Detecting the Methods under Test in Java

Philippe Marschall
Software Composition Group
University of Bern
www.iam.unibe.ch/∼scg

April 20, 2005

−Id: testscaper.tex,v 1.13 2005/04/10 13:07:36 marschal Exp −

1 Introduction

Unit tests are a well accepted part of software engineering. *JUnit* is the de facto standard for unit testing in *Java*. It collects, organizes and runs tests. Each test focuses on one or several methods. These are called the *methods under test*. They can be used for a variety of tasks including test navigation, test coverage and test analysis in general.

There are no rules for determining on which methods a test focuses. Sometimes it is obvious, but there are cases where we cannot say on which methods a test focuses. Among others we observed two test patterns that look similar but are the inverse of each other. The first consists of an initial setup method and then focuses on one or several methods. The second one invokes the focused method first and then uses accessors to test the side effects.

As a result there are no established and foolproof ways to detect the methods under test automatically. In the following we discuss several different, automated ways of detecting the methods under test. Because there are no rules to determine the methods under test, automatically detecting them can never be fully accurate. But we search for different approaches and try to find out how effective they are.

First we present several different ways to annotate a test with its methods under test and choose one of them to annotate the tests of some case studies. We also build a tool that allows us to query these methods and their annotations.

Afterward we describe ways to automatically detect the methods under test. The first one called *NameAnalyzer* looks at the names of tests and test cases and uses naming conventions to determine the methods under test. We also parse the source code of a test and try to extract all methods the test directly invokes. Because this results in a lot of false positives we build a heuristic extension to reduce this noise. We run each of these approaches for analysis on some case studies and validate their output against the annotations described in the first section.

Finally we discuss these results, judge the approaches by how effective they are in detecting the methods under test, and conclude.

2 Problem

JUnit is by far the most widely used unit testing framework for *Java*. It stores the results of the tests so they can later be analyzed. But these results provide no information at all about what was tested. This is not a fault of *JUnit*. It rather comes from the fact that it was designed to be simple.

But this information would be valuable for various tasks like determining test coverage to spot classes and methods that are untested or for navigating from methods to their tests to see examples of how the tested code is intended to be used. Such examples would help us to better understand the code by seeing it in action. The other direction is also interesting because it tells the test engineer what a test is actually testing. We found it for example extremely difficult for some *Ant* tests to say what code they test.

About the scope of unit tests Binder [Bin99] writes:

> The scope of a unit test typically comprises relatively small executables. In object oriented languages, an object of a class is the smallest executable unit, but test messages must be sent to a method, so we can speak of method scope testing.
>
> Defining the scope of a test is the same as defining the collection of software components that need to be verified. The code being tested is called the implementation under test (IUT), method under test (MUT), object under test (OUT), class under test (CUT), and system under test (SUT). Traditionally a scope is designated as unit, integration, or system.

To this we add the the *package under test* (PUT) which we define as the package in which the CUT is located.

Eclipse already does something similar. It searches via *Search – Referring Tests* for the tests of a method. Although this first looks like the opposite of what we want it is not that far away because the questions "What methods does a test focus on?" and "Where is a method testes?" are interrelated.

There are however many problems with this *Eclipse* functionality. First it does not pay attention to the PUT. So it can say a method is tested somewhere where it is not in the PUT. Second it looks at tests that are protected, but *JUnit* only runs tests that are public. In some tests all the assertions are made in a helper method that is invoked by several tests. *Eclipse* does not look at these methods and therefore misses the MUTs. Additionally the search function behaves strangely for methods that are defined in an interface. For example according to *Eclipse* org.apache.maven.util.InsertionOrderedSet.InsertionOrderedSetIterator#hasNext(), that is defined in the interface java.util.Iterator, is tested in org.apache.tools.ant.taskdefs.optional.ssh.ScpTest#testMultiUploadAndDownload(). But this test only uses the #hasNext() method of java.util.ArrayList s iterator, which is a different class. The functionality is not directly available through the JDT. We think it uses the Eclipse AST Parser to find out all methods invoked by a test and doing that for all tests. Because it is a tool inside *Eclipse* it is likely to use the parser's functionality to provide type information. But this means it cannot be used outside of *Eclipse*.

Our vision for *Eclipse* is a greatly improved *Search – Referring Tests* functionality that always runs in the background. Then a marker like the already existing ones for warnings and exceptions is added to the editor on the line of the method head that expands to the name of all the tests of this method. Clicking on the name results in a jump to the test. For tests a similar marker is added on the line of the method head. A click on it displays a list

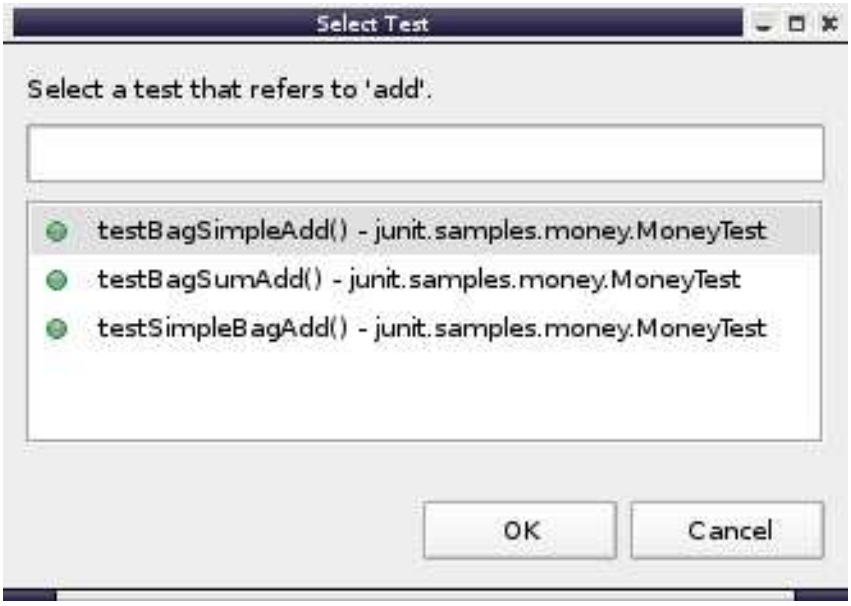

Figure 1: We let eclipse search for tests for Money#add(IMoney)

a links to the MUTs. But this is outside the scope of this project. We just focus on ways to detect the MUTs independent of *Eclipse*.

2.1 The Importance of the Package Under Test

The importance of determining the package under test is not obvious at first glance. One might think it is enough to just record all the method invocations in a test. But that might lead to false positives. Consider the code shown in Figure 2. The MUT in this case is m1() of instanceToTest's class and not java.lang.String#startsWith(String). We expect java.lang.String#startsWith(String) to work correctly and use it to test the result of m1. We can be sure about that because java.lang.String is not in the package under test. The situation would of course be different if the package under test were java.lang.

```
public void testM1(){
    java.lang.String returnValue = this.instanceToTest.m1();
    assertTrue(returnValue.startsWith("expected start"));
}
```

Figure 2: Example of the importance of the package under test

3 Possible Approaches

In the *Construction* section we will introduce ways to make the *fully qualified method under test* explicit. In the following *Analysis* section we will describe ways to automatically detect the *fully qualified method under test* in *Java*.

3.1 Construction

In this section we describe several different methods of making explicit the MUTs, the CUTs and PUT. In all examples, the MUTs are the method1 of the class Class1 and the method2 of the class Class2. So Class1 and Class2 are the CUTs. They are both from the package package.under.test, which is the PUT.

3.1.1 Comment/Javadoc

Javadoc comment is a specially structured comment in *Java*. It is intended to be processed with the javadoc tool to generate HTML documentation. Therefore the tool offers an API that lets us define our own so called tags. But there are other tools like XDoclet [1] that allow the processing of javadoc and do something other than produce documentation, for example generate a deployment descriptor.

```
/**
 * ...
 * @muts package.under.test.Class1#method1 package.under.test.Class2#method2
 */
public void testXYZ() {
    ...
}
```

Figure 3: A custom javadoc tag to annotate the methods under test

Figure 3 shows how a custom tag — @muts in this case — can be used in Javadoc comment to annotate the *fully qualified method under test*.

3.1.2 Java metadata annotations

Annotations are a new feature added to *Java* with version 1.5. They add a metadata facility to the core *Java* language. We can annotate types, methods, constructors, fields, parameters, local variables, packages and finally annotation types. The data types of annotations can be any primitive type, Class, String, enum types (another new feature), an annotation type or arrays of these types. So we would probably use a special formatted String that follows this pattern: "package.under.test.Class1#method1(ArgumentTypes)". This format is also used by Javadoc to identify methods.

Annotations can be made available at runtime so that they can be accessed via the *Java* Reflection API.

Metadata annotations were actually introduced to replace the abuse of Javadoc for metadata. For example the @deprecated Javadoc tag was replaced by the @Deprecated annotation. Sun ships a tool with *Java* 1.5 called *apt* — not to be confused with the Debian Advanced Packaging Tool — that processes annotations at compile time.

Java annotations are much like attributes in C# or VB.NET with the main difference that attributes are used widely in .NET for various tasks ranging from transactions to security and threading.

Figure 4 shows how a custom annotation — @Testscape in this case — can be used to annotate a test with the *fully qualified methods under test*. The annotation has a method

[1] http://xdoclet.sourceforge.net/xdoclet/index.html

```
@Testscape(muts = {
        "package.under.test.Class1#method1(ArgumentTypeOfMethod1)",
        "package.under.test.Class2#method2(ArgumentTypeOfMethod2)"})
public void testXYZ() {
...
}
```

Figure 4: An annotated test

muts() that returns an array of strings where each element is a fully qualified method under test.

3.1.3 External XML-File

The information about the *fully qualified methods under test* can also be stored in an other file than the source file. Various formats are possible but XML is a well accepted industry standard and various high level tools like DOM [2], JDOM [3], DOM4J [4] or XMLBeans [5] are available for processing XML Documents with *Java*.

```
<test class="SampleTest" selector="testXYZ">
    <mut>
        <package>package.under.test</package>
        <class>Class1</class>
        <method>method1</method>
    </mut>
    <mut>
        <package>package.under.test</package>
        <class>Class2</class>
        <method>method2</method>
    </mut>
</test>
```

Figure 5: An XML-File that describes the methods under test

Figure 5 shows the structure of a sample XML-File for describing the *fully qualified methods under test*.

3.1.4 Simple Return Value

In *Java* – unlike *Smalltalk* – methods can have the return type void, which means they do not return anything. *JUnit* tests in general are of this return type. But we would like our test to return a value. This value can be not only used to determine the *fully qualified methods under test*, but also for the composition of unit tests. So we will have to change the return type of the tests from void to some reference type.

In this approach the test returns a simple value that contains the tested instance, the result that is asserted and the parameters that are passed to the MUT of the tested instance.

[2] http://www.w3.org/DOM/
[3] http://www.jdom.org/
[4] http://www.dom4j.org/
[5] http://xmlbeans.apache.org/

This value does not directly describe what is tested but represents the result of the test. The information about what was tested would be gained through parsing of byte or source code and finding out how the result was constructed.

```
public ExtendedTestResult testXYZ() {
    Class1 aInstanceOfClass1 = new Class1();
    ParameterClass aParameter = new ParameterClass();
    Object aResult = aInstanceOfTestedClass.method1(aParameter);
    assertNotNull(aResult);
    // normal test code up until here
    return new ExtendedTestResult(aInstanceOfClass1, aParameter, aResult);
}
```

Figure 6: A test that returns a simple return value

Figure 7 shows how an example usage of a *Simple Return Value*. The return value is called ExtendedTestResult and not TestResult in order to avoid confusion with the existing TestResult in the *JUnit* framework. For the sake of clarity of the test code, method2 of Class2 was left out.

3.1.5 Complex Return Value

Like in the approach described above the test is changed so that it returns a value. But in this case the value is more complex (see Figure 7). It contains information about what the *fully qualified method under test* is, so it can be directly retrieved from the return value. The return value is preferably constructed in the test itself so it is as close to the test as possible. This way if the test changes, no other method has to be changed.

```
public TestDescripor testXYZ() {
    ... //test code here
    TestDescriptor descriptor = new TestDescriptor();
    descriptor.addMethodUnderTest("package.under.test.Class1", "method1");
    descriptor.addMethodUnderTest("package.under.test.Class2", "method2");
    return descriptor;
}
```

Figure 7: Construction of a complex return value in a test after the test code

3.2 Analysis

In this section we describe several different ways of automatically detecting the *fully qualified method under test*.

3.2.1 NameAnalyzer / Heuristics

We first need a simple, lightweight, easy to extend solution to identify the methods under test. We call it the *NameAnalyzer*. The idea is using a naming convention in combination with the Java Reflection API to get the information about what the *fully qualified methods under test* are.

JUnit tests that use a common fixture are grouped into a subclass TestCase. Each test in such a class is a public method, whose name starts with "test" and takes no arguments. This

way *JUnit* can find out what tests exist by using the *Java* Reflection API on such a class and invoke them dynamically.

An example how the naming convention looks like is shown in Figure 8.

```
package com.acme.test.billing;

public class BillTest extends TestCase {
    public void testAddItem() {
        ...
    }
}
```

Figure 8: An example of the naming convention

The PUT is com.acme.billing, the CUT is Bill, and in the testAddItem test the MUT is addItem.

Determine the PUT. To determine the PUT, we look at the package tree the test is located in. If it is in a sub package node with the name test, we construct a new path without this node. Else we assume it is the current package. So for example org.jhotdraw.test.samples.net would become org.jhotdraw.samples.net whereas org.apache.ant.taskdefs.cond would remain the same.

Ant and *Maven* store their tests in the same package as the PUT but at a different location in their CVS-Repository. *JHotDraw* makes a test subpackage and there subpackages for each package, for example in org.jhotdraw.test.samples.net are the tests for org.jhotdraw.samples.net.

Determine the CUT. A *JUnit* TestCase only tests one class. If the name of the TestCase ends or starts with "Test" we just remove the "Test" from the name and we have the name of the CUT. So all tests in the TestCase BillTest test the class Bill.

To get the fully qualified name we assume this class is in the PUT. In our example the fully qualified CUT would be com.acme.billing.Bill.

Determine the MUT. Each test is named testMethodUnderTest. So we chop off the leading "test" and convert the first character to lowercase (*Java* naming convention says method names start with a lowercase letter). So in this case the MUT would be methodUnderTest. The MUT belongs to the CUT in the PUT.

Later we make a little tweak. We noticed that in *JHotDraw* and *Maven* a lot of tests were named testSetGetSomething or testSetIsSomething. So if a test method starts with testSetGet or testSetIs we assume it tests a setter and a getter. In this case setSomething, getSomething and isSomething. Later it turned out that all these tests were generated by *JUnitDoclet*[6].

Determine if a Test is Empty. To find out if a test is empty, we analyze the byte code with *BCEL*[7]. *BCEL* is the *Byte Code Engineering Library* from the *Apache Jakarta Project*[8]. It allows users to analyze, modify and create binary Java class files. We chose *BCEL* because in this particular case byte code is easier to deal with than source code because we only have to look for the code of the return instruction and count the number of bytecodes. And we already use Java reflection that bases on compiled java classes.

A method is empty if it only has a single return instruction. This is a byte with the value *0xB1*[9]. Even if there is no return statement in the source code, the compiler automatically

[6] http://www.junitdoclet.org/

[7] http://jakarta.apache.org/bcel/

[8] http://jakarta.apache.org/

[9] see "The Java(TM) Virtual Machine Specification", Second Edition, Chapter 9

generates this instruction.

3.2.2 Parsing, All Methods Invoked

Description
The NameAnalyzer alone did not prove to be sufficient in all circumstances (see Section 5.2.2) so another approach is needed. We decide to parse the source code because that allows us to look into a test and see what the programmer of the test did. This is our implementation of what we think eclipse is doing. It tries to detect all methods directly invoked by a test by looking at the source code.
Implementation
We parse the source code to construct an abstract syntax tree (AST). Then we analyze this tree. We chose to use the AST parser that is part of the JDT from *Eclipse* because compared to the AST parsers generated by parser generators it does not feel generated and it is possible to use it outside of *eclipse*.

One problem with all the parsers we looked at is that they do not provide any type information. But the CUT is part of the *fully qualified method under test* so we need to know to what class an invoked method belongs to. Additionally the type of the arguments would be valuable to distinguish overloaded methods. The *Eclipse* AST parser could do that if used in an eclipse plug-in. But we chose to make our parser not an *Eclipse* plug-in, because this way it can be run outside of *Eclipse*. This means we have to find out to what class a method belongs ourselves. We do this by keeping track of all instance- and local variables and their types.

Simplified and in pseudocode our algorithm looks like this:

```
for each field in the current subclass of TestCase
    remember name and static type

analyze setUp() method
    remember dynamic types for fields if possible

for each expression in the current test method
    if is method invocation
        if is not assertion
            find out class of method and add method to invoked methods
        recurse for all method arguments
    if is local variable declaration
        remember name and static type
        if has initializer
            recurse for initializer
            remember dynamic type if possible
```

As you can see we do not handle every possible expression, but instead choose to focus on what we consider the important parts in order to reduce implementation expense at the cost of exactness.

We completely ignore inner classes because it would complicate the parser more, we had limited time and few tests use them. One exception of this rule is org.apache.maven.util.DVSL-PathToolTest. Also we ignore array access.

3.2.3 Parsing, Asserted Methods

Description

An example for situations where just collecting all invoked methods does not work is showed in Figure 9. In this case *Parsing, All Methods Invoked* would say that create is a MUT but it is only used for setup. Note that this method is in the PUT, which is junit.samples.money.

```
public void testMoneyBagHash() {
        IMoney equal= MoneyBag.create(new Money(12, "CHF"), new Money(7, "USD"));
        assertEquals(this.fMB1.hashCode(), equal.hashCode());
}
```

Figure 9:

We investigated why just listing all methods invoked produces so much noise for *MoneyTest* sand derived a rule that constructs a subset of all methods directly invoked by a test that is closer to the MUTs. We call this set *asserted methods*. We call a method that creates an argument that is passed to an assertion an *asserted method*. We added this heuristic to the previously described parser.

An *asserted method* is not inevitably a MUT and vice versa. But we hope that they are close to the MUTs or at least closer than all the methods invoked by a test. The advantage of *asserted methods* over MUTs is that there is a rule for determining them.

In Figure 9 #hashCode would be an asserted method because the second argument passed to the assertion is created by invoking this method on the local variable equal. If the second argument was just equal, create would be an *asserted method* because it created the value of equal.

Implementation

We add a second pool of invoked methods for the *asserted methods*. If an argument of an assertion is an identifier then the method that generates the identifier's current value is added to this pool. If an argument of an assertion is a method invocation (like equal.hashCode()), this method is added to the *asserted methods* pool.

There is one flaw in our implementation that is not fixed because of time constraints. In a situation as illustrated in Figure 10 someMethod would be added to the *asserted methods* although its return value is not checked and merely used as a message if the assertion fails.

```
assertTrue(var1.someMethod(), var2.anotherMethod());
```

Figure 10:

4 Our Approach

We chose *Ant*[10], *Maven*[11], *JHotDraw 6.0 beta1*[12] and *MoneyTest* that comes with *JUnit* as case studies. *Ant* and *MoneyTest* are standard case studies for unit tests. We chose *JHotDraw* because it is a rather big application with lots of tests. Later we added *Maven* because *JHotDraw* did not turn out to be a good case study, it took us a while to get *Ant* to compile, *MoneyTest* is not representative for a real world application, and, last but not least,

[10] http://ant.apache.org/
[11] http://maven.apache.org/
[12] http://www.jhotdraw.org/

Maven is a well known tool that builds itself and generates a lot of code quality reports. We did not include all tests of the optional *Ant* tasks because we could not find the necessary JARs to compile them.

Construction We chose to use metadata annotations because they are easy to handle, are together with the code and we do not have any requirements about the *Java* version. We annotated 50 random not empty *Maven* tests and all 22 tests of MoneyTest. We did not count constructors as MUTs. We did not annotate any *JHotDraw* or *Ant* tests because of the problems described in Section 5.1. We did not use fully qualified class names because our parser can not handle them.

```
@Target({METHOD}) @Retention(RUNTIME) public @interface Testscape {
    public String[] muts();
}
```

Figure 11: The annotation we used

```
    @Testscape(muts= {"Money#subtract"})
    public void testSimpleSubtract() {
        // [14 CHF] - [12 CHF] == [2 CHF]
        Money expected= new Money(2, "CHF");
        assertEquals(expected, this.f14CHF.subtract(this.f12CHF));
    }
```

Figure 12: An annotated test.

Analysis We first implemented the *NameAnalyzer* because it was a simple and lightweight way to start. Later we added the first parser because the *NameAnalyzer* did not work well enough in all situations and we needed a way to find out what methods a test invokes. Finally we added a heuristic extension to reduce noise.

5 Results

Here we present the findings we made in our experiments. Our goal was to find out how our different approaches for analysis perform. We validated them with the approach for construction described above.

First we discuss the quality of our case studies and the problems we encountered. Then we will present the results for each approach for analysis.

5.1 Quality of the Case Studies

During our experiments we found out that the quality of the tests varies from case study to case study and each had its own set of problems.

The problem with *JHotDraw* and to a lesser extent with *Maven* is that they contain a lot of generated tests. Many of them are empty or look the same. So if we can correctly classify two or three examples we can also automatically classify a big part of the rest, because they are essentially the same. Only the names of local variables and invoked methods differ.

The problem with *Ant* is that for the most part we cannot tell what the MUTs are. We even cannot tell what the CUT is. Many tests contain only one or two generic method invocations. If there are two, the first seems to build up a scenario depending on the arguments passed while the seconds tests it. These methods are in general provided via inheritance. There also seems to be usage of dynamic invocation or external files.

Example

The following methods should give you an impression. This is the method test4 from TarTest, Let us find out on what methods it focuses.

```
public void test4() {
    expectBuildException("test4", "tar cannot include itself");
}
```

Judging by the name, this looks like a pessimistic method example. Let us look at expectBuildException.

```
protected void expectBuildException(String target, String cause) {
    expectSpecificBuildException(target, cause, null);
}
```

We found it in the superclass of TarTest, BuildFileTest. Now let us search for expectSpecificBuildException

```
protected void expectSpecificBuildException(String target, String cause, String msg) {
    try {
        executeTarget(target);
    } catch (org.apache.tools.ant.BuildException ex) {
        buildException = ex;
        if ((null != msg) && (!ex.getMessage().equals(msg))) {
            fail("Should throw BuildException because '" + cause
                    + "' with message '" + msg
                    + "' (actual message '" + ex.getMessage() + "' instead)");
        }
        return;
    }
    fail("Should throw BuildException because: " + cause);
}
```

Now we know for sure it is pessimistic but still not what is tested. Let us search executeTarget

```
protected void executeTarget(String targetName) {
    PrintStream sysOut = System.out;
    PrintStream sysErr = System.err;
    try {
        sysOut.flush();
        sysErr.flush();
        outBuffer = new StringBuffer();
        PrintStream out = new PrintStream(new AntOutputStream(outBuffer));
        System.setOut(out);
        errBuffer = new StringBuffer();
        PrintStream err = new PrintStream(new AntOutputStream(errBuffer));
        System.setErr(err);
        logBuffer = new StringBuffer();
```

```
            fullLogBuffer = new StringBuffer();
            buildException = null;
            project.executeTarget(targetName);
        } finally {
            System.setOut(sysOut);
            System.setErr(sysErr);
        }

    }
```

This is where it ends. So on which methods does test4 focus now? Project#executeTarget? If yes then almost all *Ant* tests would test just this single method.

When analyzing in which cases the detection of the MUT failed we saw that in many cases it was a testVault() method. We looked at some of these methods and they all look the same. See Figure 13.

```
// JUnitDoclet begin javadoc_method testVault
/**
 * JUnitDoclet moves marker to this method, if there is not match
 * for them in the regenerated code and if the marker is not empty.
 * This way, no test gets lost when regenerating after renaming.
 * <b>Method testVault is supposed to be empty.</b>
 */
// JUnitDoclet end javadoc_method testVault
public void testVault() throws Exception {
        // JUnitDoclet begin method testcase.testVault
        // JUnitDoclet end method testcase.testVault
}
```

Figure 13: An sample testVault

These tests are generated by *JUnitDoclet* [13], do not test anything and are correctly recognized as empty tests. *JUnitDoclet* is a tool that generates skeletons of TestCases based on the source code of an application. A quote from the website says:

> Please don't get the (deadly wrong) impression, that the number of tests says anything about the quality of testing. If most of the tests are empty, you didn't get the point! It remains your job to write the tests, but JUnitDoclet is helping you as good[sic] as it can.

It does not look like this was done for *Maven* or *JHotDraw*. *Maven* has 23 of these tests, *JHotDraw* 170. *Ant* and *MoneyTest* have none. When inspected 50 random *JHotDraw* tests and found out that most of them were empty. We let our tool search *Ant*, *Maven*, *JHotDraw* and *MoneyTest* for empty tests.

All the testSetGet and testSetIs tests we found in *Maven* and *JHotDraw* are generated by *JUnitDoclet*. We therefore consider it unlikely that such methods occur in hand written tests. We also counted how often they appear. We counted only the not empty and not set/get tests as interesting because only these are likely to be hand written.

In JHotDraw almost all of the tests are empty. In Maven it is still about 50 percent. *Ant* and *MoneyTest* do not rely on *JUnitDoclet* and do not have any empty tests.

[13] http://www.junitdoclet.org/

case study	total tests	empty tests	set/get tests	interesting tests
Ant	1215	0	0	1215
Maven	209	101	53	55
JHotDraw	1221	1181	36	4
MoneyTest	22	0	0	22

Table 1: Quality summary for the case studies

5.2 NameAnalyzer Case Studies

In the following sections we discuss the results of using naming convention based heuristics to detect the different aspects of the *fully qualified method under test*.

5.2.1 Determine the PUT and CUT

First we validated the output of the *NameAnalyzer* against the annotations. Then we applied a simple heuristic to all tests that says the detection of the CUT and PUT is successful if such a class exists in the package. We applied this heuristic to the annotated tests and all tests.

case study	total CUTS	found CUTs	false negatives	false positives	recall
MoneyTest (Annotations)	34	13	21	21	38 %
MoneyTest (Heuristic)	22	21	1	1	95 %
Maven (Annotations)	84	70	14	14	83 %
Maven (Heuristic, 50)	50	42	8	8	84 %
Maven (Heuristic, all)	209	176	33	33	84 %
Ant (Heuristic)	1215	888	327	327	73 %
JHotDraw (Heuristic)	1221	1221	0	0	100 %

Table 2: Success rate of determining the package and class under test

In this special case recall and precision as well as false positives and negatives are the same. This is because the *NameAnalyzer* produces just one CUT and in all the tests from the sample there is just one CUT per test case. So if the *NameAnalyzer* fails to determine the actual CUT, he also returns a class that is not the CUT. The heuristic produces about the same results as the annotations for *Maven* but is too optimistic for MoneyTest.

5.2.2 Determine the MUT

We not only tested the *NameAnalyzer* against the annotations, but also against another heuristic. It is similar to the one described above. The detection of the MUT is successful if such a method exists in the fully qualified CUT. This implies that the detected MUT can only be validated if the detection of the fully qualified CUT is successful. We applied this heuristic again once only on the annotated tests and once on all tests.

case study	total MUTS	found MUTs	false negatives	false positives	recall	precision
Maven (Annotation)	84	56	28	20	67 %	73 %
Maven (Heuristic, 50)	???	55	???	12	???	82 %
Maven (Heuristic)	???	175	???	34	???	83 %
MoneyTest (Annotation)	34	1	33	21	3 %	5 %
MoneyTest (Heutristic)	34	1	33	45	3 %	2 %
Ant (Heutristic)	???	37	???	852	???	4 %
JHotDream (Heutristic)	???	1085	???	6	???	99 %

Table 3: Accuracy and Precision of the NameAnalyzer for MUT

The success rate for MoneyTest is low because often a "Bag" or "MoneyBag" is added to the test name. In other cases a "simple" is added and some tests are numbered like testNormalize2. The success rate for *Ant* is low because many tests are numbered like test1 up to test23 and for other tests often semantics are used in the test name.

The precision could be increased if we would discard non-existing candidate MUTs.

Out of curiosity we checked how the success rate of the NameAnalyzer is for the empty tests of *JHotDraw* and *Maven* judged by the heuristic. Quite surprisingly it is not 100%, because some names of tests refer to methods that do not exist. We suppose they were removed during a refactoring.

case study	success rate empty
Maven	85 %
JHotDraw	99 %

Table 4: The success rate for empty tests

5.3 Parsing, all methods invoked

case study	total MUTs	found MUTs	false negatives	false positives	recall	precision
Maven	84	72	12	84	86 %	46 %
MoneyTest	34	34	0	8	100 %	81 %

Table 5: Accuracy and Precision of all methods invoked

False Negatives are MUTs that were not found.

False Positives are found methods that were not MUTs.

Some MUTs in *Maven* are not found because the assertions are made in a method that is invoked in the test.

case study	total MUTs	found MUTs	false negatives	false positives	recall	precision
Maven	84	38	46	21	45 %	64 %
MoneyTest	34	34	0	0	100 %	100 %

Table 6: Accuracy and Precision of asserted methods

5.4 Parsing, asserted methods

False Negatives are MUTs that were not found.

False Positives are found methods that were not MUTs.

Some MUTs are found because our parser does not take the PUT into account.

6 Discussion

6.1 Approaches for Construction

In this section we will discuss the proposed ways to annotate a test with the *fully qualified method under test* and summarize their advantages and disadvantages.

6.1.1 Comment/Javadoc

Using javadoc to annotate the MUTs works with all Java versions. Parsing and processing of source files would of course be required. The integration of the MUT in the HTML documentation would be simple. Additionally the Javadoc can also be processed to generate external files if the requirements change.

On the downside this approach needs the source files.

6.1.2 Metadata annotations

Annotations are easy to handle and work well in collaboration with the *Java* Reflection API. This approach does not need source files or processing them because annotations can be kept in the byte code so that they are available at run time. Additionally the annotations can be processed to generate external files if the requirements change.

The disadvantage is that they require *Java* 1.5. It is still quite new, the only 1.5 JVMs available are the ones from *Sun* and *BEA* and there is no version available for MacOS yet. There is no official word from *Apple* yet when one will be released and whether there will be a version for OSX 10.3 and below too or only for OSX 10.4. Additionally the support for *Java* 1.5 of *Eclipse*, especially the JDT, was incomplete even with 3.1 Milestone 3 so *Ant* was required to compile and run the code. Milestone 3 was the current version at the time we worked on this project. At the time of writing this document, the latest version available from *www.java.com* was 1.4.2.

6.1.3 external XML-File

We can make external files easy to handle because we can choose the format with that goal.

The main disadvantage is that the information is not together with the source. That would lead to more cluttering. Additionally writing XML by hand is cumbersome so we would possibly need to provide a tool to simplify the process.

6.1.4 Simple Return Value

This approach is compatible with *JUnit* as it exists, because *JUnit* does not care about return values. It only checks that the test is public, starts with "test" and takes no arguments.

The return value can be used in other tests that need it as a prerequisite. This makes this method well suited for the composition of unit tests. Like *Complex Return Value* it is compatible with JUnit. Besides that, the method is similar to the methods for analysis that use parsing. That creates two problems. The first is that we might validate parsing with parsing. We will likely share code between them, but if there is a bug in this code, the validation might produce false results. And the second is that the *Java* syntax and therefore also the AST are quite complex.

6.1.5 Complex Return Value

Like the approach above this approach is compatible with *JUnit*.

But the return value can only be accessed if the test completes successfully. Additionally this method bloats the test code. The code gets bloated even more if the *Java* Reflection API is used instead of simple Strings.

6.1.6 Summary

This is a quick summary that compares all the different approaches.

	Metadata annotations	**comment javadoc**	**external XML File**	**simple return value**	**complex return value**
additional files	no	no	yes	no	no
needs source files	no	yes	no	perhaps 2)	no
together with source code	yes	yes	no	yes	yes
stable under refactorings	no 1)	limited 1)	no 1)	yes	no 1)
suited for compositions	no 3)	no 3)	no 3)	yes	no 3)
implementation expense	low	medium	low	high	low

Table 7: Summary of all approaches

Legend:
1) might be added/improved see section below
2) depends on implementation
3) might still be added on an other way

General Note about Stability under Refactorings Eclipse can update external files when refactoring. It automatically updates javadoc links that use the { *@link* } format. Even

if an approach would not be stable under refactorings it could still be made so by writing a custom *RenameParticipant*, see *JDT overview p78* [14].

6.2 NameAnalyzer

6.2.1 Determine the CUT and PUT

Finding the package and class under test works quite well in all scenarios, given that it is so simple. There are cases where it fails for example org.apache.maven.JAXPTest where the PUT is javax.xml.parsers. This is case where an application tests the J2SE library it uses.

6.2.2 Determine the MUTs

Finding the method under test works well for *Maven* and *JHotDraw*. The heuristic does not do too badly but in general it yields too few MUTs per test. However it fails for *Ant* and *MoneyTest*. Both *Maven* and *JHotDraw* heavily rely on tests generated by *JUnitDoclet*. The tests for *Ant* and *MoneyTest* seem all to be handwritten. This leads us to believe that this approach for finding the MUTs only works reliably for tests generated by *JUnitDoclet* and not those written by hand. But more case studies would be needed to confirm that. A possible improvement might be to cut off trailing numbers from the test name to better handle numbered tests.

6.2.3 Determine if a Test is Empty

The results are quite shocking with half of the tests for *Maven* and almost all tests of *JHotDraw* being empty. We have faith in these results because we analyzed several random tests by hand and they were empty too. Also the reports generated by *Maven* for *Maven* list a lot of tests that use no time. These are likely the empty ones.

Our approach only works if the test's return type is void. That is the case for all tests we encountered but not a requirement from *JUnit*. If we used a *Simple or Complex Return Value*, our approach would not work anymore. However it could be easily improved to consider a test, that consists only of a return null as empty too.

6.3 Parsing, All Methods Invoked

In the majority of the cases the methods under test are found. Where that is not the case this is because methods invoked by the test are not parsed and the assertions are made there. Of course parsing all invoked methods by a test would not be appropriate, but a simple heuristic that parses invoked methods only if they are defined in the current TestCase or a superclass that is not Assert might be doable.

But a lot more than actual MUTs are generated. These might be reduced to a certain extent by paying attention to the PUT. An other weakness of parsing in general is that it does not work if methods are invoked dynamically. So *Ant* would likely be a problem for the parser too, even if the problems above were addressed.

Finding out to what class an invoked method belong was one of the hardest parts. Looking back it would perhaps been easier to infer the class to which a method belongs by just searching

[14] http://www.eclipse.org/documentation/pdf/

all classes in the PUT. Also to make our parser an *Eclipse* plug-in in order to use the capability of the *Eclipse* AST parser to resolve types has to be reconsidered.

6.4 Parsing, Asserted Methods

Our parser works very well for *MoneyTest*. This is no surprise because it was designed with this case study in mind. For *Maven* the results are lower but still encouraging. This method would certainly also benefit by paying attention to the PUT. For example in Figure 2 the *asserted methods* without paying attention to the put are just startsWith. But the MUT is m1. If the PUT is used the *asserted methods* are just m1.

There remains however the problem that it is just a heuristic and there are cases where it fails, even if implemented correctly. One example is shown in below. In this case we consider only getProject as the MUT. All the other methods are only used to check if it worked correctly. The heuristic however considers only getProject as a setup method and all the accessors as the *asserted methods*. What we in this case consider as the methods under test is the difference between *all methods invoked* and the *asserted methods*. A heuristic that checks if all *asserted methods* are getters could be used to distinguish those two cases.

```
public void testProjectMappingExtends() throws Exception
{
    Project p = MavenUtils.getProject( new File( TEST_DOCUMENT2 ) );

    // Make sure the groupId is inherited correctly.
    assertEquals( "maven", p.getGroupId() );

    assertEquals( "Child Project", p.getName() );
    assertEquals( "maven:child", p.getId() );

    // Test organization inheritance.
    assertNotNull( p.getOrganization() );
    assertEquals( "Apache Software Foundation", p.getOrganization().getName() );
    assertEquals( "http://www.apache.org/", p.getOrganization().getUrl() );

    // Test siteAddress / siteDirectory
    assertEquals( "theSiteAddress", p.getSiteAddress());
    assertEquals( "theSiteDirectory", p.getSiteDirectory());

    boolean found = false;
    List resources = p.getBuild().getResources();
    for ( Iterator i = resources.iterator(); i.hasNext(); )
    {
        Resource r = ( Resource ) i.next();
        File dir = new File( p.getFile().getParent(),
                             "src" + FS + "messages" ).getCanonicalFile();
        if ( r.getDirectory().equals( dir.getPath() ) )
        {
            assertEquals( "check target path",
                          "org/apache/maven/messages",
                          r.getTargetPath() );
            assertEquals( "check includes",
                          Arrays.asList( new String[] { "messages*.properties" } ),
                                                        r.getIncludes() );
            assertEquals( "check excludes", Collections.EMPTY_LIST, r.getExcludes() );
            found = true;
```

```
            }
        }
        assertTrue( "Check found resources for src/messages", found );
    }
```

7 Conclusion

The detection of the *fully qualified method under test* is more complicated than we initially expected. Each case study was different from the others. *Maven* and *JHotDraw* were the most similar due to their generated tests. *Ant* turned out to be the most difficult case study because of the structure of its test. *JHotDraw* turned out to be of limited value because of its huge amount of generated tests that all look the same.

Using naming convention to detect the CUT and PUT works quite well. However the results for the MUTs vary greatly from case study to case study. Given the results of the case studies we conclude that it works only reliable for generated tests.

Parsing the source code of tests did not turn out to be simple. The two main problems are the complexity of the *Java* language and trying to find out to what class an invoked method belongs. As a result our parsers are not very sophisticated and leave much to be desired. The first parser produces encouraging recall values but also produces a lot of noise, resulting in a low precision value. The heuristic extension to this parser manages to increase this value at the cost of recall. We are confident that with a more sophisticated implementation both values for both parsers can be increased. An interesting future project would be to extend the parser so that it could automatically classify tests.

We think that further effort should be put in the *Eclipse Search – Referring Tests* function. It should use more sophicticated ways to search for tests always run in the background and allow a way to navigate directly from methods to their tests and vice versa.

8 Related Work

Binder [Bin99] discriminates between methods under test (MUT) and classes under test (CUT) but he does not discriminate between unit tests which focus on one or on several $MUTS$.

Beck [Bec03] argues, that isolated tests would lead to easier debugging and to systems with high cohesion and loose coupling. *One-method commands* are isolated tests, whereas *multiple method-commands* execute several tests and in the case of *cascaded method test suites* or *multi-facet test suites* depend on each other or on a common scenario. Having had a big influence by his eXtreme programming methodology and by the development of the XUnit framework, it is imaginable that developers have widely adopted his views, leading to a high percentage of *one-method commands*, at least when developed within the XUnit-framework.

Eclipse [Ecl03] provides a *Search – Referring Tests* menu entry, where one can navigate from a method to a JUnit Test, which executes this method. But no difference is made, if this method is used for setting up a test scenario or it was the method under test.

Van Deursen *et al.* [DMBK01] talk explicitly about unit tests that focus on one method and start to categorize them using bad smells like *indirect testing*, which describe tests that we would categorize as independent tests. In another paper [DM02] Van Deursen and Moonen explore the relationships between testing and refactoring, they suggest that refactoring of the

code should be followed by refactoring of the tests. Many of these dependent test refactorings could be automated or at least made easier, if the exact relationships between the unit tests and their methods under test would be known. objects whose time intensive setup has to be repeated for many unit tests and explain the refactorings involved. They mock the loading and initialization of Eclipse, which is a typical usage of mock objects [MFC00]: mocking the behavior of external software. We have not yet categorized unit tests using mock objects, but suggest to

Bruntink *et al.* [BvD04] show that classes which depend on other classes require more test code and thus are more difficult to test than classes which are independent. Using *cascaded test suites*, where a test of a complex class can use the tests of its required classes to set up the complex test scenario, should improve the testability of complex classes.

Thomas [Tho04] argues that the message-centric view deserves more attention .– *one-method tests*, *optimistic and pessimistic method examples* are all reifications of messages and are the atoms of all *one-method commands* and *multiple-method test suites*. Test cases are implemented in XUnit using the "pluggable selector" pattern, which avoids to create a new class for each new test case on the cost of using the reflection capabilities of the system, thus making the *"code hard to analyze statically"* [Bec03].

9 Appendix

Here we present the *Maven* tests that were not generated by *JUnitDoclet* and categorize them according to a taxonomy developed by Markus Gälli [GLN04] . Each is annotated with the MUTS.

9.1 One-method tests

org.apache.maven.JAXPTest.testSAXParser()

```
@Testscape(muts={"SAXParserFactory#newSAXParser"})
public void testSAXParser() throws Exception
{
    // the sax parser should be piccolo
SAXParserFactory factory = SAXParserFactory.newInstance();
factory.setNamespaceAware( true );
SAXParser parser = factory.newSAXParser();
    assertTrue("Wrong sax parser",
        parser.getClass().getName().startsWith("com.bluecast.xml."));
}
```

org.apache.maven.util.InsertionOrderedSetTest.testAddAll()

```
@Testscape(muts={"InsertionOrderedSet#addAll"})
public void testAddAll()
{
    set.addAll( INPUT_SET );
    assertEquals( OUTPUT_SET, set );
}
```

org.apache.maven.JAXPTest.testXMLReader()

```
@Testscape(muts={"XMLReaderFactory#createXMLReader"})
public void testXMLReader() throws Exception
{
```

```
XMLReader reader = XMLReaderFactory.createXMLReader();
    assertTrue("Wrong xml reader",
        reader.getClass().getName().startsWith("com.bluecast.xml."));
}
```

org.apache.maven.project.ProjectInheritanceTest.testProjectMappingExtends()

```
@Testscape(muts={"MavenUtils#getProject"})
public void testProjectMappingExtends() throws Exception
{
    Project p = MavenUtils.getProject( new File( TEST_DOCUMENT2 ) );

    // Make sure the groupId is inherited correctly.
    assertEquals( "maven", p.getGroupId() );

    assertEquals( "Child Project", p.getName() );
    assertEquals( "maven:child", p.getId() );

    // Test organization inheritance.
    assertNotNull( p.getOrganization() );
    assertEquals( "Apache Software Foundation", p.getOrganization().getName() );
    assertEquals( "http://www.apache.org/", p.getOrganization().getUrl() );

    // Test siteAddress / siteDirectory
    assertEquals( "theSiteAddress", p.getSiteAddress());
    assertEquals( "theSiteDirectory", p.getSiteDirectory());

    boolean found = false;
    List resources = p.getBuild().getResources();
    for ( Iterator i = resources.iterator(); i.hasNext(); )
    {
        Resource r = ( Resource ) i.next();
        File dir = new File( p.getFile().getParent(), "src" + FS + "messages" ).getCanonicalFile();
        if ( r.getDirectory().equals( dir.getPath() ) )
        {
            assertEquals( "check target path", "org/apache/maven/messages", r.getTargetPath() );
            assertEquals( "check includes", Arrays.asList( new String[] { "messages*.properties" } ), r.getIn
            assertEquals( "check excludes", Collections.EMPTY_LIST, r.getExcludes() );
            found = true;
        }
    }
    assertTrue( "Check found resources for src/messages", found );
}
```

org.apache.maven.MavenUtilsTest.testMergeMaps()

```
@Testscape(muts={"MavenUtils#mergeMaps"})
public void testMergeMaps()
{
    Map dominantMap = new HashMap();
    dominantMap.put( "a", "a" );
    dominantMap.put( "b", "b" );
    dominantMap.put( "c", "c" );
    dominantMap.put( "d", "d" );
    dominantMap.put( "e", "e" );
    dominantMap.put( "f", "f" );

    Map recessiveMap = new HashMap();
    recessiveMap.put( "a", "invalid" );
```

```
        recessiveMap.put( "b", "invalid" );
        recessiveMap.put( "c", "invalid" );
        recessiveMap.put( "x", "x" );
        recessiveMap.put( "y", "y" );
        recessiveMap.put( "z", "z" );

        Map result = MavenUtils.mergeMaps( dominantMap, recessiveMap );

        // We should have 9 elements
        assertEquals( 9, result.keySet().size() );

        // Check the elements.
        assertEquals( "a", result.get( "a" ) );
        assertEquals( "b", result.get( "b" ) );
        assertEquals( "c", result.get( "c" ) );
        assertEquals( "d", result.get( "d" ) );
        assertEquals( "e", result.get( "e" ) );
        assertEquals( "f", result.get( "f" ) );
        assertEquals( "x", result.get( "x" ) );
        assertEquals( "y", result.get( "y" ) );
        assertEquals( "z", result.get( "z" ) );
    }
```

org.apache.maven.util.DVSLFormatterTest.testFormatNumberNegative()

```
    @Testscape(muts={"DVSLFormatter#formatNumber"})
    public void testFormatNumberNegative() {
        assertFormat("-1", "-1", "0");
//        testFormat("-1.0", "-1", "0.0");
//        testFormat("-1.00", "-1", "0.00");
//        testFormat("-1.000", "-1", "0.000");
//        testFormat("-1.0000", "-1", "0.0000");
//
//        testFormat("-1.10", "-1.10", "0.00");
//        testFormat("-1.100", "-1.1000", "0.000");
//        testFormat("-1.123", "-1.1234", "0.000");
//        testFormat("-1.1234", "-1.1234", "0.0000");
    }
```

org.apache.maven.plugin.PluginManagerTest.testUpgrade()

```
     @Testscape(muts={"PluginManager#installPlugin"})
    public void testUpgrade() throws Exception
    {
        installPlugin("maven-clean-plugin-1.2-SNAPSHOT.jar");

        assertTrue(
            "upgraded clean plugin is not loaded properly",
            pluginManager.getGoalNames().contains("clean:other"));
        //assertFalse("original clean plugin is not removed properly",
        //pluginManager.getGoalNames().contains("clean:original"));
    }
```

org.apache.maven.util.MD5SumTest.testSum()

```
    @Testscape(muts={"MD5Sum#getChecksum"})
    public void testSum()
        throws Exception
    {
```

```
    MD5Sum md5 = new MD5Sum();
    String basedir = System.getProperty("basedir");
    assertNotNull("basedir not provided", basedir);
    md5.setFile( new File( basedir + "/src/test/checksum/input.jar" ) );
    md5.execute();
    String checksum = md5.getChecksum();

    assertEquals( "627ce116c350da6fee656177b2af86eb", checksum );
}
```

9.2 One-method test suites

org.apache.maven.util.StringToolTest.testSplitStringAtLastDelim()

```
@Testscape(muts={"StringTool#splitStringAtLastDelim"})
public void testSplitStringAtLastDelim()
{
    testSplitStringAtLastDelim("org.apache.maven.StringTool", ".", "org.apache.maven", "StringTool");
    testSplitStringAtLastDelim("org/apache/maven/StringTool.java", ".", "org/apache/maven/StringTool", "java"
    testSplitStringAtLastDelim("org.apache.maven.StringTool", "|", "org.apache.maven.StringTool", "");
    testSplitStringAtLastDelim(null, null, null, null);

}
```

org.apache.maven.project.LegacyIdTest.testLegacyToStandardId()

```
@Testscape(muts={"Project#legacyToStandardId"})
public void testLegacyToStandardId()
{
    // test 'ant' -> 'ant:ant'
    String legacyId = "ant";
    String standardId = "ant:ant";
    assertEquals("Single id conversion failed", standardId,
        Project.legacyToStandardId(legacyId));

    // test 'ant:ant' unchanged
    legacyId = "ant:ant";
    assertEquals("Standard id conversion failed", standardId,
        Project.legacyToStandardId(legacyId));

    // test 'ant+optional' -> ant:ant-optional
    legacyId = "ant+optional";
    standardId = "ant:ant-optional";
    assertEquals("Plus format id conversion failed", standardId,
        Project.legacyToStandardId(legacyId));
}
```

org.apache.maven.project.RepositoryTest.testGetCvsRoot()

```
@Testscape(muts={"Repository#getCvsRoot"})
public void testGetCvsRoot() throws Exception
{
    repository.setConnection("scm:cvs:pserver:anoncvs@cvs.apache.org:/home/cvspublic:module");
    String root = repository.getCvsRoot();
    assertEquals("Wrong root returned", ":pserver:anoncvs@cvs.apache.org:/home/cvspublic", root);
    repository.setConnection("scm|svn|http://svn.apache.org/repos");
    root = repository.getCvsRoot();
    assertEquals("Wrong root for non CVS string", "", root);
}
```

org.apache.maven.plugin.PluginManagerTest.testInstallTwice()

```
/**
 * Make sure the plugin manager can install a plugin twice without bad
 * effects.
 * @throws Exception when any error occurs
 */
@Testscape(muts={"PluginManager#installPlugin"})
public void testInstallTwice() throws Exception
{
    installPlugin("maven-java-plugin-1.3.jar");

    assertTrue("java plugin is not loaded properly",
    pluginManager.getGoalNames().contains("java:compile"));

    installPlugin("maven-java-plugin-1.3.jar");

    assertTrue("java plugin is not loaded properly",
    pluginManager.getGoalNames().contains("java:compile"));
}
```

9.3 Pessimistic method examples

org.apache.maven.util.HttpUtilsTest.testParseInvalidUrl()

```
@Testscape(muts={"HttpUtils#parseUrl"})
public void testParseInvalidUrl() throws Exception
{
    String invalidUrl = "a@username:http://password";
    try {
        String[] up = HttpUtils.parseUrl( invalidUrl );
        fail("expected RuntimeException to be thrown but got " + up.length + " elements.");

    } catch (RuntimeException e) {
        assertTrue(true); // expected
    }
}
```

9.4 One-method example commands

org.apache.maven.util.InsertionOrderedSetTest.testAdd()

```
@Testscape(muts={"InsertionOrderedSet#add"})
public void testAdd()
{
    set.add( STRING_1 );
    set.add( STRING_2 );
    set.add( STRING_1 );
    set.add( STRING_3 );

    assertEquals( OUTPUT_SET, set );
}
```

References

[Bec03] Kent Beck. *Test Driven Development: By Example.* Addison-Wesley, 2003.

[Bin99] Robert V. Binder. *Testing Object-Oriented Systems: Models, Patterns, and Tools.* Object Technology Series. Addison Wesley, 1999.

[BvD04] Magiel Bruntink and Arie van Deursen. Predicting class testability using object-oriented metrics. In *Proceedings of the Fourth IEEE International Workshop on Source Code Analysis and Manipulation (SCAM).* IEEE Computer Society Press, September 2004.

[DM02] Arie van Deursen and Leon Moonen. The video store revisited - thoughts on refactoring and testing. In M. Marchesi and G. Succi, editors, *Proceedings of the 3nd International Conference on Extreme Programming and Flexible Processes in Software Engineering (XP2002)*, May 2002.

[DMBK01] Arie van Deursen, Leon Moonen, Alex van den Bergh, and Gerard Kok. Refactoring test code. In M. Marchesi, editor, *Proceedings of the 2nd International Conference on Extreme Programming and Flexible Processes (XP2001)*, pages 92–95. University of Cagliari, 2001.

[Ecl03] Eclipse Platform: Technical Overview, 2003. http://www.eclipse.org/white-papers/eclipse-overview.pdf.

[GLN04] Markus Gälli, Michele Lanza, and Oscar Nierstrasz. Towards a taxonomy of unit tests, 2004. Submitted to CSMR05.

[MFC00] T. Mackinnon, S. Freeman, and P. Craig. Endotesting: Unit testing with mock objects, 2000.

[Tho04] Dave Thomas. Message oriented programming. *Journal of Object Technology*, 3(5):7–12, May 2004.