b
UNIVERSITÄT
BERN

Test name recommendation

A study of the unit test naming and naming traditions

Bachelor Thesis

Christian Zürcher
from
Liebefeld, Switzerland

Philosophisch-naturwissenschaftlichen Fakultät
der Universität Bern

Summer 2019

Prof. Dr. Oscar Nierstrasz
Dr. Mohammad Ghafari

Software Composition Group
Institut für Informatik
University of Bern, Switzerland

Abstract

The name of a unit test is an essential part of it and helps the developers to understand its purpose and to identify tests inside test suites. And even though such names have a lot of benefits, many tests end without a descriptive name. This occurs not only in automatically generated tests but also in manually written ones. Automatically generating descriptive names is confronted with the challenge that there is a vast variety of tests, written by different developers with different conventions, or generated by different tools. In this thesis, we present an automated approach to generate descriptive names based on the test body by finding the focal method of the test, around which it was written. We compared our results to the original names and to other publications to find out that our approach provides good results for all kind of tests even though in specific scenarios, other approaches may work better. Finally, we found out, that the names created by developers, when done correctly, are still the most descriptive.

Contents

1 Introduction

In this thesis, we answer the following research questions:

- **RQ1:** ***Does the information about focal method under test help to identify the different sections of a test case (setup, execution and oracle)*?**
- **RQ2:** ***How can the information about the different parts of a test help to generate descriptive test names*?**

One of the most difficult aspects of software maintenance is comprehension – understanding the software that needs to be modified. The amount of time needed by developers to locate and understand code is usually greater than the amount of time that they spend on making modifications [1]. This has motivated the creation of automated code summarization tools, which analyse code, extract "important" statements, and produce natural language summaries. Through these summaries, it is easier to gain a quick understanding of what the code is doing. Many existing works have shown the feasibility and benefits of summarizing Java methods and classes [2–6] and the generation of method names. Høst and Østståvold [7] mine naming rules from a corpus of source code, and then suggest new names for methods that do not match these rules. More recently, Allamanis *et al.* [8] applied a log-bilinear neural network to learn a model that can suggest method names based on features (embeddings of calls, co-occurrences,...) extracted from the source code. Recently, the summarization of unit test code has been shown to improve the understandability of test cases [9–12].

When trying to understand code, unit tests can be consulted as usage examples. When maintaining code, unit tests help to identify undesired side-effects. In this context, one of the most crucial pieces of information to appear in such a summary is of the purpose of a test. For example, when a test fails, it is necessary to understand the purpose of the test as a first step towards identifying the cause of the failure. Knowing the purpose of a test is necessary to decide whether the test should be left alone, modified, or removed in response to changes in the application under test.

The potentially most useful source of information for quickly understanding a test is its name. Ideally, test names are descriptive in that they accurately summarize both the scenario and the expected outcome of the test [13]. If a test name is descriptive, developers no longer have to read through the body to understand its purpose. In addition, descriptive names (1) make it easier to quickly tell if some functionality is not being tested – if a behaviour is not mentioned in the name of a test, then the behaviour is not being tested, (2) help prevent tests that are too large or contain unrelated assertions – if a test cannot be summarized, it likely should be split into multiple tests, and (3) serve as documentation for the class under test – the responsibilities of a class under test can be identified by reading the names of its tests. Providing tests with good names simplifies all these tasks, which is important considering the substantial costs and effort of software maintenance [14].

For example, consider the example class Package (Listing 1.1), which has two methods `addWeight` and `getTotalWeight`. Given a test named *addWeightThrowsIllegalArgumentException* we can immediately see, even without using the test code, what the purpose of the test is (call `addWeight` with an argument that makes it throw an `IllegalArgumentException`), which part of the code it uses (method `addWeight`), and it is reasonable to assume that the test provides an example of unintended usage of the class Package. Tests named *addWeightIsTrue* and *addWeightIsFalse* would immediately reveal with their name that they provide two different scenarios for the `addWeight` method. When modifying the `addWeight` method, a developer would know that these tests are the first ones to run, and when one of these tests fails during continuous integration, the developer would know immediately where to start debugging by looking at the failed assertion and the information provided in the test name.

Nevertheless, not all tests have descriptive names. Developers often write tests with poor names because naming is difficult and there is no immediate downside if a bad name is chosen. For example, developers may create generic test names (e.g., test1, test2, etc.) or test names that contain little information (e.g., testAdd, testSubtract, etc.). Also, a test name can become erroneous when it is out of sync with the test body. For instance, a developer may modify a test body to reflect a changed behaviour of the class but fail to make the corresponding changes to the test name. Such erroneous names no longer accurately summarize the test purpose. In practice, erroneous names can be more harmful than poor names. Because poor test names are often easily identifiable, developers are unlikely to consider them as a useful source of information. Conversely, erroneous names

```java
1  public class Package {
2    private int totalWeight = 0;
3    private final static int MAX = 5000;
4
5    public boolean addWeight(int weight) throws IllegalArgumentException {
6      if (weight <= 0)
7        throw new IllegalArgumentException("Weight cannot be negative or zero");
8      if (totalWeight + weight > MAX)
9        return false;
10     totalWeight += weight;
11     return true;
12   }
13 }
```

Listing 1.1: A package class that keeps track of the weight of the items inside it and has a maximal weight limit.

often appear plausible and can easily lead developers into making incorrect assumptions.

The same problem arises with automatically generated tests which save time and effort and can improve the code coverage achieved by manually written tests. Although automated test generation tools can produce tests that achieve high code coverage, these tests typically come without meaningful names. For example, the EvoSuite [15] and Randoop [16] tools name their tests `"test0"`, `"test1"`. These names give no hint to the content of the tests, and navigating such tests by name is impossible. Thus, even though the tests might achieve good code coverage, there is reason for concern when it comes to understanding, debugging, and maintaining such tests. The challenge, however, is that automatically generated tests tend to be nonsensical and have no clear purpose other than covering code, which makes it difficult to apply standard conventions to derive good names. Indeed, when the only purpose of a generated unit test is to cover line 8 of a method, then naively capturing this with a name like `"testCoversLine8"` is not helpful either.

There is already work done in improving the names of test cases. Daka *et al.* [17] have created an extension for EvoSuite [15], which largely improves the names of generated tests by using the information of the test generation process. However, it is restricted to tests during their creation and cannot be applied to already existing tests. Also, Zhang *et al.* [12] have created a prototype tool with a natural language program analysis based approach that can be run on JUnit tests. Although it has promising results, the approach is limited to unit tests with one assertion and with meaningful method names.

In this work, we use another approach to this naming problem that should work on all the different forms of unit tests. We recover test-to-code traceability links at method level to identify the actual method under test in a unit test case and distinguish the setup, execution and oracle part of this test. Since these pieces of information are available in all tests, we use them to create a name for the test case.

But even though understanding the relationship between test code and source code is essential for software evolution by maintaining unit test cases in sync with the changes to the source code, the practical automated realization of test-to-code traceability at method-level has received little attention in research.

XUnit[1] testing frameworks lack predefined structures for explicitly linking program source code and test cases [18, 19]. In consequence, developers have to resort to costly manual detection and maintenance of the traceability links [20].

Previous research mostly derived traceability links between test cases and classes under test [21]. Although the knowledge of the class under test (CUT) is useful for program comprehension and maintenance, test cases are composed of method invocations that play different roles in the test, and more useful information, especially for creating descriptive test names, can be derived by analysing test cases with a method-level precision.

```java
public void testRegisterAndRemoveProxy() {
  // register a proxy, remove it, then try to retrieve it
  IModel model = Model.getInstance();
  IProxy proxy = new Proxy("sizes", new String[]{"7","13","21"});
  model.registerProxy(proxy);
  // remove the proxy
  IProxy removedProxy = model.removeProxy("sizes");
  // assert that we removed the appropriate proxy
  assertEquals(removedProxy.getProxyName() , "sizes");
  // ensure that the proxy is no longer retrievable from the model
  proxy = model.retrieveProxy("sizes");
  assertNull("Expecting proxy is null", proxy);
}
```

Listing 1.2: Unit test case for the Model class

Consider an example unit test case from *PureMVC*[2] in Listing 1.2. This test asserts that once a proxy is registered to a model, it can also be removed, and once it was removed, the model does not contain it any more. In the example, `Model` is the CUT. The knowledge of the CUT can help to identify the method under test. However, CUT information is insufficient to identify the tested method (or focal method under test) – in the example, three methods in the test case belong to the CUT `Model` any of which can be the method under test.

The real intent in the test case in Listing 1.2 is to check the `removeProxy()` method. An expert engineer can identify this with the aid of comments, test method names, and assertions. Without this knowledge or additional analysis, one might mistakenly conclude that the goal of the test is to check `registerProxy()` or `retrieveProxy()` methods. Method `registerProxy()` seems to be a relevant method to the test case, but this method is ancillary, it brings the `model` object to

[1]XUnit is a collective name for unit testing frameworks for different programming languages.
[2]http://puremvc.org

an appropriate state in which it is possible to invoke the `removeProxy()` method. The `retrieveProxy()` helps to inspect the state of the class under test and it is the method `removeProxy()` that causes a side-effect on the current object and is the focal method under test.

Ghafari *et al.* [22], propose an approach for detecting such methods, which are called *focal methods under test* (F-MUT) in unit test cases. They focus on classes that define stateful objects, where the FMUTs are responsible for system state changes that are verified through assertions by test cases.

In this thesis, we create a tool that identifies F-MUTs and creates a test name based on that result. For this purpose, we based ourselves on the approach proposed by Ghafari *et al.* and implemented these ideas with the Soot framework [23] for static analysis and the ByteBuddy framework for dynamic analysis. We have applied the tool to 3 random sets of 50 test cases from 3 different projects (plus 313 test cases that helped in the development). The tool detects F-MUTs with 67,7% precision and 72,5% recall. We then investigated whether F-MUT information helps to distinguish the various parts of a unit test case, namely, setup, execution and oracle. Finally, we used the obtained information from each test to recommend a name and then compared it to the existing ones and the ones proposed by NameAssist of Zhang *et al.* [12].

The rest of the thesis is structured as follows. In chapter 2 we elaborate the different concepts of F-MUTs and related subjects, and the motivation behind using them. In chapter 3 we concretely described the contributed tool and in chapter 4 we discuss the results we obtain from it. In chapter 5 we see further work and will find a conclusion. Finally, in chapter 7, we find a practical manual to start using Soot.

2 Related work

2.1 Related groundwork

The ultimate aim of providing unit tests with good names is to improve understandability, and thus to improve software maintenance activities that involve unit tests. There are, however, alternative approaches that aim to achieve the same. Natural language test summaries [24] or test documentation [10] can help to make tests understandable [24]. Tests can be made easier to understand by simplifying them, for example by reducing the number of statements [25, 26], by reducing the number of assertions [27], or by splitting tests to one for each assertion [28]. Search-based approaches make it possible to include additional optimisation goals aiming to make tests more understandable, for example by making them more similar to realistic object usage scenarios [29], by making generated strings resemble natural language text [30], by optimising the syntactic readability [31], or by improving coupling and cohesion [32].

Since generating names for tests is a somewhat different problem than generating generic method names and because there is no consensus on an ideal test name, only some common recommended guidelines and conventions, Zhang *et al.* [12] proposed a natural language program analysis based approach to generate names for unit tests based on the common structure of tests and their existing names. In particular, for a given test they identify (1) the action (e.g., the method under test, also defined as focal method under test or **F-MUT** by Ghafari [22]), (2) the scenario under test (e.g., the parameters and context of the action defined in the setup), and (3) the expected outcome (e.g., the assertion, called oracle part). These three parts are then converted to text using a template-based approach, resulting in three alternative names which include (1) only the action, (2) the action and the expected outcome, or (3) the action, the expected outcome and a summary of the scenario under test. The question which of the three alternative names should be used in practice remains open, moreover, there are still some problems: The description of the scenario under test relies on descriptive variable names, which are not always available, especially in automatically generated tests. The expected outcome is assumed to be checked by a single test assertion, but tests often have many assertions. Finally, since automatically generated tests are often generated for coverage, they may target more than one method under test [17].

Dake, Rojas and Fraser [17] proposed an extension to the open-source EvoSuite test generation tool [15] which generates meaningful names instead of the ones given by the tool, thus fixing the problem of poor naming by the test generation tool (`"test#"` where # represents a number).

Zhang *et al.* [12] found that 29% of the test names in a corpus of 213,423 manually written tests were constructed using the word `"test"`, optionally followed by a number. In comparison, 62% of names include the name of the method under test which is surprisingly much. And if the method under test is simple and only has a single behaviour, naming a test by the method it calls may be sufficient (e.g., *testGetTotal*).

However, often methods are not as simple. A widely used strategy explained by Osherove in "The Art of Unit Testing" [33] and Trenk in his article for "Testing on the toilet" [13], and also used by Zhang *et al.* for their approach, requires three parts for a good unit test name: the method under test, the state under test, and the expected behaviour; all three parts should be included in a single name, although the order of expected output and state under test can vary. For instance, a test for the `Package` example class of listing 1.1 in which the `IllegalArgumentException` is triggered (expected output) when the `addWeight` method is called with negative input (state under test), could be named:

`addWeight_WithMinus1_ThrowsIllegalArgumentException`

Whether or not underscore characters should be used in a test name is a controversial question without clear consensus. Also, we could argue if the expected value should be displayed with the actual value or a generalized goal (e.g. "WithNegative") [17]. Generally, naming guidelines and conventions expect good names to encode a fair amount of detail and to explain the specific scenario under test. And even though the majority of tests have at least somewhat more descriptive names, they do not fulfil the proposed guidelines above. All this reveals the necessity of an approach to generate names for all kind of unit tests, not only automated ones.

3
The Tool

3.1 Background

Unit test cases are commonly structured in three logical parts: setup, execution, and oracle. The **setup part** instantiates the class under test and includes any dependencies on other objects that the the focal method under test will use. This part contains initial method invocations that bring the object under test (instance of the CUT) into a state required for testing. In other words, the setup creates the context of the test. The **execution part** stimulates the object under test via a method invocation, i.e., the focal method under test (**F-MUT**). This action is then checked with a series of inspector methods (the term is used by Ghafari [22] and refers to a method that returns a value but does not modify the state of the object) and assert statements in the **oracle part** that controls the side-effects of the focal method to determine whether the expected outcome is obtained.

Despite clear logical differentiation of test parts each having their purpose, in practice, the parts are often hardly discernible either manually or automatically. This hinders identifying F-MUTs without expert knowledge of the system. It is difficult to establish whether a method invocation belongs to the setup or execution parts of a test. Even the oracle part associated with assert statements may contain method invocations that may be confused with the execution part of the test case [22].

The main challenge in automatically generating tests names is that we have a large population of different tests, all demanding different approaches. There can be automatically generated tests, poorly written cases following no conventions, tests with multiple assertions or even multiple F-MUTs and scenarios, or simple, conventional tests. So we have to assume that there is a lack of real scenario or purpose (because of the automatically generated tests) that could be used to derive meaningful names. Further, we cannot rely on variable naming to describe input and output behaviour. Finally, we have to assume that there will be tests with lots of results due to multiple assertions, F-MUTs or even CUTs. However, it should still be possible to generate names that provide some of the benefits good names for manually written tests also offer. Daka *et al.* have identified three requirements for good names for generated tests [17]:

- R1 Test names should have a clear relation to the code under test (the F-MUT); they should allow developers to identify the tests concerning this method without having to inspect the test code.
- R2 Test names should be descriptive of the test code; there should be an understandable, intuitive relation between the test code and its name, meaning that the scenario of the test (setup, execution and oracle part) should be recognized by reading the name.
- R3 Test names should uniquely distinguish tests within a test suite, so that developers can use them to navigate the test suite.

So first, we need to add the information of the execution part, which represents the tested behaviour and is, therefore, the code under test, to create the test to code link. Then we also need information about the setup and the oracle part, to gain an understanding of the state the code is tested in (setup) and of the expected outcome (oracle). Finally, we have to make sure that the generated name is unique in its test class, since this is a JUnit requirement.

3.2 The Analysis Pipeline

To analyse the tests, we mostly use Soot [23], a Java bytecode optimization framework which is also largely used for analysing Java applications. The tool analyses each test case one by one. For this, it uses two main parts. First, it detects the FMUT (or multiple FMUTs) of the test case by searching for the different inspector methods with static analysis, based on Soot, and mutator methods with dynamic analysis, based on ByteBuddy, a runtime code manipulation framework. In this work, the term mutator method is used to design not only setter methods, but all methods that modifies the state of an object. We then use this information to find the required FMUT. Afterwards, this information is used to generate a possible name for the test case. The flow graph in the figure 3.1 below shows an overview of the whole procedure and we will illustrate it with the help of the fictive example of the listings 3.1.

```
1 public void testDifferentJimpleExprAndValues() {
2   Person person = new Person();
3
4   person.setNumberOfFingers(10);
5
6   int numberOfFingers = person.getArms().get(0).getFingerCount();
7
8   assertTrue(numberOfFingers > 4);
9 }
```

Listing 3.1: Example Java method

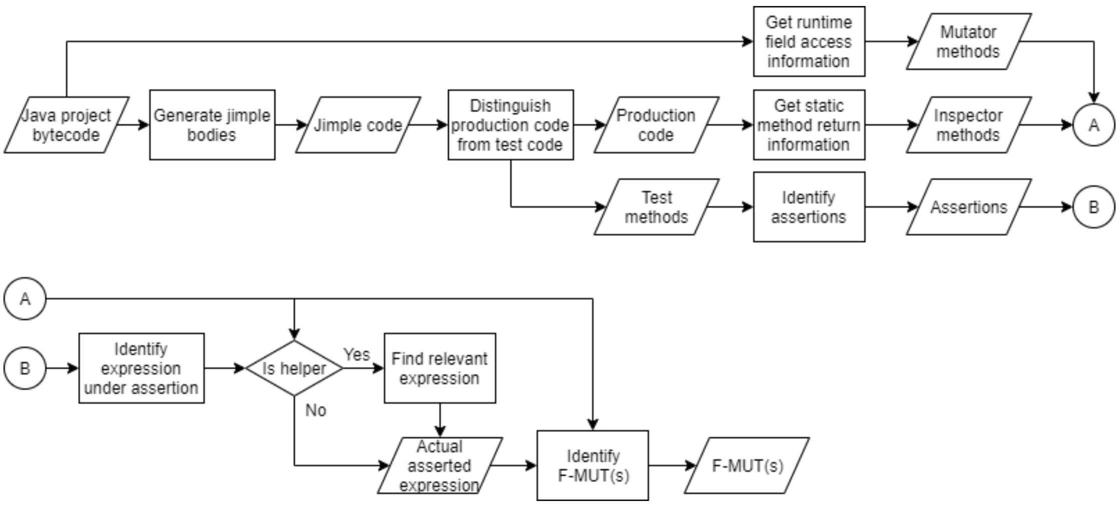


Figure 3.1: Flow graph of the analysis.

We start with the **Java binaries** of the project including its test cases. These can be in the form of a .jar file or as raw folder structure. In the second case, the folder structure has to be ordered in the conventional package system (the files are in the folder of their respective package), otherwise, Soot will not be able to retrieve the files.

These binaries will then be **transformed into Jimple bodies** for each method in each class. Jimple is a concrete three-address code language provided by the Soot framework. It is used to simplify optimization, and in our case, analysis of Java code. As we can see in listing 3.2, which is the Jimple body of the code in listing 3.1, each expression is broken down to simpler sub-expressions each representing one operation (function call, primitive operation, ...) with an optional value assignment (e.g. `c = function(a,b)`). A good example here is the line 6 of the Java code that is split up into the lines 7 to 10 in the Jimple code.

```
1  public void testDifferentJimpleExprAndValues() {
2    person = new project.Person;
3    specialinvoke person.<project.Person: void <init>()>();
4
5    virtualinvoke person.<project.Person: void setNumberOfFingers(int)>(10);
6
7    $stack1 = virtualinvoke person.<project.Person: java.util.List getArms()>();
8    $stack2 = interfaceinvoke $stack1.<java.util.List: java.lang.Object get(int)>(0);
9    $stack3 = (project.Arm) $stack2;
10   numberOfFingers = virtualinvoke $stack3.<project.Arm: int getFingerCount()>();
11
12   if numberOfFingers <= 4 goto label1;
13   $stack5 = 1;
14   goto label2;
15
16  label1:
17   $stack5 = 0;
18
19  label2:
20   staticinvoke <org.junit.Assert: void assertTrue(boolean)>($stack5);
21 }
```

Listing 3.2: Example Jimple method

Once this process is done, the different methods are **categorized into production methods and test methods**, by checking for the JUnit `Test` annotation, the “test-” prefix and existing assertions inside the method. To be considered as test methods, the method has to at least have the annotation or the prefix and contain a valid assertion provided by JUnit. Customized assertions, assertions called indirectly via helper methods, are not detected. These type of assertions were mainly used in one project and constitute around a fifth of the cases of that project. Also, most of these cases use them alongside JUnit assertions, so we can still get acceptable results based on the JUnit assertons. Then we also ignore all methods with parameters since JUnit3 and JUnit4 tests do generally not include parameters. In the example, we would have the methods of the class `Person` as production methods and the method `testDifferentJimpleExprAndValues()` as the test method.

The production code is then subject to a static **inspector analysis** done directly with the Jimple code. The production methods (e.g. `setNumberOfFingers()`, `getArms()`) will then be classified as inspector or not depending on the return statements. The code is also subject to a **mutator analysis** which is not done statically as the first one but at runtime, meaning, the tests are ran and the states analysed. Both of these procedures are further explained in details in the coming subsections.

The analysis then takes each test method one by one and traverses them to find the **actual asserted expression(s)** (**AAE**). For this, we start at each assertion (in this case only line 20) and then follow the declarations of the used variables (use-definition chain or use-def chain for short). In the listing 3.2, the main use-def chain will be: $\$stack5- > numberOfFingers- > \$stack3- > \$stack2- > \$stack1- > person$

as for example, the initialization of the variable `$stack3` on line 9 uses the variable `$stack2`. And then, by using the mutator and inspector results, we identify the F-MUT(s) of the given AAE and finally merge it to the other ones found for this test method (in this case there will be no merge since there is only one assertion and one focal method).

3.2.1 Inspector Analysis

The inspector analysis creates a map of all the methods used in the test case and categorizes them into the different categories which will be used to identify a method as an inspector. They also offers the possibility to change the definition of inspector methods by adding or removing categories to the inspector definition list. The categories do not impact the generation of the names directly but simplify the continuation of the analysis by avoiding to redo the same checks again later on (we can now just check the corresponding category). The analysis traverses the obtained Soot Jimple bodies of the test to check the body of the methods that are used. Based on these bodies, we analyse what is returned since when the method does not return anything, by definition, it will not be an inspector. So we start with the return statements and look for the first relevant element.

Algorithm 3.1: The inspector search

```
  input  : Jimple body of the method $BODY$
  output : classification of the method
1 inspectorSearch ($BODY$)
2     if there is no actual $BODY$ then
3         return $NO\_BODY$ or $ABSTRACT$ depending on the case;
4     $RETURNS$ = find all returns of $BODY$;
5     for each $RET$ in $RETURNS$ do
6         find the category of $RET$ and add it to $CAT$;
7     return merge $CAT$ for the method category;
```

First, the different returns are classified in the following categories:

- **FIELD_RETURN**: returns a field of the instance.
- **CONSTANT_RETURN**: returns a constant value.
- **STATIC_FIELD**: returns a field of a static instance.
- **DEEP_NO_BODY**: returns the result of a method that has **NO_BODY**.

- **DEEP_SEARCH**: returns the result of another inspector method.
- **NEW_INSTANCE**: returns a new instance of an object (e.g. a local variable).
- **JAVA_LIBRARY**: returns the result of a Java library method which is non-production code defined in the Java JRE and JDK.
- **RECURSION**: returns the result of the same method (or multiple methods) called recursively which would lead this analysis to throw a StackOverflowError. So these cases are detected here to avoid this error in the rest of the analysis.
- **IF_DEPENDENT_RETURN**: returns a value depending on the result of an if statement. In Jimple, this is most of the time a boolean return of another method.
- **NO**: the return does not match any of these criteria.

Once all the returns of the method are classified, these results are merged to classify the method itself. If there is only one return in the method, the method is classified in the same way as its return, but in the case of multiple returns, the following categories are used:

- **NO**: if any of the returns is a **NO**. These methods are never inspectors.
- **MULTI_RETURN**: the method contains different returns where there is at least one return that is not **FIELD_Return** or **CONSTANT_Return**.
- **MULTIPLE_CONST_OR_FIELD_RETURN**: The method returns only fields or constants depending on a branch.
- **COMPUTATION_RETURN**: With the observation that complex methods (large methods with many variables) are generally not inspectors, we also introduced the concept of computation method. These methods may not always alter the fields but are complex enough so that they should be tested. But there is no clear criteria, so we inspected over 50 inspector methods and 50 non-inspector methods and established the following thresholds: a computation method has at least 30 Jimple code lines and 25 Jimple variables (defined and temporary ones). In all the checked occurrences of this case, this approach worked.

There are also some special cases for the methods where the above procedure cannot be used:

- **NO_BODY**: the method has no Soot body. This may occur due to different reasons:
  - The test uses a mock object and does not define this method.

  - The test uses a non-production object instance which is not loaded by Soot and therefore does not contain a body in the Jimple code.

- **ABSTRACT**: similar to **NO_BODY**. Since Soot creates the SootMethod with the declared type and not the runtime type, it may try to retrieve the body (code) of an abstract method or an interface which does not have a body. The identifier **ABSTRACT** also helps to identify such methods later so that they can be linked to their runtime types when needed. As example, in the listing below, when trying to retrieve the body of the `setName()` method, Soot will get the non existent body of the abstract method defined in `Human` instead of the runtime implementation in `Woman`.

```java
// Human is an interface and Woman its implementation.
Human person = new Woman();
person.setName("Sara");
```

3.2.2 Mutator Analysis

The mutator analysis creates a map of all the methods used in a test case to find out if a method changes any variables of its class or its parameters. Using ByteBuddy[1], an agent is created to intercept all method calls and monitor the variables and parameters before and after the execution of that method. Each variable and parameter is compared with its previous state and if changes occur, they are registered. The first approach was to compare them using JSON strings of the variables, but we encountered problems with objects referencing each other, leading to infinite loops in the JSON serialization. So we opted for a deep clone approach. The changes are also differentiated between parameter changes and variable changes. Also, the changed name of the changed variable is stored for later use.

This analysis depends on the tests being successful, which was not always given either due to wrong Eclipse settings and imports or because the project is still in development. If the test fails, parts of the test will not be recorded since the failed assertion stops the execution of the test.

3.2.3 Focal method Analysis

A method has focal character if it is not an inspector or if it is a mutator. Here, the results of the inspector and the mutator analysis are used to classify the method as focal or not. Depending on the case, there are 3 different ways to determine if a method is focal or not:

[1]https://bytebuddy.net

- if the initialization is seen as a mutation (can be configured before running the analysis) it is focal.
- the simple check looks if the method is an inspector or a mutator (changes a variable or parameter depending on the use). And if the condition `(!inspector || mutator)` returns true, the method is focal.
- the complex check is a combination of the two methods above and also checks if the method belongs to the Java library, in which case it is declared an inspector. It uses slightly different parameters that are not always available, that is why the other two cases are sometimes used.

Procedure Using the inspector and mutator information, we can identify the F-MUT(s) in each test. This part starts with the variable asserted in the associated assertion. It then repeats the whole process by changing the actual tested variable (**ATV**) until the F-MUT is found or the whole test case is traversed.

Algorithm 3.2: The FMUT search

```
input  : assertion $ASRT$, actual tested variable $ATV$
output : F-MUT(s)
1 FMUTSearch ($ASRT$,$ATV$)
2     $INIT$ = initialization of $ATV$;
3     for each $UNIT$ from $ASRT$ to $INIT$ do
4         if $UNIT$ has focal character then
5             return $UNIT$;
6     for each $VAR$ in variables of $UNIT$ do
7         return FMUTSearch ($ASRT$,$VAR$);
```

We start with the assertion and its asserted variable which is our first ATV. We then follow the following process:

1. We search for the initialization (or declaration) of the ATV, which is always an assignment statement in Jimple, making it easier to find.
2. Once we have this initialization, we check all the lines (which are always one operation) between the assertion and the initialization, starting from the assertion, for focal character. If one is found, it is returned as F-MUT and the process ends.
3. If no F-MUT was found, we start the whole process again for each of the variables used in the declaration of the ATV. It is because of this part that we can find multiple F-MUTs for a single assertion, even though this happens only on rare occasions.

Since Soot has to convert all kinds of Java code, the framework introduces a lot of different statement types, and all of these can potentially appear in a test case. So we had to cover as many of these cases as possible. In the following list, we have grouped the major cases used to declare a variable:

- **Direct assignment:**
  A fixed value is assigned to the variable. If the variable is assigned with a primitive value, this branch ends. Otherwise, if the value is another variable, we continue the search with that variable. There is one exception to the dead end due to primitives. If the assigned primitive is a 1 or a 0 and that this assignment is preceded by a `label` token, it is part of a `if-goto` block in which case the branch goes on with the condition of the block (there is an example in the listing 3.4).

- **Instance method invocation:** `[var].[method]([args]);`
  This is the most basic kind of variable declaration used, assigning the return value of a method to the variable. In this case, if no F-MUT(s) are found, the search goes on with the variable (`[var]`) and the arguments (`[args]`) if they exist.

- **Static method invocation:** `[this/Class].[method]([args]);`
  This works basically like the **Instance method invocation** except there are only parameters and no variable, and that we also search for other methods being statically called on the class.

- **Field reference:** `[this/Class].[field];`
  First, we check the test case to see if this field (`[field]`) is used somewhere before and is affected by a focal method. If there is no such occurrence, we check the setUp method of the test class (and also its parents if necessary) to see if one of those contains at least one focal method acting on this `[field]`.

- **Object declaration:** `new [Class];`
  If the class is a project class (declared inside the analysed project), the declaration will count as F-MUT (this behaviour can be disabled). Otherwise, the search goes on with the parameters of the initialisation. If an array is declared, we do the same steps but check if the array type is a project class or not.

- **Unary operator:** `[op][arg];`
  These are Java operators that take one argument (e.g. `instanceof`, `!`, a cast, ...). In these cases, we just go on with that argument. In the case of `instanceof`, and casts, we classified them as a unary operator since one of their arguments only serves as a type identifier.

- **Binary operator:** `[arg1][op][arg2];`
  These are Java operators that take two arguments (e.g. +, >, `||`, ...). In these cases, we again create two branches, one for each argument.

Example We will now illustrate the procedure above with an example code. This code was written for the only purpose to contain as much different cases in as few lines as possible.

```java
@Test public void testSetAverageConeCellsTo5MHasOver8MConeCells() {
  Person person = new Person();
  person.setAverageConeCells(5000000);
  List<Eye> eyes = person.getEyes();
  int numberOfCones = eyes.get(0).getConeCells() + eyes.get(1).getConeCells();
  assertTrue(numberOfCones > 8000000);
}
```

Listing 3.3: Example Java method

```
 1 public void testSetAverageConeCellsTo5MHasOver8MConeCells() {
 2   $stack4 = new testProject.Person;
 3   specialinvoke $stack4.<testProject.Person: void <init>()>();
 4 
 5   virtualinvoke $stack4.<testProject.Person: void setAverageConeCells(int)>(5000000);
 6 
 7   eyes = virtualinvoke $stack4.<testProject.Person: java.util.List getEyes()>();
 8 
 9   $stack6 = interfaceinvoke eyes.<java.util.List: java.lang.Object get(int)>(0);
10   $stack7 = (testProject.Eye) $stack6;
11   $stack8 = virtualinvoke $stack7.<testProject.Eye: int getConeCells()>();
12 
13   $stack9 = interfaceinvoke eyes.<java.util.List: java.lang.Object get(int)>(1);
14   $stack10 = (testProject.Eye) $stack9;
15   $stack11 = virtualinvoke $stack10.<testProject.Eye: int getConeCells()>();
16 
17   numberOfCones = $stack8 + $stack11;
18   if numberOfCones <= 8000000 goto label1;
19 
20   $stack12 = 1;
21 
22   goto label2;
23 
24  label1:
25   $stack12 = 0;
26 
27  label2:
28   staticinvoke <org.junit.Assert: void assertTrue(boolean)>($stack12);
29 }
```

Listing 3.4: Example Jimple method

As explained in the algorithm 3.2, the analysis is called recursively on the different variables of the use-def chain. For the example in listing 3.4, we have the following recursion steps:

- **1st recursion:** The initial ATV is the $stack12 in the assertion on line 28 of listing 3.4. When searching for the initialization, we find line 25 and 20, but our analysis tells us that those are results of a condition (due to the `label1` before the line 25) block, so we search further to find the `goto label1` on line 18 and the corresponding if statement.

- **2nd recursion:** When looking at the condition, we have a binary operator, so we have a branch split. The first branch will follow the variable `numberOfCones` and the other one the value `8000000`. But that last one will be stopped directly since it is a direct primitive assignment, so only one branch prevails. The analysis then looks for the initialization of the variable `numberOfCones` that is found on line 17. And since there is not any other occurrence of this variable, this step ends here.

- **3rd recursion:** Here the analysis finds another binary operator: the addition. But this time, none of the variables is a primitive, so there are two search branches created. Since in this example, both branches are identical (except the variable name), we will explain them together, basing ourselves on `$stack8` and only add the information for `$stack11` in parentheses. We then look for the initialization and we find it on line 11 (and 15). But there is no other occurrence of this variable and the method used for the initialization `getConeCells()` is an inspector, there is still no F-MUT found and this step ends.

- **4th recursion:** The only variable used in the previous initialization is `$stack7` (and `$stack10`), so we simply continue with this one. Its initialization is on the next line, line 10 (and 14) and there is no other occurrence.

- **5th recursion:** Here we find a unary operator in the form of a cast. So we search for the initialization of the casted variable `$stack6` and (`$stack9`) found on line 9 (and 13). The method `get(int)` used here is a Java library method so it is not the F-MUT and the step ends there.

- **6th recursion:** In this method, there are two values used: the variable `eyes` and a primitive integer, so, again, there is only one track to follow. The variable `eyes` is initialized on line 7 and all the occurrences of this variables are inspectors: the two instances of the method `get(int)` (on lines 9 and 13) and the initialization method `getEyes()` on line 7 too.

- **7th recursion:** Again, we follow the only variable in the last initialization, `$stack4` and end up on line 2 with its initialization. Here we have an object initialization of the project class `Person` which in Soot, is done on two lines. First, the variable is created (line 2) and then the object is initialized (on line 3). This would be a candidate to be the F-MUT. But we see one other occurrence of the variable `$stack4` that comes after the first candidate, the `setAverageConeCells(int)` method on line 5. And it also has a focal character. Therefore this method is returned as F-MUT (for both branches). When the two branches merge again, the tool will return only 1 F-MUT since both branches returned the same.

3.2.4 Name generation

We have the following information: F-MUT(s), AAE(s), assertion(s), oracle part(s). For each of them, if there are more than one, the naming would become difficult. The tool generates one name component for each F-MUT and also one for each assertion. Then it uses those components to synthesize a final name. For each one of the following steps, some exceptions also lead to bad and misguiding names. They are listed and explained in section 6.2.4 of the appendix.

3.2.4.1 Name with F-MUT

Just by using the F-MUT, the tool generates a name component that corresponds to the name of the method. If we want to create a test name solely based on this component, we will end up with test[*F-MUT*]. This is not optimal since it does not fully describe the test, and a tested method has most of the times different behaviours that need to be tested, hence different test cases. And all of these cases would end up with the same name (which is impossible or would just end with test[*F-MUT*][someNumber], which is bad naming and will not resolve our main goal. Hence this naming convention is only used as a backup if there is too much information or not enough.

3.2.4.2 Name with assertion

In this section, each assertion is parsed into an English like name, similar to the approach by by Gonzalez [34] who proposed a method to interpret assertions as English sentences.

- **assertTrue:** [*oracle*]IsTrue
- **assertFalse:** [*oracle*]IsFalse
- **assertNull:** [*oracle*]ReturnIsNull
- **assertNotNull:** [*oracle*]ReturnIsNotNull
- **assertEquals** & **assertArrayEquals:** [*oracle1*]Equals[*oracle2*]
- **assertNotEquals:** [*oracle1*]DifferentThan[*oracle2*]
- **assertSame:** [*oracle1*]IsSameAs[*oracle2*]
- **assertNotSame:** [*oracle1*]IsNotSameAs[*oracle2*]

The *oracle* part(s) depends on the method that returns the asserted value (this method will often be an inspector but can also be the F-MUT) or the constant value used in the assertion. In the case of the method, we simply use its name but in the case of a constant, there are different possibilities: (1) the value could be replaced by its type. This is the easiest way but the least descriptive one. (2) if present, the name of the variable is used to generate the name component. This can be the best method but is too dependent on useful variable names and their existence and hence not ideal for real world purposes. (3) the actual value is used. This gives a lot of information and is almost always accessible. But here we have a problem with special characters (since only `$` and _ are allowed in Java method names) and with long strings.

We decided to use the third approach and replace the special characters with an identifying string: + becomes *Plus*, `.` becomes *Dot*, / becomes *Slash*, ... The only problem that remains is the possible size of these name components that make the resulting name so big that it is no longer easy to read.

3.2.4.3 Combine and choose the name

During the whole process, the name components generated with the F-MUT(s) are separated from the components generated with the assertions. First of all, the different components are classified by number of occurrences, which also indicates their relevance, and in some cases, is even added to the name. After this step, duplicates are removed and finally, with the remaining components, we create a final name:

- if there is only one F-MUT based component and no assertion based component, the F-MUT based component is returned. This only happens if there are too many assertions that would overfill the name and therefore we remove them.

```
Email email = new Email("test@mail.com","Test mail");
assertEquals("test@mail.com",email.getReceiver());
assertEquals("Test Mail",email.getTitle());
assertNull(email.getCC());
assertNull(email.getAttachment());
assertEquals("",email.getText());
```

  In this case, we have one F-MUT based component **Initialization** and five assertion based components, **GetReceiverEqualsTestAtMailDotCom**, **GetTitleEqualsTestMail**, **GetCCReturnIsNull**, **GetAttachmentReturnIsNull** and **GetTextEqualsEmptyString**. Since there is too much information in five assertion based components and we can not sort them by importance, we will ignore them, so only the F-MUT based component remains and we get the name: `"testInitialization"`.

- if there is only one assertion based component and no F-MUT based components, the assertion based component is returned. This happens if the F-MUT also appears in the assertion based component or there was not any F-MUT found.

```
Email email = new Email("test@mail.com","Test mail");
assertTrue(email.send());
```

In this case, we have one F-MUT based component **Send** and one assertion based components, **SendIsTrue**. Since the F-MUT based component is included inside the assertion based component, we ignore the F-MUT based component and we get the name `"testSendIsTrue"`.

- if there is exactly one assertion based component and one F-MUT based component, they are combined as ”test[*Assertion*]After[*FMUT*]” and the result is returned.

```
Email email = new Email("test@mail.com","Test mail");
assertEquals("Test Mail",email.getTitle());
```

In this case, we have one F-MUT based component **Initialization** and one assertion based components, **GetTitleEqualsTestMail**. We therefore get the name `"testGetTitleEqualsTestMailAfterInitialization"`.

- if there is exactly one F-MUT based component but multiple assertion based components, they are combined as “test[*Assertion1*]And[*Assertion2*]And...After[*F-MUT*]” and the result is returned. This might occur if multiple assertions have the same F-MUT.

```
Email email = new Email("test@mail.com","Test mail");
assertEquals("test@mail.com",email.getReceiver());
assertEquals("Test Mail",email.getTitle());
```

In this case, we have one F-MUT based component **Initialization** and two assertion based components, **GetReceiverEqualsTestAtMailDotCom** and **GetTitleEqualsTestMail**. We concatenate the different assertion based components and then add the F-MUT based component to get the final result `"testGetReceiverEqualsTestAtMailDotComAndGetTitleEqualsTestMailAfterInitialization"`.

- if the number of F-MUT based components and assertion based components is the same, we check if each F-MUT based component has its corresponding assertion based component, then we combine them and return the result as following: “test[*Assertion1*]After[*Method1*]And[*Assertion2*]After[*Method2*]And...”

```
Email email = new Email("test@mail.com","Test mail");
assertEquals("Test Mail",email.getTitle());
email.send();
assertTrue(email.wasSent());
```

In this case, we have two F-MUT based component **Initialization** and **Send**, and two assertion based components, **GetTitleEqualsTestMail** and **WasSentIsTrue**. We then associate each assertion based component to its F-MUT based component and get the final name `"testGetTitleEqualsTestMailAfterInitializationAndWasSentIsTrueAfterSend"`.

- if there are multiple F-MUT based components and there is not a matching amount of assertion based components, only the F-MUT based components are combined and the assertion based components are ignored since there is not a general method to combine them. The returned result will look like:
  "test[*Method1*]And[*Method2*]And...".

```java
Email email = new Email("test@mail.com","Test mail");
assertEquals("test@mail.com",email.getReceiver());
assertEquals("Test Mail",email.getTitle());
email.send();
assertTrue(email.wasSent());
```

  In this case, we have two F-MUT based component **Initialization** and **Send**, and three assertion based components, **GetReceiverEqualsTestAtMailDotCom**, **GetTitleEqualsTestMail** and **WasSentIsTrue**. The number of F-MUT based components and assertion based components does not correspond, so we ignore them and get the final name `"testInitializationAndSend"`.

If possible, the names are shortened by using common naming components. It is important to note that common components are not based on single words but components (e.g. "IsTrue", "Equals", method name,...). Here we have two cases:

- if the different assertion based components have a common prefix, join them with the common prefix as follows:
  "[*Prefix*][*Component1*]Then[*Component2*]Then..."

```java
Email email = new Email("test@mail.com","Test mail");
assertFalse(email.wasSent());
email.send();
assertTrue(email.wasSent());
```

  In this case, we have two F-MUT based component **Initialization** and **Send**, and two assertion based components, **WasSentIsFalse** and **WasSentIsTrue**. In both assertion based components, we have the same prefix block **EmailWasSent**. So we can combine them and then get the final name `"testEmailWasSentIsFalse-AfterInitializationAndIsTrueAfterSend"`.

- if the different assertion based components have a common suffix, join them with the common suffix as following:
  "[*Component1*]And[*Component2*]And...[*Suffix*]"

```java
Email email = new Email("test@mail.com","Test mail","Test mail");
assertEquals("Test Mail",email.getTitle());
assertEquals("Test Mail",email.getText());
```

  In this case, we have one F-MUT based component **Initialization** and two assertion based components, **GetTitleEqualsTestMail** and **WasSentIsTrue**. Both assertion based components have the same suffix **TestMail**, so we combine them and get the name `"testGetTitleAndGetTextEqualsTestMailAfterInitialization"`.

4 Results

We analysed five different open source projects. First, we used the test cases of the project **net.sourceforge.barbecue** [1], with 142 test cases, to create a prototype of the tool. We chose this specific project since the tool presented by Zhang *et al.* [12] also featured this project. By using this project, we can then easily compare the results of our approach to the one taken by Zhang *et al.*. Then, the tests of **net.sourceforge.htmlparser** [2] were used to improve the tool. We only used the first 171 test cases since the tool already became rather robust and we saw the necessity to evaluate the tool on a wider base.

We then took three projects to evaluate the tool: **com.tagtraum.perf.gcviewer** [3], **uk.ac.ebi.ena.sra.cram** [4] and **org.terrier** [5]. Based on these results, the tool was further improved. The only real criteria for selecting these projects was that they had to contain JUnit4 test cases and that they were easy to import into eclipse. For these three projects, we only took 50 random test cases. This ensured that we used a wider range of projects and therefore a wider range of test cases. Also, a manual analysis takes time, so we restricted ourselves to these 50 test cases.

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

[2]http://barbecue.sourceforge.net

[3]https://github.com/chewiebug/GCViewer/wiki

[4]https://www.ebi.ac.uk/ena/software/cram-toolkit

[5]https://github.com/terrier-org/terrier-core/tree/4.4

4.1 Focal methods

The focal method under test (F-MUT) is the critical part of a test around which the whole test is constructed. Therefore it is important to be able to identify it correctly. In this section, we will discuss the success of our tool in finding the F-MUT(s).

4.1.1 Evaluation

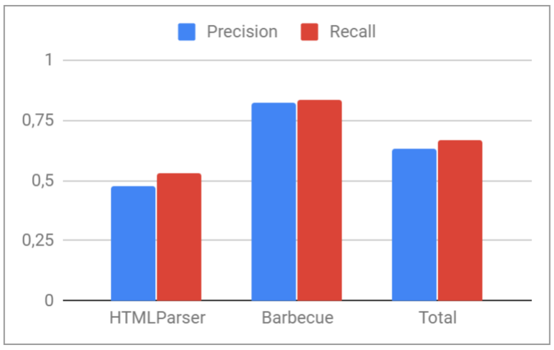
In total, there were 461 unit tests that were analysed and verified over all five projects. All of these tests were subject to a manual analysis to find the intended F-MUT(s). The results of the manual analysis were then compared to the output of the tool. Here we encountered an unforeseen problem. The intended F-MUT found by the manual analysis, for which the test was written, did not always match the actual F-MUT found by the tool. As illustration, the example in listing 4.1, shows a test, written for `write_int_LSB_0()` on line 5 but `flush()`, on line 6, is the real F-MUT. `flush()` has focal character because it alters the `baos` field of `bos`, but its only purpose is to clean the `DefaultBitOutputStream` for later uses in the same test class. In such a case, the intended F-MUT cannot be detected since the real one has all the focal characteristics that the intended one possesses hence leading to a lower precision and recall.

```
 1  @Test public void test_write_int_32bits() throws IOException {
 2    ByteArrayOutputStream baos = new ByteArrayOutputStream();
 3    DefaultBitOutputStream bos = new DefaultBitOutputStream(baos);
 4
 5    bos.write_int_LSB_0(-1073741823, 32);
 6    bos.flush();
 7
 8    byte[] buf = baos.toByteArray();
 9    assertThat(buf.length, is(4));
10  }
```

Listing 4.1: Excerpt from class *uk.ac.ebi.ena.sra.cram.io.BitOutputStreamTest*

4.1.2 Results

When comparing the resulting F-MUTs with the intended ones found by the manual analysis, we have a precision of 63.3% and a recall of 67.1% for the development projects and a precision of 77.4% and a recall of 83.8% for the test projects. Over all test cases, this results in a precision of 67.7% and a recall of 72.5%.



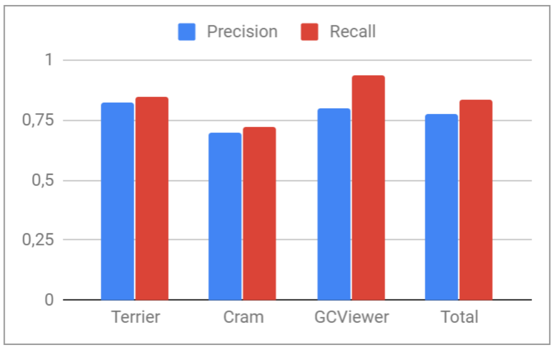


Figure 4.1: The precision and recall of the tool in finding the correct F-MUT(s). **Left:** development projects, **Right:** verification projects.

The fact that the test projects end up better than the development projects might seem contradictory, but in figure 4.1, we can see that the HTMLParser project has a 20% lower precision and recall than the other projects. This is because some of the tests have their setup and execution part inside a method invoked by the test and this method does not have any direct connection to the assertion (no common used variables). Recognizing these kind of helper methods is not implemented, otherwise, we would apply our analysis recursively on these methods. One of these examples is shown in the listing 4.2 where the `getParameterTableFor()` method contains the setup and the actual F-MUT. For this reason, we cannot identify the F-MUT(s) and therefore precision and recall are both 0%. Since this case appears 47 times in this project and only two times in all the four others combined, the precision and recall of the project are lower than for the others. If only observing the Barbecue project, the precision goes up to 82.4% and the recall to 83.8% for the development projects.

```java
public void testParseParameters() {
  getParameterTableFor("a b = \"c\"");
  assertEquals("Value","c",((Attribute)(attributes.elementAt (2))).getValue ());
}
```

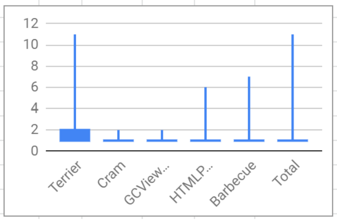
Listing 4.2: Excerpt from class *org.htmlparser.tests.lexerTests.AttributeTests*

Over all projects, recall is mostly lost because of the mutator analysis failing, leading to the real F-MUT being ignored since its focal character could not be detected. This can have different reasons. The most common ones are if the creation of the deep clone failed or the attached ByteBuddy agent leads the test to crash. On the other hand, precision was lost by detecting not the intended F-MUT(s), for which the test was created, but the real F-MUT(s) based on our definition.

Finally, the performance of the tool, when the inspector and mutator analysis succeed, is at a precision of 91.8% and a recall of 92.6%. So, when knowing mutators and inspectors, our approach yields very good results.

One other finding in relation with the F-MUT(s), that can be seen in figure 4.2, is that a majority of test cases have one F-MUT: the median is 1 and even the upper quartile is 1. The average number of F-MUTs is also at 1.264 F-MUTs/test.

	Min	Q1	Q3	Max	Avg	Med
Terrier	1	1	2	11	1,723	1
Cram	1	1	1	2	1,146	1
GCViewer	1	1	1	2	1,061	1
HTMLParser	1	1	1	6	1,263	1
Barbecue	1	1	1	7	1,221	1
Total	1	1	1	11	1,264	1



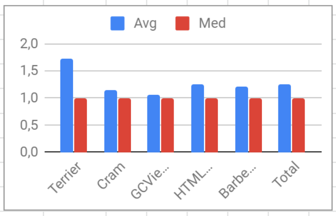


Figure 4.2: The average and median number of F-MUTs per test case

In figure 4.3, we see the same result but in other numbers. When we observe all F-MUTs, 85.536% of all tests have only one focal method, 9.476% have two F-MUTs and only the remaining 4.987% have more than two F-MUTs. If we look at distinct F-MUTs (if there are two scenarios with the same F-MUT used multiple times, it only counts as one F-MUT), the results are even more extreme: 93.267% have one distinct F-MUT, 5.985% have two and only 0.748% (in total three cases) have more than three distinct F-MUTs.

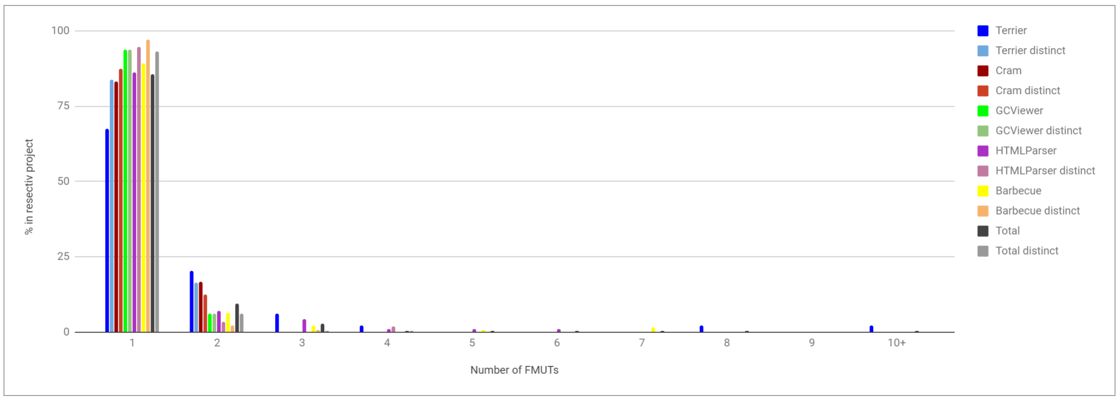


Figure 4.3: Number of F-MUTs per test case (apparition and distinct)

4.2 Separate parts

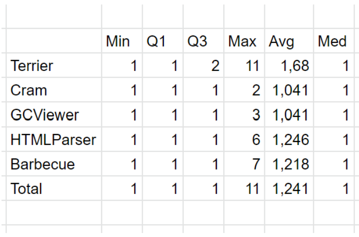
The notion of scenarios is important. A scenario is the combination of setup, execution and oracle part which tests a specific action or unit. Not all the tests were composed of one scenario but there were also multi-scenario tests where we had to differentiate between the different sub-scenarios. In this part, we will therefore study these scenarios, their relation with focal methods and assertions, and see if they can and should be simplified.

4.2.1 Evaluation

One big question was if the information about the focal method under test helps to identify the different sections of a test case (setup, execution and oracle). In these cases, we used the intended F-MUTs found through the manual search. Once we had the F-MUT, we inspected the body of the test to find all three parts by starting with the F-MUT. For this purpose, we looked for the number of F-MUTs, scenarios and assertions in each test.

4.2.2 Findings

The results of figure 4.4, showing the number of scenarios per test, are almost similar to the results obtained in figure 4.2 for the number of F-MUTs per test. We have again a median and an upper quartile of 1 and an average of 1.241 scenarios/test (compared to 1.264 F-MUTs/test).



	Min	Q1	Q3	Max	Avg	Med
Terrier	1	1	2	11	1,68	1
Cram	1	1	1	2	1,041	1
GCViewer	1	1	1	3	1,041	1
HTMLParser	1	1	1	6	1,246	1
Barbecue	1	1	1	7	1,218	1
Total	1	1	1	11	1,241	1

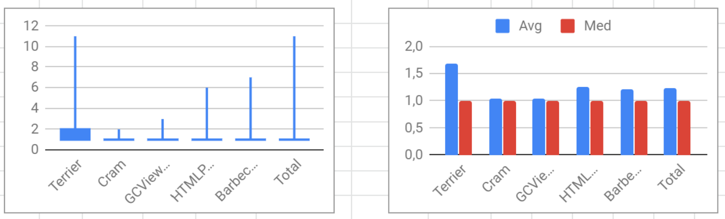


Figure 4.4: The average and median number of scenarios per test case

Figure 4.5 shows the same thing again: 88.286% of tests have one scenario, 6.291% of tests have two scenarios, 3.036% of tests have three scenarios and only the last 1% has more than three scenarios.

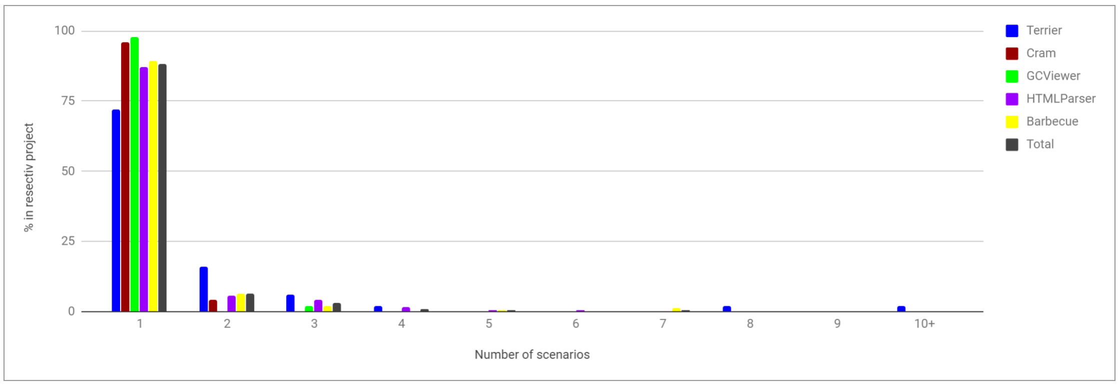
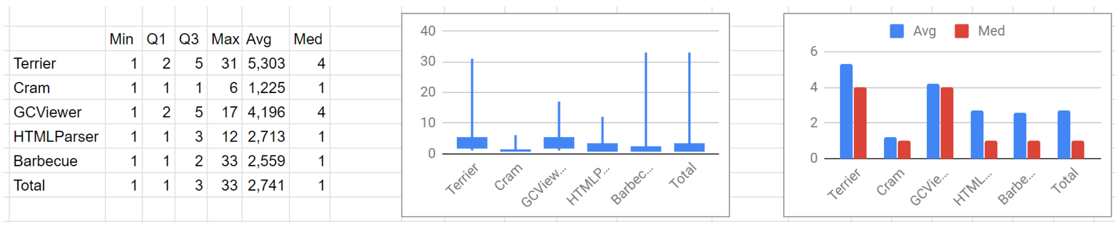


Figure 4.5: Number of scenarios per test case

Finally, when we observe the number of assertions in all the tests with one F-MUT, we can see a little difference with a total average of 2.741 assertions per test but we still have a median of 1 assertion per test. The high average, compared to the median, is due to the big maximum values with tests at 31 to 33 assertions.



	Min	Q1	Q3	Max	Avg	Med
Terrier	1	2	5	31	5,303	4
Cram	1	1	1	6	1,225	1
GCViewer	1	2	5	17	4,196	4
HTMLParser	1	1	3	12	2,713	1
Barbecue	1	1	2	33	2,559	1
Total	1	1	3	33	2,741	1

Figure 4.6: The average and median number of assertions per test case

In total, 52.88% of the tests with one F-MUT also have one assertion but we also observe that the number of assertions per test is not as clearly defined as the number of F-MUTs and scenarios. By looking at the test cases in Terrier and Cram we even see that there are more tests with two assertions than tests with one assertion, though in total we still observe that we have more tests with few assertions.

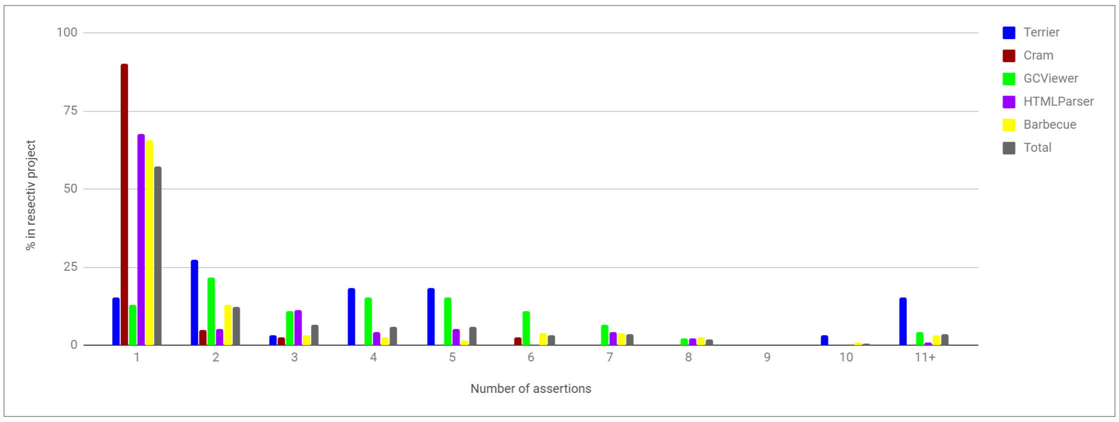
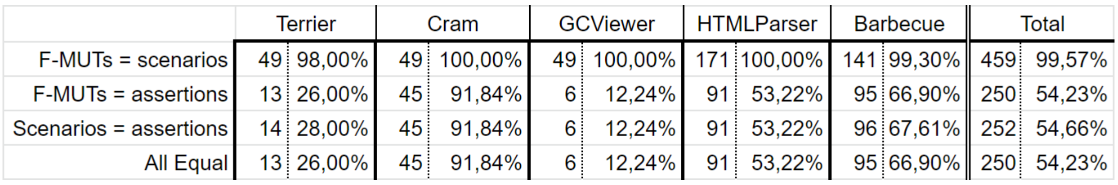


Figure 4.7: Number of assertions per test case with one F-MUT

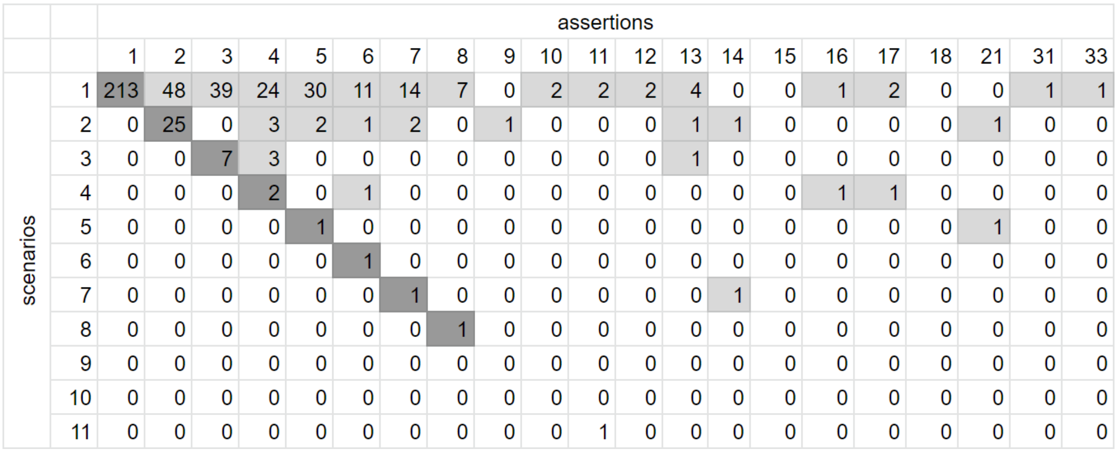
These numbers lead us to make correlations between the number of F-MUTs, assertions and scenarios and we expected to see that there should be a majority of cases where we have as many F-MUTs as scenarios and ideally as assertions. For each scenario, there should be one F-MUT and one or more assertions. It is exactly this behaviour that we observe in figure 4.8 below. There, we have the number of tests that have as many F-MUTs as scenarios, F-MUTs as assertions and scenarios as assertions. Also, in the last row, we see the number of tests where the number of F-MUTs is equal to the number of scenarios **and** the number of assertions. In 99.57%, we have as much scenarios as F-MUTs and in 54.23% even the same amount of assertions.



	Terrier		Cram		GCViewer		HTMLParser		Barbecue		Total	
F-MUTs = scenarios	49	98,00%	49	100,00%	49	100,00%	171	100,00%	141	99,30%	459	99,57%
F-MUTs = assertions	13	26,00%	45	91,84%	6	12,24%	91	53,22%	95	66,90%	250	54,23%
Scenarios = assertions	14	28,00%	45	91,84%	6	12,24%	91	53,22%	96	67,61%	252	54,66%
All Equal	13	26,00%	45	91,84%	6	12,24%	91	53,22%	95	66,90%	250	54,23%

Figure 4.8: Are the number of F-MUTs, scenarios and assertions equal?

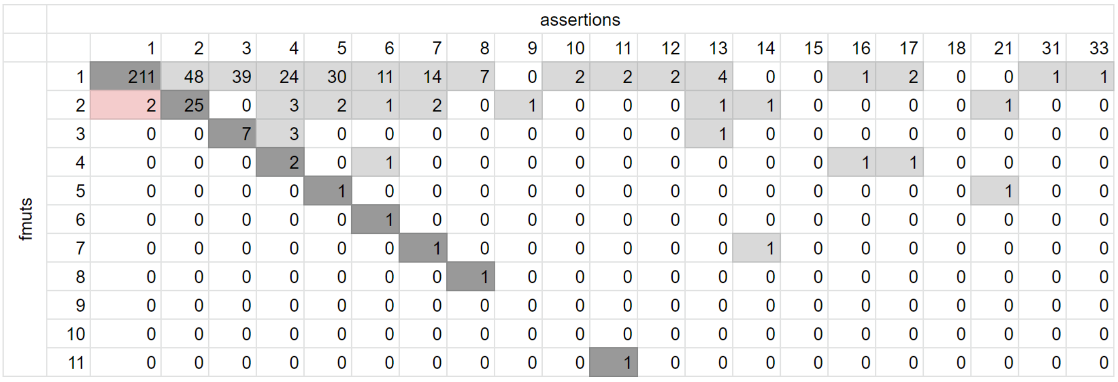
In figure 4.9, we see that there are never more sub-scenarios than assertions which is obvious since a scenario must have an assertion per definition. Otherwise, we do not see a clear relationship between them.



		assertions																				
		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	21	31	33
scenarios	1	213	48	39	24	30	11	14	7	0	2	2	2	4	0	0	1	2	0	0	1	1
	2	0	25	0	3	2	1	2	0	1	0	0	0	1	1	0	0	0	0	1	0	0
	3	0	0	7	3	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0
	4	0	0	0	2	0	1	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0
	5	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0
	6	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	7	0	0	0	0	0	0	1	0	0	0	0	0	0	1	0	0	0	0	0	0	0
	8	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0
	9	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	10	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	11	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0

Figure 4.9: Table that shows the relationship between the number of scenarios and the number of assertions.

In figure 4.10, we see the same phenomenon with as many or more assertions than F-MUTs. There are two exceptions here since an assertion can, in rare cases, have more than one actual asserted expression, which leads to a greater number of F-MUTs. Below, in listing 4.3 we have one of these exceptions where one single assertion leads to two F-MUTs (both `hashCode()` methods) due to the binary operator on line 4.



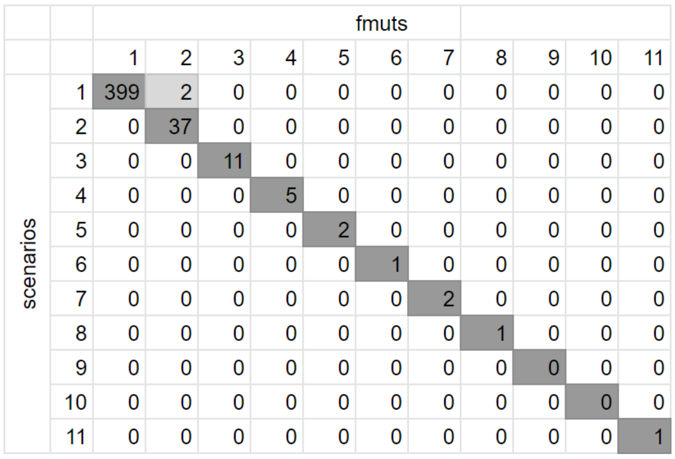
		assertions																				
		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	21	31	33
fmuts	1	211	48	39	24	30	11	14	7	0	2	2	2	4	0	0	1	2	0	0	1	1
	2	2	25	0	3	2	1	2	0	1	0	0	0	1	1	0	0	0	0	1	0	0
	3	0	0	7	3	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0
	4	0	0	0	2	0	1	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0
	5	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0
	6	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	7	0	0	0	0	0	0	1	0	0	0	0	0	0	1	0	0	0	0	0	0	0
	8	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0
	9	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	10	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	11	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0

Figure 4.10: Table that shows the relationship between the number of focal methods and the number of assertions.

```
1 @Test public void testHashCodesAreNotEqualsIfNotEquals() throws Exception {
2   Module mod = new Module(new int[] {2, 2, 2, 1, 2, 4});
3   Module mod2 = new Module(new int[] {2, 2, 2, 1, 4, 2});
4   assertFalse(mod.hashCode() == mod2.hashCode());
5 }
```

Listing 4.3: Excerpt from class *net.sourceforge.barbecue.ModuleTest*

Finally, in figure 4.11, we see a very clear relation. For almost all cases, the number of F-MUTs and the number of sub-scenarios are equal. The only two exceptions are the same as seen in the previous paragraph. If the actual asserted expression splits and depends on two separate F-MUTs, the number of F-MUTs will be greater than the number of scenarios.



		fmuts										
		1	2	3	4	5	6	7	8	9	10	11
scenarios	1	399	2	0	0	0	0	0	0	0	0	0
	2	0	37	0	0	0	0	0	0	0	0	0
	3	0	0	11	0	0	0	0	0	0	0	0
	4	0	0	0	5	0	0	0	0	0	0	0
	5	0	0	0	0	2	0	0	0	0	0	0
	6	0	0	0	0	0	1	0	0	0	0	0
	7	0	0	0	0	0	0	2	0	0	0	0
	8	0	0	0	0	0	0	0	1	0	0	0
	9	0	0	0	0	0	0	0	0	0	0	0
	10	0	0	0	0	0	0	0	0	0	0	0
	11	0	0	0	0	0	0	0	0	0	0	1

Figure 4.11: Table that shows the relationship between the number of focal methods and the number of scenarios.

We have seen that there exist tests with more than one scenario. This raises the question on how these tests could be improved. For the following part, three aspects have to be considered. Either the test is realistic, theoretically ideal or useful for the analysis. In theory, a unit test should only test one aspect and hence contain one assertion and one F-MUT and therefore one scenario. But such tests are not realistic. Now the question arises if multiple scenarios should be separated at any cost, simplified or left unchanged.

In total, we had 60 (13.02%) of the tests with multiple scenarios and half of them could have been easily broken down to multiple single scenario tests. Most of the tests where the scenarios are somewhat more difficult to separate are because the first scenario entirely depends on the second one. But even there we can separate them by putting the whole setup and execution part of the fist scenario into the test of the second scenario and only omit the oracle part.

Below we have a simple example of a `Login` class that has the method `login()` which returns false if the login and the password do not match the expected value and throws a `ToMuchTriesException` if the password is false for the third time in a row. To test this behaviour, we have the following three test cases. Here they are as they should be in theory: one behaviour, one assertion, one scenario.

In this second example, there is one single test with the same intent and the same result. In this case, the first and the second tests are included inside the third one by asserting each instance of `login()`. This does not change the result of the test since it

```java
@Test public void testLoginOnceWithFalsePasswordReturnsFalse() throws
    TooManyTriesException {
  Login login = new Login();
  assertFalse(login.login("username", "falsePassword"));
}

@Test public void testLoginTwiceWithFalsePasswordReturnsFalse() throws
    TooManyTriesException {
  Login login = new Login();
  login.login("username", "falsePassword");
  assertFalse(login.login("username", "falsePassword"));
}

@Test public void testLoginThreeTimesWithFalsePasswordThrowsTooManyTriesException()
    throws TooManyTriesException {
  Login login = new Login();
  login.login("username", "falsePassword");
  login.login("username", "falsePassword");
  try {
    login.login("username", "falsePassword");
    fail("Should throw TooManyTriesException");
  } catch (TooManyTriesException e) {}
}
```

Listing 4.4: Example of theoretically correct test

fails if one of the three `login()` calls would return true. But they added scenarios to the test case that are not necessary and transformed the test case into a multi-scenario test.

```java
@Test public void
    testLoginThreeTimesWithFalsePasswordReturnsFalseFalseThrowsTooManyTriesException()
    throws TooManyTriesException {
  Login login = new Login();
  assertFalse(login.login("username", "falsePassword"));
  assertFalse(login.login("username", "falsePassword"));
  try {
    login.login("username", "falsePassword");
    fail("Should throw TooManyTriesException");
  } catch (TooManyTriesException e) {}
}
```

Listing 4.5: Example of same tests with multiple scenarios

Another question that was raised is around one-line-test collections: tests that have setup, F-MUT and assertion on the same line and appear multiple times with different inputs. We found both; either they were contained in one test case or each line represents a different scenario. In this first example, we have such one-line-tests, each in their dedicated test case. We only show the first three but there are 41 of them.

In the second example, we also have these one-line-tests but now merged into one single big test which covers all the cases. Although this kind of test seems easier to write, especially for the naming, where we only have to decide and write one single test name,

```java
public void test1 () throws ParserException {
  assertEquals ("test1 failed", "https:h", mPage.getAbsoluteURL ("https:h"));
}
public void test2 () throws ParserException {
  assertEquals ("test2 failed", "http://a/b/c/g", mPage.getAbsoluteURL ("g"));
}
public void test3 () throws ParserException {
  assertEquals ("test3 failed", "http://a/b/c/g", mPage.getAbsoluteURL ("./g"));
}
```

Listing 4.6: Excerpt from class *org.htmlparser.tests.lexerTests.PageTests*

it is less practical to debug since only one error can be thrown at a time.

```java
@Test public void testParseCommaDelimitedInt() throws Exception {
  assertTrue(Arrays.equals(new int[0], ArrayUtils.parseCommaDelimitedInts("")));
  assertTrue(Arrays.equals(new int[0], ArrayUtils.parseCommaDelimitedInts(" ")));
  assertTrue(Arrays.equals(new int[]{1}, ArrayUtils.parseCommaDelimitedInts("1")));
  assertTrue(Arrays.equals(new int[]{1}, ArrayUtils.parseCommaDelimitedInts(" 1")));
  assertTrue(Arrays.equals(new int[]{1}, ArrayUtils.parseCommaDelimitedInts("1 ")));
  assertTrue(Arrays.equals(new int[]{1}, ArrayUtils.parseCommaDelimitedInts(" 1 ")));
  assertTrue(Arrays.equals(new int[]{1,2},
      ArrayUtils.parseCommaDelimitedInts("1,2")));
  assertTrue(Arrays.equals(new int[]{1,2}, ArrayUtils.parseCommaDelimitedInts("1
      ,2")));
  assertTrue(Arrays.equals(new int[]{1,2}, ArrayUtils.parseCommaDelimitedInts("1,
      2")));
  assertTrue(Arrays.equals(new int[]{1,2}, ArrayUtils.parseCommaDelimitedInts("1 ,
      2")));
}
```

Listing 4.7: Excerpt from class *org.terrier.utility.TestArrayUtils*

So in theory, test cases should be broken down to only one scenario[17], and if possible with one assertion. It helps the debugging process, the understanding of the test and simplifies the process of finding a matching name. This is also observed by Kuhn [35] who found out that such tests can and should be broken down into cascaded tests (with the output of one test being the input/setup of another). But it is far more time-consuming and therefore not as widespread as its simpler version.

4.3 Naming

The goals for test names consists of three important points that these names have to consider, as seen in chapter 3:

- R1 Test names should have a clear relation to the code under test (the F-MUT); they should allow developers to identify the tests concerning this method without having to inspect the test code.

- R2 Test names should be descriptive of the test code; there should be an understandable, intuitive relation between the test code and its name, meaning that the scenario of the test (setup, execution and oracle part) should be recognized by reading the name.
- R3 Test names should uniquely distinguish tests within a test suite, such that developers can use them to navigate the test suite.

For the last requirement, it is important to note that it could always be achieved by adding a number behind the name. But this should, if possible, be avoided because of the points described in the first two requirements: the names should be descriptive and help the reader to understand the test by the name alone, and if the only difference is a number, this purpose will not be fulfilled.

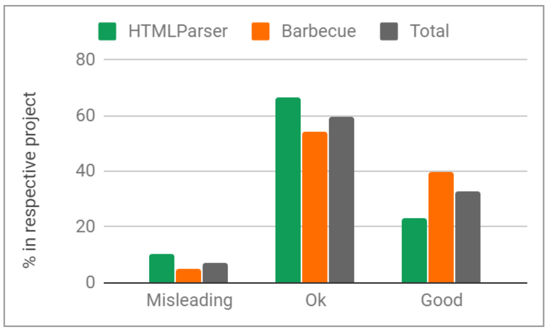
4.3.1 Evaluation

We then compared the computed name with the test body to see if the behaviour of the method is reflected in the name. If the name matched all three criteria, it was labelled as a good name. To be qualified as an acceptable name, the test has to at least satisfy the first requirement by containing the right F-MUT information and not add other, erroneous, information. This is part of the second requirement. The third requirement is less important since it can be easily fixed by adding an incrementing number behind the name. It is not desired to do so, but the names might still be enough descriptive to be considered as acceptable.

Finally, there where two other papers that contributed to the topic of automated unit test name generation: Zhang *et al.* propose a natural language program analysis approach [12] restricted to single assertion tests whereas Daka *et al.* proposed a plug-in for an automated test generation tool [17] that cannot generate names for existing test cases. So we compared the results of their specialized tools to the generated names of our general tool.

4.3.2 Findings

For the cases where we correctly identified the F-MUT, we have 61.6% of OK names, 31.4% of good names and only 6.9% of misleading ones, meaning that they describe a false behaviour. Even though this seems to be a very good number of acceptable names, we should not forget that this only applies to the cases where we correctly identified the F-MUT(s). Over all test cases we still have around 25% of misleading names. And since many of the misleading names are due to the difference between actual and intended F-MUTs, even improving the tool will not be sufficient. What we need is to improve the rules of how F-MUTs are defined, but there will never be a complete one, because we have a high diversity of test cases. What can be improved is the generation of the names by improving the readability and hence the fraction of good names by improving the acceptable ones.



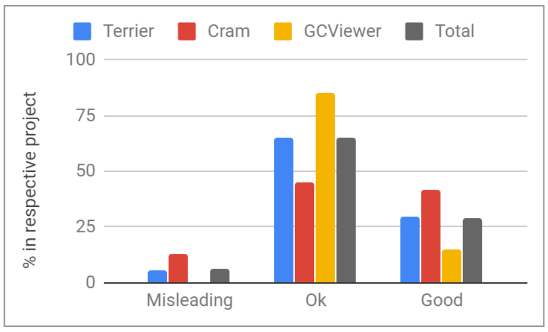


Figure 4.12: Fraction of misleading, OK and good test name recommendations per project. **Left:** development projects, **Right:** verification projects.

4.3.2.1 Improving acceptable names

One of these aspects that can be improved is to reduce the number of too complex and difficult to read names. In many cases, this is due to the way we handle the special characters (as plus, question mark, dots,...) from strings or primitives inside the generated name. We replaced each of these characters by a descriptive string. And almost half of the acceptable names are not rated as good names for this reason. Below we have one example where this convention has led to the following test names:

```
"testGetAbsoluteURLEqualsHttpColonSlashSlashaSlashbSlashcSlashgDot"
"testGetAbsoluteURLEqualsHttpColonSlashSlashaSlashbSlashcSlashDotg"
"testGetAbsoluteURLEqualsHttpColonSlashSlashaSlashbSlashcSlashgDotDot"
"testGetAbsoluteURLEqualsHttpColonSlashSlashaSlashbSlashcSlashDotDotg"
```

The names are correct and also differentiate the different cases, at least for the computer, but are difficult to read. We could also just ignore special characters or replace the string with "*SomeString*" or in case of numbers, differentiate between whole and decimal numbers and positive and negative ones. But by doing this in the given example, all the tests would end up with the same name hence breaking the rule R2.

```
public void test27 () throws ParserException {
  assertEquals ("test27 failed", "http://a/b/c/g.", mPage.getAbsoluteURL ("g."));
}
public void test28 () throws ParserException {
  assertEquals ("test28 failed", "http://a/b/c/.g", mPage.getAbsoluteURL (".g"));
}
public void test29 () throws ParserException {
  assertEquals ("test29 failed", "http://a/b/c/g..", mPage.getAbsoluteURL ("g.."));
}
public void test30 () throws ParserException {
  assertEquals ("test30 failed", "http://a/b/c/..g", mPage.getAbsoluteURL ("..g"));
}
```

Listing 4.8: Excerpt from class *org.htmlparser.tests.lexerTests.PageTests*

On the other hand, around 25% of the acceptable names are due to such a simplification that was implemented. When there were too many assertions, the generated name became far too long and hence we decided to remove the information gained through these assertions leaving only the information gained through the F-MUT(s). And this has led to names that are somehow descriptive but appear multiple times in the same test class and therefore break the rule R2. What would be interesting in this part, is to be able to classify the assertions by importance, and in such cases, only take into account the information of the most important assertions.

4.3.2.2 Need for human interpretation

In many cases, we have unique names that contain information about the F-MUT and the oracle part, but they lack human interpretation and are therefore difficult to understand. In some of these cases, it would also be helpful to add information about the setup part, which is sometimes even more helpful than the oracle part.

In the example below, we have a bit reader that reads bits from left to right. The generated names are `"testReadBitsIs0"` and `"testReadBitsIs4"`. Those are not wrong, but they do not help one to understand the test cases without knowledge of the code. For example, the tool cannot recognize that the test is based on the byte representation of the number and will generate the names with the decimal formatted numbers and not their byte alternative. `"testReadBitsIs000"` and `"testReadBitsIs100"` would have been much clearer. The original names, as `"test_ReadBits_3_bits_of_00000100"` for the first case, are far more descriptive since they include the setup instead of the oracle part information. But even those names need some clarifications that are not provided. First, the expected result is not shown at all. Also, the `readBits()` method reads the bits from left to right. This is not necessary to be mentioned, but it adds a simple additional layer of understanding to the test name. Finally, the second test case even has a misleading original name since the byte that is processed is 1000000 and not 0100000. The best name for the second case would have been `"test_ReadBits_3_leftmost_bits_of_10000000_equals_100"`. But such naming needs human interpretation of the code and cannot be computed by this tool.

4.3.2.3 Our tool compared to others

In their work, **Daka** *et al.* [17] proposed a approach restricted to tests during creation. The core idea is similar, since it uses the informations of the method the test is created for (F-MUT) but the way to retrieve the results are different. Where we had to mine these information of the test body, they received these information automatically by the

```java
@Test public void test_ReadBits_3_bits_of_00000100() throws IOException {
  byte value = 4;
  byte[] buf = new byte[] { value };
  ByteArrayInputStream bais = new ByteArrayInputStream(buf);
  BitInputStream bis = new LongBufferBitInputStream(bais);
  int readBits = bis.readBits(3);

  assertThat(readBits, is(0));
}

@Test public void test_ReadBits_3_bits_of_01000000() throws IOException {
  byte value = (byte) (1 << 7);
  byte[] buf = new byte[] { value };
  ByteArrayInputStream bais = new ByteArrayInputStream(buf);
  BitInputStream bis = new LongBufferBitInputStream(bais);
  int readBits = bis.readBits(3);

  assertThat(readBits, is(4));
}
```

Listing 4.9: Excerpt from class *uk.ac.ebi.ena.sra.cram.io.LongBufferBitInputStreamTest*

test generation tool (since the method to be tested is know during the test generation procedure). The problem in comparing our approaches was that they created a plug-in for a coverage test generation, we cannot directly compare the results, so the only comparison is based on the examples they use in their paper, which is unfortunately only the three shown in the listing 4.10 below. For these examples, our tool generated the names (1) `"testGetTotalEquals0LAndAddPriceIsFalseAfterInitializationOf-ShoppingCart"`, (2) `"testAddPriceIsTrue"` and (3) `"testGetTotalEquals0LAfter-InitializationOfShoppingCart"` respectively. In comparison, their tool generated the names (1) `"testAddPriceReturningFalse"`, (2) `"addPriceReturningTrue"` and (3) `"testGetTotal"`. When looking at these names, we can see that our tool generates longer and more complete names which may seem verbose at first sight but contain much more useful pieces of information. Compared to the default generated names by the test generation tool there is a real improvement, but they are not enough since the naming convention might change depending on the names of other tests in the test class: when one test is named after the F-MUT, a second test, with the same F-MUT will be named after the expected output but without the F-MUT information. This could be improved, by combining these results.

```java
@Test public void testAddPriceReturningFalse() {
  ShoppingCart cart0 = new ShoppingCart();
  boolean boolean0 = cart0.addPrice(2298);
  assertEquals(0, cart0.getTotal());
  assertFalse(boolean0);
}

@Test public void testAddPriceReturningTrue() {
  ShoppingCart cart0 = new ShoppingCart();
  boolean boolean0 = cart0.addPrice(1);
  assertEquals(1, cart0.getTotal());
  assertTrue(boolean0);
}

@Test public void testGetTotal() {
  ShoppingCart cart0 = new ShoppingCart();
  int int0 = cart0.getTotal();
  assertEquals(0, int0);
}
```

Listing 4.10: Example classes used by Daka et al.

On the other hand, **Zhang** *et al.* [12] proposed a different approach. Instead of an inter-procedural approach, they used a natural language program analysis (NLPA) using a dependency graph and different rule-based procedures. They also implemented three different kinds of name levels that can all be generated with their tool. The generated names are quite promising when looking at the examples given in their paper.

For the first example they generate the names `"testDoGet"`, `"testDoGetResolution-Is72"`, `"testDoGetResolutionIs72WhenParamsResolutionIs72AndSettingParameters"` which do a perfect job for each of their categories. Our tool would have generated the name `"testGetResolutionEquals72"` if the variables in the assertion were not used in the wrong order (more information can be found in the appendix). This name is not convenient either since we have a getter method that has focal character (here again, refer to the appendix) with `getResolution()` on line 8. Otherwise, the generated name would be `"testGetResolutionEquals72AfterDoGet"` which is similar to the medium sized name of *NameAssist*, `"testDoGetResolutionIs72"`.

```
1  @Test public void testSettingResolutionChangesDefaultResolution() throws Exception {
2    servlet = new BarcodeServletMock();
3    params.put("height", "200");
4    params.put("width", "3");
5    params.put("resolution", "72");
6    req.setParameters(params);
7    servlet.doGet(req, res);
8    Barcode barcode = servlet.getBarcode();
9    assertEquals(barcode.getResolution(), 72);
10 }
```

Listing 4.11: Excerpt from class *net.sourceforge.barbecue.BarcodeServletTest*

In the second example they used, they generated the names `"testToString"`, `"test-ToStringExpressionIsSuper4"` and `"testToStringExpressionIsSuper4WhenSetting-SpecSuperLevelAndSettingSpecId"`. Here again, their names are qualified. In comparison, our tool generated `"testToStringEqualsSuperLParentheses4RParentheses"` as name. Here, we once again have a little difference due to the used approach. We see that in their tool, they removed all special characters to generate the name. In most of the cases, as in this example, this works perfectly and creates a more aesthetic name.

But when the special characters play an important role in the test, they should also be included in the name (in our case by replacing them with their respective String).

```java
public void testGetExpression_1() throws Throwable {
  NavigatorExpression navExpr = new NavigatorExpression();
  navExpr.setSpecId(NavigatorExpression.SPEC_SUPER_ID);
  navExpr.setSpecSuperLevel(4);
  String expression = navExpr.toString();
  assertEquals("super(4)", expression);
}
```

Listing 4.12: Excerpt from class *com.agiletec.aps.system.services.page.widget.TestNavigatorExpression*

In the third example, we illustrate one of the big flaws of the tool: it strongly depends on the already existing names (especially test class name and getter names) to find the CUT and F-MUT, but reality shows that these cannot be counted as given. For this purpose, we used three test cases that are doing the same thing, but were executed from different test classes and with different getter names:

1. Executes from a wrong test class and the getter does not contain the prefix “get”.
2. Executes from the correct test class and the getter does not contain the prefix “get”.
3. Executes from the correct test class and the getter contains the prefix “get”.

The only focal method in these tests is the `changeField()` method. The other used methods are simple getters. Our tool generates the same test name for all the three test cases: `"testChangedIsTrueAfterChangeField"` which correctly identifies the F-MUT. The tool of Zhang *et al.* on the other hand generates three different sets of names (with a small (S), a medium (M) and a large (L) name):

1. (S) `"testChanged"`, (M) `"testChangedIsTrue"`, (L) `"testChangedIsTrue"`
2. (S) `"testRetreiveField"`, (M) `"testRetreiveFieldChangedIsTrue"`, (L) `"test-RetreiveFieldChangedIsTrueWhenChangeField"`
3. (S) `"testChangeField"`, (M) `"testChangeFieldChangedIsTrue"`, (L) `"testChange-FieldChangedIsTrue"`

In the first case, NameAssist detected the class `Param` as the CUT because of the test name. This leads to the method `changed()` to be recognized as F-MUT which is false and hence the name is useless. Also, since the `param` object is not used apart from the line 6, there is not any setup detected either for the long names.

In the second case, the class name is correct, so the `changed()` method is ignored (defined in another class). Checking at the last method invocation of the object `obj`, which is now correctly identified as CUT, the `retreiveField()` method does not have the prefix “get”, so it is tagged as the tested method. And even though the real

F-MUT appears in the long name as a setup method, the name still does not reveal the purpose of the test.

Finally, in the last case, all is named correctly and the CUT and F-MUT are identified, leading to meaningful names which are almost similar to the generated name by our tool for all three of these test cases.

```
public class ParamTest {
  @Test public void testChangeField(){
    Obji obj = new Obji();
    obj.changeField();
    Param param = obj.retreiveField();
    boolean val = param.changed();
    assertTrue(val);
  }
}

public class ObjiTest {
  @Test public void testChangeField(){
    Obji obj = new Obji();
    obj.changeField();
    Param param = obj.retreiveField();
    boolean val = param.changed();
    assertTrue(val);
  }

  @Test public void testChangeField2(){
    Obji obj = new Obji();
    obj.changeField();
    Param param = obj.getField();
    boolean val = param.changed();
    assertTrue(val);
  }
}
```

Listing 4.13: Example methods

The same thing can be observed when changing some method names of the first example. We changed `getBarcode()` on line 8 to `receiveBarecode()` meaning that this method will not be recognized as a getter and hence becomes the F-MUT, and then changing `getResolution()` on line 9 to `getRes()` so that the link between lines 5 and 9 is not made (based on the String "resolution"). This then leads to the names `"testReceiveBarcode"`, `"testReceiveBarcodeResIs72"`, `"testReceiveBarcodeResIs72"` instead of `"testDoGet"`, `"testDoGetResolutionIs72"`, `"testDoGetResolutionIs72WhenParamsResolutionIs72AndSettingParameters"`.

The Tool NameAssist also came with a small dataset of 11 cases based on the Barbecue project. When comparing our results to the ones of this dataset NameAssist had 27% (3) of clearly more explicit names, 36% (4) of almost similar names but where NameAssist had a nicer generation method, 18% (2) where our tool created a slightly better name and 27% (3) where our tool created a better name. So only looking at these numbers, it seems that both tools are almost equal, but as already mentioned, NameAssist has a slightly better approach on how to assemble the pieces of information to generate aesthetic names. But even though the approach of Zhang *et al.* implemented in NameAssist gets more complete results, they only work if the code and test writers did the job to name the different parts correctly (like test class and getter methods), so that the test to code traceability and the mutator analysis can be skipped.

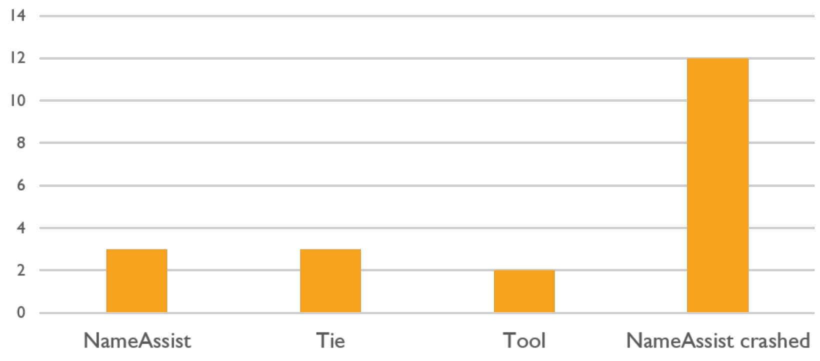


Figure 4.13: Comparison with NameAssist.

We also compared 20 single assertion test cases taken from all of the five projects, to adapt to the restrictions of *NameAssist*, to see which tool generates better names. As the graph above shows us, if *NameAssist* generates a name that works, it is most of the time preferred to or at least equal to the one generated with our tool. But one big difference we see is the stability difference of the tools. Whereas our tool always generated a name, including one misleading name, *NameAssist* generated two misleading names and crashed on 12 cases. Important to mention is that over half the cases were taken from the Barbecue project which was also used to develop *NameAssist*.

So, in their restrained fields, the two other tools yield slightly more descriptive results, but when taking into account that our tool is applicable to a much wider range of tests, these slight differences are acceptable.

4.4 Research questions

Now to answer the **RQ1** and see if the focal method under test information helps to identify the different sections of a test case (setup, execution and oracle), we first identified these focal methods manually and tried to understand the test, based on this information. The results depend on the number of scenarios inside the test. For single scenario tests, the task was easy. As seen, we have as many F-MUTs as scenarios (except two special cases) and since the execution part of the scenario only consists of the F-MUT, the execution part is automatically found when having the F-MUT information. Once the execution part of the scenario is found, we can quickly identify the setup part (everything before) and the oracle part (everything after). In the cases of multiple scenarios, the oracle part of the scenarios is still easy to identify by taking all the assertions between the F-MUT of the first and the second scenario including all the calls they depend on. However, identifying the setup became a bit more difficult since the scenarios tend to share a part of the whole setup part. Also, the setup part of the second scenario might depend on a part of the oracle of the first scenario.

To respond to the **RQ2** on how the information of the focal method under test helps to generate descriptive test names, we gathered that information with our tool and generated a name based on it. We then compared the generated name to the original one to see which of them describes the test better. Also, we compared our results with the results of two other tools. Here we had to notice that the F-MUT(s) alone does not provide enough information for a good name. A name, based merely on the F-MUT(s), is still better than a mere `"test1"` but will also, in most cases, be accompanied by a number to avoid two similar names. For a test to really be descriptive, we also need information about setup and oracle part to accompany the F-MUT information. But here we have the problem, that often there are more than one element and hence more than one piece of information in each of these parts, meaning the name will become too long and difficult to read. In such cases, and also to improve the description of the setup and the oracle part, we still are dependent on human interpretation.

5 Conclusion and Future Work

5.1 Further work

The tool is still not complete and could be improved by finding a way to include all custom assertions into the analysis which should not be difficult. We could also improving the mutator analysis to work properly in more cases. This is more complicated since the main problem resides with compatibility between the used framework (ByteBuddy) and the execution of some of the test cases. Also, the tool does not yet automatically change the names of the methods, so there is not any mechanism that helps to differentiate the names inside a test class. This could also be added by adding a new layer of results that regroups the results for each class and then proceeds to a last check for each method in this class to adapt the possible conflicts. To further evaluate the tool, there could also be a case study proposing this tool to developers to find out if it is usable and how it could be improved. In this context, we could also check if the naming improves when a developer uses the tool. Also, the tool could be changed to create a name recommendation plugin for developers and hence exclude the danger of misleading names sneaking into the tests. In such a tool, the roughly 75% of acceptable names would still benefit the developer while the misleading ones can be rejected. The tool could then even integrate optional fields to specify the class under test and the focal method under test to improve the naming even further.

As mentioned in section 2.1, there are also alternative approaches that aim to achieve the same goal of improving the unit test comprehension and these approaches could even be combined with our works. For example by reducing the number of statements [25, 26], by reducing the number of assertions [27], by splitting tests to one for each assertion [28], or by separating the different scenarios [35–37]. These would, for example, help to reduce the problem of multiple scenarios naming and also the long names due to multiple assertions.

5.2 Conclusion

In this work, we picked up the idea of focal method under tests as the central point of the unit test, to use it to improve test names. We implemented a tool which generates unit test names based on their source and byte code by (1) extracting the focal method information and (2) build a name around this and other information picked up in the test case. The results were promising, and we managed to generate over 70% of test names based on these focal method under test information. But names that are only based on this information are not descriptive enough and need to feature other pieces of information as the information about the oracle and setup parts. We also found that these are easy to find once we know the focal method under test. Only in more complex and rare cases with multiple scenarios, finding these parts became a bit more challenging. Finally, even if the tool provides good results, it still has too many mistakes to use it as an automatic test name generator and should rather be used as a test name recommendation plugin.

6 Appendix

During the work on this thesis, an excel sheet was created where all the results were collected. In this appendix, we give a detailed explanation about this document starting by explaining its structure and the we explain the different abbreviations (special cases) that are listed there.

6.1 Data set

The above mentioned Excel sheet contains all the results from the Excel output of the tool but most of the lines are the result of the manual analysed. Only the ones marked with "*(cmp.)*" are computed results directly returned from the tool.

6.1.1 General information

- **Test Class Package** *(cmp.)*: the package name of the analysed test class.
- **Test Class Name** *(cmp.)*: the name of the class in which the analysed method is defined.
- **Test Method Name** *(cmp.)*: the name of the analysed method. The name does not include the parameters since unit test usually do not feature them.

6.1.2 Complete results for all assertions

In this part, are all the results coming from the analysis with the option `keepDuplicate-Fmuts()` set to true. So each line is the result from one assertion. But in some cases, there can no result for an assertion (due to a failure or something else) or more than one result per assertion. In this part, there is always the same number of each of the element ant they correspond to each other.

- **All computed AAE types** *(cmp.)*: consists of the type of the AAE (package and class name).
- **All computed AAE name** *(cmp.)*: shows the variable name defined by Jimple. If the variable is named, declared in the Java code, this name is passed on, otherwise the variable will either receive "$stack[*number*]" or "tmp$[*number*]" as name depending on their initialization order.
- **All computed FMUT class type** *(cmp.)*: consists of the actual class type of the F-MUTs (package and class name).
- **All computed FMUT** *(cmp.)*: shows the Jimple method definition of the method: "[*return type*] methodName([*parameter types*])>([*parameter variable names*])".
- **All computed FMUT Line Number** *(cmp.)*: the corresponding line number of the F-MUT in the original Java code.

6.1.3 Different unique results

In this part, we broke the results of the previous part down to remove the duplicates. Now we also have two groups: the AAE group of unique pairs and the F-MUT group of unique triplets.

- **Computed AAE types**: consists of the type of the AAE (package and class name).
- **Computed AAE name**: shows the variable name defined by Jimple. If the variable is named, declared in the Java code, this name is passed on, otherwise the variable will either receive "$stack[*number*]" or "tmp$[*number*]" as name depending on their initialization order.
- **Computed FMUT class type**: consists of the actual class type of the F-MUTs (package and class name).
- **Computed FMUT**: shows the Jimple method definition of the method: "[*return type*] methodName([*parameter types*])>([*parameter variable names*])".
- **Computed FMUT Line Number**: the corresponding line number of the F-MUT in the original Java code.

6.1.4 Manual correction

If, for whatever reason, we do not agree with the F-MUT (or one of them), the actual expected result is shown here.

- **Manual FMUT class type**: consists of the actual class type of the F-MUTs (package and class name).
- **Manual FMUT**: shows the Jimple method definition of the method: "[*return type*] methodName([*parameter types*])>([*parameter variable names*])".

6.1.5 Statistic numbers

In this part are collected all the numbers of the previous parts, so they can be used in statistic analysis.

- **number of FMUTs**: number of total F-MUTs found. This may contain a lot of duplicates since all the F-MUTs are collected for each assertion. It corresponds to the number of F-MUTs in subsection 6.1.2.
- **number of different FMUTs (different line numbers)**: number of different F-MUTs found by appearance in the test case (with different name and different line number). This corresponds to the number of F-MUTs in subsection 6.1.3.
- **number of different FMUTs (different method)**: number of different methods used as F-MUT in the test case (different name and class type).
- **number of different FMUT classes**: number of different classes declaring the F-MUTs in the test case.
- **number of AAEs**: number of total AAEs found. This may contain a lot of duplicates since all the AAEs are collected for each assertion. It corresponds to the number of AAEs in subsection 6.1.2.
- **number of different AAEs**: number of different variables used as AAE in the test case (different name and type). This corresponds to the number of AAEs in subsection 6.1.3.
- **number of different AAEs types**: number of different types for the used AAEs.

6.1.6 Analysis success

- **Analysis success?**: 1 if the analysis was successful in this test case based on the predefined parameters. It may also count as success if there are more F-MUTs as probably intended by the test writer. 2 if the results correspond to the manual F-MUT. Else if the analysis failed somehow it is a 0. For the different special cases see section 6.2.1.

6.1.7 Scenarios

- **Number of sub-scenarios**: Number of sub-scenarios detected in the test case.
- **Can we separate sub-scenarios?**: This field only contains a value if there are more than one sub-scenario. In this case, 1 if the different sub-scenarios can be re-factored to different test cases containing one sub-scenario each. There are different cases described in section 6.2.2.

6.1.8 Assertions

- **Number of regular assertions**: Number of assertions in the test case (custom assertions included).
- **Number of custom assertions**: Number of custom assertions in the test case.
- **Are assertions in loop?**: 1 if a part or all assertions are inside a loop.

6.1.9 Original name

- **Original name meaningful?**: 1 if the original test case name given by the test developer is meaningful, 2 if it contains al desired pieces of information and is readable and 0 if the name is meaningless or even erroneous. There are different cases described in section 6.2.3.

6.1.10 Computed name

- **Computed name** *(cmp.)*: the name computed by the tool.
- **Computed name meaningful?**: 1 if the new test case name we could generate out of the received results is meaningful, 2 if it contains al desired pieces of information and is readable and 0 if the name is meaningless or even erroneous. There are different cases described in section 6.2.4.

- **Is computed name better?**: 1 if the newly generated name is as good or comparable (with different aspects) as the original one, 2 if it surpasses this last one and 0 if the new name is worse than the original. There are different cases described in section 6.2.5.

6.1.11 Does F-MUT help with sections

- **Does name help understanding?**: 1 if the knowing the F-MUT helps discerning the setup part of the test from the other parts. There are different cases described in section 6.2.6.

6.2 Encountered exceptions

In this section, we grouped all the exceptions that we encountered during our work that were not fixed or just partially. The abbreviations are the same as in the Excel sheet containing all the results.

6.2.1 Analysis not successful and other general cases

In this section are listed all the exceptions found in relationship to the general behaviour of the tool, why it might have failed or other interesting information about the test.

0aij, Too much nested Java Library variables Actual asserted expression search fails due too many Java library variables building up the asserted object.

In the following case, the analysis first checks the variable `test` from the assertion (line 20). The array is initialized with an integer value, the return of `buffer.length()` (which is not followed any further). Next, we see that `test` is used in the `buffer.getChars()` (line 19), which is also a Java library method. And the analysis stops here since nothing was found. But it should continue to search the variable `buffer` so find the variable `string` on line 12 trough the `append()` method (which is also Java library). This last one would finally guide us to the real F-MUT, `node.toHtml()` on line 9.

```
1  // Test the fidelity of the toHtml() method.
2  public void testFidelity () throws ParserException, IOException {
3    Node node;
4    URL url = new URL ("http://sourceforge.net");
5    Lexer lexer = new Lexer (url.openConnection ());
6    int position = 0;
7    StringBuffer buffer = new StringBuffer (80000);
8    while (null != (node = lexer.nextNode ())) {
9      String string = node.toHtml ();
10     if (position != node.getStartPosition ())
11         fail ("non-contiguous" + string);
12       buffer.append (string);
13       position = node.getEndPosition ();
14       if (buffer.length () != position)
15         fail ("text length differed after encountering node " + string);
16     }
17     char[] ref = lexer.getPage ().getText ().toCharArray ();
18     char[] test = new char[buffer.length ()];
19     buffer.getChars (0, buffer.length (), test, 0);
20     assertEquals ("different amounts of text", ref.length, test.length);
21     ...
22   }
```

Listing 6.1: Excerpt from class *org.htmlparser.tests.lexerTests.LexerTests*

0baf, No back-step from setup implemented The focal method needs a forward search into the setup method of the test class then a backwards search into the actual test case (which is not covered).

Even if the assertion on line 20 does not follow the JUnit convention (see paragraph 0nca), this case still helps explain this problem.
In the assertions on line 18 and 19 methods of the variable `rects` are asserted. Both methods are inspectors so we look for the declaration which is on line 17 with `g.getRects()`. And since this method also is an inspector, we search for `g` (which is also the variable asserted by the third assertion on line 20). But this variable is not found in the test case itself but in the `setup()` method on line 10. So we set the initialization of `g` as temporary F-MUT. But when we go further in the `setup()` method, on line 11, we see that `g` is used in the initialization of the variable `output`. With this in mind we can check even further but the `setup()` ends here and if we look on line 16, a mutator method `drawBar()` is executed on `output`. But my tool does not yet include a back-step from `setup()` to the actual test case. That is why `drawBar()` is not correctly detected as F-MUT.

```
1  private Color fgColour;
2  private Color bgColour;
3  private GraphicsMock g;
4  private GraphicsOutput output;
5
6  protected void setUp() throws Exception {
7    super.setUp();
8    fgColour = Color.black;
9    bgColour = Color.pink;
10   g = new GraphicsMock();
11   output = new GraphicsOutput(g, DefaultEnvironment.DEFAULT_FONT, fgColour, bgColour);
12 }
13
14 @Test
15 public void testDrawBarDrawsRectangle() throws Exception {
16   output.drawBar(0, 0, 10, 100, true);
17   List rects = g.getRects();
18   assertEquals(1, rects.size());
19   assertEquals(new Rectangle(0, 0, 10, 100), rects.get(0));
```

```
  assertEquals(g.getColor(), fgColour);
}
```

Listing 6.2: Excerpt from class *net.sourceforge.barbecue.output.GraphicsOutputTest*

0bbb, Byte-Buddy failure Something with the ByteBuddy agent is not working (memory overload when and only when attaching an interceptor, even with empty one that does nothing). This leads to no mutator

0cni, Java conventions not implemented The Java conventions are not implemented in the analysis (except for the assertion conventions), so there are more F-MUTs than there should be.

The first case encountered is the method `Object.equals()` that is not implemented as explained in the documentation[1] nor any other `equals` method overriding it, so both compared values are taken into account in the search for the F-MUTs, which leads to too much (and false) results alongside the correct one. The approach would suggest to ignore the expected value of the `equals()` method, since it is the reference object to compare to, and only focus on the actual value, but this has to be implemented as a special case in many different places and for all the official `equals` methods.

In this second case, illustrated below, the `Arrays.equals(expected, actual)` on line 10 has two parameters: `expected` and `encoded` (or actual). In this case, the `expected` could be ignored due to the Java convention. But, as above, the tool does not implement this special exclusions. So not only the correct F-MUT `barcode.encodeData()` from `encoded` on line 3 is detected but also the `ModuleFactory.getModule()` from `expected` on the lines 5-8.

```
1  @Test public void testCSetEncodesDigitPairs() throws Exception {
2    barcode = new Code128Barcode("01990199", Code128Barcode.C);
3    Module[] encoded = barcode.encodeData();
4    Module[] expected = new Module[] {
5      ModuleFactory.getModule("01", Code128Barcode.C),
6      ModuleFactory.getModule("99", Code128Barcode.C),
7      ModuleFactory.getModule("01", Code128Barcode.C),
8      ModuleFactory.getModule("99", Code128Barcode.C)
9    };
10   assertTrue(Arrays.equals(expected, encoded));
11 }
```

Listing 6.3: Excerpt from class *net.sourceforge.barbecue.linear.code128.Code128BarcodeTest*

0csi, Asserting incrementing index The assertion checks the size of an object with a loop and an incrementing index.

[1]https://docs.oracle.com/javase/8/docs/api/java/lang/Object.html

We see that the variable `index` that is asserted is initialized on line 5 with a primitive integer value and only used as index identifier of the `node` list on line 7. It is then incremented in the same line (which is translated to `index = index + 1`. As we see this, we can assume that the object assigned to the list on this index is part an important part of the test. We would then check `e.nextNode()` which is doing a computation and would therefore be the F-MUT (and if we continue, `e.hasMoreNodes()` is a mutator and `e = parser.elements()`, the initialization of `e` would be the method the tester intended to test, judging by the test name).

But the tool does not support backtracking trough the index of lists so the F-MUT is not found (in this case there is no F-MUT detected by the tool).

```
1  public void testElements() throws Exception {
2    StringBuffer hugeData = new StringBuffer();
3    for (int i=0;i<5001;i++) hugeData.append('a');
4    createParser(hugeData.toString());
5    int index = 0;
6    for (NodeIterator e = parser.elements();e.hasMoreNodes();) {
7      node[index++] = e.nextNode();
8    }
9    assertEquals("There should be 1 node identified",1,index);
10 }
```

Listing 6.4: Excerpt from class *org.htmlparser.tests.ParserTest*

0fcm, F-MUT in test method The analysis finds a F-MUT, but it belongs to the test class. The real one is inside this "false" F-MUT.

In the following example, the asserted variables `source` and `query` both only invoke inspectors, so the `processString` on line 2 is detected as F-MUT. But it is a test class and should not. If we look inside this method and follow the `rtr` variable and enter the `getQuerySource` method on line 15, we find the actual F-MUT: the initialization of `TRECQuery` on line 21.

```
1  @Test public void testOneLongQID() throws Exception {
2    QuerySource source = processString("<top>\n<num> Number: 4444\n<title> defination
         Gravitational\n</top>");
3    assertTrue(source.hasNext());
4    String query = source.next();
5    assertEquals("defination gravitational", query);
6    assertEquals("4444", source.getQueryId());
7    assertFalse(source.hasNext());
8  }
9
10 protected QuerySource processString(String fileContents) throws Exception {
11   File tmpFile = tmpfolder.newFile("tmpQueries.trec");
12   PrintWriter pw = new PrintWriter(Files.writeFileWriter(tmpFile));
13   pw.print(fileContents);
14   pw.close();
15   QuerySource rtr = getQuerySource(tmpFile.toString());
16   assertNotNull(rtr);
17   return rtr;
18 }
```

```

protected QuerySource getQuerySource(String filename) throws Exception {
  return new TRECQuery(filename);
}
```

Listing 6.5: Excerpt from class *org.terrier.structures.TestTRECQuery*

0fih, Detected F-MUT is used as oracle method When looking at the test, we see that sometimes, the detected F-MUT is only helping the oracle part of the test. It is not an analysis failure as it has focal character, but by inspecting the code and the test name, we can see that the real intent is another method.

In the given case the method `getBounds()` on line 6 has focal character but the real intent of the tester is the `draw()` method on line 5 (which is even mentioned in the test name).

Since most of these “false F-MUTs” are getters (by the name), we could add an optional exclusion for methods starting with “get” (so they would automatically be declared inspectors).

```
1 @Test
2 public void testDrawingDoesNotAffectBounds() throws Exception {
3   Barcode barcode = new BarcodeMock("12345", false);
4   Rectangle bounds1 = barcode.getBounds();
5   barcode.draw((Graphics2D) new BufferedImage(500, 500,
       BufferedImage.TYPE_BYTE_GRAY).getGraphics(), 0, 0);
6   Rectangle bounds2 = barcode.getBounds();
7   assertEquals(bounds1, bounds2);
8 }
```

Listing 6.6: Excerpt from class *net.sourceforge.barbecue.BarcodeTest*

0fne, Multiple F-MUTs in scenario The F-MUT is not the only relevant tested method (different setters are tested but only the last one is can be the F-MUT).

In the following excerpt, we have the variable `attribute` as AAE and `setRawValue()` on line 7 as F-MUT. But `setName()` (line 5) and `setAssignment()` (line 6) should also count as F-MUT since they change another part of the asserted values.

```
1  public void testProperties () {
2    Attribute attribute;
3    ...
4    attribute = new Attribute ();
5    attribute.setName ("label");
6    attribute.setAssignment ("=");
7    attribute.setRawValue ("The civil war.");
8    assertTrue ("should not be standalone", !attribute.isStandAlone ());
9    assertTrue ("should not be whitespace", !attribute.isWhitespace ());
10   assertTrue ("should be valued", attribute.isValued ());
11   assertTrue ("should not be empty", !attribute.isEmpty ());
12   ...
13 }
```

Listing 6.7: Excerpt from class *org.htmlparser.tests.lexerTests.AttributeTests*

0git, Tested getter The tool detects the correct F-MUT but the intent of the tester is probably to test a getter method (which cannot be a F-MUT).

In the following test case, the result of `getKeys()` on line 5 is asserted. When looking at the original test name, we can see that the tester intended to create the test for this getter method. But since this method just returns a field of `metaindex` it is correctly not detected as the F-MUT but the initialization on line 3 (which is also the F-MUT for the assertion on line 4).

```
1  @Test
2  public void test_getKeys() throws Exception {
3    MemoryMetaIndex metaindex = new MemoryMetaIndex();
4    assertNotNull(metaindex);
5    assertArrayEquals(keys, metaindex.getKeys());
6  }
```

Listing 6.8: Excerpt from class *org.terrier.realtime.memory.TestMemoryMetaIndex*

0iwtc, Test in wrong test class The method is clearly in the wrong test class. The assertion checks the size of the variable `model` (line 7), initialized on line 6 by `reader.read()` which is the F-MUT. But the `reader` object is of type `DataReaderIBM1_3_0` and not `DataReaderIBM1_2_2`. The only occurrence of `DataReaderIBM1_2_2` is in the form of the input stream source on line 4.
And since this is the only method of this class and there is no existing `TestDataReaderIBM1_3_0` test class and no `DataReaderIBM1_2_2` type class, either the test class name and the name of the input stream file or the name of the `DataReaderIBM1_3_0` class should be adapted (or the sources changed) otherwise this naming is confusing.

Another possibility would be that the `DataReaderIBM1_3_0` is tested to be backwards compatible with IBM1_2_2 data, but in this case, this should be mentioned in the test name.

```
1  public class TestDataReaderIBM1_2_2 {
2     @Test
3     public void testParse1() throws Exception {
4        InputStream in = UnittestHelper.getResourceAsStream(UnittestHelper.FOLDER_IBM,
            "SampleIBM1_2_2.txt");
5        DataReader reader = new DataReaderIBM1_3_0(in);
6        GCModel model = reader.read();
7        assertEquals("number of events", 28, model.size());
8     }
9  }
```

Listing 6.9: Excerpt from class *com.tagtraum.perf.gcviewer.imp.TestDataReaderIBM1_2_2*

0mnv, Mutator analysis not working The mutator analysis fails. This can have multiple causes. One of them is that the deep cloning fails.

0nca, Wrong use of the JUnit assertion convention To avoid unwanted F-MUTs by checking all the parameters of the assertions (e.g. the initialization of the expected value), the tool is restricted on the JUnit conventions for the assertions[2]. But in some cases these are not respected.

For example, for the assertion used below, we have `assertEquals(expected, actual)`, so the tool only checks the second parameter, in this example `bgColour` instead of `g.getColor()` (line 11).

```
1  protected void setUp() throws Exception {
2    ...
3    bgColour = Color.pink;
4    g = new GraphicsMock();
5    output = new GraphicsOutput(g, DefaultEnvironment.DEFAULT_FONT, fgColour, bgColour);
6  }
7
8  @Test
9  public void testDrawBarUsingBackgroundColourActuallyDrawsWithBackgroundColour() throws
        Exception {
10   output.drawBar(0, 0, 10, 100, false);
11   assertEquals(g.getColor(), bgColour);
12 }
```

Listing 6.10: Excerpt from class *net.sourceforge.barbecue.output.GraphicsOutputTest*

0nf, No F-MUT There is no F-MUT detected, despite the tool running successfully. There were two of such tests (other tests had no F-MUTs due to an analysis failure). These two tests just test getters of a static class (otherwise the initialization would be the F-MUT). The shorter example is the following.

Here, the result of `getIndex()` on line 3 is asserted. But this method just returns the index of a char in a `java.util.List` so it gets classified as an inspector. But there is nothing else to check, so no F-MUT is detected.

```
1  @Test
2  public void testCorrectIndexReturnedForChar() throws Exception {
3    assertEquals(8, ModuleFactory.getIndex("(", Code128Barcode.A));
4  }
```

Listing 6.11: Excerpt from class *net.sourceforge.barbecue.linear.code128.ModuleFactoryTest*

0nit, Nothing is tested Nothing is tested (e.g. a simple constant return without anything else). In this case, the method `getPostAmble()` returns `ModuleFactory.START_STOP` (line 2 of listing 6.13. In the test, in line 3 (listing 6.12), this results to

[2]http://junit.sourceforge.net/javadoc/org/junit/Assert.html

compare the `ModuleFactory.START_STOP` constant to itself. And since the setup is done inside the constructor of the test class and not in a designated setup method, no F-MUT is detected.

```java
@Test
public void testPostAmbleIsStopCharOnly() throws Exception {
  assertEquals(ModuleFactory.START_STOP, barcode.getPostAmble());
}
```

Listing 6.12: Excerpt from class *net.sourceforge.barbecue.linear.code39.Code39BarcodeTest*

with

```java
protected Module getPostAmble() {
  return ModuleFactory.START_STOP;
}
```

Listing 6.13: Excerpt from class *net.sourceforge.barbecue.linear.code39.Code39Barcode*

0oiv, Actual type not visible Only the interface is visible so the real F-MUT is also abstract and can not be detected, since its body can not be analysed.

In this case, all assertions point to the `reader.read()` method on line 7 (since `event` points to `model` on line 11). But as explained above, the return type of `getDataReader()` on line 3 is the abstract `DataReader` type. So the `read()` method on line 7 does not have any body and is therefore an inspector. In the case where the mutator analysis fails or the method would be a computation, it is therefore ignored. Finally, the F-MUT found by the tool is `getDataReader()` on line 3.

```java
1  @Test public void testParseTsGCReportPrioPauseTime() throws Exception {
2    InputStream in =
         getInputStream("SampleJRockit1_4_2ts-gcreport-gcpriopausetime.txt");
3    DataReader reader = new DataReaderFactory().getDataReader(in);
4    ...
5
6    GCModel model = reader.read();
7
8    assertEquals("count", 64, model.size());
9
10   GCEvent event = (GCEvent) model.get(0);
11   assertEquals("timestamp", 18.785, event.getTimestamp(), 0.000001);
12   assertEquals("name", Type.JROCKIT_GC.getName(), event.getExtendedType().getName());
13   assertEquals("before", 32260, event.getPreUsed());
14   assertEquals("after", 4028, event.getPostUsed());
15   assertEquals("total", 32768, event.getTotal());
16   assertEquals("pause", 0.024491, event.getPause(), 0.0000001);
17 }
```

Listing 6.14: Excerpt from class *com.tagtraum.perf.gcviewer.imp.TestDataReaderJRockit1_4_2*

0sdm, Non-conventional setup method The setup is done by an invoked helper method (not detectable since it is not marked with setup annotations) and the actual asserted expression is in this setup part.

To illustrate, the assertion on line 3 asserts the return of `getValue()` called on the return value of `elementAt()`, leading to the variable `attributes` (none of those two has focal character). But the track ends here because there is no official setup method, all the setup and the execution of the F-MUT is done in the `getParameterTableFor()` method on line 2. So there is no F-MUT detected.

```
1  public void testParseParameters() {
2    getParameterTableFor("a b = \"c\"");
3    assertEquals("Value","c",((Attribute)(attributes.elementAt (2))).getValue ());
4  }
```

Listing 6.15: Excerpt from class *org.htmlparser.tests.lexerTests.AttributeTests*

0sjf, Soot Jimple format problem When creating arrays without assigning them to a direct variable (as on line 7), the Soot Jimple compiler transforms the array to a `java.lang.Object` array (line 4 of Jimple code). When values are retrieved from this array, they are still of the type `java.lang.Object` (`r` on Jimple line 3 and 7). And only when the variable is needed for a method invocation it is cast to a new temporary object ("tmp$number", on Jimple lines 10 and 13). But each time a new temporary variable is generated, so there is no direct way to link two uses of the same variable `r` in this case.

So when checking the assertion on Jimple line 17, we follow the variables from $stack17 on Jimple line 16 and 17 to `$stack16` (jimple line 16 and 15), `$stack15` (jimple line 15 and 14) and finally to `tmp$2` on Jimple line 15. But when checking `tmp$2` and even `r` (which follows `tmp$2` in the def-chain), we do not find a method with focal character since `sort()`, on Jimple line 11, is invoked on another temporary variable which was cast from `r` (jimple line 10). So the detected F-MUT are the `initialise()` on the lines 3 and 5.

The real variable names where `tmp$729037495` as `tmp$1` and `tmp$605216264` as `tmp$2` (changed it to improve readability).

```
1  @Test public void testSorting() {
2    ResultSet r1 = new CollectionResultSet(2);
3    r1.initialise();
4    ResultSet r2 = new QueryResultSet(2);
5    r2.initialise();
6
7    for (ResultSet r : new ResultSet[]{r1,r2}) {
8      r.getDocids()[0] = 10;
9      r.getScores()[0] = 5d;
10     r.getDocids()[1] = 9;
11     r.getScores()[1] = 10d;
12     r.sort();
13     assertEquals(9, r.getDocids()[0]);
14     assertEquals(10, r.getDocids()[1]);
15
16     assertEquals(10d, r.getScores()[0], 0.0d);
```

```
    assertEquals(5d, r.getScores()[1], 0.0d);
  }
}
```

Listing 6.16: Excerpt from class *org.terrier.matching.TestResultSets*

```
public void testSorting() {
  ...
  java.lang.Object r;
  java.lang.Object[] $stack9;
  ...
  // line 7
  r = $stack9[14];
  ...
  // line 12
  tmp$1 = (org.terrier.matching.ResultSet) r;
  interfaceinvoke tmp$1.<org.terrier.matching.ResultSet: void sort()>();
  // line 13
  tmp$2 = (org.terrier.matching.ResultSet) r;
  $stack15 = interfaceinvoke tmp$2.<org.terrier.matching.ResultSet: int[]
      getDocids()>();
  $stack16 = $stack15[0];
  $stack17 = (long) $stack16;
  staticinvoke <org.junit.Assert: void assertEquals(long,long)>(9L, $stack17);
  ...
}
```

Listing 6.17: Excerpt from the Jimple code of *org.terrier.matching.TestResultSets*

0sma, Use of uncovered assertions The test either uses custom assertions or the fail assertion[3]. This can lead to no results at all or just a part of the expected results (like in the example below where the tool does not find an entry point since there is no conventional assertion, only the custom `assertStringEquals()` is present).

```
public void testConstructors () {
  Vector attributes;
  Tag tag;
  String html;

  attributes = new Vector ();
  // String, null
  attributes.add (new Attribute ("wombat", null));
  // String
  attributes.add (new Attribute (" "));
  // String, String
  attributes.add (new Attribute ("label", "The civil war."));
  attributes.add (new Attribute (" "));
  // String, String, String
  attributes.add (new Attribute ("frameborder", "= ", "no"));
  attributes.add (new Attribute (" "));
  // String String, String, char
  attributes.add (new Attribute ("name", "=", "topFrame", '"'));
  tag = new TagNode (null, 0, 0, attributes);
  html = "<wombat label=\"The civil war.\" frameborder= no name=\"topFrame\">";
```

[3]http://junit.sourceforge.net/javadoc/org/junit/Assert.html

```
  assertStringEquals ("tag contents", html, tag.toHtml ());
}
```

Listing 6.18: Excerpt from class *org.htmlparser.tests.lexerTests.AttributeTests*

6.2.2 Can it be refactored?

In this section are listed all the main cases of multi-scenario tests. It also explains if they can be separated or not.

1ioo, Inspector has focal character In reality, there is only one scenario, but since there is an inspector with focal character (also see paragraph 6.2.1), the analysis detects more than one sub-scenarios.
Below we should only have `servlet.doGet(req, res)` (line 5) as only F-MUT and hence only one scenario but since `barecode.getWidth()` (line 9) has focal character the analysis detects two F-MUTs and scenarios. A way of solving this would be annotating `Barecode.getWidth()` so the analysis will not take it into account.

```
1  public void testSettingWidthChangesDefault() throws Exception {
2    params.put("data", "12345");
3    params.put("width", "3");
4    req.setParameters(params);
5    servlet.doGet(req, res);
6    assertEquals("image/png", res.getContentType());
7    assertTrue(res.hasOutput());
8    Barcode barcode = servlet.getBarcode();
9    assertEquals(330, barcode.getWidth());
10 }
```

Listing 6.19: Excerpt from class *net.sourceforge.barbecue.BarcodeServletTest*

with

```
1  public int getWidth() {
2    return (int) getActualSize().getWidth();
3  }
```

Listing 6.20: Excerpt from class *net.sourceforge.barbecue.Barcode*

where “Barecode.getActualSize()” (line 2) exercises a computation.

1ms, Only one dependant scenario Different independent sub-scenarios can be separated into different test cases, but the last part needs all the others as setup.

The different scenarios all work in standalone apart of the last one which needs all the previous ones as setup. In the code, we even distinguish between scenarios and the setup parts for the last scenario (all the lines that are not between the declaration of the AAE and the last assertion are only useful for the last scenario, in the example below, this would be lines 18, 27, 39-46 and 58). This last one though, since it depends on the others, raises the problem discussed in the paragraph paragraph 1nc.

```
1  public void testProperties () {
2    Attribute attribute;
3    Attribute space;
4    Vector attributes;
5    Tag tag;
6    String html;
7
8    attributes = new Vector ();
9    attribute = new Attribute ();
10
11 // Part 1 ---------------------------------------
12   attribute.setName ("wombat");
13   assertTrue ("should be standalone", attribute.isStandAlone ());
14   assertTrue ("should not be whitespace", !attribute.isWhitespace ());
15   assertTrue ("should not be valued", !attribute.isValued ());
16   assertTrue ("should not be empty", !attribute.isEmpty ());
17 // Setup for last assertion : part 5 -----------
18   attributes.add (attribute);
19 // Part 2 ---------------------------------------
20   space = new Attribute ();
21   space.setValue (" ");
22   assertTrue ("should not be standalone", !space.isStandAlone ());
23   assertTrue ("should be whitespace", space.isWhitespace ());
24   assertTrue ("should be valued", space.isValued ());
25   assertTrue ("should not be empty", !space.isEmpty ());
26 // Setup for last assertion : part 5 -----------
27   attributes.add (space);
28 // Part 3 ---------------------------------------
29   attribute = new Attribute ();
30   attribute.setName ("label");
31   attribute.setAssignment ("=");
32   attribute.setRawValue ("The civil war.");
33   assertTrue ("should not be standalone", !attribute.isStandAlone ());
34   assertTrue ("should not be whitespace", !attribute.isWhitespace ());
35   assertTrue ("should be valued", attribute.isValued ());
36   assertTrue ("should not be empty", !attribute.isEmpty ());
37 // Setup for last assertion : part 5 -----------
38 // Could also be a proper scenario but has no direct assertions
39   attributes.add (attribute);
40   attributes.add (space);
41   attribute = new Attribute ();
42   attribute.setName ("frameborder");
43   attribute.setAssignment ("= ");
44   attribute.setRawValue ("no");
45   attributes.add (attribute);
46   attributes.add (space);
47 // Part 4 ---------------------------------------
48   attribute = new Attribute ();
49   attribute.setName ("name");
50   attribute.setAssignment ("=");
51   attribute.setValue ("topFrame");
52   attribute.setQuote ('"');
53   assertTrue ("should not be standalone", !attribute.isStandAlone ());
54   assertTrue ("should not be whitespace", !attribute.isWhitespace ());
55   assertTrue ("should be valued", attribute.isValued ());
56   assertTrue ("should not be empty", !attribute.isEmpty ());
57 // Setup for last assertion : part 5 -----------
58   attributes.add (attribute);
// Part 5 --------------------------------------
  tag = new TagNode (null, 0, 0, attributes);
  html = "<wombat label=\"The civil war.\" frameborder= no name=\"topFrame\">";
  assertStringEquals ("tag contents", html, tag.toHtml ());
}
```

Listing 6.21: Excerpt from class *org.htmlparser.tests.lexerTests.AttributeTests*

1nc, Dependent scenarios The different scenarios check different aspects of the tested object and/or how they react to changes (before and after scenarios). So, most of them could be separated so that the different scenarios each figure in their test case. But they would have to depend one on another which does not make it practical, so it is better to keep them as they are in one common scenario.

To illustrate this, we have the following case below with two scenarios around two F-MUTS: `makeIndexSinglePass()` (line 2) and `createDirectIndex()` (line 8). To separate both, either (1) the first test looses its test case status (becomes a private setup invoked by the second test), (2) the one who sets the other up is tested twice (once on it own and once in the setup part of the second test) or (3) the second test repeats the setup and execution part of the first one.

```
1  @Test public void testQuickly() throws Exception {
2    Index index = IndexTestUtils.makeIndexSinglePass(
3      new String[]{"doc1", "doc2"},
4      new String[]{"Quick fast brown fox", "jumped huge black lazy dog"});//no stopwords
5    assertFalse(index.hasIndexStructure("direct"));
6    assertFalse(index.hasIndexStructure("direct-inputstream"));
7    assertTrue(index instanceof IndexOnDisk);
8    new Inverted2DirectIndexBuilder((IndexOnDisk) index).createDirectIndex();
9    assertTrue(index.hasIndexStructure("direct"));
10   assertTrue(index.hasIndexStructure("direct-inputstream"));
11 }
```

Listing 6.22: Excerpt from class *org.terrier.structures.indexing.singlepass.TestInverted2DirectIndexBuilder*

to

```
private Index setUp_makeIndexSinglePass() throws Exception {
  Index index = IndexTestUtils.makeIndexSinglePass(
    new String[]{"doc1", "doc2"},
    new String[]{"Quick fast brown fox", "jumped huge black lazy dog"});//no stopwords
  assertFalse(index.hasIndexStructure("direct"));
  assertFalse(index.hasIndexStructure("direct-inputstream"));
  assertTrue(index instanceof IndexOnDisk);
  return index;
}

@Test public void testQuicklyV1() throws Exception {
  Index index = setUp_makeIndexSinglePass();
  new Inverted2DirectIndexBuilder((IndexOnDisk) index).createDirectIndex();
  assertTrue(index.hasIndexStructure("direct"));
  assertTrue(index.hasIndexStructure("direct-inputstream"));
}
```

Listing 6.23: Example of refactoring 1nc (v1)

or

```java
private Index index;

@Test public void testMakeIndexSinglePass() throws Exception {
  index = IndexTestUtils.makeIndexSinglePass(
    new String[]{"doc1", "doc2"},
    new String[]{"Quick fast brown fox", "jumped huge black lazy dog"});//no stopwords
  assertFalse(index.hasIndexStructure("direct"));
  assertFalse(index.hasIndexStructure("direct-inputstream"));
  assertTrue(index instanceof IndexOnDisk);
}

@Test public void testQuickly() throws Exception {
  // to be sure it is ran before
  testMakeIndexSinglePass();
  new Inverted2DirectIndexBuilder((IndexOnDisk) index).createDirectIndex();
  assertTrue(index.hasIndexStructure("direct"));
  assertTrue(index.hasIndexStructure("direct-inputstream"));
}
```

Listing 6.24: Example of refactoring 1nc (v2)

or

```java
@Test public void testMakeIndexSinglePass() throws Exception {
  Index index = IndexTestUtils.makeIndexSinglePass(
    new String[]{"doc1", "doc2"},
    new String[]{"Quick fast brown fox", "jumped huge black lazy dog"});//no stopwords
  assertFalse(index.hasIndexStructure("direct"));
  assertFalse(index.hasIndexStructure("direct-inputstream"));
  assertTrue(index instanceof IndexOnDisk);
}

@Test public void testQuickly() throws Exception {
  Index index = IndexTestUtils.makeIndexSinglePass(
    new String[]{"doc1", "doc2"},
    new String[]{"Quick fast brown fox", "jumped huge black lazy dog"});//no stopwords
  new Inverted2DirectIndexBuilder((IndexOnDisk) index).createDirectIndex();
  assertTrue(index.hasIndexStructure("direct"));
  assertTrue(index.hasIndexStructure("direct-inputstream"));
}
```

Listing 6.25: Example of refactoring 1nc (v3)

1olt, Gives many little scenarios Yes, the sub-scenarios can be separated into different test cases but they will be many little tests, testing different aspects of the same method. In the case below, we could create 11 separate test cases which are all very small and difficult to make distinguishable by the name only. So we could just keep them together in one test case.

```
@Test public void testParseCommaDelimitedStrings() throws Exception {
  assertTrue(Arrays.equals(new String[0], ArrayUtils.parseCommaDelimitedString("")));
  assertTrue(Arrays.equals(new String[0], ArrayUtils.parseCommaDelimitedString(" ")));
  assertTrue(Arrays.equals(new String[]{"1"},
      ArrayUtils.parseCommaDelimitedString("1")));
  assertTrue(Arrays.equals(new String[]{"1"}, ArrayUtils.parseCommaDelimitedString("
      1")));
  assertTrue(Arrays.equals(new String[]{"1"}, ArrayUtils.parseCommaDelimitedString("1
      ")));
  assertTrue(Arrays.equals(new String[]{"1"}, ArrayUtils.parseCommaDelimitedString("
      1 ")));
  assertTrue(Arrays.equals(new String[]{"1","2"},
      ArrayUtils.parseCommaDelimitedString("1,2")));
  assertTrue(Arrays.equals(new String[]{"1","2"},
      ArrayUtils.parseCommaDelimitedString("1 ,2")));
  assertTrue(Arrays.equals(new String[]{"1","2"},
      ArrayUtils.parseCommaDelimitedString("1, 2")));
  assertTrue(Arrays.equals(new String[]{"1","2"},
      ArrayUtils.parseCommaDelimitedString("1 , 2")));
  assertTrue(Arrays.equals(new String[]{"12","256"},
      ArrayUtils.parseCommaDelimitedString("12,256")));
}
```

Listing 6.26: Excerpt from class *org.terrier.utility.TestArrayUtils*

1rem, Out of context assertion An "out of context assertion" check is made that does not belong to the scenario and should be tested in its test case. This would lead to less F-MUTs in the test cases and hence improve the naming of these tests.

In the example below, we have two F-MUTs: `term()` (line 5) coming from the assertion on line 6 and `new MemoryLexicon()` (line 2) coming from the assertion on line 3. As we can see from the title, the check of the assertion on line 3 does not have any relationship with the intent of the test and should be done in its test case as shown in the refactoring example below.

```
1  public void test_incrementTerm1() throws Exception {
2    MemoryLexicon lexicon = new MemoryLexicon();
3    assertNotNull(lexicon);
4    for (int i = 0; i < 10; i++)
5      lexicon.term(terms[i].toString(), entries[i]);
6    assertEquals(10, lexicon.numberOfEntries());
7  }
```

Listing 6.27: Excerpt from class *org.terrier.realtime.memory.TestMemoryLexicon*

```
@Test public void test_initializationIsNotNull() throws Exception {
  MemoryLexicon lexicon = new MemoryLexicon();
  assertNotNull(lexicon);
}

@Test public void test_incrementTerm1() throws Exception {
  MemoryLexicon lexicon = new MemoryLexicon();
  for (int i = 0; i < 10; i++)
    lexicon.term(terms[i].toString(), entries[i]);
  assertEquals(10, lexicon.numberOfEntries());
}
```

Listing 6.28: Example of refactoring 1rem

1ss, Need a common setup Yes, the different sub-scenarios can be separated into different test cases but they need to have the same setup part.

Already by reading the test name, we can see that there are more than one aspect of the method being tested: that the resolution cannot be negative and not zero. We could separate them into two separate tests with the same setup part (line 3 and 4). Here again, the question is raised if the setup should be done before every test case in the class or only a few selected (then it should be used as private method invoked only in the needed cases).

```
1  @Test
2  public void testResolutionCannotBeSetNegativeOrZero() throws Exception {
3    BarcodeMock barcode = new BarcodeMock("12345");
4    int expected = barcode.getResolution();
5    barcode.setResolution(-1);
6    assertEquals(expected, barcode.getResolution());
7    barcode.setResolution(0);
8    assertEquals(expected, barcode.getResolution());
9  }
```

Listing 6.29: Excerpt from class *net.sourceforge.barbecue.BarcodeTest*

```
BarcodeMock barcode;
int expected;

@Before public void setUp() throws Exception {
  barcode = new BarcodeMock("12345");
  expected = barcode.getResolution();
}

@Test public void testResolutionCannotBeSetNegative() {
  barcode.setResolution(-1);
  assertEquals(expected, barcode.getResolution());
}

@Test public void testResolutionCannotBeSetZero() {
  barcode.setResolution(0);
  assertEquals(expected, barcode.getResolution());
}
```

Listing 6.30: Example of refactoring 1ss

6.2.3 The Original naming

In this section we look at the different bad habits of test writers regarding the names they gave. This is also based on the three important points for test names that we saw in chapter 3:

- R1 Test names should have a clear relation to the code under test (the F-MUT); they should allow developers to identify the tests concerning this method without having to inspect the test code.
- R2 Test names should be descriptive of the test code; there should be an understandable, intuitive relation between the test code and its name, meaning that the scenario of the test (setup, execution and oracle part) should be recognized by reading the name.
- R3 Test names should uniquely distinguish tests within a test suite, such that developers can use them to navigate the test suite.

2nia, Only vague and repetitive description The name gives a general description of the test does not contain any information about the concrete F-MUT, nor the setup or the oracle part. So it can be largely improved. The names, if alone in a test case, could be acceptable but often end up with multiple names, only differentiated by a number, which is a bad practice. Here, the additional information like F-MUT, setup or oracle part would largely improve the name and help to distinguish the different similar cases. But overall, it does not completely satisfy R1 and R2 since it gives only one part of the important information.

In the following example, we have three tests which only differ on line 9, 11 and 16 ; the first only has i as index, the second (featured here) adds 42 to the index and the third multiplies the index by 42. So, the tests could have been named "`testAppendSimple`", "`testAppendWithAddition`" and "`testAppendWithMultiplication`" in the following order, instead of "`testAppend1()`", "`testAppend2()`" and "`testAppend3()`".

```java
1  public void testAppend2 () {
2    PageIndex index;
3    int pos;
4    int[] list;
5
6    index = new PageIndex (null);
7
8    for (int i = 0; i < 10000; i++) {
9      pos = index.row (i + 42);
10     assertTrue ("append not at end", pos == i);
11     assertTrue ("wrong position", pos == index.add (i + 42));
12   }
13
14   list = index.get ();
15   for (int i = 0; i < 10000; i++)
16     assertTrue ("wrong value", list[i] == i + 42);
17 }
```

Listing 6.31: Excerpt from class *org.htmlparser.tests.lexerTests.PageIndexTests*

2nic, No information at all Although the test class name shows the right class being tested, the names are just named `"test#"` or `"test_#"` where # represents a number. In the case of `org.htmlparser.tests.lexerTests.PageTests` we even have 42 of such tests which all test the same method `getAbsoluteURL()` on line 2 with slightly different parameters. At least, they could have been named `"testGetAbsoluteURL_#"`, but ideally they would be named with a name describing their difference. In the documentation, we see that the first 22 tests are "normal" examples and the others "abnormal" examples. Using this information, we could name the case presented below: `"testNormalGetAbsoluteURLWithValidURL"` (since the first test returns the parameter as a result). This breaks R1 and R2.

```
1 public void test1 () throws ParserException {
2   assertEquals ("test1 failed", "https:h", mPage.getAbsoluteURL ("https:h"));
3 }
```

Listing 6.32: Excerpt from class *org.htmlparser.tests.lexerTests.PageTests*

2nicf, Does not match test class name In this test case, the name of the class is misleading or the test case is added to the wrong test class, hence making the relationship to the class under test difficult and misleading, hence breaking R1.

As an example, in the class `TestDataReaderSun1_7_0G1`, the important class is the `DataReaderSun1_6_0G1` class since the F-MUT, `read()` on line 6 is declared in it. In the whole test case, we see that the `DataReaderSun1_6_0G1` class is used with the `TestDataReaderSun1_7_0G1` data format (line 4 and 5), but this should be specified in the name to avoid confusion. So either the test class should be renamed to `TestDataReaderSun1_7_0G1FormatOnDataReaderSun1_6_0G1` for example or the test case moved to the class `TestDataReaderSun1_6_0G1` (which by the way does not even exist).

```
1  @Test public void youngPause_u1() throws Exception {
2    ...
3
4    final InputStream in = getInputStream("SampleSun1_7_0-01_G1_young.txt");
5    final DataReader reader = new DataReaderSun1_6_0G1(in, GcLogType.SUN1_7G1);
6    GCModel model = reader.read();
7
8    assertEquals("gc pause", 0.00631825, model.getPause().getMax(), 0.000000001);
9    assertEquals("heap", 64*1024, model.getHeapAllocatedSizes().getMax());
10   assertEquals("number of errors", 0, handler.getCount());
11 }
```

Listing 6.33: Excerpt from class *com.tagtraum.perf.gcviewer.imp.TestDataReaderSun1_7_0G1*

2opi, No indicated F-MUT In these cases, the name does not indicate the actual tested method, but the parameters given to the tested method. It would be useful to add this indication to the test name. It is basically the counterpart to paragraph 2nia and also does not completely satisfy R1 and even breaks R2 since the relation to the tested method is not given.

In the example below, the same class also contains a `"testOr"`, `"testString"` and other similar tests. All of them test the method `extractAllNodesThatMatch()` on line 10 with different input classes which all inherit `"NodeFilter"`. So we could either (1) name all methods `"extractAllNodesThatMatchXFilter"` (with X being the tested filter name) or (2) we could rename the test class `"ExtractAllNodesThatMatch-FilterTest"` and keep the method names as they are.

```
1  // Test and filtering.
2  public void testAnd () throws ParserException {
3    String guts;
4    String html;
5    NodeList list;
6
7    guts = "<body>Now is the <a id=one><b>time</b></a> for all good <a
         id=two><b>men</b></a>..</body>";
8    html = "<html>" + guts + "</html>";
9    createParser (html);
10   list = parser.extractAllNodesThatMatch (
11     new AndFilter (
12       new HasChildFilter (new TagNameFilter ("b")),
13       new HasChildFilter (new StringFilter ("men")))
14     );
15   assertEquals ("only one element", 1, list.size ());
16   assertType ("should be LinkTag", LinkTag.class, list.elementAt (0));
17   LinkTag link = (LinkTag)list.elementAt (0);
18   assertEquals ("attribute value", "two", link.getAttribute ("id"));
19 }
```

Listing 6.34: Excerpt from class *org.htmlparser.tests.filterTests.FilterTest*

2tm, Tests without identifier In these tests, the prefix “test” is missing at the start of the name. In all the occurring cases, the test case is annotated with the `@org.junit.Test` annotation, which could count as a test identifier. But it would still be better to include the “test” prefix even though, as already mentioned, there is not a general convention for naming test cases and this case also does not break any of our criteria.

6.2.4 The Computed naming

In this section are listed all the exceptions found in relationship to the general behaviour of the tool, why it might have failed or other interesting information about the test.

3aii, Inner class used in naming In some cases, if the F-MUT is declared in an inner class, it might generate long and confusing names (with $). This might give unclear test case names. This could be improved by removing the part before and with the $, but this is not implemented yet.

As an example, in `org.terrier.indexing.TestWARC10Collection.testDocumentSpecific`, the F-MUT is the initialization of the inner class of `TestWARC10Collection`, `2RedirWARC10Collection`. So the generated name is `"testInitializationOfTestWARC10Collection$2RedirWARC10Collection"` which is weird and confusing. With the proposed improvement, it would result in `"testInitializationOf2RedirWARC10Collection"` which is way better.

3ecs, Jimple conversions In some cases, Jimple makes a conversion that makes the name to be incorrect or misguiding. The name gives a false or confusing image of the test case hence violate criteria R2.

If there is a negation of a boolean value, Jimple transforms it to an integer (0 or 1) and compares it to another integer (see lines 8-13 in the Jimple code below). For example, the assertion `assertTrue(!threadsException.isThrown())` on line 5 below, will generate the name `"IsThrownEquals0IsTrue"`. Also, this shows a bad usage of the assertion `assertTrue()` since the tester could have used `assertFalse()` and remove the negation. This would have led to the name `"IsThrownIsFalse"`, which is much clearer.

```
1  @Test public void multiThreadIndexingTest() throws Exception {
2    UncaughtIRException threadsStatusException = new UncaughtIRException();
3    Thread.setDefaultUncaughtExceptionHandler(threadsStatusException);
4    ...
5    assertTrue(!threadsStatusException.isThrown());
6    ...
7  }
```

Listing 6.35: Excerpt from class *org.terrier.tests.SimultaneousIndexingRetrievalTest*

```
1  public void multiThreadIndexingTest() throws java.lang.Exception {
2    $stack12 = new
         org.terrier.tests.SimultaneousIndexingRetrievalTest$UncaughtIRException;
3    specialinvoke
         $stack12.<org.terrier.tests.SimultaneousIndexingRetrievalTest$UncaughtIRException:
         void <init>(org.terrier.tests.SimultaneousIndexingRetrievalTest)>(this);
4    staticinvoke <java.lang.Thread: void
         setDefaultUncaughtExceptionHandler(java.lang.Thread$UncaughtExceptionHandler)>($stack12);
5    ...
6    $stack22 = virtualinvoke
         $stack12.<org.terrier.tests.SimultaneousIndexingRetrievalTest$UncaughtIRException:
         boolean isThrown()>();
7
8    if $stack22 == 0 goto label1;
9    $stack35 = 0;
10   goto label2;
11
12  label1:
13   $stack35 = 1;
14
label2:
  staticinvoke <org.junit.Assert: void assertTrue(boolean)>($stack35);
}
```

Listing 6.36: Excerpt of Jimple conversion of *org.terrier.tests.SimultaneousIndexingRetrievalTest*

3csi, Asserting incrementing index As seen in paragraph 0csi when asserting the size of an object with a counter and a loop, the tool fails to find the F-MUT. But not only this, it also fails in the name generation from the concerned assertion.

As we can see in the example, the variable `index` is initialized with the value 0 on line 5. The real F-MUT is `elements()` on line 6, so the generated name by the tool is `"test0Equals1AfterElements"` since the initialization of the actual value is 0 and the tool cannot detect the real number.

And since this case often occurs in relation with the case 0csi, we have a name that is not logical and does not give any relation with the CUT so violating the criteria R1 and R2.

```
1  public void testElements() throws Exception {
2    StringBuffer hugeData = new StringBuffer();
3    for (int i=0;i<5001;i++) hugeData.append('a');
4    createParser(hugeData.toString());
5    int index = 0;
6    for (NodeIterator e = parser.elements();e.hasMoreNodes();) {
7      node[index++] = e.nextNode();
8    }
9    assertEquals("There should be 1 node identified",1,index);
10 }
```

Listing 6.37: Excerpt from class *org.htmlparser.tests.ParserTest*

3eni, Equals not implemented As in the paragraph 0cni the method `Object.equals()` is not implemented as special case, so the name generator is treating it the same way as all other methods ending with a weird naming hence break the criteria R2.

In this example, the F-MUT is the initialization of `Parser` on line 4. When generating the name with the assertion of line 7, the tool first encounters the `equals()` method on line 7 and generates a name with its expected return value, giving `"testEqualsIsTrueAfterInitializationOfParser"`. But since the `equals()` method has a defined purpose by the Java conventions, a better name would be `"testGetEncodingEqualsUTF8AfterInitializationOfParser"`.

```
1  public void testSingleQuotedCharset () throws ParserException {
2    String url = "http://htmlparser.sourceforge.net/test/SinglequotedCharset.html";
3
4    Parser parser = new Parser(url);
5    for (NodeIterator e = parser.elements();e.hasMoreNodes();)
6      e.nextNode();
7    assertTrue ("Wrong encoding", parser.getEncoding ().equals ("UTF-8"));
8  }
```

Listing 6.38: Excerpt from class *org.htmlparser.tests.ParserTest*

3fni, False F-MUTs lead to false names Since the F-MUT (or one of the F-MUTs) is not the intent of the tester, but most methods used in the oracle part of the test (see paragraph 0fih), the name is completely wrong or at least part of it (if the real F-MUT is also part of the detected F-MUTs). This violates point R2 and can also lead to violation of R1 (when the focal inspector belongs to another class than the real F-MUT).

In the example below, the AAE is `bos` declared on line 3 (through `buf` and `boas`). And the detected F-MUT is `flush()` (line 9) which is only helping to reset the tests so that the other test cases work afterwards and `toBitString()` on line 13 which is only helping the test as an inspector. The real, intended F-MUT in this test, is the method `write_int_LSB_0()` on line 10. But the generated name is "`testFlushAndToBitString`" which is misleading.

```
1  @Test public void test_write_int_32bits() throws IOException {
2    ByteArrayOutputStream baos = new ByteArrayOutputStream();
3    DefaultBitOutputStream bos = new DefaultBitOutputStream(baos);
4    int value = 0;
5    value = (1 << 31) | (1 << 30);
6    value = value | 1;
7
8    bos.write_int_LSB_0(value, 32);
9    bos.flush();
10   byte[] buf = baos.toByteArray();
11   assertThat(buf.length, is(4));
12   assertThat(buf, equalTo(Utils.toBytes(value)));
13   assertThat(Utils.toBitString(buf), equalTo("11000000000000000000000000000001"));
14 }
```

Listing 6.39: Excerpt from class *uk.ac.ebi.ena.sra.cram.io.BitOutputStreamTest*

3ntl, Too long The generated name is too long. This can be caused if (1) there are too much detected F-MUTs (mostly happens when there are too much F-MUTs detected, see paragraph 0fih and 3fni), (2) too many assertions that are taken into account or (3) because one of the constants is too long (e.g. a long String or number). This makes it difficult to distinguish anything in the test name, hence does not satisfy either criteria R2 nor R1.

In the example below, we have the situation with a long string: "`testGenerateSummaryEquals`*X*" (with X, the String initialized on line 4). This becomes way too long and confusing. This could be fixed by fixing a maximum String length and if longer, cut the string in this place or replace it with “SomeString” for example.

```
1  @Test public void testTwoOfThreeSentences() {
2    Summariser s = new DefaultSummariser();
```

```
  String summary = s.generateSummary(doc1, new String[]{"lorem", "ipsum"});
  String expected = "Lorem Ipsum is simply dummy text of the printing and typesetting
      industry...Lorem Ipsum has been the industry's standard dummy text ever since
      the 1500s, when an unknown";
  assertEquals(expected, summary);
}
```

Listing 6.40: Excerpt from class *org.terrier.querying.summarisation.TestDefaultSummariser*

3rfm, F-MUT twice in the name In some cases the F-MUT appears twice in the name. This happens if the F-MUT is used to generate the name with the assertion and that there is another F-MUT from another assertion. For example, we have the name `"testCalculateMod43Equals24AfterCalculateMod43AndGetSymbolEquals0AfterGetModuleForIndex"`. The first assertion (line 2) generates the name component `"CalculateMod43Equals24"` and the second one (line 4) `"GetSymbolEquals0AfterGetModuleForIndex"`. When simply combining those two, we obtain `"testCalculateMod43Equals24AndGetSymbolEquals0AfterGetModuleForIndex"`. But when reading the name, we might think that there are two assertions for one F-MUT which is false. So to clarify this, the F-MUT of the first assertion is repeated.

```
1 @Test public void testCalculateMod43() throws Exception {
2   assertEquals(24, Code39Barcode.calculateMod43("I050000001"));
3   assertEquals("0",
        ModuleFactory.getModuleForIndex(Code39Barcode.calculateMod43("I050000001")).getSymbol());
4 }
```

Listing 6.41: Excerpt from class *net.sourceforge.barbecue.linear.code39.Code39BarcodeTest*

3tcc, Test class method is F-MUT The focal method is a class from the test case (something inside is the true F-MUT) as in paragraph 0fcm. But if the test class is correctly named, the name generation can even be improved. But even then, the relation with the CUT cannot be found only by reading the test name. So these cases break the criteria R1.

In the below example, the detected F-MUT is `countImageTagsWithHTMLParser()` on line 6. But the actual F-MUT would be `setNodeFactory()` (line 3) of the detected F-MUT. But when comparing the generated names, we end with `"testCountImageTagsWithHTMLParserEqualsFindImageTagCount"` when using the test class method, and would end up with `"test0EqualsFindImageTagCountAfterSetNodeFactory"` (which is even false due to the case explained in paragraph 3csi).

```
1 public void testNumImageTagsInYahooWithoutRegisteringScanners() throws ParserException
      {
2    // this page is full of bad comments like <!---resources--->
3    Lexer.STRICT_REMARKS = false;
4    // First count the image tags as is
5    int imgTagCount;
```

```
   int parserImgTagCount = countImageTagsWithHTMLParser();
   imgTagCount = findImageTagCount(getParser ());
   assertEquals("Image Tag Count",imgTagCount,parserImgTagCount);
}

public int countImageTagsWithHTMLParser() throws ParserException {
   Parser parser = new Parser("http://education.yahoo.com/",new
       DefaultParserFeedback());
   parser.setNodeFactory (new PrototypicalNodeFactory (new ImageTag ()));
   setParser (parser);
   int parserImgTagCount = 0;
   Node node;
   for (NodeIterator e= parser.elements();e.hasMoreNodes();) {
      node = e.nextNode();
      if (node instanceof ImageTag) {
         parserImgTagCount++;
      }
   }
   return parserImgTagCount;
}
```

Listing 6.42: Excerpt from class *org.htmlparser.tests.FunctionalTests*

3uia, Assertion generated name is useless The information generated by the assertion is not useful at all hence in conflict with requirement R2. In the example below, we would have the name `"testToByteArryEqualToNewByteArrayAfterWrite"`. Here we see that using the initialization of the Array as name does not give any useful information, rather we should have used the value used for the initialization “10000000” (which is given by the expression “`1 >> 7`” on line 14) to get the name `"testToByteArryEqual-To10000000AfterWrite"`.

```
1  @Test public void test_write_2() throws IOException {
2    Byte[] values = new Byte[] { 1, 2 };
3    int[] charFreqs = new int[] { 1, 2 };
4    HuffmanTree<Byte> tree = HuffmanCode.buildTree(charFreqs, values);
5
6    HuffmanByteCodec codec = new HuffmanByteCodec(tree);
7    ByteArrayOutputStream baos = new ByteArrayOutputStream();
8    BitOutputStream bos = new DefaultBitOutputStream(baos);
9
10   codec.write(bos, (byte) 2);
11   bos.flush();
12   byte[] buf = baos.toByteArray();
13
14   assertThat(buf, equalTo(new byte[] { (byte) (1 << 7) }));
15 }
```

Listing 6.43: Excerpt from class *uk.ac.ebi.ena.sra.cram.encoding.HuffmanByteCodecTest*

3ofm, Only one F-MUT in name In some cases, the tool generates a name only on the base of the F-MUT. This can become problematic for tests with the same F-MUT since Java does not allow multiple methods with the same name in the same class. So these names must be made unique to implement requirement R3, either by a number (which is what we try to avoid) or additional information that the tool does not provide or that would be inconvenient to show (e.g. generates names that are too long).

In the example below, we have three different cases of the same F-MUT: `createLexiconIndex()` on lines 2, 10 and 20. But all original names miss to mention the F-MUT. For the first case, we generated the name `"testNumberOfEntriesEquals1LAndGetLexiconEntryReturnIsNotNullAndGetFrequencyEquals2LAfterCreateLexiconIndex"` which is long but contains the F-MUT and also mentions that we have one entry that should appear twice (as in the original name `"testOneTermTwoOccurrence"`). But this name is already very long because of its three assertions. And if we look at the second and third examples, `"testOneTermOneOccurrence"` and `"testTwoTermThreeOccurrence"`, we even have five assertions instead of three (even if in each case, two will be merged because they generate the same name: lines 15 and 16 for the second example and lines 23 and 25 for the third example). This would become too large, that is why the tool has a limit of three assertions to be included in the name, and if more are present, the assertion names will be ignored. So the generated names for the second and third case are `"testCreateLexiconIndex"`. It only shows the F-MUT used and not the additional information as the first example and the original name and because of this, we have the problem of two methods with the same generated name.

```
1  @Test public void testOneTermTwoOccurrence() throws Exception {
2    Index index = createLexiconIndex(new String[]{"a", "a"});
3    Lexicon<String> lexicon = index.getLexicon();
4    assertEquals(1, lexicon.numberOfEntries());
5    assertNotNull(lexicon.getLexiconEntry("a"));
6    assertEquals(2, lexicon.getLexiconEntry("a").getFrequency());
7  }
8
9  @Test public void testOneTermOneOccurrence() throws Exception {
10   Index index = createLexiconIndex(new String[]{"a"});
11   Lexicon<String> lexicon = index.getLexicon();
12   assertEquals(1, lexicon.numberOfEntries());
13   assertNotNull(lexicon.getLexiconEntry("a"));
14   assertEquals(1, lexicon.getLexiconEntry("a").getFrequency());
15   assertEquals("a", lexicon.getIthLexiconEntry(0).getKey());
16   assertEquals("a", lexicon.getLexiconEntry(0).getKey());
17 }
18
19 @Test public void testTwoTermThreeOccurrence() throws Exception {
20   Index index = createLexiconIndex(new String[]{"a", "b", "a"});
21   Lexicon<String> lexicon = index.getLexicon();
22   assertEquals(2, lexicon.numberOfEntries());
23   assertNotNull(lexicon.getLexiconEntry("a"));
24   assertEquals(2, lexicon.getLexiconEntry("a").getFrequency());
25   assertNotNull(lexicon.getLexiconEntry("b"));
26   assertEquals(1, lexicon.getLexiconEntry("b").getFrequency());
27 }
```

Listing 6.44: Excerpt from class *org.terrier.structures.TestFSOMapFileLexicon*

6.2.5 Is the computed name better?

In this section we find the main difference between acceptable original names and computed names.

4diff, Show different aspects The computed name and the actual given name show different aspects of the test case that are both valid. The automated names often better reflect the tested F-MUT but the real names give more insights into the parameters (and other differences) of test cases. This might even be crucial for cases with the same F-MUTs which may have the same name if only based on the computing ones (see paragraph 3ofm.

If we look at the example below, we have three cases (there are even more in the test class) which all have the same F-MUT `read()` on lines 5, 18 and 30. And each of these cases has a name which does not mention the F-MUT (`"testYoungToSpace-Overflow"`, `"testPartialToSpaceOverflow"` and `"testGcMemoryPausePattern"`) but the important aspect of the setup (lines 3, 17 and 28). But on the other side, all three cases generate the name `"testRead"` (since they all have more than three assertions). So both names have useful information but the combination of them would be the best result: `"testReadYoungToSpaceOverflow"`, `"testReadPartialToSpaceOverflow"` and `"testReadGcMemoryPausePattern"`.

```
1   @Test public void testYoungToSpaceOverflow() throws Exception {
2     // special type of GC: 0.838: "[GC pause (young) (to-space overflow)..."
3     InputStream in = getInputStream("SampleSun1_6_0G1_young_toSpaceOverflow.txt");
4     DataReader reader = new DataReaderSun1_6_0G1(in, GcLogType.SUN1_6G1);
5     GCModel model = reader.read();
6
7     assertEquals("nummber of events", 1, model.size());
8     assertEquals("number of pauses", 1, model.getPause().getN());
9     assertEquals("gc pause sum", 0.04674512, model.getPause().getSum(), 0.000000001);
10    assertEquals("gc memory", 228*1024 - 102*1024, model.getFreedMemoryByGC().getMax());
11    assertEquals("max memory", 256*1024, model.getFootprint());
12  }
13
14  @Test public void testPartialToSpaceOverflow() throws Exception {
15    // special type of GC: 0.838: "[GC pause (partial) (to-space overflow)..."
16    InputStream in = getInputStream("SampleSun1_6_0G1_partial_toSpaceOverflow.txt");
17    DataReader reader = new DataReaderSun1_6_0G1(in, GcLogType.SUN1_6G1);
18    GCModel model = reader.read();
19
20    assertEquals("nummber of events", 1, model.size());
21    assertEquals("number of pauses", 1, model.getPause().getN());
22    assertEquals("gc pause sum", 0.00271976, model.getPause().getSum(), 0.000000001);
23    assertEquals("gc memory", 255*1024 - 181*1024, model.getFreedMemoryByGC().getMax());
24    assertEquals("max memory", 256*1024, model.getFootprint());
25  }
26
27  @Test public void testGcMemoryPausePattern() throws Exception {
28    InputStream in = new ByteArrayInputStream(("0.360: [GC cleanup 19M->19M(36M),
          0.0007889 secs]").getBytes());
29    DataReader reader = new DataReaderSun1_6_0G1(in, GcLogType.SUN1_6G1);
30    GCModel model = reader.read();
```

```java

  assertEquals("count", 1, model.size());
  assertEquals("full gc pause", 0, model.getFullGCPause().getN());
  assertEquals("gc pause", 0.0007889, model.getGCPause().getMax(), 0.0000001);
  assertEquals("memory", 0, model.getFreedMemoryByGC().getMax());
}
```

Listing 6.45: Excerpt from class *org.terrier.structures.TestFSOMapFileLexicon*

6.2.6 Does the F-MUT help find the setup/oracle part?

In this section we see the main cases where finding the setup or the oracle part, only with the help of the F-MUT(s) information, was more difficult.

5rfs, Wrong F-MUT does not help Because the detected F-MUT is the wrong one (see special cases in section 6.2.1), it will automatically give a wrong impression on the setup and oracle part.

5nut, Not adapted to integration tests The tool only works on unit tests. If the test case represents an integration test, the tool will fail to generate a meaningful name since it only checks the last mutator before the assertion, but an integration test has many different mutators that work together. And to generate a meaningful name for a whole integration scenario, we would need a completely different approach and even more human interpretation.

```java
@Test public void test1() throws IOException, CramCompressionException {
  CramRecordFormat format = new CramRecordFormat();
  CramRecord record = format.fromString(recordSpec);

  CramStats stats = new CramStats(new CramHeader(), null);
  Logger.getLogger(CramStats.class).setLevel(Level.ERROR);
  stats.addRecord(record);

  RecordCodecFactory factory = new RecordCodecFactory();
  CramRecordBlock block = new CramRecordBlock();
  block.setUnmappedReadQualityScoresIncluded(true);
  CramCompression compression = CramCompression.createDefaultCramCompression();
  block.setCompression(compression);

  stats.adjustBlock(block);
  BitCodec<CramRecord> codec = factory.createRecordCodec(null, block, null);

  ByteArrayOutputStream baos = new ByteArrayOutputStream();
  DefaultBitOutputStream bos = new DefaultBitOutputStream(baos);

  codec.write(bos, record);
  bos.close();

  ByteArrayInputStream bais = new ByteArrayInputStream(baos.toByteArray());
  DefaultBitInputStream bis = new DefaultBitInputStream(bais);
  codec = factory.createRecordCodec(null, block, null);
  CramRecord derivedRecord = codec.read(bis);
  derivedRecord.setFlags(derivedRecord.getFlags());

  assertThat(derivedRecord, equalTo(record));
}
```

Listing 6.46: Excerpt from class *uk.ac.ebi.ena.sra.cram.impl.CramRecordCodecRoundTripTests*

5sis, Setup is in dedicated setup method The F-MUT helps to find the oracle part but the setup part is in the dedicated `setup()` method.

```
protected void setUp() throws Exception {
  super.setUp();
  fgColour = Color.black;
  bgColour = Color.pink;
  g = new GraphicsMock();
  output = new GraphicsOutput(g, DefaultEnvironment.DEFAULT_FONT, fgColour, bgColour);
}

@Test public void testDrawBarDrawsRectangle() throws Exception {
  output.drawBar(0, 0, 10, 100, true);
  List rects = g.getRects();
  assertEquals(1, rects.size());
  assertEquals(new Rectangle(0, 0, 10, 100), rects.get(0));
  assertEquals(g.getColor(), fgColour);
}
```

Listing 6.47: Excerpt from class *net.sourceforge.barbecue.output.GraphicsOutputTest*

7 Anleitung zu wissenschaftlichen Arbeiten

7.1 Requirements

The recommended way to use the tool is using Eclipse but the tool also works inside other IDEs or the command line (for a quick guide for the command line, check the appendix).

1. Install Eclipse: Eclipse can be downloaded from `http://www.eclipse.org`.
2. Import the project to analyse: click on `File -> Import -> General -> Existing Projects`, insert the path of the project and select it.
3. Include `FMUTAnalysis` in your build path: Go to `Java Build Path -> Libraries` and click Add External JARs... and select the FMUTAnalysis.jar. There is a jar with all the dependencies build into it. If you want to add the dependencies by yourselves, the list can be found in the appendix.

For the purpose of this guide, we provide a sample project that can be imported (in this case, step 3 can be skipped since FMUTAnalysis is already included).

7.2 Tool setup

The tool also supports projects in a folder structure or build to jars (in this example, we use a project inside a folder structure, for a quick guide for jar projects, check the appendix).

We need to create the class shown in listing 7.1. This class is used to configure and run the analysis.

```java
1 import ch.unibe.fmut.FMUTAnalysis;
2 import ch.unibe.fmut.soot.WrongSootPathException;
3 public class Runner {
4   public static void main(String[] args) throws WrongSootPathException {
5     FMUTAnalysis analysis = new FMUTAnalysis("packageName", "pathToFileClasses",
          "pathToTestClasses");
6     // add configurations here
7     analysis.start();
8   }
9 }
```

Listing 7.1: Example runner for Eclipse inside project to analyze

There is one class in the jar that can be used: FMUTAnalysis (line 5).The initialization of this class uses the following parameters:

⇒ **String packageName**: the parent package name. In the case of our sample project, it is “sample.project”.

⇒ **String pathToFileClasses**: main directory path of the class files (e.g. “target/-classes” or “bin/main”). In the case of our sample project, we use “bin/main”.

⇒ **String pathToTestClasses**: main directory path of the test class files (e.g. “target/test-classes” or “bin/test”). It may be the same as to the normal class files. In the case of our sample project, we use “bin/test”.

By default, the analysis checks all the test methods it finds. If for some reason, this is not desired, there is a way to include or exclude specific methods by using one of the following commands:

⇒ **setIncludedFiles(List<String>)**: If only certain test cases have to be analyzed, they can be added to the analysis by this method in the form of a List containing the String “package.Class.method” for each method. Once this method is called all changes done by **setExcludedFiles** are ignored. Optionally, the different method identifiers can also be added as varargs (array of parameters without having to explicitly create the array).

```java
analysis.setIncludedFiles(
  "sample.project.PersonTest.testInitializePersonHasTwoEyes",
  "sample.project.PersonTest.testInitializePersonIsNotColorBlind",
  "sample.project.EyeTest.testSetNumberOfConeCells"
);
```

In our example, by adding this code to line 6, only these three methods will be analysed.

⇒ **setExcludedFiles(List<String>)**: If only certain test cases have to be skipped, they can be removed from the analysis by this method in the form of a List containing the String “package.Class.method” for each method. Once this method is called all changes done by **setIncludedFiles** are ignored. Optionally, the different method identifiers can also be added as varargs.

```
analysis.setExcludedFiles(
  "sample.project.PersonTest.testInitializePersonHasTwoEyes",
  "sample.project.PersonTest.testInitializePersonIsNotColorBlind",
  "sample.project.EyeTest.testSetNumberOfConeCells"
);
```

In our example, by adding this code to line 6, all but these three methods will be analysed.

Once the setup is finished only possibilities remain:

⇒ **start()**: The whole analysis is started with all the configurations set in advance. If the “pathToTestClasses” or “pathToFileClasses” is not set correctly, a “WrongSootPathException” is thrown.

⇒ **getResults()**: returns a Map with the different test case identifiers (“package.Class.method”) as keys and a list of the results as values. This method has to be called after the **start()** method otherwise, no results will be returned.

⇒ **reset()**: resets the whole analysis to run a second time. If not done, the results might not be correct.

For a complete list of options for the analysis, you can refer to the appendix.

7.3 Output

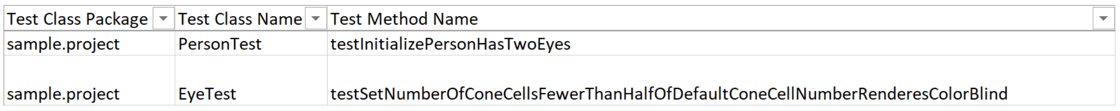
The analysis outputs its results as an excel file in the root directory of the project. It contains all the analysed test methods and the respective results. The information is grouped into different categories:

7.3.1 General

The three first columns, as shown in Figure 7.1, constitute the basic information of the test case. We have (from left to right):

- **Test Class Package**: the package name of the analysed test class.

- **Test Class Name**: the name of the class in which the analysed method is defined.
- **Test Method Name**: the name of the analysed method. The name does not include the parameters since unit tests usually do not feature them.

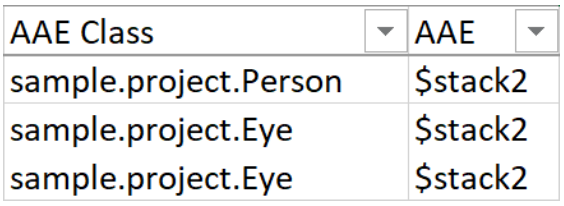


Test Class Package	Test Class Name	Test Method Name
sample.project	PersonTest	testInitializePersonHasTwoEyes
sample.project	EyeTest	testSetNumberOfConeCellsFewerThanHalfOfDefaultConeCellNumberRenderesColorBlind

Figure 7.1: General information of the excel sheet.

7.3.2 Actual asserted expression

In Figure 7.2, we have the same two examples as in Section 7.3.1 but this time with the AAEs. The first column, “AAE Class” consists of the type of the AAE (package and class name). The second column shows the variable name defined by Jimple. If the variable is named, declared in the Java code, this name is passed on, otherwise the variable will either receive “$stack[*number*]” or “tmp$[*number*]” as name depending on their initialization order. A test case may contain different AAEs as shown in the second example.

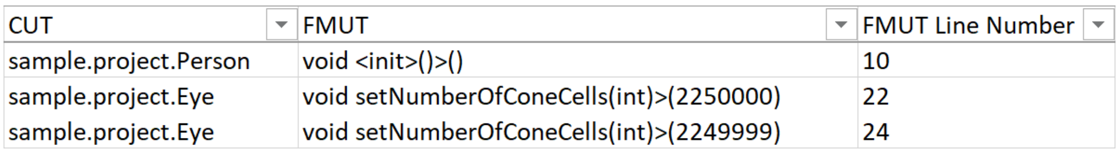


AAE Class	AAE
sample.project.Person	$stack2
sample.project.Eye	$stack2
sample.project.Eye	$stack2

Figure 7.2: Information about the actual asserted expression

7.3.3 Focal method under test

In Figure 7.3, we have the same two examples as in Section 7.3.1 but this time with the F-MUTs. The first column, “CUT” consists of the class the F-MUT is declared under test (package and class name). The second column shows the Jimple method definition of the focal method: “[*return type*] methodName([*parameter types*])>([*parameter variable names*])”. The last column shows the line number of the F-MUT in the original Java test code.

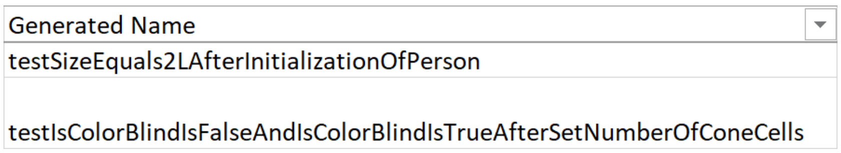


CUT	FMUT	FMUT Line Number
sample.project.Person	void <init>()>()	10
sample.project.Eye	void setNumberOfConeCells(int)>(2250000)	22
sample.project.Eye	void setNumberOfConeCells(int)>(2249999)	24

Figure 7.3: Information about the focal method under test

7.3.4 Naming

The last column shows the generated names.



Generated Name
testSizeEquals2LAfterInitializationOfPerson
testIsColorBlindIsFalseAndIsColorBlindIsTrueAfterSetNumberOfConeCells

Figure 7.4: The generated name for the test

8 Appendix to the Anleitung

8.1 Tool setup

Here is additional information for the setup of the tool.

8.1.1 JAR project

The tool also provides support to analyse jars directly (if the tests are included). For some features, such as line numbers and original variable names, the jar has to be built to contain the information, otherwise, they won't be given.

```java
import ch.unibe.fmut.FMUTAnalysis;
import ch.unibe.fmut.soot.WrongSootPathException;
public class Runner {
  public static void main(String[] args) throws WrongSootPathException {
    FMUTAnalysis analysis = new FMUTAnalysis("packageName", "jarPath/Name.jar",
        "jarPath/Name.jar");
    // add configurations here
    analysis.start();
  }
}
```

Listing 8.1: Example runner for Eclipse inside FMUTAnalysis project

The only thing that changes is that, instead of the output folder directories, we must now provide the jar directory and name.

8.1.2 Command prompt

Aside from Eclipse, our tool can also be run from the command line. For this, the following Java file of listing 8.2 has to be in the same directory as the FMUTAnalysis.jar and the jar of the project to analyse (which also has to contain the test cases).

```java
import ch.unibe.fmut.FMUTAnalysis;
import ch.unibe.fmut.soot.WrongSootPathException;
public class Runner {
  public static void main(String[] args) throws WrongSootPathException {
    String s = args[0];
    FMUTAnalysis analysis = new FMUTAnalysis(s,s,s);
    // add configurations here
    analysis.start();
  }
}
```

Listing 8.2: Example runner for CMD

The analysis can then be started with the following commands:

In Linux or MacOS:
First execute: `javac -cp .:ch.unibe.fmut.jar Runner.java`
Then: `java -cp .:ch.unibe.fmut.jar:`*`jarname`*`.jar Runner `*`jarname`*`.jar`

In Windows:
First execute: `javac -cp .;ch.unibe.fmut.jar Runner.java`
Then: `java -cp .;ch.unibe.fmut.jar;`*`jarname`*`.jar Runner `*`jarname`*`.jar`

8.1.3 Dependencies

There is a .jar file with all dependencies included if required but it is recommended to import the actual dependencies to keep the newest versions. For a list of all dependencies see table 8.1.

8.1.4 Available methods

The class FMUTAnalysis contains more configuration methods than explained in the previous chapter. The whole list of methods is found below:

The initialization parameters:

⇒ **String packageName**: the parent package name (e.g. “ch.unibe”)

⇒ **String pathToFileClasses**: main directory path of the class files (e.g. “target/-classes” or “bin/main”).

Name	Oldest tested version	Newest tested version	Maven XML
Soot	3.0.0 SNAPSHOT	3.1.0 SNAPSHOT	`<dependency> <groupId>ca.mcgill.sable</groupId> <artifactId>soot</artifactId> </dependency>`
Junit4	4.12	4.12	`<dependency> <groupId>junit</groupId> <artifactId>junit</artifactId> </dependency>`
Apache POI	3.11	4.1.0	`<dependency> <groupId>org.apache.poi</groupId> <artifactId>poi</artifactId> </dependency>`
Apache POI OOXML	3.11	4.1.0	`<dependency> <groupId>org.apache.poi</groupId> <artifactId>poi-ooxml</artifactId> </dependency>`
Byte-Buddy	1.7.5	1.9.12	`<dependency> <groupId>net.bytebuddy</groupId> <artifactId>byte-buddy</artifactId> </dependency>`
Byte-Buddy Dep.	1.7.5	1.9.12	`<dependency> <groupId>net.bytebuddy</groupId> <artifactId>byte-buddy-dep</artifactId> </dependency>`
Byte-Buddy agent	1.7.5	1.9.12	`<dependency> <groupId>net.bytebuddy</groupId> <artifactId>byte-buddy-agent</artifactId> </dependency>`
Unitils core	3.4.2	3.4.6	`<dependency> <groupId>org.unitils</groupId> <artifactId>unitils-core</artifactId> </dependency>`
Unitils mock	3.4.2	3.4.6	`<dependency> <groupId>org.unitils</groupId> <artifactId>unitils-mock</artifactId> </dependency>`
Logback	1.2.3	1.2.3	`<dependency> <groupId>ch.qos.logback</groupId> <artifactId>logback-classic</artifactId> </dependency>`

Table 8.1: Needed dependencies for the .jar

⇒ **String pathToTestClasses**: main directory path of the test class files (e.g. “target/test-classes” or “bin/test”). It may be the same as to the normal class files..

Configuration methods:

⇒ **setIncludedFiles(List<String>)**: If only certain test cases have to be analyzed, they can be added to the analysis by this method in the form of a List containing the String “package.Class.method” for each method. Once this method is called all changes done by **setExcludedFiles** are ignored. Optionally, the different method identifiers can also be added as varargs .

⇒ **setExcludedFiles(List<String>)**: If only certain test cases have to be skipped, they can be removed from the analysis by this method in the form of a List containing the String “package.Class.method” for each method. Once this method is called all changes done by **setIncludedFiles** are ignored. Optionally, the different method identifiers can also be added as varargs.

⇒ **setMutatorAnalysis(boolean)**: default value is true. If set to false, the analysis will run without the mutator analysis.

⇒ **setInspectorAnalysis(boolean)**: default value is true. If set to false, the analysis will run without the inspector analysis.

⇒ **generateNames(boolean)**: default value is true. If set to false, the analysis will not generate names for the results and the excel output. If set to true, *keepDuplicateFmuts* bellow will also be set to true (so that the name generation works properly).

⇒ **setIsInitializationMutator(boolean)**: default value is true. If set to false, the initialization of a variable will not count as mutator.

⇒ **keepDuplicateFmuts(boolean)**: default is true. If set to false, the analysis will only keep one each F-MUT once (so there will not be an F-MUT for each assertion).

⇒ **List<CaseOfResult>) noInspectorCases**: a list of cases that do not count as inspectors. The cases are defined in advanced and each method receives one case during the inspector analysis. Depending on the contained in this list the linked method may be counted as inspector (if there is no mutating character detected).

⇒ **generateExcelOutput(boolean)**: default value is true. If set to false, there will be no Excel output of the results.

⇒ **getNamingOptions()**: returns an object which contains the settings for the name generation. It contains three values: (1) **keepWholeString** to keep raw Strings and other primitive inside the name, (2) **removeAllSpecialCharacters** to remove all special characters from the name (otherwise they will be marked with a $), and (3) **replaceCharactersWithName** which will replace the all special characters with their identifying String (e.g. + becomes *plus*).

Log configurations:

⇒ **showSootLog(boolean)**: default value is false. If set to true, all the Soot logs are shown in the log.

⇒ **showInternExceptionsLog(boolean)**: default value is false. If set to true, all exceptions and errors that are thrown and cached internally are shown in the log.

⇒ **showByteBuddyExecutionErrorsLog(boolean)**: default value is false. If set to true, all exceptions and errors that are thrown and cached during the execution of the test cases trough ByteBuddy are printed in the log.

⇒ **setLogLevel(Level)**: set the level of the output logs (see Running Analysis).

Execution methods:

⇒ **start()**: The whole analysis is started with all the configurations set before. If the “pathToTestClasses” or “pathToFileClasses” is not set correctly, a “WrongSootPathException” is thrown.

⇒ **getResults()**: returns a Map with the different test case identifiers (“package.Class.method”) as keys and a list of the results as values. This method has to be called after the **start()** method otherwise, no results will be returned.

⇒ **reset()**: resets the whole analysis to run a second time. If not done, the results might not be correct.

8.2 How to use Soot

In this section, we describe how we used Soot and we give some general information about the Soot framework.

8.2.1 Create project with Soot

1. Install Eclipse: Eclipse can be downloaded from `http://www.eclipse.org`.
2. Create a new Java Project: click on `File -> New -> Project -> Java Project`, insert a name for your project and choose at least Java 1.5 to build your project.
3. Include `Soot` in your build path: download `Soot` from `https://www.sable.mcgill.ca/soot/soot_download.html`. In Eclipse right click on your project and select Properties. Go to `Java Build Path -> Libraries` and click Add External JARs... and select the downloaded soot.jar

or

1. Install Eclipse: Eclipse can be downloaded from `http://www.eclipse.org`.
2. Create a new Java Maven Project: click on `File -> New -> Project -> Maven Project`, insert group and artefact id and a name for your project and choose at least Java 1.5 to build your project.
3. Add `Soot` to your dependencies: In Eclipse open the pom.xml and add Soot to the dependencies (see listing 8.3).

```
<dependency>
  <groupId>ca.mcgill.sable</groupId>
  <artifactId>soot</artifactId>
  <version>3.1.0-SNAPSHOT</version>
</dependency>
```

Listing 8.3: Soot in tool

8.2.2 How to use Soot

```
public static void main(String[] args) {
  soot.PackManager.v()
      .getPack("jtp")
      .addTransform(new Transform("jtp.unitGraph", new BodyTransformer() {
    @Override protected void internalTransform(Body body, String phase, Map options) {
      if (!body.getMethod().getSubSignature().matches("void <init>(.*)")) {
        UnitGraph unitGraph = new EnhancedUnitGraph(body);
        TestCase testCase = new TestCase(unitGraph, unitGraph.getBody().getUnits());
        if(isATestMethod(body))
          //Has to have valid assertions and no parameters
          if (!testCase.getTestUnits().isEmpty() &&
              body.getMethod().getParameterCount()==0)
            testCases.add(testCase);
      }
```

```java
14     }
15 
16     private boolean isATestMethod(Body body) {
17       List<Tag> tags = body.getMethod().getTags();
18       for(Tag t:tags) {
19         if(t instanceof VisibilityAnnotationTag) {
20           List<AnnotationTag> annos = ((VisibilityAnnotationTag)t).getAnnotations();
21           for(AnnotationTag anno:annos) {
22             if(anno.getType().equals("Lorg/junit/Test;"))
23               return true;
24           }
25         }
26       }
27       if(body.getMethod().getName().startsWith("test") ||
             body.getMethod().getName().endsWith("Test"))
28         return true;
29       return false;
30     }
31   }));
32 
33   soot.options.Options.v().set_keep_line_number(true);
34 
35   soot.Main.v().run(getSootArgs());
36 
37   startPrivateAnalysis();
38 }
39 
40 /**
41  * Creates the different soot options needed in this analysis
42  *
43  * @return A string array containing all options needed to run soot.
44  */
45 private static String[] getSootArgs() {
46   String fileClassPath = pathToFileClasses.replace("/", File.separator);
47   String testClassPath = pathToTestClasses.replace("/", File.separator);
48   // see http://soot-build.cs.uni-paderborn.de/doc/sootoptions/
49   return new String[] {
50     // Soot runs in whole-program mode therefore scans the whole program
51     "-w",
52     // mock classes that are not found on the soot class path so we can work with them
53     "-allow-phantom-refs",
54     // don't let soot output anything
55     "-output-format", "none",
56     // whenever there is a variable name detected use the original name in jimple
57     "-p", "jb", "use-original-names:true",
58     // set project main and test classes as classes that can be found
59     "-cp", fileClassPath + File.pathSeparator + testClassPath,
60     // set test classes as application classes, of which test methods are found
61     "-process-dir", testClassPath,
62     };
63 }
```

Listing 8.4: Soot in tool

The listing 8.4, presents the main setup of Soot used for the tool.

In the method `getSootArgs()` on lines 46 to 63, you can configure the general behaviour of the Soot tool. These options are the same as the Soot command-line options. For a list of all possible command-line options, check https://soot-build.cs.uni-paderborn.de[1].

Soot's execution is divided in a set of different packs and each pack contains different phases. The pack represents one step of the execution. The different packs are represented on figure 8.1. There we can see a difference between whole program packs, which are executed over the whole project (as `cg`, `wjtp`,...) and the others, which are applied over each body separately.

On line 2 to 15, we use the `soot.PackManager` to add a Soot transform to the `jtp` pack. The transform represents a phase. Either a phase can be modified or a new one created. On line 3, we select the pack in which we want to add our transform, here we selected `jpt`, so the phase is applied to each method body individually. This pack is empty by default and is usually the one where we add the intra-procedural analyses. On line 4, we then add the needed transform. The string *jpt.unitGraph* is the id given to the transform. It must be unique and generally starts with the pack id to which it is added. And then there is the initialization of the transform. It might either be a `BodyTransform` (that is applied on each body individually) or a `SceneTransform` (applied over the whole project). In the case of the `jpt` pack, we have to use the `BodyTransform` since only the whole program packs can use the `SceneTransform`. The code we put inside the method `internalTransform()` on line 5 is then executed once Soot gets to the `jpt` pack, and is executed once for each body. For more information about Soot packs and phases, see `https://github.com/Sable/soot/wiki/Packs-and-phases-in-Soot`.

[1]https://soot-build.cs.uni-paderborn.de/public/origin/develop/soot/soot-develop/options/soot_options.htm#options

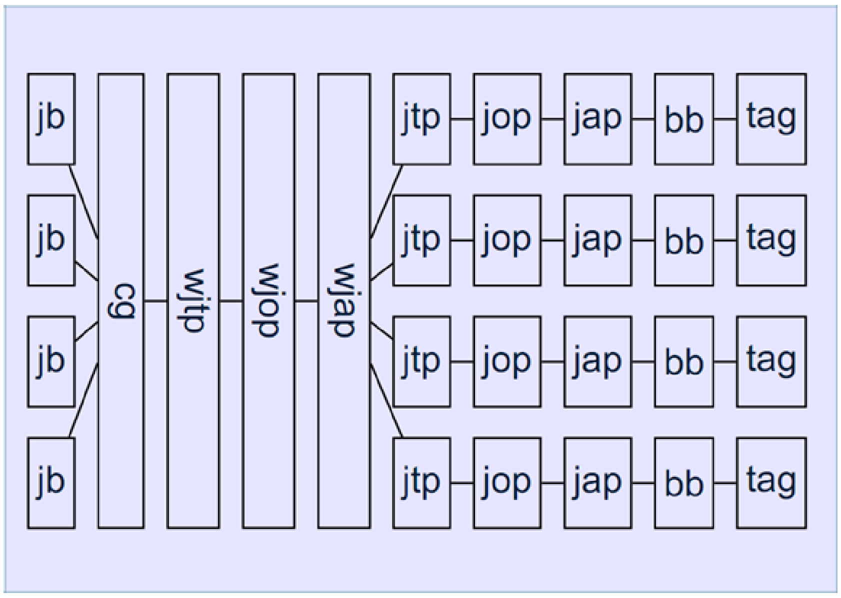


Figure 8.1: The different packs of Soot

Source: https://github.com/Sable/soot/wiki/Packs-and-phases-in-Soot

On line 6, we start to use the Soot objects. The most interesting feature here, is the `Body`, which is the jimple representation of the method. This `Body` then offers, among other things:

- `getUnits()`: returns a list of all the `Units` of the body. A `Unit` represents one operation and one line of the code.
- `getLocals()`: return a list of all variables used in the body, independent of its nature (static, temporary,...).
- `getTraps()`:
- `getMethod()`: returns the `SootMethod` of the body. This will give access to some other information:
  - `getName()` and `getSignature()`: return the method name and package.
  - `getTags()`: returns a list of all the Tags. Most of the Tags will represent the annotations of the method, but there is also a tag for the line number and other properties. Although, the same method exists for the body, most of the tags are only available inside the `SootMethod`.
  - `getExceptions()`: returns all the exceptions that the method can throw.

  - `getReturnType()` and `getParameterTypes()`: return the type that is returned by the method and the type of the different parameters.
  - `is...()`: there are many methods to check if the method is abstract, Java library (JDK), native, phantom, private, protected, public, static, synchronized,...

On line 6, we see the first use of the freshly gained `Body`. We retrieve the sub signature of the method with the help of the `SootMethod` to exclude the initialization methods of all classes. These are methods that Jimple automatically creates and should not be confused with the constructor.

On line 7, we crate a `UnitGraph`. The graph facilitates the navigation inside the `Body` of the method. But it is not necessary since everything can be implemented with the unit-, local- and trap-chains of the body. The different types of `UnitGraphs` can be found on the page `https://www.sable.mcgill.ca/soot/doc/soot/toolkits/graph/UnitGraph.html`.

From lines 17 to 26 we check the `Tags` of the method and search for the the Test annotation to identify text cases.

On line 33, we see another method to configure Soot. It is a bit easier to use but does not have all the possibilities. The Soot package `soot.options` contains different option classes that contain statics methods to configure Soot. All the options can be found on `https://www.sable.mcgill.ca/soot/doc/soot/options/package-summary.html`.

Finally, on line 35, we start the the Soot execution with the given command line arguments. The transforms are now executed in the respecting pack and phase.

One important aspect to understand and work with the Jimple bodies is the `Value`. It can either be a `Local` (a variable), a `Ref` (a read only variable), a `Constant` (primitive values and `java.lang.Strings`), or an `Expr` (an expression). This last one is more complex than the others and has many different representations that can be found on `https://www.sable.mcgill.ca/soot/doc/soot/jimple/Expr.html`. The most important ones are:

- `InvokeExpr` represents a call of a method. It can be static (example on line 24), a method declared by an interface where the actual type is unknown (example on line 9) or form a known class (example on lines 7 and 11).

- `BinopExpr` represent an atomic expression that uses two `Values`. The most known examples are`ConditionExpr` which represents all the relational operators, but we also have the arithmetic, bitwise and logical operators. Here we have examples on line 13 and 14

- `NewExpr` represents the declaration of a class as we can see on line 2.

The `Values` are then used to create the `Units` which are the ground pieces of the Soot `Body`. It represents one operation. Each line represents one `Unit`. In Jimple, the interface `Stmt` is the used implementation of a `Unit`. And here, we have again a wide variety of implementations that can be found on `https://www.sable.mcgill.ca/soot/doc/soot/jimple/Stmt.html`, but the most used ones are:

- `InvokeStmt` represents a method call where the return is ignored. These are often methods that return nothing (`void` methods) but may also be methods that do return something, but the return is not used. The `InvokeStmt` only contains one single `Value` which is of type `InvokeExpr`. In the example, we have `InvokeStmts` on the lines 3, 5 and 24.

- `AssignStmt` represent an assignment of a `Value` (called `rightOp`) to another `Value` (called `leftOp`). The `rightOp` can be any kind of `Value` that possesses a return value. The `leftOp` on the other hand, must be a `Local`. The lines 2, 7, 9, 10, 13, 16 and 21 are all `AssignStmts`.

- `IfStmt`: represents an if statement. It contains a condition `Value` and a target `Stmt`. The target is reached with the help of a `GotoStmt`. The condition on its side will be of type `ConditionExpr`. For this, we have an example on line 14 and the respecting target on line 21.

- `ReturnStmt`: represent return of the method. It has one `Value` that possesses a return value. In the example, we have it on line 26.

```
1  public void testDifferentJimpleExprAndValues() {
2    person = new project.Person;
3    specialinvoke person.<project.Person: void <init>()>();
4
5    virtualinvoke person.<project.Person: void setNumberOfFingers(int)>(10);
6
7    $stack1 = virtualinvoke person.<project.Person: java.util.List getArms()>();
8    $stack2 = interfaceinvoke $stack1.<java.util.List: java.lang.Object get(int)>(0);
9    $stack3 = (project.Arm) $stack2;
10   numberOfFingers = virtualinvoke $stack3.<project.Arm: int getFingerCount()>();
11
12   if numberOfFingers <= 4 goto label1;
13   $stack5 = 1;
14   goto label2;
15
16  label1:
17   $stack5 = 0;
18
19  label2:
20   staticinvoke <org.junit.Assert: void assertTrue(boolean)>($stack5);
21 }
```

Listing 8.5: A Jimple example

8.2.3 Links

For more information, you can look at the following sites:

- `https://soot-build.cs.uni-paderborn.de/public/origin/develop/soot/soot-develop/options/soot_options.htm`
- `https://github.com/Sable/soot/wiki/Tutorials`
- `https://www.sable.mcgill.ca/soot/doc/overview-summary.html`

Bibliography

[1] A. J. Ko, B. A. Myers, M. J. Coblenz, and H. H. Aung, “An exploratory study of how developers seek, relate, and collect relevant information during software maintenance tasks,” *IEEE Transactions on Software Engineering*, vol. 32, 2006.

[2] S. Haiduc, J. Aponte, and A. Marcus, “Supporting program comprehension with source code summarization,” *2010 ACM/IEEE 32nd International Conference on Software Engineering*, vol. 2, pp. 223–226, 2010.

[3] P. W. McBurney and C. McMillan, “Automatic documentation generation via source code summarization of method context,” in *ICPC*, 2014.

[4] L. Moreno, J. Aponte, G. Sridhara, A. Marcus, L. L. Pollock, and K. Vijay-Shanker, “Automatic generation of natural language summaries for java classes,” *2013 21st International Conference on Program Comprehension (ICPC)*, pp. 23–32, 2013.

[5] S. Rastkar, G. C. Murphy, and A. W. J. Bradley, “Generating natural language summaries for crosscutting source code concerns,” *2011 27th IEEE International Conference on Software Maintenance (ICSM)*, pp. 103–112, 2011.

[6] G. Sridhara, E. Hill, D. Muppaneni, L. L. Pollock, and K. Vijay-Shanker, “Towards automatically generating summary comments for java methods,” in *ASE*, 2010.

[7] E. W. Høst and B. M. Østvold, “Debugging method names,” in *ECOOP*, 2009.

[8] M. Allamanis, E. T. Barr, C. Bird, and C. A. Sutton, “Suggesting accurate method and class names,” in *ESEC/SIGSOFT FSE*, 2015.

[9] M. Kamimura and G. C. Murphy, “Towards generating human-oriented summaries of unit test cases,” *2013 21st International Conference on Program Comprehension (ICPC)*, pp. 215–218, 2013.

[10] B. Li, C. Vendome, M. L. Vásquez, D. Poshyvanyk, and N. A. Kraft, “Automatically documenting unit test cases,” *2016 IEEE International Conference on Software Testing, Verification and Validation (ICST)*, pp. 341–352, 2016.

[11] S. Panichella, A. Panichella, M. Beller, A. Zaidman, and H. C. Gall, "The impact of test case summaries on bug fixing performance: An empirical investigation," *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, pp. 547–558, 2016.

[12] B. Zhang, E. Hill, and J. Clause, "Towards automatically generating descriptive names for unit tests," *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 625–636, 2016.

[13] A. Trenk, "Testing on the toilet: Writing descriptive test names.." http://googletesting.blogspot.com/2014/10/testing-on-toilet-writing-descriptive.html, 2015.

[14] K. H. Bennett and V. Rajlich, "Software maintenance and evolution: a roadmap," in *ICSE - Future of SE Track*, 2000.

[15] G. Fraser and A. Arcuri, "Evosuite: automatic test suite generation for object-oriented software," in *SIGSOFT FSE*, 2011.

[16] C. Pacheco and M. D. Ernst, "Randoop: feedback-directed random testing for java," in *OOPSLA Companion*, 2007.

[17] E. Daka, J. M. Rojas, and G. Fraser, "Generating unit tests with descriptive names or: would you name your children thing1 and thing2?," in *ISSTA*, 2017.

[18] P. Hamill, *Unit Test Frameworks*. O'Reilly, first ed., 2004.

[19] G. Meszaros, *xUnit Test Patterns: Refactoring Test Code*. Addison Wesley, first ed., 2007.

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

[21] R. M. Parizi, S. P. Lee, and M. Dabbagh, "Achievements and challenges in state-of-the-art software traceability between test and code artifacts," *IEEE Transactions on Reliability*, vol. 63, pp. 913–926, 2014.

[22] M. Ghafari, C. Ghezzi, and K. Rubinov, "Automatically identifying focal methods under test in unit test cases," *2015 IEEE 15th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pp. 61–70, 2015.

[23] R. Vall233e-Rai, P. Co, E. Gagnon, L. J. Hendren, P. Lam, and V. Sundaresan, "Soot: a java bytecode optimization framework," 2010.

[24] S. Panichella, A. Panichella, M. Beller, A. Zaidman, and H. C. Gall, "The impact of test case summaries on bug fixing performance: An empirical investigation," *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, pp. 547–558, 2015.

[25] Y. Lei and J. H. Andrews, "Minimization of randomized unit test cases," *16th IEEE International Symposium on Software Reliability Engineering (ISSRE'05)*, pp. 10 pp.–276, 2005.

[26] A. Leitner, M. Oriol, A. Zeller, I. Ciupa, and B. Meyer, "Efficient unit test case minimization," in *ASE*, 2007.

[27] G. Fraser and A. Zeller, "Mutation-driven generation of unit tests and oracles," *IEEE Transactions on Software Engineering*, vol. 38, pp. 278–292, 2010.

[28] J. Xuan and M. Monperrus, "Test case purification for improving fault localization," in *SIGSOFT FSE*, 2014.

[29] G. Fraser and A. Zeller, "Exploiting common object usage in test case generation," *2011 Fourth IEEE International Conference on Software Testing, Verification and Validation*, pp. 80–89, 2011.

[30] S. Afshan, P. McMinn, and M. Stevenson, "Evolving readable string test inputs using a natural language model to reduce human oracle cost," *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation*, pp. 352–361, 2013.

[31] E. Daka, J. Campos, G. Fraser, J. Dorn, and W. Weimer, "Modeling readability to improve unit tests," in *ESEC/SIGSOFT FSE*, 2015.

[32] F. Palomba, A. Panichella, A. Zaidman, R. Oliveto, and A. D. Lucia, "Automatic test case generation: what if test code quality matters?," in *ISSTA*, 2016.

[33] R. Osherove, "The art of unit testing: With examples in .net," 2009.

[34] D. Gonzalez, S. Prentice, and M. Mirakhorli, "A fine-grained approach for automated conversion of junit assertions to english," in *NL4SE@ESEC/SIGSOFT FSE*, 2018.

[35] A. Kuhn, B. V. Rompaey, L. Haensenberger, O. Nierstrasz, S. Demeyer, M. Gälli, and K. V. Leemput, "Jexample: Exploiting dependencies between tests to improve defect localization," in *XP*, 2008.

[36] L. Hänsenberger, "Jexample," March 2008.

[37] L. Hänsenberger, "Defect isolation as responsibility of the framework," Master's thesis, University of Bern, September 2009.