as created by the `toString` method of `Object`. So the test generated by the `TestThisInstance` annotation for the `toString()` method verifies if the `toString()` method of the class under test does not return the `null` value, an empty `String` or the default value as created by `Object.toString()`. This implies that JTA requires the developer to override the `toString()` method.

For the `equals(Object o)` and `hashCode()` methods we will use the `EqualsVerifier`[3], which is quite strict and requires all best practices to be applied when implementing `equals(Object o)` and `hashCode()`.

`EqualsVerifier` provides several configuration options, which are not used in the code generated by the `TestThisInstance` annotation for two reasons. First, most configuration options are options for the developer, because the `EqualsVerifier` could not determine them heuristically, and neither can the `TestThisInstance` annotation. Second, to provide these options, we need to add them to the options of the `TestThisInstance` annotation. We aimed to keep our annotation simple and concise, and adding all options of `EqualsVerifier` would clutter up the annotation. Additionally, if a developer needs to configure `EqualsVerifier`, for example when testing the `equals(Object o)` in a class hierarchy, it would be the best practice to configure `EqualsVerifier` directly in a test.

4.1.3 No coverage

To allow programmers to easily verify the correct working of their getter and setter methods for fields of a class, the `TestThisInstance` annotation will generate tests for specified fields (like the "setters" in listing 4.1), or, by default, test all fields. To test all fields, we generate a test that uses Java's reflection mechanism to gather all declared fields, find out which ones have a getter or setter method defined and test those methods. JTA follows the JavaBeans Specification[4], which states that a setter method for field `someField` of type `someType` is required to be `setSomeField` with one single parameter of type `someType`. A getter method for field `someField` of type `someType` is required to be `getSomeField` with return type `someType`, with the exception that if `someType` is `boolean`, then the getter method is required to be named `isSomeField`. If a field does not comply with these requirements, it is skipped in the test.

[2]http://docs.oracle.com/javase/7/docs/api/java/lang/Object.html#toString()

[3]http://code.google.com/p/equalsverifier/

[4]http://www.oracle.com/technetwork/java/javase/documentation/spec-136004.html

```java
public class SomeClass {
  String someString;
  ...
  public void setSomeString(String str) {
    someString = "Hello␣World!" + str;
  }
  public String getSomeString() {
    return someString.substring(12); // Chop Hello World! off
  }
}
```

Listing 4.3: *Example indicating the applicability of testing a getter method using its setter method*

```java
public class Point {
  int x;
  int y;
  ...
  public void setX(int newx) {
    x = newx;
  }
  public int getX() {
    return x;
  }
  public void setY(int newy) {
    x = newy; // Error
  }
  public int getY() {
    return x; // Error
  }
}
```

Listing 4.4: *Example showing an error that would not be detected when testing getter methods using setter methods.*

There are two ways to test getter and setter methods. If we have both a getter and a setter method for a field, we can first set the value using the setter, retrieve it using the getter and compare if we have the original value. The second way is to set the field using Java's reflection mechanism and then use the getter to see if the correct field was gotten. Similarly for the setter method, the setter method is called and the test then uses reflection to see if the correct field was set. Both methods have their use cases, as is indicated by listings 4.3 and 4.4. Listing 4.3 shows an example where using reflection to inspect the field will not work, since a different value is stored in the field then the one passed into the setter method. However, the getter method returns the original value, so it may be considered a correct implementation. Listing 4.4 shows an example where calling a setter and a getter will not detect the error that `getY` and `setY` operate on *x*. Calling `setY(value)` will ensure `getY` returns *value*, but when inspected using reflection the test detects that *y* is not set to *value*. To accommodate the two methods, the behavior of the tests can be toggled in the `TestThisInstance` annotation with a boolean `testGettersUsingSetters` flag.

```java
public class SomeClass {
  final String someString;

  public SomeClass(String str) {
    someString = str;
  }
  public String getSomeString() {
    return someString;
  }
}
```

Listing 4.5: *Example showing the setting of fields using the constructor.*

```java
public class SomeClass {
  String someString;

  public SomeClass(String str) {
    someString = str;
  }
  public String getSomeString() {
    return someString;
  }
  public void setSomeString(String newString) {
    // Do nothing
  }
}

public class SomeClassTest {
  SomeClass instance = new SomeClass("");

  @Test
  public void testSomeStringSetter() { // Passes
    String test = "";
    instance.set(test);
    assertEquals(test, instance.getSomeString());
  }
}
```

Listing 4.6: *Example showing a test that can not detect if the field is set by the setter or by the constructor.*

There are many classes that do not have setter methods for fields, but instead set their fields using the parameters of the constructor, as in listing 4.5. Since a mapping from the parameters of the constructor to the fields they set is not available without complex analysis, we cannot test the setting of fields in these cases. Using the reflection method of testing we can however test the getter method. This motivates the separated settings for setters and getters in the `TestThisInstance` annotation.

The potential setting of fields using the parameters of the constructor raises another problem. For the instantiation of the class under test we can not use a default instance constructed by our heuristic, because it will call the constructor with test values, so some fields might already be set to the values that the setter method will apply. This problem is illustrated in listing 4.6: the test would not be able to detect an empty setter method. We solve this by setting the field with Java's initial values, i.e. false, 0 or `null`, at the start of the test.

Of course, the generated tests for `toString()`, `equals(Object o)` and `hashCode()` also aid in increasing the coverage of a test suite.

4.1.4 An (unmodifiable) view or copy of collections

To verify if the getter of a field containing a `Collection` or `Map` provides this field as a view, we need to test if all *modification operations* on the field also reflect in the original collection, i.e. the one which is set using the setter method. We also need to test the other way around, i.e. modifications to the original collection need to be visible in the getter method. The modification operations of `Collection`, `List` and `Map` are listed in table 4.1. Note that the `l.subList()` method of some `List l` returns a view of (a part of) `l`, so modifications made on this sublist will reflect in the original list `l`. The same holds for the `m.entrySet()`, `m.keySet()` and `m.values()` methods on a `Map m`.

In a similar fashion as the view, we can also define test cases for a copy. All modification operations performed on the original collection of a copy should not reflect in the getter method and vice versa. The unmodifiable view and unmodifiable copy need the same set of tests as their modifiable counterparts, but require an `UnsupportedOperationException` to be thrown whenever a modification operation is called on the collection stored in the field.

We implemented four different test classes for the four different behavior types of collections. Each test class contains tests that cover the modification operations of `Collection`, `List` and `Map`, that are checked in the "covered" column of table 4.1. The developer using JTA specifies in the `TestThisInstance` annotation which fields should be tested for which behavior and the test will select the appropriate modification operations to test.

4.2 Experiment

Our experiment consists of annotating the test suites of several projects and measuring several metrics before and after applying JTA. The test suites we annotated are from the JFreeChart axis, Maven model, TOPdesk application, application-utils and core projects. When annotating tests, we removed tests that were superseded by the generated tests, i.e. the generated test asserts the same or even more than the already present test. For example, a test calls the `toString()` method on a default instance and checks if it does not return

Modification operation	`Collection`	`List`	`Map`	Covered
add(E)	✓	✓		✓
addAll(Collection<E>)	✓	✓		✓
clear()	✓	✓	✓	✓
iterator().next().remove()	✓	✓		✓
remove(E)	✓	✓		✓
removeAll(Collection<E>)	✓	✓		✓
retainAll(Collection<E>)	✓	✓		✓
add(index, E)		✓		✓
addAll(index, Collection<E>)		✓		✓
listIterator().next().set(E)		✓		✓
listIterator().next().add(E)		✓		✓
remove(index)		✓		✓
set(index, E)		✓		✓
all modifications on subList()		✓		
put(K, V)			✓	✓
putAll(Map<K, V>)			✓	✓
remove(K)			✓	✓
entrySet().setValue()			✓	✓
all modifications on entrySet()			✓	
all modifications on keySet()			✓	
all modifications on values()			✓	

Table 4.1: *Modification operators for `Collection`, `List` and `Map`. The covered column lists the modification operators that are covered by the tests generated by JTA.*

`null`. Our test for the `toString()` method tests this exact same assertion and more, so we remove the original test and let JTA generate a test.

We also inspected the code to find occurrences of (unmodifiable) views or copies. If the behavior is specified either in comments or in code (for example, the `unmodifiableList` method of `Collections` is called), we will test for the specified behavior. In all other cases we will assume that a view is used.

If no test class was present for a class, and the class has properties testable by JTA, we created a new test class containing only an instance definition and the annotation. The default settings of JTA, i.e. test all getters, setters, `equals(Object o)`, `hashCode()` and `toString()`, might not be applicable to all classes under test. We configured the annotation in each test class such that we can apply it in as many cases as possible. If `equals(Object o)`, `hashCode()` or `toString()` are not overridden, we will disable their respective tests.

4.2.1 Expectation

We added annotations and even new test classes to place the annotations in, if the class under test was previously untested. So we expect the code coverage to go up, since previously untested code will now be tested. Unless the added tests only exercise code that is not mutated by the mutation tool PIT, higher code coverage should lead to more killed mutations.

During the annotating of tests, we removed some tests, especially in the Maven model test suite. Since all tests in this test suite were duplicated, removing them should lead to a decrease in duplication. Additionally, removing methods also decreases the size of the test code.

So in summary, we expect JTA to increase the code coverage, decrease the number of survived mutations, decrease the amount of duplication in test code and decrease the size of test code.

4.2.2 Results

Tables 4.2 and 4.3 show the difference in metrics on the test suites before and after applying JTA. Note that we did not include the generated tests, since these are a direct, automatic translation of the `TestThisInstance` annotation, and are not written by developers themselves. Additionally, we used the Cloned Lines Of Code (CLOC) metric to list the change in duplicated test code, instead of the CVRL metric. The CVRL metric is defined as CLOC / SLOC, where SLOC is the count of lines, in our case the *Lines Of Test Code* (LOTC) excluding lines without any valid tokens. So since the LOTC increased by adding annotations, the CLOC increased, and the CVRL value would be lower, even if we did not remove any duplication.

Code coverage increased in all test suites, with the biggest improvement in Maven model and TOPdesk application, the two projects that originally had the lowest coverage.

The number of killed mutations also increased in all test suites, in roughly the same ratio as

	Before		After		Change	
Project name	Mutation	Cover.	Mutation	Cover.	Mutation	Cover.
JFreeChart axis	0.248	0.472	0.304	0.491	1.23	1.04
Maven model	0.0105	0.0227	0.0431	0.0953	4.10	4.19
TOPdesk application	0.0337	0.0739	0.0905	0.172	2.68	2.33
TOPdesk app-utils	0.522	0.613	0.529	0.637	1.01	1.04
TOPdesk core	0.447	0.435	0.469	0.523	1.05	1.20

Table 4.2: *Result of the mutation and coverage metrics for test suites before and after applying JTA. Mutation indicates the ratio of mutations that were killed. Coverage indicates the ratio of code covered by the tests. The change ratio is calculated by After / Before. For all values hold: the higher the better.*

```java
public class SomeClassTest {
  @TestThisInstance
  SomeClass instance;
}
```

Listing 4.7: *Testing `SomeClass` using JTA.*

```java
public class SomeClassTest {
  SomeClass instance;

  @Before
  public void before() {
    instance = new SomeClass();
  }

  @Test
  public void testToString() {
    assertNotNull("toString␣method␣returned␣null", instance.toString());
    assertFalse("toString␣method␣returned␣an␣empty␣String",
        instance.toString().equals(""));
    String defaultToString = instance.getClass().getName() + '@' +
        Integer.toHexString(instance.hashCode());
    assertFalse("toString␣method␣returned␣the␣default␣toString.␣Override␣
        Object.toString()!", instance.toString().equals(defaultToString));
  }

  @Test
  public void testEqualsAndHashCode() {
    EqualsVerifier.forClass(SomeClass.class).verify();
  }

  @Test
  public void testSomeStringSetter() {
    String test = "";
    instance.set(test);
    assertEquals(test, instance.getSomeString());
  }
  // If a class has more fields, more tests are added here.
}
```

Listing 4.8: *Testing `SomeClass` in the same way as listing 4.7 with all tests written down.*

	Before		After		Change	
Project name	CLOC	LOTC	CLOC	LOTC	CLOC	LOTC
JFreeChart axis	2081	9096	2081	9167	1.00	1.01
Maven model	777	2072	629	2066	0.810	0.997
TOPdesk application	26	424	26	669	1.00	1.58
TOPdesk application-utils	288	903	288	916	1.00	1.01
TOPdesk core	574	3899	574	4075	1.00	1.05

Table 4.3: *Result of the CLOC and LOTC metrics for test suites before and after applying JTA. CLOC is the number of the lines including at least one token of a code fragment of a code clone. LOTC stands for Lines Of Test Code. The change ratio is calculated by After / Before. For all values hold: the lower the better.*

the coverage increased. It is possible that the change in mutations is larger than the change in coverage. One possible reason is that the code that was already covered is again tested by JTA and the generated test detects more mutations than the original test.

The number of CLOC lowered only in the Maven model test suite. CCFinderX does not mark our annotation as duplication, so we never introduced new duplication by annotating the test suites. As a result, we can conclude that we removed duplicated methods only in Maven model.

The number of lines of test code increased in all suites, because we added annotations to all test suites. Maven model is an exception, since most tests we removed were in Maven model. The large increase of LOTC in the TOPdesk application test suite can be easily explained. Before annotating the suite, it contained 7 test classes. Now, after annotation, it has 36 test classes, of which 29 only include the `TestThisInstance` annotation.

4.3 Analysis

For completely untested classes, our annotation provides a simple and declarative way to generate test methods. We can hypothesize as to why these classes are untested: the classes might be considered too simple to break, or maybe there was a lack of time or interest of the developer for testing. In our previous listings we have shown several small bugs that might occur in simple code, so it is advisable to even test simple code. The time needed to test this code is drastically decreased with JTA, since all a developer has to do is call our annotation. Additionally, the generated tests prevent the developer of writing boiler-plate tests, as can be seen by comparing listings 4.7 and 4.8.

Our goal of boiler-plate reduction leans on the notion of testing more with less code. Since we added Lines Of Test Code with our annotation, we need to adjust our mutation and coverage results to get a better indication whether we follow this notion. The adjustment we applied is the normalization of the change in mutations killed and coverage on the change of LOTC. The amount of testing is assessed by the code coverage and the number of mutations killed. Table 4.4 shows the results after adjusting. In all cases we still have an improvement

	Before	Change			Change, adjusted	
Project name	Cover.	Mutation	Cover.	LOTC	Mutation	Cover.
JFreeChart axis	0.472	1.23	1.04	1.01	1.22	1.03
Maven model	0.0227	4.10	4.19	0.997	4.11	4.20
TOPdesk application	0.0739	2.68	2.33	1.58	1.70	1.47
TOPdesk app-utils	0.613	1.01	1.04	1.01	1.00	1.03
TOPdesk core	0.435	1.05	1.20	1.05	1.00	1.14

Table 4.4: *The change in mutation and coverage metrics, adjusted for the change in LOTC, Lines Of Test Code. For all values hold: the higher the better.*

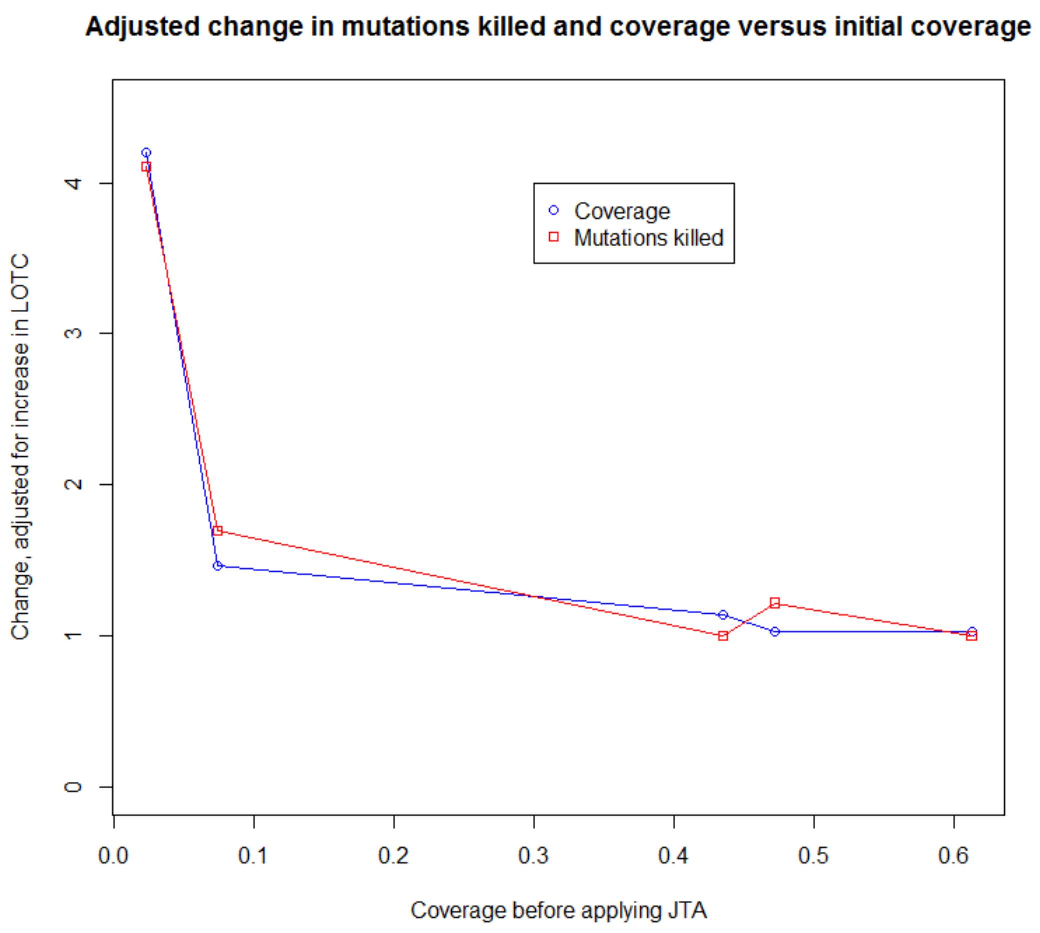

Figure 4.1: *Change in mutations killed and coverage, adjusted for change in lines of test code, plotted against the original coverage of each test suite*

in coverage, indicating that for every project, the application of JTA results in a test suite that covers more code per line of test code, compared to before applying JTA. For the mutations killed, this holds for three projects. The application-utils and core projects of TOPdesk have an adjusted change of 1.00, indicating that the generated tests detect the same amount of mutations per LOTC as the original test suite.

Additionally, there seems to be a relation between the coverage before application of JTA and the adjusted change in mutations killed and coverage caused by applying JTA. We illustrated this relation in figure 4.1. The underlying cause of this explanation is that JTA tests relatively simple properties, most of which are covered in the original test suite. In other words, the profit of having JTA generate tests is the highest on untested classes, while still having use on already tested classes.

4.3.1 Usability and applicability of the features

During the application of JTA we also evaluated the features we implemented. The most interesting findings are listed below, sorted per feature.

Assumption free tests

Our tool was able to create test instances in most of the cases. We identified two cases that were troublesome. The first one occured when testing `toString()` methods on objects that return one of their string fields, e.g. a name field, in the `toString()` method. Since the no-argument constructor of `String` is used by our heuristic for filling this field, the field is empty, because that constructor returns a `String` object describing the empty string. This results in a failing test, because we require the `toString()` method to not return an empty string. The test does signal a potential fault, but if this is the specified behavior, our test produces a false negative. We circumvented this by allowing the developer to actually instantiate the annotated instance, so it could be instantiated with its name not set to the empty string.

The second case that was a problem for creating default test instances, occurs when fields are objects that are mocked by our heuristic. If the constructor performs operations on this field (e.g. before setting a field file of type `File`, the setter method or constructor checks if it is an actual file), then these operations are performed on a mock, returning undefined results, which might throw exceptions (e.g. `FileNotFoundException` if the file can not be found). Letting the developer instantiate the instance under test, also provides a solution in this case.

`Equals(Object o)`, `hashCode()` and `toString()` tests

The generated tests for the `equals(Object o)`, `hashCode()` and `toString()` methods require these methods to be overridden in the class under test. In general, all `toString()` tests passed, with the exception mentioned above.

```java
public class SomeClass {
  List<String> someStringList = new ArrayList<String>();
  ...
  public void addSomeStringList(String str) {
    someStringList.add(str);
  }
  public String getSomeStringList(int index) {
    return someStringList.get(index);
  }
}
```

Listing 4.9: *Example showing the wrapping of a field.*

We could only sparsely make use of the `EqualsVerifier` to test the `equals(Object o)` and `hashCode()` methods. The problem `EqualsVerifier` encountered the most, was that the methods were not marked `final`. The only reason to not mark these methods as `final`, is if the class under test will be subclassed and its subclasses provide their own implementation. In this case `EqualsVerifier` expects the developer to provide some examples of subclasses to test with, but we can provide none. We suspect that in some of the cases, the `equals` and `hashCode()` methods could be marked as final.

If the Maven model classes would correctly implement `equals(Object o)` and `hashCode()`, JTA would have completely removed the duplication in the test suite as well as drastically decreased the lines of test code.

Getter and setter methods tests

We encountered relatively few problems in test suites with testing getters and setters. Some classes did not adhere to the Java Bean Specification and named the getter methods for their boolean field `someBoolean` `getSomeBoolean()` instead of `isSomeBoolean()`.

We did see the delegating construction from listing 4.9 in several classes. The field `someStringList` is only accessed using methods that delegate to the field. JTA can not test `someStringList`, because there are no dedicated getter or setter methods for the entire field.

An (unmodifiable) view or copy of collections

Our tests to verify the behavior of collections require the collection under test to have a setter and getter method available. This is not often the case; we could only apply it four times in the experiment. Most collection fields, and especially the immutable collections, are declared `final` and are only set once, through the constructor. The only way to test these cases is to use the constructor, which requires a mapping from constructor parameters to fields. As mentioned before, this mapping requires complex analysis.

4.3.2 Maintainability of Java Test Assistant and applicability in test-first development

Usage of the default settings of JTA, i.e. testing the getters and setters of all fields, has the advantage that the test code need not be updated if the class under test changes, e.g. a new field is added. Van Rompaey et al. state that "the quality of the tests should not only be measured in terms of coverage (i.e. increase the likelihood of detecting bugs) but also in terms of maintainability (i.e. reduce the cost of adapting the tests)." [22]. JTA in the default configuration is highly maintainable.

However, the default settings might not appeal to people who like test-first development, where tests are explicitly written before the code under test. JTA can also be used effectively in test-first development. For example, if a new field needs to be implemented, the developer specifies that field as getter and setter. This will generate failing tests until that field and its getter and setter method are implemented. The same holds also for the `equals(Object o)`, `hashCode()` and `toString()` methods.

4.4 Conclusion

We implemented several proposed solutions from the previous chapter in JTA and applied it on the JFreeChart axis, Maven model, TOPdesk application, application-utils and core projects. The code coverage increased with factors between 1.04 and 4.19. The number of mutations killed increased with factors between 1.01 to 4.10. All at the cost of a little more lines of test code. Relatively, the increase in coverage and mutations killed was greater than or equal to the increase in lines of test code. There is an inverse relation between the coverage before applying JTA and the increase in killed mutations and coverage, i.e. the lower the coverage before, the greater the increase in coverage and mutations killed will be. Removal of duplicated code was less successful; only one of the five tested projects had its number of duplications reduced.

The tests generated by JTA are maintainable and lower the amount of boiler-plate code. The tests for `equals(Object o)` and `hashCode()` are very strict, and they show that many implementations in the classes under test are not entirely correct. The generated tests for the behavior of collection fields could be applied less often then expected, due to requiring both a getter and a setter for the collection field.

Chapter 5

Related Work

Three of the problems addressed in this thesis, namely the detection of the class under test, test case prioritization and the automatic generation of tests, fall in a well explored domain in computer science. This chapter will briefly list some of the research in these fields.

5.1 Detection of the class under test

Detecting the class under test is part of the ongoing research on test traceability. Software Traceability is defined by the Institute of Electrical and Electronics Engineers as the degree to which a relationship can be established between two or more products of the development process [23]. The two products of interest in test traceability are the code under test and the test code.

Van Rompaey and Demeyer compared four different approaches to obtaining the traceability links between test code and code under test. The first approach relies on naming conventions, more specifically dropping the “Test” prefix from the test class name, e.g. `TestSomeClass` contains the tests for class `SomeClass`. The results of their experiment show that the naming conventions are the most accurate way of determining the class under test, but is not always applicable. The second approach, called “Last Call Before Assert” (LCBA), uses a static call graph to determine which classes are called in the statement before the last assertion. This approach does not suffer from a dependency on naming conventions, but is much less accurate. Their third approach looks for textual similarities between the test class and classes under test. The fourth approach assumes that the test class co-evolves with the class under test. These latter two approaches were less successful than the first two [24].

Qusef et al. took the LCBA approach and refined it, resulting in an approach that uses Data Flow Analysis (DFA). They show that DFA provides more accurate results than both LCBA and the naming conventions, but their case study also highlighted the limitations of the approach [25]. To overcome the limitations, they introduced another approach using dynamic program slicing to identify the set of classes that affected the last assertion. The approach is implemented in a tool called SCOTCH (Slicing and Coupling based Test to Code trace Hunter). They show that SCOTCH identifies traceability links between unit test classes and tested classes with a high accuracy and greater stability than existing techniques [26].

Test traceability was also researched by Hurdugaci and Zaidman, who developed TestNForce. TestNForce is a Visual Studio plugin that helps developers to identify the unit tests that need to be altered and executed after a code change. It uses test traceability to facilitate the co-evolution of production code and test code [27].

Our approach of detecting the class under test uses naming conventions and dynamic analysis and therefore leans on the work of Van Rompaey and Demeyer and Qusef et al.

5.2 Test case prioritization

One of the motivations for JUnitCategorizer is to improve error traceability by running unit tests before integration tests. This assures that the integration tests are performed with correctly operating units, so detected test failures occur at the integration level. This is a form of test case prioritization.

Most test case prioritiziation techniques are focused on running tests in a cost-effective manner [28]. Parrish et al. described a method to start with fine grained tests (unit tests) and then increase the granularity to move to coarser tests (integration tests). Their method has to be used when writing the test specifications and is not useful to apply at the moment tests fail. [29]. Gälli et al. focused on the same problem we address, by focusing on the error traceability through failed tests. They create a coverage hierarchy based on the sets of covered method signatures and use that hierarchy to present the developer with the most specific test case that failed [30].

5.3 Automated test case generation

Our approach focused on removing boiler-plate tests by generating tests and simple test data automatically. Most research in this field takes another approach and focuses on generating test data as input for any method. For example JMLUnit by Cheon et al., which generates unit tests based on annotations in the Java Modeling Language (JML) [20]. Cheon and Avila also created an implementation for AspectJ based on the Object Constraint Language instead of JML [31].

Cheon et al. incorporated genetic algorithms to generate object-oriented test data for the test methods generated by JMLUnit [32]. The usage of genetic algorithms was based on earlier work by Pargas et al. [33].

Another mention in test data generation is for JMLUnitNG, which was created in part to handle performance problems of JMLUnit. JMLUnitNG takes a similar approach to the test data generation as we do, but in the absence of a no argument constructor for a class under test, JMLUnitNG looks reflectively to a developer written test class of the class under test and uses the test objects defined there [34]. An effective way and a possible extension to our test data generation heuristic.

Some researchers capture the generation of test methods in formal specifications based on an algebraic formalism of Abstract Data Types by Guttag and Horning [35]. Examples are DAISTS [36] and its successor for object-oriented programming languages (specifically C++) Daistish [37]. The Junit Axioms, or JAX, are the Java specific version of ADT and come in two versions: full or partial automation. Both versions require the developer to specify some testing points, i.e. the test data. Full automation also requires the developer to capture the entire code under test in algebraic axioms, after which all test methods can be generated. In partial automation, the test methods are written by the developer, but JAX will apply all combinatorial combinations of the test points on the test methods. Partial automation is essentially parameterized unit testing [38].

Chapter 6

Conclusions and Future Work

This chapter gives an overview of the project's contributions. After this overview, we will reflect on the results and draw some conclusions. Finally, some ideas for future work will be discussed.

6.1 Contributions

One of the contributions of this thesis is a clear definition of the distinction between unit and integration tests. We implemented this distinction in our tool JUnitCategorizer and used our tool to provide an initial classification of test suites on the ratio of integration tests in the test suite. This classification showed that the majority of JUnit test suites consist of integration tests, with several test suites being purely integration, i.e. with more than 85% integration tests.

For the correct working of JUnitCategorizer, we implemented an algorithm to detect the class under test in a test class, which determines at least 90% of the classes under test correctly.

Using JUnitCategorizer, we found that boiler-plate code and common testing errors occur in both unit as integration tests. We detected the following problems: duplicate code in tests, tests inadequately covering code, corner cases not fully tested, incomplete tests of the `equals(Object o)` and `hashCode()` methods and untested behavior of collections.

Another contribution is our Java Test Assistant (JTA) tool, which was originally meant as extension to Project Lombok, but due to incompatibility issues became a stand alone tool. It has the ability to automatically generate several tests; replacing several boiler-plate tests by a single call to our tool. JTA generates assumption free tests, thus requiring little to no configuration when applied.

Application of JTA on five projects increased the code coverage by a factor of 1.04 to 4.19 and the number of killed mutations by a factor of 1.01 to 4.10. All at the cost of a little more lines of test code. Removal of duplicated code was less successful: only the Maven model project saw a decrease in cloned lines of test code by a factor of 0.81.

6.2 Conclusions

We summarize our conclusions by answering the research questions we posed in the introduction.

6.2.1 How can we distinguish unit and integration tests in JUnit?

In our definitions, we stated that when a mock object is replaced by the real implementation of that object, we speak of an integration test. Conversely, if a test only uses the object under test and mock objects, it is likely a unit test. Our tool JUnitCategorizer determines all objects called from a test method, determines the class under test and identifies mock objects, to make an educated guess if the test is unit or integration.

JUnitCategorizer uses on the fly instrumentation of Java class files to determine which classes are called in a test method. This list is filtered in three ways: using a blacklist, a suppressor and a whitelist. Any method that uses other classes than a single class under test, is marked as integration.

We determine the class under test by a heuristic that scores potential classes under test, where the highest score is the most likely class under test. We identified five cases that could be used in the scoring, but an exploratory data analysis showed that the best results were gained by only using two of the cases. The heuristic awards points if the name of the test class is either `SomeClassTest`, `SomeClassTests` or `TestSomeClass`, or if there are tests that only exercise `SomeClass`, i.e. they have exactly one called class, namely `SomeClass`. The points for a matching file name are multiplied by the number of test methods in the test class.

In an analysis to determine how well JUnitCategorizer distinguishes between unit and integration tests, it reached an accuracy score of 95.8%.

6.2.2 Which types of tests are best suitable to extend Project Lombok to?

An experiment to answer this question could not determine whether most testing problems are in unit or integration tests. We concluded that the problems occur in both, so we focused the rest of our research on both unit and integration tests.

6.2.3 Which code can be considered as boiler-plate in tests based on JUnit?

We found occurrences of inter and intra test class duplication. Inter test class duplication is duplication that occurs between different test classes, e.g. common tests for `hashCode()` and `equals(Object o)` applied on each class under test. We consider these tests as boiler-plate, so we created JTA to automatically generate these tests.

Intra class duplication is duplication that occurs inside a single test class, for example testing various inputs to a single method. It can be solved by using JUnit's parameterized tests and is therefore not handled in JTA.

6.2.4 Which common test errors could be alleviated by using the extension of Project Lombok (e.g. trivial tests a tester typically omits)?

The biggest problem with tests is the lack thereof; we found several cases of low code coverage. For example, getter and setter methods are sometimes considered too simple to break, but we showed that it is wise to still test them, using some simple examples that show what can go wrong. We also found that some behavior of the class under test is not tested correctly. The `hashCode()` and `equals(Object o)` method are hard to correctly implement and test. The `toString()` method often goes untested. The behavior of fields that contain a collection is often unspecified; it can be a view, an unmodifiable view, a copy, or an unmodifiable copy. JTA can easily generate tests for all these aforementioned cases.

6.2.5 What does the extension of Project Lombok add to existing tooling?

The most important feature of JTA is that it generates assumption free tests. We developed a heuristic for creating simple test values that are suitable in most cases. The downside of these assumption free tests is that the generated tests are not suitable for testing corner cases.

6.2.6 Does the extension of Project Lombok increase the quality of unit testing, compared to manually writing unit test code?

We have shown that the application of JTA on several test suites increased the code coverage and number of mutations killed while hardly adding new test code. Additionally, several tests that are generated by JTA are stricter than the tests currently present in the test suites, since they fail on the current implementation.

6.3 Discussion/Reflection

Overall, we are satisfied with the results in this master's thesis and how we got to those results. But, even in a large project as the master's thesis is, there are time constraints. These constraints limited the implementations and analyses of JUnitCategorizer and JTA. More time would enable the experiments to be applied on more projects, leading to more accurate results. One pertinent example of a time constraint is that the validation of the hypothesis in section 2.3.4 is performed on a very small test set.

JUnitCategorizer and JTA both approach known (sub)problems in research in their own ways. There are state of the art tools that solve similar (sub)problems, where they might have been compared against. These are different experiments from the ones we conducted.

Our research should be seen as exploratory, e.g. whether our approaches work at all, opposed to comparing our tools to state of the art tools.

6.4 Future work

Our suggestions for future work are based around the tools we implemented: JUnitCategorizer and Java Test Assistant (JTA).

6.4.1 JUnitCategorizer

In section 2.2.5 we already mentioned future work for JUnitCategorizer, namely the detection of dependence of system defaults for locales. To determine whether a default locale is compared to a specific locale or whether a default locale has been set prior, JUnitCategorizer needs to apply a kind of taint analysis. This analysis will improve the correct distinction of unit and integration tests for classes depending on locales.

Since version 4.8 JUnit supports test groups, i.e. a test method can be annotated with zero or more groups and then groups can be run in isolation. Applying separate JUnit groups for unit and integration tests can result in better error traceability, since unit tests are run before the integration tests. JUnitCategorizer could be extended to automatically annotate test methods with the correct JUnit group.

Using the analysis JUnitCategorizer performs, we can even further improve error traceability by introducing *layered testing*. Many applications use a layered architecture to separate major technical concerns. Every layer has its own responsibilities and it is preferable to test every layer in isolation [39]. JUnitCategorizer currently distinguishes between two layers: the unit layer and the integration layer. An extension to JUnitCategorizer is to use n layers, where every layer i only uses objects from layer j, with $j < i$. Conceptually, tests on layer i are the integration tests of units from layer $i-1$. Using more layers results in an even better error traceability than with two layers. Suppose we look at testing in n rounds, where every round i runs all tests for layer i and if tests in a certain round fail, we skip the remainder of the tests. Problems with units of layer i are detected in round i and don't show in higher layers, since those tests will not be run. Integration problems between units of layer i are detected in layer $i+1$, so an error in units of a lower layer will not propagate into higher layers. This is a variation of the research on ordering broken unit tests performed by Gälli et al. [30].

6.4.2 Java Test Assistant

There are several possibilities to extend and improve JTA. Besides the issues mentioned in section 4.3, a possible extension to JTA can be generation of tests for the `clone()` method. Just like the `equals(Object o)` and `hashCode()` methods, it is very difficult to get the implementation correct and it should be tested extensively. There are probably more extensions possible to further increase code coverage in an assumption free manner.

We used metrics to show that, in theory, JTA improves developer testing. It would also be interesting to perform a user study on JTA, to assess its usability in practice.

Both tools are now used in relatively small experiments. A larger experiment will increase confidence in the results and provides potentially more insights in the distribution between unit and integration tests and the applicability of JTA in test suites.

Additionally, both tools solve (sub)problems that are also solved in other tools. Certain properties might be compared to other state of the art tools, e.g. JUnitCategorizer's ability to detecting the correct class under test could be compared to the class under test detecting part of Qusef et al.'s Data Flow Analysis.

Bibliography

[1] R. Binder. *Testing object-oriented systems: models, patterns, and tools*. Addison-Wesley, 1999.

[2] M. Pezzè and M. Young. *Software Testing and Analysis: Process, Principles, and Techniques*. Wiley, 2008.

[3] L. Koskela. *Test driven: practical tdd and acceptance tdd for java developers*. Manning Publications Co., 2007.

[4] K. Beck. *Test-driven development: by example*. Addison-Wesley Professional, 2003.

[5] M.C. Feathers. *Working effectively with legacy code*. Prentice Hall PTR, 2005.

[6] T. Mackinnon, S. Freeman, and P. Craig. Endo-testing: unit testing with mock objects. *Extreme programming examined*, pages 287–301, 2001.

[7] M. Bruntink, A. Van Deursen, R. Van Engelen, and T. Tourwe. On the use of clone detection for identifying crosscutting concern code. *Software Engineering, IEEE Transactions on*, 31(10):804–818, 2005.

[8] F.M. Dekking, C. Kraaikamp, H.P. Lopuhaä, and L.E. Meester. *A modern introduction to probability and statistics. Understanding why and how*. Springer Texts in Statistics, 2005.

[9] C.D. Manning, P. Raghavan, and H. Schutze. *Introduction to information retrieval*, volume 1. Cambridge University Press Cambridge, 2008.

[10] T. Kamiya, S. Kusumoto, and K. Inoue. Ccfinder: a multilinguistic token-based code clone detection system for large scale source code. *Software Engineering, IEEE Transactions on*, 28(7):654–670, 2002.

[11] Y. Jia and M. Harman. An analysis and survey of the development of mutation testing. *Software Engineering, IEEE Transactions on*, (99), 2010.

[12] I. Moore. Jester-a junit test tester. *Proc. of 2nd XP*, pages 84–87, 2001.

[13] Y.S. Ma, J. Offutt, and Y.R. Kwon. Mujava: an automated class mutation system. *Software Testing, Verification and Reliability*, 15(2):97–133, 2005.

[14] S.A. Irvine, T. Pavlinic, L. Trigg, J.G. Cleary, S. Inglis, and M. Utting. Jumble java byte code to measure the effectiveness of unit tests. In *Testing: Academic and Industrial Conference Practice and Research Techniques-MUTATION, 2007. TAICPART-MUTATION 2007*, pages 169–175. IEEE, 2007.

[15] D. Schuler and A. Zeller. Javalanche: efficient mutation testing for java. In *Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 297–298. ACM, 2009.

[16] Q. Yang, J.J. Li, and D. Weiss. A survey of coverage based testing tools. In *Proceedings of the 2006 international workshop on Automation of software test*, pages 99–103. ACM, 2006.

[17] G. Kudrjavets, N. Nagappan, and T. Ball. Assessing the relationship between software assertions and faults: An empirical investigation. In *Proceedings of the 17th International Symposium on Software Reliability Engineering*, pages 204–212. IEEE Computer Society, 2006.

[18] M. Potel. Mvp: Model-view-presenter the taligent programming model for c++ and java. *Taligent Inc*, 1996.

[19] N. Tillmann and W. Schulte. Parameterized unit tests. In *ACM SIGSOFT Software Engineering Notes*, volume 30, pages 253–262. ACM, 2005.

[20] Y. Cheon and G.T. Leavens. A simple and practical approach to unit testing: The jml and junit way. In *ECOOP 2002-object-oriented programming: 16th European Conference, Malága, Spain, June 10-14, 2002: proceedings*, volume 2374, page 231. Springer-Verlag New York Inc, 2002.

[21] T. Xie, N. Tillmann, J. De Halleux, and W. Schulte. Future of developer testing: Building quality in code. In *Proceedings of the FSE/SDP workshop on Future of software engineering research*, pages 415–420. ACM, 2010.

[22] B. Van Rompaey, B. Du Bois, and S. Demeyer. Characterizing the relative significance of a test smell. In *Software Maintenance, 2006. ICSM'06. 22nd IEEE International Conference on*, pages 391–400. IEEE, 2006.

[23] J. Radatz, A. Geraci, and F. Katki. Ieee standard glossary of software engineering terminology. *IEEE Standards Board, New York, Standard IEEE std*, pages 610–12, 1990.

[24] B. Van Rompaey and S. Demeyer. Establishing traceability links between unit test cases and units under test. In *Software Maintenance and Reengineering, 2009. CSMR'09. 13th European Conference on*, pages 209–218. IEEE, 2009.

[25] A. Qusef, R. Oliveto, and A. De Lucia. Recovering traceability links between unit tests and classes under test: An improved method. In *Software Maintenance (ICSM), 2010 IEEE International Conference on*, pages 1–10. IEEE, 2010.

[26] A. Qusef, G. Bavota, R. Oliveto, A. De Lucia, and D. Binkley. Scotch: Test-to-code traceability using slicing and conceptual coupling. In *Software Maintenance (ICSM), 2011 27th IEEE International Conference on*, pages 63–72. IEEE, 2011.

[27] V. Hurdugaci and A.E. Zaidman. Aiding developers to maintain developer tests. In *Software Maintenance and Reengineering, 2012. Proceedings. 16th European Conference on*, pages 11–20. IEEE, 2012.

[28] S. Elbaum, A.G. Malishevsky, and G. Rothermel. Test case prioritization: A family of empirical studies. *Software Engineering, IEEE Transactions on*, 28(2):159–182, 2002.

[29] A. Parrish, J. Jones, and B. Dixon. Extreme unit testing: Ordering test cases to maximize early testing. *Extreme Programming Perspectives*, pages 123–140, 2001.

[30] M. Galli, M. Lanza, O. Nierstrasz, and R. Wuyts. Ordering broken unit tests for focused debugging. In *Software Maintenance, 2004. Proceedings. 20th IEEE International Conference on*, pages 114–123. IEEE, 2004.

[31] Y. Cheon and C. Avila. Automating java program testing using ocl and aspectj. In *Information Technology: New Generations (ITNG), 2010 Seventh International Conference on*, pages 1020–1025. IEEE, 2010.

[32] Y. Cheon, M.Y. Kim, and A. Perumandla. A complete automation of unit testing for java programs. In *Proceedings of the 2005 International Conference on Software Engineering Research and Practice*, pages 290–295, 2005.

[33] R.P. Pargas, M.J. Harrold, and R.R. Peck. Test-data generation using genetic algorithms. *Software Testing Verification and Reliability*, 9(4):263–282, 1999.

[34] D. Zimmerman and R. Nagmoti. Jmlunit: the next generation. *Formal Verification of Object-Oriented Software*, pages 183–197, 2011.

[35] J.V. Guttag and J.J. Horning. The algebraic specification of abstract data types. *Acta informatica*, 10(1):27–52, 1978.

[36] J. Gannon, P. McMullin, and R. Hamlet. Data abstraction, implementation, specification, and testing. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 3(3):211–223, 1981.

[37] M. Hughes and D. Stotts. Daistish: Systematic algebraic testing for oo programs in the presence of side-effects. In *ACM SIGSOFT Software Engineering Notes*, volume 21, pages 53–61. ACM, 1996.

[38] D. Stotts, M. Lindsey, and A. Antley. An informal formal method for systematic junit test case generation. *Extreme Programming and Agile MethodsXP/Agile Universe 2002*, pages 365–385, 2002.

[39] G. Meszaros. *xUnit test patterns: Refactoring test code*. Addison-Wesley Professional, 2007.