Bachelor Thesis

August 21, 2017

TestDescriber

Generating comprehensible Test Cases

Antonio Galluccio-Saez

of Barcelona, Spain (06-197-271)

supervised by

Prof. Dr. Harald C. Gall

Dr. Sebastiano Panichella

University of Zurich[UZH]

s.e.a.l.

software evolution & architecture lab

Bachelor Thesis

TestDescriber

Generating comprehensible Test Cases

Antonio Galluccio-Saez

University of Zurich[UZH]

s.e.a.l.
software evolution & architecture lab

Bachelor Thesis

Author: Antonio Galluccio-Saez, antonio.gallucciosaez@uzh.ch

URL: https://github.com/antoio/TestDescriber

Project period: 20.02.2017 - 20.08.2017

Software Evolution & Architecture Lab
Department of Informatics, University of Zurich

Acknowledgements

The author, Antonio Galluccio-Saez, wants to thank Professor Dr. Harald C. Gall for supervising this bachelor thesis and especially Sebastiano Panichella and Carmine Vasallo for the permanent support and supervision of- and time invested in this bachelor thesis.

Abstract

Nowadays, it is very common in software development to write code that will change multiple times over a rather short period of time, depending on the business requirements. As this is an integral part of agile software development, it is very important for developers to be able to change parts of a program without introducing new errors that would compromise the existing system. Because of this, software tests have become a vital part of software development, too. However, the task of testing the developed software takes up a lot of time for a programmer and even becomes a great part of the software project itself. In fact, the creation and maintenance of software tests can take up to 50% of the overall project effort [**?**]. That is why automatic test generation suites like EvoSuite or JTest, etc... exist. These tools are able to automatically create unit tests that can be useful to increase test coverage of a project and help in reducing the test creation time. Unfortunately, the software developer still needs to verify the created tests and check whether they are correct and cover all the important parts of the production code. As these tests are automatically generated, it isn't given that they are easily understandable and the checking of the automatically generated tests always involves looking up the relevant parts in the production code as well. This is where TestDescriber aims to help in the development cycle. The tool tries to add comments in natural language to automatically generated tests, that are supposed to help the developer gain a faster and better understanding of the test code before him and therefore make it easier to find bugs and modify the created test cases.

Zusammenfassung

Heutzutage ist es üblich, dass sich der Quellcode eines Projekts während dem Softwareentwicklungszyklus innerhalb kürzester Zeit mehrere Male ändert, immer abhängig von den Business Requirements. Diese Eigenschaft ist ein Teil der agilen Softwareentwicklung und macht es deswegen für den Entwickler umso wichtiger, dass er in der Lage ist den Quellcode anzupassen oder umzuschreiben ohne dabei neue Fehler einzuführen oder ein laufendes System zu kompromittieren. Aufgrunddessen sind Softwaretests ein wichtiger Bestandteil für Softwareentwickler geworden. Allerdings benötigt ein Entwickler zum Schreiben von Softwaretests sehr viel Zeit und die Softwaretests erweitern ein Projekt auch um mehr Komponenten, welche gewartet werden müssen. Tatsächlich ist in der Industrie die Aussage verbreitet, dass das Schreiben und Warten von Softwaretests bis zu 50% der Zeit eines Programmierers in Anspruch nimmt [**?**]. Deswegen gibt es automatische Testgenerationstools wie EvoSuite oder JTest, etc... . Diese Tools sind dazu in der Lage automatisch Softwaretests zu erstellen, was zu erhöhter Codeabdeckung führen kann und die Zeit zur Erstellung der Softwaretests signifikant verringert. Unglücklicherweise sind diese automatisch generierten Tests aber nicht fehlerfrei und somit benötigen sie immernoch einen Programmierer der die Gültigkeit der Tests überprüft. Ausserdem sind sie nicht immer einfach verständlich und oftmals ist der Prüfer der Tests gezwungen sich durch den ursprünglichen Produktionscode zu lesen um zu verstehen was der generierte Test überhaupt macht. Mithilfe von TestDescriber soll den Entwicklern geholfen werden Tests leichter zu verstehen indem automatisch generierte Kommentare in natürlicher Sprache in die Testklasse eingeführt werden. Dadurch soll es dem Entwickler gelingen Tests leichter zu verstehen und mehr Bugs zu finden.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Since agile software development has become more and more ubiquitous in todays software development projects the emphasis on it's core advantages is very strong. By being able to deliver working, deployable and tested software on an incremental basis notably reduces project risk. Because of the many changes that can occur on the software project in an agile environment it is essentially vital that the software has been tested thoroughly. These tests, however, have to be written by one or more developers which means that they have to divide their time and attention between writing, improving and maintaining the regular code base of the software product as well as the corresponding test cases for it. Automatic test generators can be used to hugely improve code coverage of tests in general and reduce the time developers need to spend to write the tests at all. Drawbacks of these methods are that these tests are hard to really understand and maintain. Also the naming of code parts like variables and methods often follow an algorithmic logic, which can be confusing for a human to try to follow and understand. Apart from that it is almost always necessary to have a look at the production code in order to verify if the generated test case does make sense and is feasible. By considering all these aspects it becomes clear that the automatic test generation alone does not provide enough comfort in a software project to use it in the software development life cycle without any intervention of a software developer. But given that a developer is still needed for maintaining the software tests his time spent on creating them is drastically cut by the test generators. Under these circumstances it would be nice to improve the developers ability to understand automatically generated tests better by providing him with additional information on the test cases and the generated code. Based on the former work by Sebastiano Panichella *et al.* [**?**] this bachelor thesis built upon the approach coined TestDescriber to create an *Eclipse Plugin* that is able to annotate test classes with comments and information that can help to better understand the test class and to give a better overview of the test class in relation to other test classes. The approach can be divided into three different steps the first being the creation of the test classes, either manually or automatically, the second step consists of analysing the coverage to gather enough information for the third step which finally parses the actual code written in the test class and the production code and creates comments in natural language mixed with all the collected information. In addition to the original approach the aim of this bachelor thesis was to build upon it and create a simple to use tool that is available to everybody and can be easily used and integrated into any project. Since *Eclipse* is a widespread Integrated Development Environment (IDE), to create an *Eclipse Plugin* was a logical choice. It is possible to easily add it to the existing *Eclipse* environment and to run it on the current tests in a project. The created comments are added to the existing test class, thus giving the developer more information on how the test works as well as meta information on how much lines the test case covers in the production code and how significant it is compared to other test cases in the same class. The benefit of this addition should be for the developer to generally understand tests better, to help him find more bugs and even maintaining tests easier in the long term.

Chapter 2

Related Works

There exist a lot of different ideas and theoretical assumptions on how a developer can be supported during the task of testing and how these tests should be approached to be more efficient, to cover more production code or to expose more bugs. As a part of this work a few representative papers were selected to support the hypothesis that automatically generated, additionally added information can be used to easier understand a test and find more bugs in the long run.
Based on a large scale field study the authors Beller, Gousios and Panichella, Zaidman [**?**] explored the question how much code developers *are* testing and *should* test in general. They found out that even though a majority of software developers think they spend more than 50% of their productive time on engineering software tests in reality the time spent circles around a quarter of their time and tests are neither as thorough nor as extensive as the average software developer perceives them to be. The paper revealed that most participants didn't strictly follow the TDD rules. In the study conducted developers were mostly concerned with adding new features, bug fixing or refactoring the production code but the tests and the production code didn't co-evolve. Instead the immediate reaction of developers after a failing test tended to be to make adjustments in the production code thus showing that the TDD process is not followed strictly and for all the modifications made. Possible explanations for this behavior varied between uselessness of the strict process in relation to the changes made to the production code up to misunderstanding of the TDD procedure by developers in general. However, against common perception the researchers also noted that instead of the often assumed 50% of time of a developer taken up by testing it were in fact only 25% of their time, showing that developers spent less time engineering tests in general than they actually thought they would. Obviously the consequences of this realization can lead up to more error prone software and less qualitative software products.
Although there exist tools nowadays that can help a developer generate tests automatically, thereby reducing the work load of the developer, recent studies have highlighted that automatically generated tests are not necessarily easier to understand [**?**] due to their randomized, algorithmic nature [**?**]. Thus, as a consequence, generated tests do not improve the ability of the developer to detect bugs when compared to manual testing. On this premise we want to present TestDescriber, a tool that is able to summarize automatically generated- as well as manually written JUnit tests.

Chapter 3

Approach

The main goal of this thesis is to create an *Eclipse Plugin* that is built upon the TestDescriber approach as described by Panichella *et al.* [**?**] and to be able to integrate the tool into an existing project in the *Eclipse* IDE.

3.1 Approach Overview

Since TestDescriber is a tool that should help developers with their testing process, it comes into play when the production code and test cases already exist. Whereas the production code has to be programmed according to the business requirements, the test cases can either be designed and written manually or be generated automatically. With the aid of existing tools like EvoSuite [**?**] or Randoop [**?**], it is fairly easy to create a test class for each production code class. For TestDescriber to be able to compute the summaries, it will need the production code class from here on referred to as class under test (CUT) and the corresponding test class as input.
Even though TestDescriber is also able to run on manually written test classes the original approach used EvoSuite[1] for the test generation. As this approach is bound to give a more uniform test output we also relied on EvoSuite generated test classes.
The whole TestDescriber run cycle is built upon following steps:

- ***1.*** *Test execution and coverage analysis:*

  The test class is executed by Maven [2] and the test coverage is analyzed by Cobertura [3]. Cobertura tries to gather information about the CUT that will be summarized in the end.

- ***2.*** *Summary generation:*

  Based on the information gathered in the step before, the summaries are generated and virtually added to the test class. To each test case, the percentage of how many lines of code in the production code were covered is added.

- ***3.*** *Summary aggregation and clean up:*

  In the final step, the virtually added summaries are extended with coverage information about how high a percentage of this test case is also covered by the other test cases in the

[1] http://www.evosuite.org/
[2] https://maven.apache.org
[3] https://github.com/cobertura/cobertura

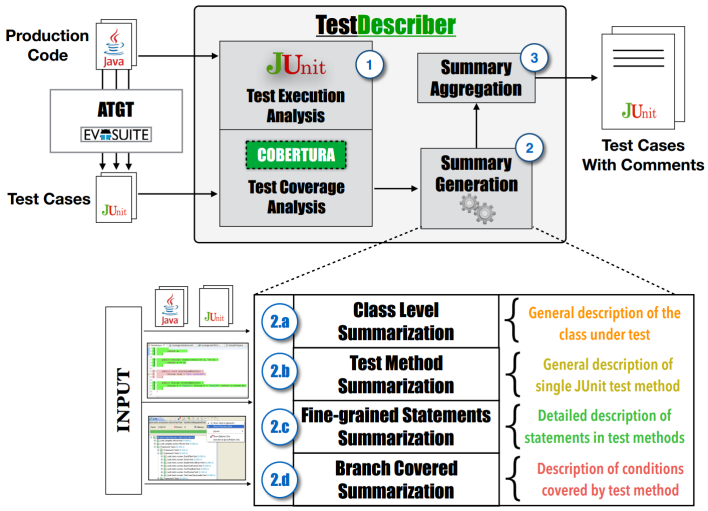

Figure 3.1: Overview of the TestDescriber approach

same test class. Additionally, if during the test execution phase an exception arises, any information about it is added to the line that threw the exception. Finally, the summaries are cleaned up and written into the original test class, thereby replacing the old test class.

3.2 Step 1: Execution and Analysis

The TestDescriber *Eclipse Plugin* relies heavily on Maven for the tests to run and to calculate the coverage of each production class. Therefore, for TestDescriber to work it is a necessary requirement that the project is a valid Maven project and follows the standard Maven project structure. For the coverage analysis, the Cobertura Maven Plugin[4] was used. On the first run of the plugin, the *'pom.xml'* of the project gets copied into a temporary alternative *Pom File* with the dependencies for the Cobertura Maven Plugin injected and will furtheron be used from TestDescriber.

3.2.1 Test Execution

As a preparation step for the coverage analysis, it is important to run the current test class. The goal is to collect information on the various code parts that are covered by each test case. This information builds the main vocabulary that is required for the generation of the summaries later on. Hence, the TestDescriber Plugin relies heavily on the Cobertura analysis to exactly identify programming statements and code branches (e.g. If-Statements) that are executed in the CUT by each individual test case. Additionally, by executing each test class, a report about the test results is created as part of the Maven Surefire Plugin[5] in case of a test failure or in case an exception occurs. In this report, various information is collected and is used in the later stages of the plugin to identify the exact line where an exception occured in the test class and what the problem of the exception was.

[4]http://www.mojohaus.org/cobertura-maven-plugin/
[5]http://maven.apache.org/surefire/maven-surefire-plugin/

3.2.2 Coverage Analysis

The coverage analysis heavily relies on an extended Parser that is built upon the project JavaParser[6] and follows the following steps:

- **I.** Extract a list of all methods of the CUT that are invoked by the test case
- **II.** Extract a list of statements of the CUT along with all class attributes and variables
- **III.** Extract any indirect calls to other methods of the CUT
- **IV.** Add the boolean values to the statements depending on the corresponding If-Statement

With the help of the coverage analysis information on each code element is extracted from the coverage report. The coverage report is stored in a File called *'coverage.xml'* in the project folder and is basically the Cobertura analysis. It contains information about the amount of hits on each line and about whether a line got executed by being in the 'true'- or 'false'-branch of an If-Statement. By extracting this information on each class, TestDescriber gets a detailed "description" of the parts that were actually executed of the CUT. At this point, it is possible to exactly identify the parts relevant to the corresponding test case based on the line numbers extracted from Cobertura.

3.3 Step 2: Summary Generation

The main goal of the summary generation is to provide the developer with an abstract high-level view of which part of the CUT each test case is going to execute. In order to achieve this abstract view, TestDescriber uses an approach called the Software Word Usage Model (SWUM), which was proposed by Hill *et al.* [**?**]. The idea of SWUM is that *actions, themes* and any *secondary arguments* can be deduced from an arbitrary fragment of code by making assumptions about the most common Java naming conventions and effectively using these assumptions to connect linguistic information to programming language structures and semantics. In fact, method signatures (including class names, method names, data types and formal parameters) as well as field signatures (including class name, data type and field name) almost always contain *verbs (actions), nouns (themes)* and *prepositional phrases (secondary arguments)* that can be used in a bigger context in order to generate a readable natural language sentence.
Consequently, TestDescriber breaks up the identifier names into single sentence parts (e.g. method names like *'getValue'* can be used as 'get value') and expands abbreviations to type names and identifiers using on the one hand an external dictionary of common short forms for English words [**?**] and on the other hand a more sophisticated technique called *'contextual-based expansion'* [**?**], where it tries to guess the most appropriate expansion for a given abbreviation. After the main terms are acquired from the identifier names, the Plugin makes use of *Language Tool* [7], a Part-of-Speech (POS) tagger to determine which word is a noun, a verb or an adjective. With the output of the POS tagger, it is further determined whether a word should be treated as a noun phrase, a verb phrase or a prepositional phrase and based on that category, a sentence in natural language is formed.

[6] https://github.com/javaparser/javaparser
[7] https://github.com/languagetool-org/languagetool

Finally, a template is used to annotate the test classes. These templates can be divided into four main categories at different levels:

- ***i.*** *Class Level Summaries*
- ***ii.*** *Method Level Summaries*
- ***iii.*** *Detail Statement Level Summaries*
- ***iv.*** *Branch Coverage Summaries*

The difference between each level of summaries will be explained in the following sections.

3.3.1 Class Summary

Class level summaries give the developer a quick outline of all the responsibilities of the CUT. Thus, if a CUT is not well documented or sparsely commented in general it will still give the developer a quick hint of what the CUT has to manage. In this respect, TestDescriber adopts an approach postulated by Moreno *et al.* [**?**] that formulates an idea for summarizing Java classes by taking into account the most relevant methods of the CUT, the super class and class interfaces and the role of the class itself within the system. Specifically, TestDescriber tries to set its focus mainly on the interfaces and the attributes of the CUT. The more detailed summaries for the CUT follow later on in the paper.

3.3.2 Method Summary

On a method level, all the information in the Cobertura report is gathered in one place, thus describing the coverage of each statement in the method as both a coverage score and in terms of lines covered in the CUT. The first line of each method comment shows the percentage of statements covered in the CUT by the given test method independent of all the other test methods. Additionally, TestDescriber also adds information on which other test cases in the same test class execute the same lines, hence presenting the developer with the percentage of the same lines covered by other test cases and their relating line numbers. By using this approach, it is possible to generate a general description of the statement coverage scores reached by each JUnit test method.

3.3.3 Statement summary

As has already been described above in reference to the fine-grained statement summarization, the plugin extracts the detailed list of *code elements* (e.g. methods, attributes or local variables) that make up each statement of the CUT and are covered by each JUnit test method. The information collected in this manner builds the input for the 'Detail Statement Level Summaries' and TestDescriber processes them in the following three steps:

- ***1.*** *Parse all the instructions contained in a test method*
- ***2.*** *Use the SWUM methodology for each instruction and classify to which expression it belongs (e.g. does it declare a variable, does it use a constructor or a method, does it use a specific assertion, etc...) and determine which part of the code is executed*
- ***3.*** *Generate a set of natural language sentences in relation to the selected kind of instructions*

```
1  /**
2   * The main class under test is TemperatureSaver.
3   *  It describes a single temperature saver and maintains information
4  regarding:
5   *  - the far of the temperature saver;
6   *  - the lower of the temperature saver;
7   * */
8  public class TemperatureSaverTest {
9  /**OVERVIEW: The test case "testTemperatureSaverDouble" covers around
10 *33.333336% [13,17,18,20,23,29] of statements.
11 *testTemperatureSaver covers 33% of this test. Same lines [23,29]
12 *testTemperatureSaverConversion covers 67% of this test. Same lines
13 [13, 17, 18, 20]*/
14 @Test
15 public void testTemperatureSaverDouble() {
16 //The test case instantiates a "TemperatureSaver" with  celsius
17 //temperature equal to 28.0, and  fahrenheit temperature equal to 0.0.
18 //The execution of this constructor implicitly covers the following 1
19 //conditions:
20 // - the condition  "celsius temperature equals to 0.0" is FALSE;
21 TemperatureSaver con1 = new  TemperatureSaver(28.0, 0.0);
22 //Then, it tests:
23 //1) whether the fahrenheit of con1 is equal to 82.4 with delta equal
24 //to 1.0; --> java.lang.AssertionError: expected:<82.4> but was:<0.0>
25 assertEquals(82.4, con1.getFahrenheit(), 1.0);
26 //2) whether the celsius of con1 is equal to 0.0 with delta equal to
27 //0.0;
28 assertEquals(0.0, con1.getCelsius(), 0.0);
29     }
30 }
```

Figure 3.2: Different summary levels created on a manually written class (see Fig. 4.1 for explanation)

In this context, it classifies each statement into one of the following three categories:

- *Constructor of a Class*
  The constructor generally implies an instantiation of an object and may contain a couple of parameters. In more abstract terms, this corresponds to an implicit action on a theme with secondary arguments. TestDescriber looks for the correct declaration of the constructor in the CUT and tries to map the *formal* and *actual* arguments. Then it uses the terms tagged by the POS to generate specific natural language sentences according to a certain template.

- *Method Call*
  The naming of method calls usually involves a verb determining the *action* while the method caller and the parameters stand for the *theme* and *secondary arguments* each respectively. For getters and setters, specific templates are used and TestDescriber does notice if the return value of a method is assigned to a local variable and in consequence adjusts the description.

- *Assertion*
  In order to generate summaries based on assertions, a test oracle is introduced into the process that gives TestDescriber the ability to test whether a CUT's feature behaves as expected. While the name of the assertion further specifies what kind of test is intended (e.g. assertEquals, assertFalse, etc...) the arguments to it both state (i) the expected and (ii) the actual behavior. Based on each assertion type, TestDescriber relies on different natural language templates. Moreover, in case of an assertion failure the Plugin also analyses the test log and displays the thrown exception on top of the failed assert statement including the exception type.

Furthermore, another summary category is included that does not fit into any of the categories above but is worth mentioning: branch coverage. These summaries are generated during runtime when the execution of a method evaluates an If-condition to decide which branch to take (here either 'true' or 'false'). For the summary, the boolean expression is extracted from the If-statement and, in the manner of the summaries before, a natural language sentence is created.

3.4 Step 3: Summary Aggregation

In the final step of TestDescriber, all collected information and all created summaries get consolidated and are added to the class. The information is added as different inline comments: all *Class Level* and *Method Level* comments are added as block comments right before the declaration of each respectively. The more detailed *Statement Level* summaries and *Branch Coverage* are inserted right before the complementing statement they are summarizing. The summaries are inserted in memory first before being written into a new test class. The new test class overwrites the old test class with all the additional information.

Chapter 4

How the tool works

4.1 Architecture

With respect to the architecture of the whole program, it is important to differentiate between the TestDescriber back end and the front end. Whereas the back end consists of the original TestDescriber approach as proposed by Panichella *et al.* in [**?**] and the extended structure of the program in order to give it easier possibilities to connect with the outside, the front end mainly encompasses the eclipse plugin and handles the integration into an existing project in the *Eclipse* IDE.

- *Back End*
  Built on top of the TestDescriber approach was a new architecture based on the *State Pattern* to easily extend the back end with new functionality and the *Decorator Pattern* to offer different entry points into the program and to use parts of it like building blocks to create new behavior or adjust it according to new requirements.
- *Front End*
  This component makes up mainly the user interface of the plugin which was kept intentionally simple. Nevertheless, some program logic had to be built in because the plugin directly resides in the *Eclipse* IDE and is responsible for the discovery of the test classes and the complementing production code classes.

Also noteworthy are the preconditions we had to assume in order to be able to find the test classes and to match them to the correct production code classes.

- *Assumption 1*
  All the projects that the *Eclipse* plugin can run on have to adhere to the standard Maven project structure (e.g. all tests have to be under *'src/test/java'*).
- *Assumption 2*
  All the tests are required to have the word 'Test' in their file- / class name. With tests automatically generated with EvoSuite, this is already standard.

Thus, the first assumption we made was that the TestDescriber Plugin could only be used in a Maven project. On the one hand, in a Maven project the project structure is fixed and all test classes have to be in a folder under *'src/java/test'*. On the other hand, the TestDescriber core relies on the Cobertura Plugin for Maven which therefore needs to be added to the project's *Pom File*.

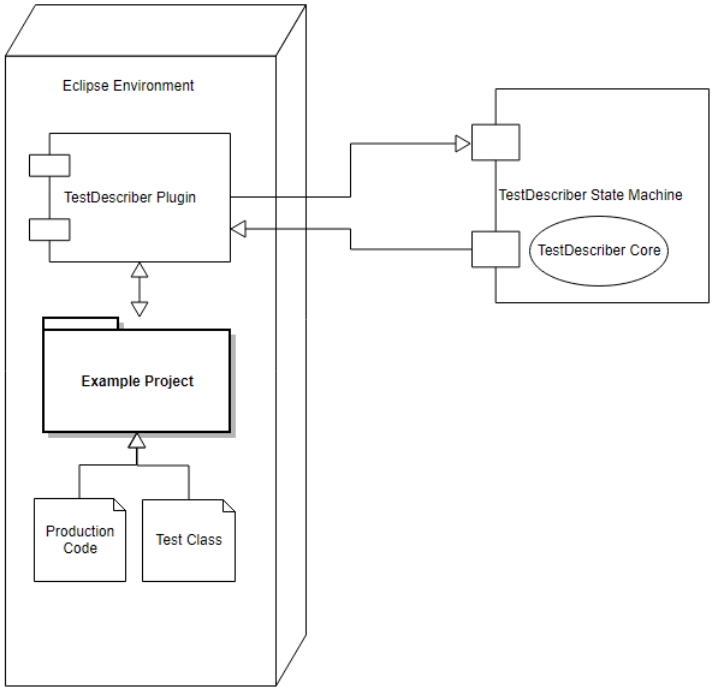

Figure 4.1: Component diagram of the new TestDescriber environment

4.1.1 Design Overview

In this section, it will be explained how the TestDescriber core and the Plugin work together and are connected. The main goal is to give a high-level outline of the single components and to show the different entry points to the TestDescriber core as well as the executive domain of the plugin.

4.1.2 Architectural Decisions

As the Plugin grew over time, a few important design decisions had to be made, taking into consideration future extensions to the program. A very important step was the introduction of the *State Pattern*. By breaking up the execution of the original approach and dividing it into different *states*, it was possible to reach a design where a new feature could be integrated into the program flow very easily. All a developer needs to do is create a new *state*, encapsulate the new functionality in a separate class (to keep the responsibility of the class separated from the state) and finally add the created *state* into the desired order of the state machine.

The state machine client offers different entry points (plugin, command line, TestDescriber unit, etc...) and uses the decorator pattern to combine single steps of TestDescriber into a new feature. Still, as some steps, namely the TestDescriber core execution flow, are dependent on each other, it is not possible to interchange them in whatever way needed.

In fact, the core has to stay grouped together so that the right steps are executed at the right time and each step has the necessary information it needs to run its computation. When presented with this problem, it seemed to make more sense to have a dedicated object called the *Configuration Ob-*

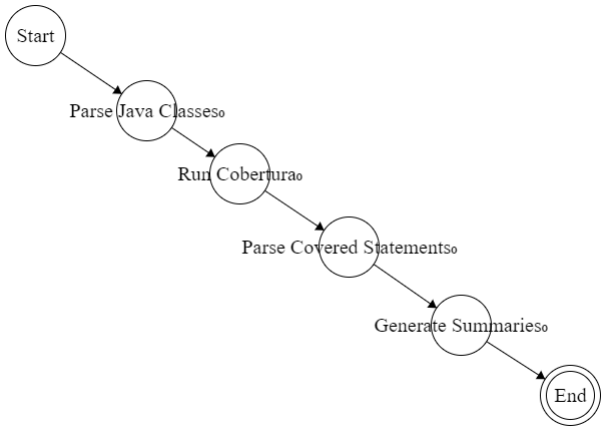

Figure 4.2: Automaton of the steps of the original TestDescriber approach

ject that can be initialized, saves intermediary computation results and keeps meta information about the end result (file path, file name, etc.). The *Configuration Object* is passed around between each step of the execution flow and filled or read accordingly.

Internally, to make the different entry points into the TestDescriber core possible, all the information the program relies on had to be identified and decoupled from the TestDescriber logic, consequently leading to a data interchange object called a *TestDescriber Unit*. This unit wraps all the information that the Plugin extracts from the *Eclipse* project, the production code and the test classes (e.g. file path, file name, package name, etc...). Thus it acts as a data exchange format for the interface between the TestDescriber core and the *Eclipse* Plugin.

State Machine Pattern

The original TestDescriber approach was based on a set amount of steps as already described in *'Chapter 2: Approach'*. Therefore, in order to add the new functionality and adapt the program into a more versatile tool, it was necessary to be able to break up that fixed structure into movable parts. For this, the use of the *State Pattern* was obvious.

In short, the *State Pattern* can be summarized as a behavioral software design pattern that implements a state machine in an object-oriented way. With the state pattern, a state machine is implemented by creating each individual state as a derived class of the state interface, and implementing state transitions by invoking methods defined by the state machines superclass.
The state pattern can be seen as a strategy pattern which is able to switch the current strategy through calling methods defined in the state interface.

If the original TestDescriber approach was seen as a machine, each individual phase of it could intuitively be translated into a state of that machine. Even though some of these states are chronologically strongly dependent upon each other, a strong emphasis lies on the possibility to integrate a new state without taking apart the whole machine. In the *State Pattern*, an addition like this is done with almost no effort. A more thorough explanation follows below.

On the downside, it is inevitable that each step needs information before it is able to start its com-

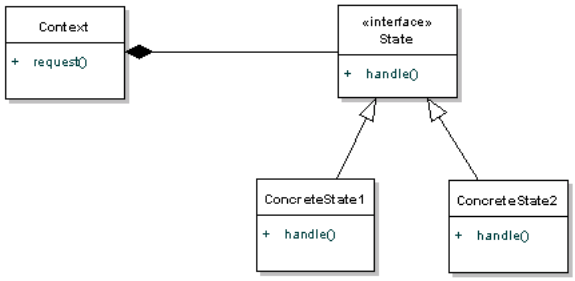

Figure 4.3: State Pattern Class Diagram [1]

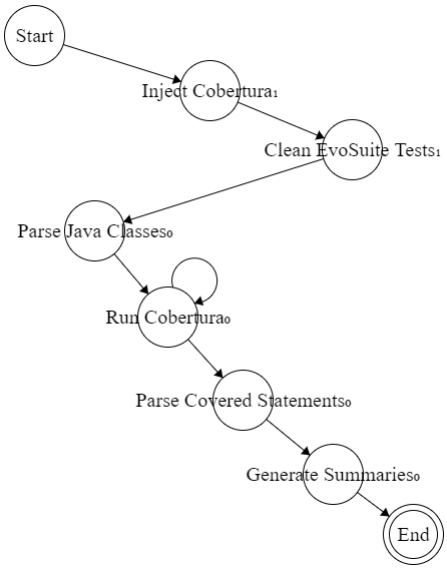

Figure 4.4: Automaton of the steps of the new TestDescriber Core

putation and it needs to store the its results after it finishes. This coupling between the different states could not entirely be resolved. As a consequence, each state either keeps its information and offers an interface for other states to access them, or a central information repository is needed. Since the states should be as uniform as possible, we decided to use the information repository approach which we called *Configuration Object*.

Adding a new Step

Through the adoption of the *State Pattern*, the addition of a new step is not only easy but also follows a strict set of working steps. If followed correctly, the program can easily be extended with new functionality. The necessary steps are as follows:

- ***1.*** Create a new state
  Create a new class that implements the *State Interface* and import the needed method *'executeState()'*. As this method builds the entry point into the created state, it receives the responsibility of the work flow during the execution of the state. Consequently, the method is tasked with a couple of responsibilities that are not enforced but should be followed *by convention*. These are mainly:
  - *Pre-Stage:*
    When entering the method, the current state needs to be prepared to have the information necessary for the follow-up stage. Therefore, all of the needed information from the *Configuration Object* should be loaded into local variables.
  - *Calculation Stage:*
    After the preparation, the information loaded into the local variables can be passed on to the logic that should get executed during this stage. As already mentioned, it is advised to keep any functionality into an object of its own. Then, all that remains is create an instance of the object and pass it the local information for its calculation.
  - *Post-Stage:*
    When the calculation is completed, all results should be saved again in the *Configuration Object*. If needed any verification of the result or clean-up of the state should be done in this stage. After this stage is completed, the *'executeState()'* method will return and give the execution back to the client.
- ***2.*** Add the new state to the context
  The state machine context needs to hold a reference to the new state and an extension to serve the new state when needed. Thus, a new *'getFooState()'* ('Foo' is a place holder) has to be added to the context. Also, in order to ensure that only one object gets created during this run of TestDescriber a *guard* has to be added to the method. The return value type is the type of the interface, hence adhering to the state pattern.
- ***3.*** Call the new state from the Client
  In the client of the state machine, the context switch method and the execute state method have to be placed as wished. At this point, the advantages of the state pattern are shown. The new functionality can, without effort, be inserted wherever it is needed. Since all states in the execution of TestDescriber use the same information repository, all calculation results up until that point are available. After the new step concludes, the client will call the context switch method again and execute the next step.
- ***4.*** (optional) Encapsulate the new functionality into a class of its own
  Even though this step is optional it is **strongly advised**. The idea is simply to develop a new feature or functionality separated from the whole program. If it is encapsulated into a class

of its own it becomes a lot easier to test the developed feature or functionality and to ensure that all necessary logic is contained within that single class. This follows a common programming principle called the *Single Responsibility Principle (SRP)* and ensures that classes do not grow too large and the program can still be maintained even when it is still growing. In addition, the developer has to thoroughly consider what kind of information is needed for his feature or functionality which makes the information exchange a lot easier.

Configuration Object

This object's purpose is to keep all the information from the current TestDescriber execution in one place, thus allowing it to be able to get information from different sources and distribute the information to everybody who asks for it. The classification of the different information is as follows:

Description	Detail
File Information	
	Source Folder
	Test Folder
	Output Folder for Result
	Package Name of Production Code and Test Class
	File Name
	Production Class Name
	Test Class Name
Cobertura Information	
	Path to coverage XML File
	Path to POM File
	Test coverage of current Production Code
Computation Results	
	Path to Resulting Class
	All Test Cases
	Effectively executed Production Class based on coverage
	Test Case Annotator
UI Information	
	Progress Monitor

Table 1: Meta information kept in the *Configuration Object*

4.1.3 TestDescriber Core

During the creation of this thesis, the original TestDescriber approach was substantially extended. The monolithic prototype was opened and split up into separate more independent parts. The input configuration for the program was worked out and designed as a data type of its own in order to create an interface for other programs. In this sense, the TestDescriber back end now can be run over the command line or the plugin but the connection with another type of front end, for example a web interface, would be possible as well.

Because the original approach relied on a locally installed version of Cobertura that was run over the command line and could lead to performance problems or runtime errors, new functionality

was introduced to run Cobertura as a Maven plugin. That required the ability to run a Maven command programmatically during runtime execution of TestDescriber and on the other hand it was necessary to inject the Cobertura plugin into the project *POM File*. Since this File is a very important part for a software project and should not be changed by an extension or a plugin in general, our solution includes a copy of the *POM File* with the cobertura plugin code injected. That way, the project structure and configuration stays unchanged but Cobertura can still be invoked for the coverage analysis. In addition, the invocation of the Cobertura plugin through Maven requires an additional plugin from Apache called Maven Invoker [2]. With this library, it was possible to invoke the appropriate Maven command and execute the desired goal.

Besides extending TestDescriber, also a new project structure was introduced so as to create a clearer path through the execution of TestDescriber. In terms of maintainability, this should prove more useful and will provide guidance for new developers.
Additionally, some changes to the core of the original approach had to be made for bug fixing purposes. These changes included the addition of Abstract Syntax Tree (AST) parsers based on the *Eclipse* AST[3] and ensured that TestDescriber could run on an entire test class without interruption.

4.1.4 TestDescriber Plugin

In this section, the details of the plugin are described and outlined. The emphasis lies on the different parts that make up the plugin as well as the connection between the plugin and the TestDescriber back end. Since the plugin is made for *Eclipse*, a short description of the IDE environment will help understand how the plugin works and interacts with its surroundings. For easier handling, TestDescriber is added to the regular *Eclipse* context menu.

Eclipse Environment

The plugin itself relies heavily on the resources that *Eclipse* provides. The developers of *Eclipse* always intended the IDE to be extensible, thus building in a lot of extension points and other services that can be used to interact with the entire IDE. Because TestDescriber needs as an input the production code and test classes, it is required that a project is developed or imported into the IDE beforehand. The IDE itself then provides services, access points and resources that the plugin can use. For example, a file in the IDE counts as resource while a selection in the IDE is handled by a Selection Service and needs to be queried to get a result. In particular, the plugin itself needs to be able to recognize a project, a package and a single file. But with knowing the type of the resource, it will not be able to achieve much. Another important point is the file path to the specific resource that is needed for the TestDescriber core. Since TestDescriber originally modified files on the file system, the plugin only collects enough information for the core to do just that. A good example of this strategy is the *File Collector* that was created to go through all the test classes and find the complementing production code classes. The matching is based on the file name of the test and the production code. Firstly, the test class is required to have the word *'test'* in it and the matching criteria is not case sensitive. Secondly, the name of the production code class has to be equal to the name of the test class without the word *'test'* in it. If the *File Collector* can create such pairs, it creates the *Testdescriber Units* and keeps them in a collection for the subsequent hand-off to the TestDescriber core. These production code and test class pairs will further on be abbreviated as "PCTC".

[2] http://maven.apache.org/shared/maven-invoker/index.html
[3] https://www.eclipse.org/articles/article.php?file=Article-JavaCodeManipulation$_AST/index.html$

Context Menu

In order to easily give access to TestDescriber, we decided to put the entry point to the tool in the *Eclipse* context menu. By adopting this way of handling the start of TestDescriber, it became possible to make the plugin context sensitive. In other words, depending on whether the user selects the entire project, a specific package or a single test class, the *File Collector* will either start creating PCTC for all files in the project or only for the files in the selected context. This choice will likewise influence the user interface since the user is given another chance to choose which tests should be analysed by TestDescriber if more than one pair were found. On a side note, we also added a descriptive icon to the context menu entry to make the entry easy recognizable.

User Interface

The user interface of the TestDescriber plugin consists of different components.

- *i. Context Menu*
- *ii. Selection List Dialog*
- *iii. Progress Bar*

As we already covered point *i. Context Menu*, we will proceed with the next point on the list.

The Selection List Dialog is presented after the entire project was selected or a specific package, once the *File Collector* has found at least two PCTC pairs. In that case, the name of the production code classes will be presented to the user in order for him to decide whether he really wants TestDescriber to run on all PCTC or only on specific ones.

As soon as the plugin is finished collecting all information, it hands the execution as well as the necessary *Testdescriber Units* over to the TestDescriber core which, in turn, will start to run on the input files. During this time, a progress bar is presented to the user indication on which class TestDescriber is running right now and how far it has progressed in the analysis of the classes. After the successful termination of the TestDescriber run, the user is presented with a message box informing them that the computation has been completed.

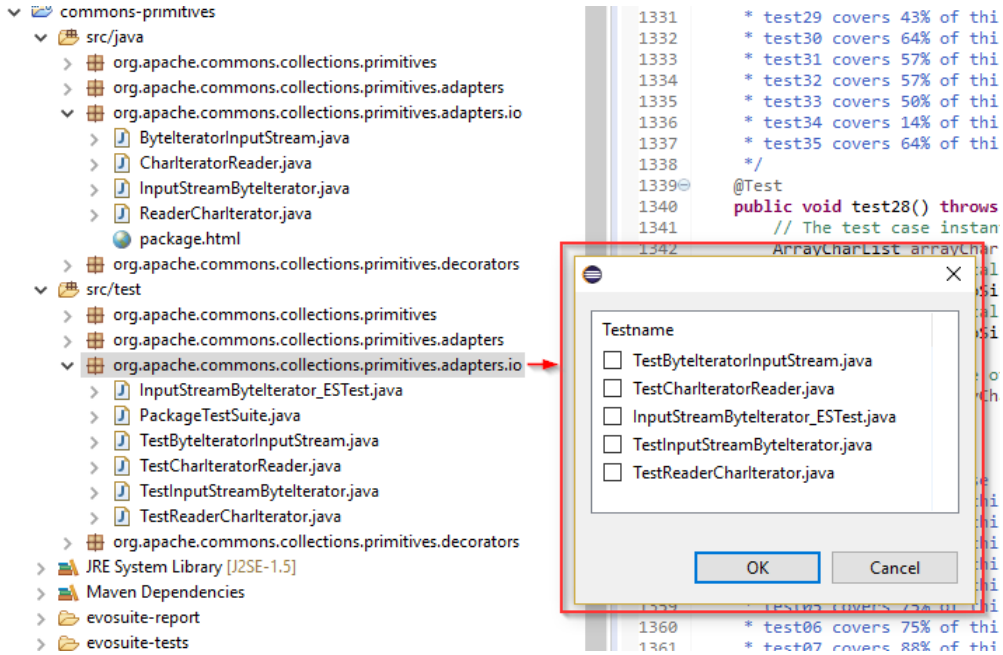

Figure 4.5: Screenshot of the selected Package and the Test Selection List Dialog, showing only valid test classes

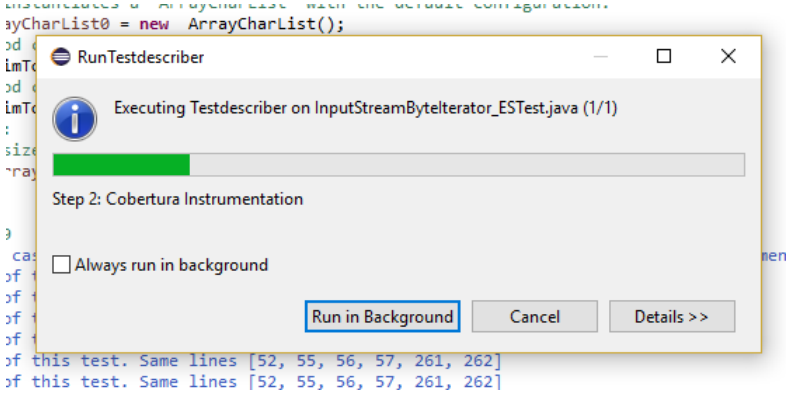

Figure 4.6: Screenshot of the Progress Bar

4.1.5 TestDescriber Unit

The *Testdescriber Unit* is the base information the TestDescriber core needs to process a PCTC pair. This represents the minimal set of information the TestDescriber core needs to run, so if an extension to TestDescriber should be programmed (e.g. a Webinterface) it would have to provide this information together with the files. In the end, the *Testdescriber Unit* merely acts as a kind of wrapper.

Field	**Description**
Package Name	Name of selected package (e.g. ch.uzh.thermometer)
File Name	Name of production code class (e.g. Temperature.java)
Test Name	Name of complementing test class (e.g. Temperature**Test**.java)
Path to Production Code	File system path to production code
Path to Test Class	File system path to test class
Path to Project Root	File system path to project root folder
Path to Resulting File	File system path to output test class (with Annotations)

Table 2: Collected information stored in the *Testdescriber Unit* by the plugin

Of course, since not all information is directly available, the *Testdescriber Unit* also holds specific methods to filter the necessary information out of the information provided by the *Eclipse* Environment.

Chapter 5

Evaluation

Finally, we ran TestDescriber on different projects and tried to establish a rough estimate about how far the tool has come since the beginning. Our benchmarks will examine the results according to the criteria of quality, structure and efficiency. The projects TestDescriber works with have to be strictly structured like a Maven project and a lot of projects we found on the internet were altered in one way or another to suit the needs of their respective developers, we found that the projects from Apache strictly adhered to the standard Maven project structure. Thus, we chose four projects from the Apache Group for our evaluation. The following projects were considered:

- **Commons-CLI**[1]
  Short description of the project according to the project homepage:

  > *"The Apache Commons CLI library provides an API for parsing command line options passed to programs. It's also able to print help messages detailing the options available for a command line tool."*

- **Commons-Lang**[2]
  Short description of the project according to the project homepage:

  > *"Lang provides a host of helper utilities for the java.lang API, notably String manipulation methods, basic numerical methods, object reflection, concurrency, creation and serialization and System properties. Additionally it contains basic enhancements to java.util.Date and a series of utilities dedicated to help with building methods, such as hashCode, toString and equals."*

- **Commons-Math**[3]
  Short description of the project according to the project homepage:

  > *"Commons Math is a library of lightweight, self-contained mathematics and statistics components addressing the most common problems not available in the Java programming language or Commons Lang."*

- **Commons-Primitives**[4]
  Short description of the project according to the project homepage:

  > *"The Java language is mostly Object based, however the original designers chose to include eight primitive types - boolean, byte, char, double, float, int, long and short. Commons*

[1] https://commons.apache.org/proper/commons-cli/source-repository.html
[2] https://commons.apache.org/proper/commons-lang/source-repository.html
[3] https://git1-us-west.apache.org/repos/asf?p=commons-math.git
[4] https://commons.apache.org/dormant/commons-primitives/source-repository.html

Primitives provides a library of collections and utilities specially designed for use with primitive types."

In order to rate the different outputs of TestDescriber according to a certain standard, we will apply the Likert scale[5]. The Likert scale is a rating system that is based on five different choices ranging from "very bad" up to "very good" based on a personal opinion. The scale we applied for our evaluation is as follows:

- 1. Strongly disagree
- 2. Disagree
- 3. Neutral (neither agree nor disagree)
- 4. Agree
- 5. Strongly agree

According to these five categories, we will examine the resulting test classes in the following three categories:

- *Effectiveness*
  Are the summaries useful?
- *Completeness*
  How many test cases are correctly summarized?
- *Performance*
  How long does it take to summarize a test class?

To further answer each questions per category following summarization table of our test results will act as base:

Project	Class	Likert Rating	No. Methods	No. LOC	Running time
CLI					
	Util	2	11	160	2 min 41s
	CommandLine	4	50	795	11 min
	Option	4	6	87	2 min 28s
Lang					
	BooleanUtils	1	122	1727	1h
	ClassUtils	2	33	486	19 min
	Conversion	1	263	4268	2h 43 min
Math					
	PolynomialFunction	2	37	568	49 min
	Primes	4	14	202	20 min
	Fraction	3	96	1379	2h 32 min
Primitives					
	ArrayCharList	4	36	573	11 min 12s
	BooleanStack	3	23	341	7 min 44s
	InputStreamByteIterator	4	13	257	5m 27s

Table 3: TestDescriber output results (the prefix 'Commons' for the projects was omitted)

[5]https://www.socialresearchmethods.net/kb/scallik.php

5.1 Tool Effectiveness

The class summaries get generated in most cases and moreover show good, understandable results.
The method summaries are also always generated but don't always add value to the understanding of the corresponding test case. The percentage information can be very useful, however, because it shows the other automatically generated test cases that cover almost the same information and differ only in small parts. For maintainability purposes, it could be beneficial to only keep one test case that covers both criteria.
The statement summary generation ranges from "not understandable" to "somewhat understandable" up to "well understandable", but they never really seem to make absolute sense. As these comments are dependent upon the naming of the methods and variables, it somehow seems that more descriptive names might be helpful but overall it is very difficult for TestDescriber to create meaningful summaries if that is not the case. Especially with automatically generated test cases, it is already known that variable names are not very descriptive. However, TestDescriber's ability to extend abbreviations does help in some cases. The best summarized statemens are clearly the *'assertEquals(...)'* statements.
With respect to effectiveness, TestDescriber still needs to improve since in the author's opinion the most important summaries, the statement summaries, are hardly read- and understandable on their own.

5.2 Tool Completeness

The completeness of the tool is very good throughout. On each test class that was chosen, all types of summaries appeared and almost all parts were annotated. Occasionally, the class summaries were missing or some unsupported statements were not summarized but overall, the completeness is very high. Of course, the impression is influenced by the correctness of the summaries, which is rather low. Moreover, if the statements are too complex or include unsupported code structures, the result is not completely satisfying.

5.3 Tool Performance

On the performance side, TestDescriber is still clearly not ready and needs improvement. Even though we achieved different results based on the randomly selected classes, the running time of TestDescriber is definitely too long. Depending on the size of the class TestDescriber is run on, it can take as long as 2 hours and more to finish a single test class. No tests were made with annotating whole projects but it can safely be assumed that it would take a very long time.
The performance bottle neck is the method level coverage that has to be computed by Cobertura on each single method. In order to get correct results, all the files from the previous Cobertura run have to be purged, which results in a large amount of steps that have to be repeated to reach the desired goal. The effect of this bottle neck is enhanced if the automatically generated test classes consist of a large number of small test cases. In this combination, some classes take up too long for the summarization.
Another point were the program could be optimized is right after the retrieval of the coverage information, when the effectively executed production code class is assembled. During that step, the analysing method has to be run for each method individually, which also adds to the slow down of the execution. By improving that method, it would be possible to gain some speed with regard to the running time of TestDescriber.

Chapter 6

Conclusion

The TestDescriber approach has come a long way from being a theoretical piece of work and a Proof-of-Concept program to a fully-fledged prototype with a user interface and an extension as a plugin. Even though the project is now able to run on various kinds of projects, it still needs to improve its core in order to generate better understandable sentences and comments. However, the biggest issue right now is the performance of the program. In a next step, it should be made a priority to optimize the general work flow of the program and to ensure a better running time of the comment generation.
However, the program's strong suits do show themselves clearly in combination with automatic test generation. As these tests tend to be somewhat cryptic and hard to read, TestDescriber is able to make reading these tests more easy in certain situations. But for the program to be production-ready, it will still need some work on the analytical side as well as the more hands-on pragmatic side. For one, the automatically generated test cases could be analysed more thoroughly in order to adjust the sentence templates for the comment generation. Another good approach would be to get more recommendations from the industry/developers on what kind of information could help them more while testing their software. Nevertheless, TestDescriber has taken another step in its evolution and with time it will become the production ready tool we already know it can be. For the author, the creation of this work has been very educational, time consuming and surely represents the peak of his undergraduate studies. It was a rich experience and it will certainly be useful on his further path as a software developer.

Bibliography

[BGPZ15] Moritz Beller, Georgios Gousios, Annibale Panichella, and Andy Zaidman. When, how, and why developers (do not) test in their ides. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, pages 179–190, New York, NY, USA, 2015. ACM.

[Bro78] Frederick P. Brooks, Jr. *The Mythical Man-Month: Essays on Softw*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 1978.

[DCF+15] Ermira Daka, José Campos, Gordon Fraser, Jonathan Dorn, and Westley Weimer. Modeling readability to improve unit tests. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, pages 107–118, New York, NY, USA, 2015. ACM.

[FAM15] Gordon Fraser, Andrea Arcuri, and Phil McMinn. A memetic algorithm for whole test suite generation. *J. Syst. Softw.*, 103(C):311–327, May 2015.

[FSM+13] Gordon Fraser, Matt Staats, Phil McMinn, Andrea Arcuri, and Frank Padberg. Does automated white-box test generation really help software testers? In *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, ISSTA 2013, pages 291–301, New York, NY, USA, 2013. ACM.

[HFB+08] Emily Hill, Zachary P. Fry, Haley Boyd, Giriprasad Sridhara, Yana Novikova, Lori Pollock, and K. Vijay-Shanker. Amap: Automatically mining abbreviation expansions in programs to enhance software maintenance tools. In *Proceedings of the 2008 International Working Conference on Mining Software Repositories*, MSR '08, pages 79–88, New York, NY, USA, 2008. ACM.

[HPVS09] Emily Hill, Lori Pollock, and K. Vijay-Shanker. Automatically capturing source code context of nl-queries for software maintenance and reuse. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 232–242, Washington, DC, USA, 2009. IEEE Computer Society.

[LMVS13] G. Sridhara A. Marcus L. Pollock L. Moreno, J. Aponte and K. Vijay-Shanker. Automatic generation of natural language summaries for java classes. in proceedings of the 21st ieee international conference on program comprehension. pages 23–32, 2013.

[PE07] Carlos Pacheco and Michael D. Ernst. Randoop: Feedback-directed random testing for java. In *Companion to the 22Nd ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications Companion*, OOPSLA '07, pages 815–816, New York, NY, USA, 2007. ACM.

[PPB+16] 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. In *Proceedings of the International Conference on Software Engineering (ICSE)*. IEEE, 2016.

[RAN07] Per Runeson, Magnus Alexandersson, and Oskar Nyholm. Detection of duplicate defect reports using natural language processing. In *Proceedings of the 29th International Conference on Software Engineering*, ICSE '07, pages 499–510, Washington, DC, USA, 2007. IEEE Computer Society.