Bachelor Thesis

January 31, 2018

TestSmellDescriber

Enabling Developers' Awareness on Test Quality with Test Smell Summaries

Ivan Taraca

of Pfullendorf, Germany (13-751-896)

supervised by

Prof. Dr. Harald C. Gall

Dr. Sebastiano Panichella

University of Zurich[UZH]

s.e.a.l.

software evolution & architecture lab

Bachelor Thesis

TestSmellDescriber

Enabling Developers' Awareness on Test Quality with Test Smell Summaries

Ivan Taraca

University of Zurich[UZH]

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

Bachelor Thesis

Author: Ivan Taraca, ivan.taraca@uzh.ch

URL: http://bit.ly/2DUiZrC

Project period: 20.10.2018 - 31.01.2018

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

Acknowledgements

First of all, I like to thank Dr. Harald Gall for giving me the opportunity to write this thesis at the Software Evolution & Architecture Lab.

Special thanks goes out to Dr. Sebastiano Panichella for his instructions, guidance and help during the making of this thesis, without whom this would not have been possible.

I would also like to express my gratitude to Dr. Fabio Polomba, Dr. Yann-Gaël Guéhéneuc and Dr. Nikolaos Tsantalis for providing me access to their research and always being available for questions.

Last, but not least, do I want to thank my parents, sisters and nephews for the support and love they've given all those years.

Abstract

With the importance of software in today's society, malfunctioning software can not only lead to disrupting our day-to-day lives, but also large monetary damages. A lot of time and effort goes into the development of test suites to ensure the quality and accuracy of software. But how do we elevate the quality of test code? This thesis presents *TestSmellDescriber*, a tool with the ability to generate descriptions detailing potential problems in test cases, which are collected by conducting a Test Smell analysis. These descriptions along with methods describing refactorings and information detailing the quality of test suites are directly augmented as comments in the source code to bring awareness on the quality of tests and to enable developers to improve their code.

Zusammenfassung

Auf Grund der hohen Bedeutung von Software in der heutigen Gesellschaft kann das Fehlverhalten von Software nicht nur zur Beeinträchtigung unseres täglichen Lebens führen, sondern auch zu grossen finanziellen Verlusten. Sehr viel Zeit und Geld wird in das Entwickeln von Testsuites investiert um die Qualität und Fehlerfreiheit von Software zu gewährleisten. Wie jedoch erhöhen wir die Qualität von Testcode? Diese These präsentiert *TestSmellDescriber*, ein Tool mit der Fähigkeit Deskriptionen zu generieren, die potentielle Probleme in Testfällen schildern, welche durch die Durchführung einer Test Smell Analyse erhoben werden. Diese Deskriptionen zusammen mit Verfahrensweisen, die Refaktorisierungen schildern und Informationen, welche die Qualität der Testsuite schildern, werden als Kommentare direkt im Quellcode hinzugefügt um das Bewusstsein im Bezug zur Qualität von Tests zu stärken und Entwicklern zum Verbessern ihres Codes zu ermächtigen.

Contents

List of Figures

List of Tables

List of Listings

Chapter 1

Introduction

The importance of software in today's society has long been established. Software is everywhere and defines our every life. So are today's major businesses and industries being run on software and delivered as online services, and industries being dominated by software companies [And11]. Is it the music industry with companies such as Spotify and iTunes, the television industry with Netflix, the book industry with the software company Amazon, or the movie industry with the former software company Pixar: "Software is eating the world" [And11]. Smartphones also accelerated this process as they enable us to gain access to the Internet and to software every minute of our lives. With such an enormous amount of software being used, required and produced every day, ensuring its quality is an everlasting challenge [And11].
To ensure the quality of the large amount of software, it has to be tried and tested. In addition to this, testing also ensures the usability of software, while it is also a mean to document code by indicating what the expected results of a method should be for typical cases and it verifies complicated functionality and unusual circumstances [VDMvdBK01]. But manual testing is very time consuming as it takes "as much as 50% of overall project effort" [PPB+16], while up to a quarter of developers work time is spend on testing [BDLMO14]. Rapid release schedules lead to even less time for software testing as well as narrower test scope [MAK+15]. Considering the amount of software in circulation and production the monetary amount grows and grows. Variance in testing outcomes can also not be reliably accounted for as results can be caused by the manual tester running the tests in different ways [RMPM12].
While the goal of creating test suites is to ensure the quality of software, assuring the quality of test suites is an equally challenging task. Automated test generation tools have been widely researched with the goal of reducing the cost of testing activities and to elevate the quality of test suites [PPB+16]. This allows developers to reduce the time and cost of the testing process [CGP+06,FA15,PKT15,RT01,Ton04], and to find violations of automated oracles [Csa04,FA15, BML07, PE07] [PPB+16]. Automatically generated tools have shown to increase confidence in the quality of the system [RMPM12], while it also leads to an improved product quality [RMPM12].
Automated testing is able to achieve similar decision coverage as manual testing in a fraction of the time, but is has not shown to be more effective in fault detection than manual testing [ESČP17]. It also produces false expectations in organizations [RMPM12] as automated test generation tools do not negate testing costs. To automate tests a process and infrastructure has to be developed that takes time to mature [RMPM12], while it also requires skilled people to run and maintain the testing process [RMPM12]. A change in technology or evolution of software requires a likewise change and maintenance in the test code, a task which is difficult to perform in test automation [RMPM12]. While the costs to produce the tests are low, the cost of checking the results is high due to the relative difficulty to understand each created test [DCF+15, RFA15, ESČP17]. A study [FSM+15,FSM+13] has shown that up to 50% of developers time is spend on understanding and analyzing automatically generated tests.

1.1 Goal of this Thesis

In recent work [PPB+16] *TestDescriber* has been developed, a tool that helps developers to better understand automatically generated test cases. The tool generates summaries of the code which is exercised by the test cases, while also delivering information regarding the coverage each case reaches in the code. The tool helps developers to better detect and find bugs, and significantly improves the comprehensibility, thereby leverages the usability of automatically generated test cases for the developer.
Due to the fact that developers still prefer manually testing their code over automatic testing methods [PPB+16], we want to complement and improve automatically as well as manually written test cases, by indicating possible areas of bugs in the test code. We are looking to achieve this by detecting test and code smells in the test cases under investigation. Code smells indicate areas in the code which are potential causes of bugs and errors as they are "symptoms of poor design and implementation choices" [TPB+15]. Code smells can also potentially lead to an increase in change- and fault-proneness, and to a decrease of software understandability or maintainability [KDPGA12]. However, bad smells can also be specific to test suites and test cases. So have van Deursen *et al.* created a catalog of test smells in [VDMvdBK01] and a collection of test refactorings to remove those smells. If your test runs fine while only one person is testing the code, but fails if more people run them, the test code smells and is affected by *Test Run War*. The cause of that failure is most likely caused by interference of resources that are used by others [VDMvdBK01]. When a test requires external resources to run, it smells. A *Mystery Guest* leads to tests no longer being self contained, while it also renders the documentational use of a test code useless and introduces hidden dependencies [VDMvdBK01].
The detection of those and other test smells enables the developer to asses and leverage the quality of test cases. To detect smells in the code we will use tools with the ability to automatically detect smells by conducting a statical analysis. The results of the smell detection will then be used to generate descriptions detailing the characteristic of the smell, along with smell information in regards to the whole project. In addition to the smell descriptions a textual description will be displayed denoting how the code can be improved, *i.e.*, refactorings. These descriptions will be displayed to the user using the *TestDescriber* method to augment information as comments into the source code. The injection of the description at the direct cause of the smell will bring awareness on the quality of tests and will enable developers to improve their code.

1.2 Research Method

To achieve this we will first conduct a literature study on code and test smells. This knowledge allows us to gain an understanding into the detection of smells, while it additionally acts as the basis for the textual descriptions of smells and their refactorings. Secondly, we will conduct a literature study on automatic smell detection. The study will yield a set of tools that we will implement into our own tool TestSmellDescriber. Before they are implemented will we asses the tools on their usability, integrability and applicability on test code on a selection of 100 projects. TestSmellDescriber will then gather the results of the smell detection and will provide the opportunity to generate smell descriptions, refactoring messages, and quantitative data. These descriptions will be generated using templates that will be defined with the knowledge gained in the literature study on code and test smells.

1.3 Structure of this Thesis

This thesis is structured as follows. In chapter 2 the results of the literature studies regarding code and test smells and their automatic detection are discussed. Chapter 3 introduces the research approach for the literature study and the assessment of the smell detection tools in addition to the results of the assessment. Chapter 4 describes and presents the different aspects of our tool TestSmellDescriber. The 5th chapter showcases the result of our prototype by presenting the output that results in running TestSmellDescriber on a test suite and a test class. Finally chapter 6 concludes our work and introduces future work that can be conducted.

Chapter 2

Background & Related Work

2.1 Smelly Code

In Martin Fowler's book *Refactoring: Improving the design of existing code* [Fow99], Fowler realized that it is easy to explain **how** to perform refactorings, but having to explain **when** to refactor turned out to be a challenging task. Fowler wanted something more solid than a "vague notion of programming aesthetics" [Fow99]. "If it stinks, change it." [Fow99] – This was Fowler's opening statement in the 3rd chapter of his book, where he explained when a refactoring should be conducted, because after brooding over the dilemma for a while with his colleague Kent Beck in Zurich, the term they came up with to explain the **when** was "(code) smells" [Fow99].

2.1.1 Code Smell Definitions, Types and Refactorings

Fowler defined code smells as "indications that there is trouble" [Fow99] in the code, which can be solved by refactoring. Code smells indicate areas, which potentially house the root of bug or error-causing code, as they are "symptoms of poor design and implementation choices" [TPB+15]. Code smells have many implications and consequences. The smell Duplicated Code denotes a code portion that can be found in more than one place [Fow99]. This duplication makes the code harder to maintain and debug [KDPGA12]. Changes to the code in one part draw changes in the duplicated version with it. This impediment can easily introduce bugs into the code leading to errors. The method to remove smells and to thereby improve the code is performed by refactoring. So can duplicated code in two methods be refactored by extracting the duplicated code into a method, which will be invoked from both previous places [Fow99]. Refactoring will ease the future development and present maintainability [Fow99], as a combination of smells significantly reduces the comprehension of the code [AKGA11].

Martin Fowler and others [Fow99, WB98, Mä03] have constructed descriptions of the possible code smells that can occur to ease the continuous cleaning process using refactoring. Following is a list (table 2.1) of the 22 code smells that were described by Fowler in [Fow99].

Table 2.1: Fowler's list of code smells

ID	Smell name	Description
F1	Duplicated Code	Duplicated Code occurs if the same expression can be found in two methods of the same class, in two sibling classes or in to unrelated classes.
F2	Long Method	A Long Method occurs if a function does more than one thing.

Continuation on the next page

Continuation of Table 2.1

ID	Smell name	Description
F3	Large Class	The class is doing too much. A Large Class often leads to duplicated code and chaos.
F4	Long Parameter List	A Long Parameter List smell is a method that requires too many parameters. A long list of parameters is hard to understand, may becomes inconsistent, difficult to use and is suspect to constant changes as you need more data.
F5	Divergent Change	A class should only have on reason to change. Divergent change occurs when a class has to be changed in different ways for different reasons.
F6	Shotgun Surgery	Shotgun survery occurs if a change requires many small changes to a lot of different classes.
F7	Feature Envy	Feature Envy occurs if a method requires more methods of another class, than it currently is in.
F8	Data Clumps	Data Clumps occurs if several data items are often used together by a field or as parameters.
F9	Primitive Obsession	Primitive Obsession occurs if a set of primitive types can be replaced by a record type that contains the primitive types.
F10	Switch Statements	A Switch Statement occurs if a switch statement is scattered multiple times in different places.
F11	Parallel Inheritance Hierarchies	Parallel Inheritance Hierarchies occurs if creating a subclass of one class, always results in the creation of a subclass of another class.
F12	Lazy Class	Lazy Class occurs if a class does too little.
F13	Speculative Generality	Speculative Generality occurs if hooks and special cases are added to a class, on the notion that they are needed one day.
F14	Temporary Field	Temporary Field occurs if an instance variable is only required for certain circumstances.
F15	Message Chain	Message Chain occurs if a class asks one object for another object. This is a violation of the Law of Demeter.
F16	Middle Man	Middle Man occurs if half the methods of an interface delegate to another class.
F17	Inappropriate Intimacy	Inappropriate Intimacy occurs if classes require too many of each others' fields and methods.
F18	Alternative Classes with Different Interfaces	Alternative Classes with Different Interfaces occurs if two classes have different method names but perform the same functions.
F19	Incomplete Library Class	Incomplete Library Class occurs if you require a library to have certain features.
F20	Data Class	Data Class occurs if a class only has fields, and those field affiliated getters and setters.
F21	Refused Bequest	Refused Bequest occurs if sub classes do not require all the methods and fields of their super class.
F22	Comments	Comments are often indicators for a smell if they try to cover bad code.

Fowler has further defined ways to eliminate those smells. In his book [Fow99] Fowler first defined several refactoring methods. He then described ways to remove smells by applying those refactorings on the smelly code. There are many instances and scenarios where smells can take

place. [Fow99] defines refactorings for many of those. Table 2.2 lists refactorings for the most common scenarios.

Table 2.2: Fowler's list of code smell refactorings

ID	Smell name	Refactoring
F1	Duplicated Code	Extract the duplicated method and invoke the code from both places.
F2	Long Method	Find parts of the method that go together and extract them into a new method.
F3	Large Class	Bundle instance variables together and extract those into new classes.
F4	Long Parameter List	Replace the parameter with a new class that holds all requested data, or if they already belong to one object, pass the whole object.
F5	Divergent Change	Identify the parts that change for a particular cause and extract them into new classes.
F6	Shotgun Surgery	Move all elements that requires changes into one entity by extracting fields and methods from their original class.
F7	Feature Envy	Move the feature envy method into the class it uses the most data from.
F8	Data Clumps	Extract the data fields into a new object. Simplify the method call by passing the whole object.
F9	Primitive Obsession	Replace the set of primitive types with an object that holds all the data.
F10	Switch Statements	Extract the switch statement into the class where polymorphism is needed.
F11	Parallel Inheritance Hierarchies	Instances of one hierarchy should refer to instances of the other.
F12	Lazy Class	Eliminate the whole class or transform the lazy class into an inline class.
F13	Speculative Generality	Remove the generality by removing unused parameters, renaming abstractly named methods, removing abstract classes or transforming them to inline classes.
F14	Temporary Field	Move the instance variable to a new class and moving all the code that concerns the variable into that class.
F15	Message Chain	Create a method that hides the delegation or move the method that performed the message chain into the correct object.
F16	Middle Man	Remove the middle man and invoke the object directly.
F17	Inappropriate Intimacy	Move the method or field that requires another field or method into the respective class.
F18	Alternative Classes with Different Interfaces	Rename the methods to make them identical. Alternatively move the method into a respective class.
F19	Incomplete Library Class	Create a method in your class with an instance of the library class as its first argument.
F20	Data Class	Hide public fields by encapsulating them. Alternatively find methods or part of methods that are better suited in the class and move them.

Continuation on the next page

Continuation of Table 2.2

ID	Smell name	Refactoring
F21	Refused Bequest	Create a new sibling and move the methods and fields to the subclass that requires them.
F22	Comments	Extract part of the method that requires commenting into a new class and rename that method to reflect what the comment states.

Mäntylä recognized that the compiled smells by Fowler & Beck lacked structure. He argued that a flat list of 22 smells were to hard to perceive and understand. His study [Mä03] resulted in a taxonomy consisting of 7 higher-level categories that are mapped to the 22 code smells. Besides listing the categories, table 2.3 additionally associates the categories to Fowler's smells by referring to them by their ID in table 2.1.

Table 2.3: Mäntylä's higher level smell categories

Category	Smells	Category description
The Bloaters	F2, F3, F4, F8, F9	Something has grown so large that it cannot be handled effectively.
The Object-Orientation Abusers	F10, F11, F18, F14, F21	Possibilities of object-oriented design have not been fully exploited.
The Change Preventers	F5, F6	Change or further development of software is hindered or prevented.
The Dispensables	F1, F12, F13, F20	An unnecessary element in the source code that should be removed.
The Encapsulators	F15, F16	Smells are concerned with data communication mechanism or encapsulation.
The Couplers	F7, F17	Minimal coupling principle is violated.
Others	F19, F22	Smells that do not fit into any of the above categories above.

2.1.2 Test Smell Definitions, Types and Refactorings

Bad smells can also be specific to test code. In the paper *Refactoring Test Code* by van Deursen *et al.* [VDMvdBK01] a list of code smells have been described that indicate trouble in test code, *i.e.*, test smells. Code smells are not specific to production code and can also be applied to test code, but van Deursen *et al.* have acknowledged that refactoring test code requires additional test-specific refactorings [VDMvdBK01]. Along with a catalog of test smells [VDMvdBK01] also defines six test smell specific refactorings. Table 2.4 lists the set of test smells that were identified by van Deursen.

Table 2.4: Van Deursen's list of test smells

ID	Smell Name	Description
D1	Mistery Guest	Part of the test is executed outside of the test case, *i.e.*, the test uses external resources.
D2	Resource Optimism	The test makes assumptions about external resources leading to test running fine one times and failing another.

Continuation on the next page

Continuation of Table 2.4

ID	Smell Name	Description
D3	Test Run War	If a test allocates resources for the run, running two tests simultaneously results in resource interference.
D4	General Fixture	The fixture is too general. Individual test cases only access and require part of the provided fixture.
D5	Eager Test	The test checks too much functionality/methods of the object under test in a single test case.
D6	Lazy Test	Several test methods check the same method using the same fixture. The tests only have meaning when they are considered together.
D7	Assertion Roulette	The test contains several assertions that have no explanation.
D8	Indirect Testing	The test class contains methods that perform tests on other objects.
D9	For Testers Only	The production class contains methods that are only used by test methods.
D10	Sensitive Equality	If a result is mapped to string and compared to literals the result may vary as it depends on many irrelevant details (*i.e.*, commas, quotes, spaces, etc.)
D11	Test Code Duplication	The test code contains duplications.

Van Deursen *et al.* have further defined refactorings for all above listed test smells. These refactorings are listed in table 2.5.

Table 2.5: Van Deursen's list of test smell refactorings

ID	Smell Name	Refactoring
D1	Mistery Guest	Incorporate the required resource into the test by setting up a fixture that holds the contents of the resource or make sure to explicitly allocate or initialize and release the resource.
D2	Resource Optimism	Make certain to explicitly allocate or initialize and release the required resources.
D3	Test Run War	Use unique identifiers for the allocated resources to find tests that do not properly release their resources.
D4	General Fixture	Extract the parts of the fixture that are not required by all methods into the methods that require it.
D5	Eager Test	Separate the test case into methods that test only one method of the class under test. Additionally assign meaningful names to the methods describing the goal of the test cases.
D6	Lazy Test	Combine the individual test cases into one test method.
D7	Assertion Roulette	Pass a message to the assertion to distinguish between different assertions.
D8	Indirect Testing	Move the test cases to the appropriate test classes.
D9	For Testers Only	Move the methods in the production code used only by test methods to a new subclass and perform the tests on that subclass.

Continuation on the next page

Continuation of Table 2.5

ID	Smell Name	Refactoring
D10	Sensitive Equality	Introduce real equality checks by adding an implementation for the equals methods in the object's class and check the equality by using this method. Alternatively construct a new object that holds all the expected values and compare the computed values to the values in the new object.
D11	Test Code Duplication	Extract the duplication into a new method.

While van Deursen described test smells on a low level, Meszaros in [Mes07] went a step further and described three different kinds of higher level smells:

- *Code smells* are smells that must be recognized by looking at code.
- *Behavior smells* are smells that are realized during the execution of a test as they affect its outcome.
- *Project smells* are smells that are usually recognized by project managers, who do not come in direct contact with the test code. These smells indicate the overall health of a project.

Since this thesis focuses on static analysis of tests, we further studied Meszaros's *Code smells*. Within his higher level smells [Mes07] splits smells into 5 more categories. Those categories rely on observations made when a smell has occurred in the code. The smells are then labeled as the *causes* of the made observations.
Note: Some of the below smells are a repetition of already discovered smells, but are listed and described for the sake of completeness.

Obscure Test. A reviewer might observe that the test "is difficult to understand at first glance" [Mes07]. This can be caused by any of the following smells:

- *Eager Test:* The test checks too much functionality/methods of the object under test in a single test case.
- *Mystery Guest:* Part of the test is executed outside of the test case, *i.e.*, the test uses external resources.
- *General Fixture:* The fixture is too general. Individual test cases only access and require part of the provided fixture.
- *Irrelevant Information:* The test exposes too many irrelevant details about the fixture. This distracts the test reader from what really affects the behavior of the system.
- *Hard-Coded Test Data:* The fixture, assertions or arguments contain hard coded data values of the system.
- *Indirect Testing:* The method interacts with the test object via another object.

Conditional Test Logic. A reviewer might observe that the "test contains code that may or may not be executed" [Mes07]. This can be caused by any of the following smells:

- *Flexible Test:* The test results varies depending on when or where it is run.
- *Conditional Verification Logic:* Conditional Test Logic is used to verify the expected outcome. *E.g.* assertions are prevented to be executed if the wrong object was returned by the class under test.
- *Production Logic in Test:* The verification section of the tests contains some form of Conditional Test Logic.
- *Complex Teardown:* Complex fixture teardown is hard to verify and can result in "data leaks".
- *Multiple Test Conditions:* The same test logic is applied many sets of input values, each with its own corresponding expected result.

Hard-to-Test Code. A reviewer might observe that the "code is difficult to test" [Mes07]. This can be caused by any of the following smells:

- *Highly Coupled Code:* Testing a class cannot be done without testing several other classes.
- *Asynchronous Code:* Testing a method requires the start of an executable (such as a thread, process, or application) and wait until its start-up has finished.
- *Untestable Test Code:* The test method is so obscure or contains enough Conditional Test Logic that we wonder whether the test is correct.

Test Code Duplication. A reviewer might observe that "the same test code is executed many times" [Mes07]. This can be caused by any of the following smells:

- *Cut-and-Paste Code Reuse:* Copies of the same code result in having to maintain all copies in parallel.
- *Reinventing the Wheel:* The same sequence of statements was written in different tests.

Test Logic in Production. A reviewer might observe that "the code that is put into production contains logic that should be exercised only during tests" [Mes07]. This can be caused by any of the following smells:

- *Test Hook:* The production code contains conditional logic that determines whether the “real” code or test-specific logic is run.
- *For Tests Only:* Code exists in the production code that is strictly used by tests.
- *Test Dependency in Production:* The production executables depend on test executables.
- *Equality Pollution:* Test specific equalities are implemented in the equals method of the class under test.

2.2 Smell Detection Tools

Fowler & Beck did not try to deliver precise criteria on how a smell can be found in the code, because "no set of metrics rivals human intuition" [Fow99]. Many later realized that their vague and informal descriptions leave too much space for interpretation and have come up with metrics that precisely define a smell occurrence [TC09b,MGDLM10,PBDP+13,Mar04]. Contrary to Fowler & Beck's opinion on automatic detection of smells, starting from the defined metrics on smell detection, tools have been researched and developed that are able to automatically detect code smell in a given code [MTSV16, MGDLM10, Pal15]. Following are the four smell detection tools that were considered for this thesis.
Note 1: Some of the below mentioned smells have been given different names in their respective papers. To give a clear summary of the papers and simultaneously link them to the in sections 2.1.1 & 2.1.2 defined smells, all further listed smells are followed by IDs corresponding to the IDs in tables 2.1 & 2.4.
*Note 2: Along with code smells JDeodorant (section 2.2.1), DECOR (section 2.2.2), and Organic (section 2.2.4) are able to detect design smells[1] and/or anti-patterns[2] in the code. For the sake of completeness these are also mentioned, but marked with **AP/DS**.*

2.2.1 JDeodorant

JDeodorant is a tool that was developed by Nikolaos Tsantalis *et al.* for 7 years. It is a plug-in for the well known IDE Eclipse and is able to detect three code smells and one design smell/anti-pattern in Java source code [jde18]:

- Long Method (F2)
- God Class (F3)
- Feature Envy (F7)
- Type/State Checking (AP/DS)

JDeodorant is also able to parse the output of a clone detection tool and provide refactoring opportunities for a duplicated code (F1) smell [MTSV16]. JDeodorant does not find smells in the code, but finds different refactoring opportunities in regard to the above mentioned smells. Examples are Extract Method Refactorings [TC11, TC09a] and Move Method Refactorings [TC09b, FTC07]. These refactoring opportunities are visualized to the user and after being accepted are automatically executed [MTSV16].

2.2.2 DECOR

Yann-Gaël Guéhéneuc *et al.* have proposed DECOR (***DE****tection &* ***COR****rection*) [MGDLM10], a method that allows the specification and detection of smells and anti-patterns. DETEX (***DE****tection* ***EX****pert*) is the implementation of DECOR and a stand-alone tool. DETEX (from this point called DECOR) defines the detection of 8 code smells and 10 design smells/anti-patterns:

- AntiSingleton (AP/DS)
- BaseClassKnowsDerivedClass (AP/DS)

[1] Design smells are "structures in the design that indicate violation of fundamental design principles and negatively impact design quality" [SSS14].
[2] "An AntiPattern is a literary form that describes a commonly occurring solution to a problem that generates decidedly negative consequences." [WB98]

- BaseClassShouldBeAbstract (AP/DS)
- Blob (AP/DS)
- ClassDataShouldBePrivate (AP/DS)
- ComplexClass (AP/DS)
- FunctionalDecomposition (AP/DS)
- LargeClass (F3)
- LazyClass (F12)
- LongMethod (F2)
- LongParameterList (F4)
- ManyFieldAttributesButNotComplex (F20)
- MessageChains (F15)
- RefusedParentBequest (F22)
- SpaghettiCode (AP/DS)
- SpeculativeGenerality (F13)
- SwissArmyKnife (AP/DS)
- TraditionBreaker (AP/DS)

DECOR analyzes the structure of a system and detects smells or anti-patterns with the help of previously defined rules.

2.2.3 TACO

Although research has explored the use of both structural and conceptual information for the removal of smells, no approach for the identification of smells with the use of conceptual information has been investigated. Palomba argues in [Pal15] that conceptual information can be used to identify smells in code. To verify this TACO (***T**extual **A**nalysis for **C**ode smell detecti**O**n*), a standalone tool, has been developed, a tool that extracts conceptual information from the code using textual analysis techniques. To achieve this TACO evaluates textual information that are contained in elements of the source code and computes the textual similarity between code elements that characterize a code component [PPDL$^+$16]. In contrast to other detection tools, including to its nouvelle detection approach, TACO has implemented detection algorithms for both code smells and test smells. The test smells that are detected by TACO are

- Eager Test (D5),
- and General Texture (D4)

2.2.4 Organic

Organic is an Eclipse plug-in that has implemented the rules published by Bavota *et al.* [BDLMO14] to detect seven code smells and eleven design smells/anti-patterns. It uses the Eclipse JDT API to parse the candidate classes, and then analyzes the syntactical structure of those classes by applying the aforementioned rules [org18, BDLDP+15]. *E.g.* a method is guilty of being Feature Envy if the methods makes more calls with another class than the one they are implemented [BDLDP+15]. Organic is able to detect 7 code smells along with 4 design smells/anti-patterns.

- Class data should be private (AP/DS)
- Complex class (AP/DS)
- Feature envy (F7)
- Blob class (AP/DS)
- Lazy class (F12)
- Long method (F2)
- Long parameter list (F4)
- Message chain (F15)
- Refused bequest (F22)
- Spaghetti code (AP/DS)
- Speculative generality (F13)

2.3 Software Testing

With as much software was and is being developed to this day, proving the correctness of a system is still beyond our abilities, and so is specifying its behavior [Mes07]. It does not take long for a new software developer to realize that having bugs in your code is inevitable, and debugging your code is an everlasting challenge. In the following sections will we look at goals of software testing in general, and limitations of manual testing compared to the benefits and limitations that are achieved by automatic software testing (AST). In this thesis automatic software testing means automatically generated unit tests unless otherwise specified.

2.3.1 Goals of Software Testing

Bug-free Code Since we cannot prove the correctness of a system one goal of quality assurance is to test so long and so often, that we cannot prove that there are still bugs in our systems. The intent of testing is to find errors in the code by executing a program [MSB11]. In addition to proving that there are currently no bugs in the code, the goal of testing is also to prevent bugs from being introduced [Mes07]. Lastly, once a bug has occured, tests allow us to localize the defect [Mes07].

Verification and Validation Having bug-free code is not the only reason to have tests. Testing is also a mean to verify the correct execution of code [MSB11]. By running tests we are looking to demonstrate that a program works as intended, but also to check whether it meets the defined requirements [MSB11, KS10]. A testing report is akin to a status report comparing the actual product to the product requirements [KS10].

Flexible Code Test suites additionally enable developers to easily and safely conduct changes in the code. "If you have tests, you do not fear making changes to the code! Without tests every change is a possible bug. No matter how flexible your architecture is, no matter how nicely partitioned your design, without tests you will be reluctant to make changes because of the fear that you will introduce undetected bugs" [Mar08]. Tests allow your system to be flexible, maintainable and reusable [Mar08]. After every change to the production code test cases can be used to verify the validity of the code.

Test as Specification Following Martin's three laws of Test-driven Development (TDD)[3] lead us to another revelation: tests enable us to specify the behavior of the system. By writing tests before building the system, it allows us to capture how it will be used, and contemplating through scenarios enables us to identify ambiguity in requirements [Mes07].

Test as Documentation Lastly tests are a mean to document code. Having well written and clean tests enables developers to learn what and how a code is executed, by running a test and stepping through it with a debugger. Furthermore without tests developers have to go over the code in order to find out the result of an input and to answer the question "What should be the result if…" [Mes07], whereas a provided test quickly delivers the answer to that question.

2.3.2 Limitations of Manual testing

Effort and Time Historically speaking testing was performed manually. But as already mentioned is manual testing very time consuming as 50% of the effort spent on a project is required to verify the produced code [PPB+16]. Every individual developer also spends up to a quarter of his work time on testing [BDLMO14]. Agile Development additionally leads to shorter release schedules, leaving less time to test, or forces developers to resort to narrower test scopes [MAK+15].

Rigid Code Time pressure can also lead to developers writing dirty tests. A result which "is equivalent to, if not worse than having no tests" [Mar08]. Dirty tests can have the opposite effects of a clean test. Instead of achieving flexible code by writing tests, changes in the code lead to more time being spent on changing and adding tests [Mar08].

Reliability Reliability also comes into question, considering that a variance in results can be caused by a manual tester running tests in different ways [RMPM12].

Benefits of Automatic Testing Table 2.6 displays the benefits of Automatic Software Testing (AST) that resulted in a literature and practitioner survey conducted by Rafi *et al.* in [RMPM12].

[3] The three laws of TDD, taken from [Mar07]:

1. You may not write production code unless you've first written a failing unit test.
2. You may not write more of a unit test than is sufficient to fail.
3. You may not write more production code than is sufficient to make the failing unit test pass.

Table 2.6: Benefits of automatic testing according to [RMPM12]

ID	Benefits of AST
B1	Improved product quality
B2	Higher test coverage
B3	Reduced testing time
B4	Higher reliability in repetition of tests
B5	Increase in confidence
B6	Higher reusability of tests
B7	Less requirement for human effort
B8	Cost reduction
B9	Higher fault detection

On one hand AST can leverage the benefits of software testing in general. A higher test coverage (B2) and higher fault detection (B9) achieve an even better proof for a bug-free code. But AST also counteracts the limitations that result from manual testing. A higher reusability of tests (B6) leads to less rigid and therefore more flexible code. Effort and time is also saved due to reduced testing time (B3), less requirement for human effort (B7) and reductions of costs (B8). It also achieves a higher reliability due to the increase in reliability in test repetitions (B4). These benefits all lead to an increase in confidence in the quality of the system (B5) and to an improved product quality (B1).

2.3.3 Limitations of Automatic Testing

Rafi *et al.* also recognized that there are not only benefits in automatic testing, but it also leads to limitations. The results of the survey can be viewed in table 2.7.

Table 2.7: Limitations of automatic testing according to [RMPM12]

ID	Limitations of AST
L1	Automation can not replace manual testing
L2	Failure to achieve expected goals
L3	Difficulty in maintenance of test automation
L4	Process of test automation needs time to mature
L5	False expectations
L6	Inappropriate test automation strategy
L7	Lack of skilled people

One of the limitations that Rafi *et al.* recognized is that test automation can not replace manual testing. Some tasks still need to be manually executed, as they require extensive knowledge in a specific domain (L1). The limitations further highlight that AST not only brings benefits, but requires effort to set up (L7) and time to mature (L4). They also reflect a wrong understanding of the actual benefits that result in AST (L5) and the set automation strategy (L6).

While the benefits of AST show a decrease of the costs in test production, the costs of checking the results of the produced test are still high. This is due to the relative difficulty to understand each created test [DCF+15, RFA15, ESČP17] as up to 50% of developers time is spend on understanding and analyzing automatically generated tests [FSM+15, FSM+13]. This is also reflected in L3, which states a difficulty in maintaining test automation. The results of automatically generated tests are often tests that are not as nice to look at as manually written tests [DCF+15].

2.3.4 Impact of Smells on Test Quality

Code smells can have or have an overall negative impact on the quality of test suites and test cases. Smells like *Duplicated Code*, *Mystery Guest* or *Test Run War* make a test prone to bugs and defects [BQO+15, VDMvdBK01]. Performing small changes on duplicated code will require changes to multiple methods, which possibly introduces unexpected behavior countering a test's intended goal of bug-free code. If adapting your test code requires more time than writing new production code, tests become increasingly viewed as a liability, an effect which will hurt the flexibility of code that is achieved by having well written tests [Mar08]. This can occur if test cases are not well separated in *Long Methods* or *Eager Tests*, but also in *Large Classes* [BQO+15, Mar08, BQO+12]. Having test cases that are well separated and which through that separation test individual operations of production code, helps developers in learning what the intent of that code is and how to correctly use it [Mar08]. Introducing *Long Methods* or *Eager Tests* make tests less concise and hurt the documentational purpose of tests [VDMvdBK01, Fow99]. From these scenarios it is apparent that smells can not only impact test suites negatively by countering intended goals of software testing, but can also amplify its limitations. Servicing tests that are smelly will require even more time and effort, than they would without smells [BQO+15, BQO+12]. A *General Fixture* will slow the execution of a test case, as much of the fixture is not required by the test case [VDMvdBK01]. Furthermore, as mentioned already, adapting code housing duplicated code is more difficult than servicing non smelly tests, this makes tests simultaneously more rigid and less reliable [BQO+12].

2.4 Summarization Techniques for Software Testing

In 2016 Panichella *et al.* tackled some of the above mentioned limitations in AST. Their thesis [PPB+16] builds on the results that were achieved in the experiment conducted in [DCF+15]. Daka *et al.* proposed a on human judgments based domain-specific model of unit test readability, which was then used to augment automatically generated tests. This resulted in more readable tests. Their subsequent study showed that the improved tests helped their test subjects to answer tests 14% faster with no change in accuracy.
Panichella *et al.* developed the tool *TestDescriber* (TD), which generates summaries of the code under test and delivers information regarding the coverage each case reaches. The summaries contribute to the understanding of the test and aims to provide empirical evidence that "readability improvements produce tangible results in terms of the number of bugs actually found by developers" [PPB+16].
The TD approach works in five steps. The tool requires a test suite along with the production code. The test suites can be provided, or generated by TD from the provided production code. This is done with the help of *EvoSuite* [PMGZ13]. In the next step TD relies on *Cobertura*[4] and a self constructed parser based on *JavaParser*[5] to provide information regarding the branches and statements that are tested by each test case. This information is then used to build the textual corpus of the summaries, and to produce a quantitative analysis of the tests. For that the information are first pre-processed, then analyzed by *LanguageTool*[6], a Part-of-speech tagger, and lastly categorized in Noun Phrases, Verb Phrases and Prepositional Phrases. The different types of phrases are used in the next step to produce summaries at three different levels of abstractions: a summary providing a general description of the class that is tested by the JUnit test, a summary of the structural code coverage that is achieved by the test cases, and a description of each statement of the

[4] http://cobertura.github.io/cobertura/
[5] https://github.com/javaparser/javaparser
[6] https://github.com/languagetool-org/languagetool

test cases. TD then aggregates and enriches the original test with the produced summaries in the last step. The subsequent study showed that the with summaries enriched tests help developers to find twice as many bugs in comparison to non-summarized tests.

Chapter 3

Approach

The goal of this thesis is to bring attention to the quality of test suites and to further help developers to improve their test code. We want to achieve this by delivering qualitative information to the tester regarding their test suite by indicating problematic areas in their tests as well as information on how to eliminate those problems. The critical areas will be located by detecting code/test smells (section 2.1). The detection of smells will be performed by automatic smell detection tools, which will be combined into one tool, TestSmellDescriber. TestSmellDescriber will then use these information to generate descriptions detailing the characteristics of the found smell, the method to refactor that smell and and to parse additional project relevant information. The deliverance of those descriptions will be performed by the from Panichella *et al.* in [PPB+16] developed tool *TestDescriber* (section 2.4). Using the *TestDescriber* method to inject descriptions directly at the cause of the smell will aid in bringing awareness to the test quality and to enable developers to localize the cause of the problem along with the refactoring description. The steps to accomplish this are:

1. We will study from the literature the types of existing Test and Code Smells (sections 3.1). This information constitutes the basis of knowledge required for this thesis and enables us to construct the textual descriptions and refactoring that are displayed to the user in step 5.
2. We will study from the literature the available tools and ways to automatically detect Test and Code Smells in the code (sections 3.2).
3. The researched list of tools in step 2 will be tested on a dataset of 100 projects (section 3.3) to analyze the effectiveness and the capability of the tool to be implemented in TestSmellDescriber.
4. The accepted tools will then be combined in a single tool to provide a detector of Test Smells (chapter 4).
5. We will generate descriptions and refactoring messages for the smelly classes and methods (chapter 4) using the output of the implemented detection tools and the gained knowledge in step 1.

3.1 Literature Study on Test and Code Smells

Following is the list of literature that was used for the study on test and code smells:

- *Improving the design of existing code*, Martin Fowler [Fow99]
- *When and why your code starts to smell bad*, Tufano *et al.* [TPB+15]

- *An exploratory study of the impact of antipatterns on class change- and fault-proneness*, Khomh *et al.* [KDPGA12]
- *An Empirical Study of the Impact of Two Antipatterns, Blob and Spaghetti Code, on Program Comprehension*, Abbes *et al.* [AKGA11]
- *An exploratory study of the impact of code smells on software change-proneness*, Khomh *et al.* [FKG09]
- *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis*, Brown *et al.* [WB98]
- *Bad Smells in Software: a Taxonomy and an Empirical Study*, Mäntylä [Mä03]
- *Refactoring test code*, van Deursen [VDMvdBK01]
- *xUnit Test Patterns: Refactoring Test Code*, Meszaros [Mes07]

The result of the literature study can be found in section 2.1. The resulting list of literature outlined above is not exhaustive, but was selected by conducting and researching papers that provide information for refactoring code in the face of code smells. By concentrating on the references from researches that actively use code smells, we used commonly accepted views and definitions regarding code smells. [Pal15] provided only a short definition for smells, but provided references to literature that extensively researched code smells, and references to papers that discussed other smell detection method, such as [KDPGA12], [AKGA11] and [FKG09]. These papers provided references to literature that represent the basis of the literature study on code smells: [Fow99], [WB98] and [Mä03]. Google Scholar was able to provide the test smell catalog by van Deursen [VDMvdBK01] by conducting the search engine test smell refactorings, while the book [Mes07] by Meszaros on the refactoring of test code was found through the Martin Fowler Signatures Series[1]. We provided definitions from Fowler and van Deursen for the individual smells in the code and test smell sections, and followed it up with a categorization by Mäntylä and Meszaros.

3.2 Literature Study on Automatic Smell Detection

Following is the list of literature that was used for the study on automatic test and code smell detection tools:

- *JDeodorant: clone refactoring*, Mazinanian *et al.* [MTSV16]
- *JDeodorant: Identification and Removal of Type-Checking Bad Smells*, Tsantalis *et al.* [TCC08]
- *Identification of Extract Method Refactoring Opportunities for the Decomposition of Methods*, Tsantalis *et al.* [TC11]
- *Identification of Extract Method Refactoring Opportunities*, Tsantalis *et al.* [TC09a]
- *Identification of Move Method Refactoring Opportunities*, Tsantalis *et al.* [TC09b]
- *JDeodorant: Identification and Removal of Feature Envy Bad Smells*, Fokaefs *et al.* [FTC07]
- *DECOR: A Method for the Specification and Detection of Code and Design Smells*, Moha *et al.* [MGDLM10]
- *Textual Analysis for Code Smell Detection*, Palomba [Pal15]

[1] https://martinfowler.com/books/

- *Automating extract class refactoring: An improved method and its evaluation*, Bavota *et al.* [BDLMO14],
- *An experimental investigation on the innate relationship between quality and refactoring*, Bavota *et al.* [BDLDP+15]

The literature study regarding Smell Detection Tools can be found in section 2.2. The paper [Pal15] constructed the start of our literature study. TACO is a tool that follows a not yet commonly used concept. Due to this Polomba contrasts the tool with other tools, namely JDeodorant [MTSV16, TCC08, TC11, TC09a, TC09b, FTC07], and DECOR [MGDLM10]. By consulting Google Scholar regarding automatic smell detections the paper [BDLDP+15] was found and subsequently its implementation Organic. To narrow down the scope of this thesis, we concentrated on the above mentioned papers and in them mentioned tools. Other Java tools with some aspects of automatic code smell detection exist and have been listed below for completeness sake. These tools were found by further inquiring Google Scholar for automatic code smell detection. Some tools were found by consulting [PDFS17] and [FBZ12].

- Checkstyle [che17]
- iPlasma [MMMW05]
- PMD [pmd17]
- StenchBlossom [MHB10]
- JSpIRIT [VVDP+15]
- ConQAT [con18]
- CloneDigger [clo18]
- JCosmo [VEM02]
- CodeVizard [ZA10]

3.3 Assessment of Smell Detection Tools

3.3.1 Overview

Starting from the literature study on automatic smell detection tools (section 3.2) we next tried to get an assessment of the developed tools and their detection on the following points.

Integrability. Due to scalability reasons TestSmellDescriber will be a stand alone tool as this allows the tool to be used flexibly by many users, for a large number of projects, as well as to be integrable into other tools. The automatic smell detection tools are therefore required to be usable without any additional requirements and environment.

Applicability on Test Suites. Since the purpose of TestSmellDescriber is to detect smells in test suites, the tool should be able to detect smells in test classes or test cases. The smell detection tools have shown to be effective in detecting code smells, but we have found no record of its use for the detection of test smells.

Detectable Smells. Only one out of four tools are specified for the detection of smells in test code. The assessment will help us to gain knowledge about which implemented code smell detections are relevant for test code.

For that we first found tools that are capable of automatically detecting code smells in the code. This was done in section 3.2. Next we selected several projects on which the tools could be applied to based on several requirements outlined in section 3.3.2. This resulted in 67 projects. And lastly we made an assessment of the candidate tools. This assessment and its result can be viewed in section 3.3.3.

Candidate Tools

The list of tools resulted from the literature study conducted in section 3.2 & 2.2. The candidate tools are

- TACO,
- DECOR,
- JDeodorant,
- and Organic.

We want to help developers to improve their tests by delivering qualitative information to the tester regarding their test suite by indicating problematic areas in their tests. These areas are symptoms of poor design and implementation choices, i.e. smells. The aforementioned tool are able to identify smells in the code and indicate those to the user. Identifying the guilty smell also enables us to provide information to the developer on how to eliminate those problems.

Candidate Smells

Not all aforementioned smells could be considered for this thesis. The smell detection process in our tool TestSmellDescriber will be conducted by the in section 3.3.1 researched tools or a subset of those tools. These tools have implemented methods and metrics for a subset of all code and test smells. The detectable smells are listed in the tool's associated subsection in section 2.2. These smells are

- Long method (F2),
- Large Class (F3),
- Long Parameter List (F4),
- Feature Envy (F7),
- Lazy Class (F12),
- Speculative Generality (F13),
- Message Chains (F15),
- Data Class (F20),
- Refused Bequest (F22),
- General Fixture (D4),

- and Eager Test (D5).

This list of smells enables us to detect smells from 70% of all categories defined by Mäntylä: *Bloaters* (F2, F3, F4), *Dispensables* (F12, F13, F20), *Encapsulators* (F15), *Couplers* (F7) and *Others* (F22). The detection of test specific smells, General Fixture and Eager Test, covers 20% of all test smell categories by Meszaros: *Obscure Test*.

3.3.2 Test Projects

The projects were selected from a number of 100 candidates from two well-known open-source repositories, the Apache Software Foundation and Black Duck Open Hub, starting from the projects with the most number of committers, respectively projects with the highest rating. Tools were rejected based on the availability of their source code, the availability of test suites, their language, their capability to build and the overextension of a time limit during the detection process. The final list of projects can be found in table 3.4.

Selection Process

The projects underwent three selection steps. In the preliminary selection we gathered 100 projects. We believe this number is large enough to leave us with a sufficiently large number of projects at the end of the selection process. Projects were selected based on the following requirements for the first selection:

1. The source code of the project has to be available. The source code is necessary to detect smells in the code with the tool TACO. More importantly are we looking to display the results of the smell detection as comments in the code. The source code is therefore required to augment the comments in the code.
2. Since we want to detect smells in test code, the project has to offer test suites. It was not relevant if those tests were manually written or automatically generated.
3. The project is written in Java.

To meet requirements number one and three we decided to gather the projects evenly split from two well known open source communities:

- the Apache Software Foundation,
- and Black Duck Open Hub.

Apache hosts 389 open source projects with over 59% of those (partially) written in Java [apa18]. Open Hub indexes 472,078 open source projects [ope18]. By querying for "Java" we received a list of projects that are (partially) written in Java. To meet the second requirement we concluded that the likelihood of projects having test suites is higher the more popular the project is, popular in terms of either by the number of committers on Apache [2], or the received user rating on Open Hub [3]. With above requirements we looked at 50 projects from Open Hub, while seven of those had to be rejected due to no available source repositories. We further looked and checked out 50 Apache projects.

[2] https://projects.apache.org/projects.html?number
[3] https://www.openhub.net/p?query=java&sort=rating

Table 3.1: Projects' availability of source code

Source available		No source available	
OpenHub	**Apache**	**OpenHub**	**Apache**
43	50	7	0
93		7	

To parse and detect code smells *DECOR* requires the code in binary format. *TestDescriber* also requires source as well as binary versions of the given code. Some projects where available in both source and binary format, while others had to be manually built and compiled. While we were able to successfully built many projects, some experienced compile error or could otherwise not be build and had to therefore be rejected. This second selection step left us with 31 Open Hub projects, and 39 Apache projects.

Table 3.2: Projects' build status

No build issues		Build issues	
OpenHub	**Apache**	**OpenHub**	**Apache**
31	39	12	11
70		23	

In the third selection step we ran those 70 projects in a preliminary test run with TACO, Organic, DECOR, and JDeodorant. Since testing and running the projects with the given tools have to be repeated many times over the course of the development, and to achieve a repeatability of the received results, projects that took more than 30 minutes to parse were rejected in this last step. Here two Open Hub projects and one Apache projects were discarded.

Table 3.3: Projects' testability

Time not exceeded		Exceeded time	
OpenHub	**Apache**	**OpenHub**	**Apache**
29	38	2	1
67		3	

The selection process resulted in 67 Java projects with test suites, in source and binary format, that are testable in less than 30 minutes. 29 of the 67 are from Open Hub and 38 from Apache. The whole list of projects along with their status can be found in table 3.4.

Table 3.4: Test projects

ID	Project name	Version	Considered?	Reason for rejection
1	Apache Hadoop	2.9.0	Yes	–
2	Apache Cloudstack	4.10.0	Yes	–
3	Apache Ambari	2.6.0	No	Build error
4	Apache Cordova Android	7.0.0	No	Build error
5	Apache Geode	1.3.0	No	Time limit exceeded
6	Apache Cocoon	2.1	No	Build error
7	Apache MyFaces Core	2.2.13	Yes	–

Continuation on the next page

Continuation of Table 3.4

ID	Project name	Version	Considered?	Reason for rejection
8	Apache Geronimo	3.0.1	Yes	–
9	Apache Hive	2.3.2	Yes	–
10	Apache HBase	1.3.1	Yes	–
11	Apache Felix	5.6.10	Yes	–
12	Apache ActiveMQ	5.15.2	Yes	–
13	Apache Camel	2.20.1	Yes	–
14	Apache Struts	2.5.13	Yes	–
15	Apache Directory DS	2.0.0	Yes	–
16	Apache Maven	3.5.2	Yes	–
17	Apache Aries Application	1.0.0	Yes	–
18	Apache Spark	2.2.0	Yes	–
19	Apache Jackrabbit	2.16.0	Yes	–
20	Apache ServiceMix	7.0.1	Yes	–
21	Apache Cassandra	3.11.1	Yes	–
22	Apache Qpid	0.23.0	Yes	–
23	Apache OODT	1.2	Yes	–
24	Apache OFBiz	16.11.03	Yes	–
25	Apache Tomcat	9.0.1	Yes	–
26	Apache CXF	3.2.1	Yes	–
27	Apache Portals Pluto	3.0.0	Yes	–
28	Apache Apex Core	3.6.0	Yes	–
29	Apache Atlas	0.8.1	No	Build error
30	Apache Sling	9	Yes	–
31	Apache Bahir Flink	2.2.0	Yes	–
32	Apache Chemistry Client	1.1.0	Yes	–
33	Apache Drill	1.11.0	Yes	–
34	Apache Sentry	1.8.0	No	Build error
35	Apache Tez	0.9.0	Yes	–
36	Apache James	3.0.0	No	Build error
37	Apache cTakes	4.0.0	Yes	–
38	Apache Metron	0.4.1	No	Build error
39	Apache ORC	1.4.1	No	Build error
40	Apache POI	3.17	Yes	–
41	Apache Storm	1.1.1	Yes	–
42	Apache Nifi	1.3.0	No	Build error
43	Apache Reef	0.16.0	No	Build error
44	Apache Synapse	3.0.0	Yes	–
45	Apache Airavata	0.16	Yes	–
46	Apache Deltaspike	1.8.0	Yes	–
47	Apache Ignite Fabric	2.3.0	Yes	–
48	Apache OpenJPA	2.4.2	Yes	–
49	Apache Phoenix HBase	4.13.0	Yes	–
50	Apache Beam	2.1.1	No	Build error
51	toxiclibs	20110103	Yes	–
52	jPOS	2.1.1	Yes	–
53	Axis2	1.7.6	Yes	–
54	iText7	7.0.5	Yes	–
55	Jackrabbit Oak	1.7.11	Yes	–

Continuation on the next page

Continuation of Table 3.4

ID	Project name	Version	Considered?	Reason for rejection
56	jBehave	4.1.3	No	Build error
57	Netty Project	4.1.17	Yes	–
58	Hippo Site Toolkit	5.0.1	No	Build error
59	M2E Android Configurator	1.4.0	No	Build error
60	Apache Karaf	4.2.0	Yes	–
61	Nifty GUI	1.4.2	No	Build error
62	oVirt engine	4.1.8	No	Build error
63	WildFly	12.0.0	Yes	–
64	Apache Tika	1.16	Yes	–
65	Apache Marmotta	3.3.0	Yes	–
66	jogl	2.3.2	No	Build error
67	Apache Zookeeper	3.4.11	Yes	–
68	OrientDB	2.2.30	Yes	–
69	Errai Framework	4.0.2	No	Time limit exceeded
70	Apache UIMA	2.10.2	Yes	–
71	Protocol Buffers	3.5.0	No	Build error
72	WSO2 Carbon	4.4.11	Yes	–
73	Apache Ant	1.10.1	Yes	–
74	Encog	3.4	No	Build error
75	Apache ODE	1.3.7	Yes	–
76	OpenNMS	21.0.0	No	Build error
77	Jope	5.1.0	No	Source not available
78	DataNucleus	5.1.3	Yes	–
79	Apache jclouds	2.0.2	Yes	–
80	OpenDaylight	Carbon SR2	No	Source not available
81	JORAM	5.14.0	Yes	–
82	Apache Cayenne	4.1	Yes	–
83	JaCoCo	0.7.9	Yes	–
84	geogson	1.4.2	Yes	–
85	Eclipse Scout	Oxygen	Yes	–
86	Aion-Unique	NA	No	Source not available
87	Caucho Resin	4.0	No	Source not available
88	Slick 2D	NA	No	Source not available
89	google-gson	2.8.2	Yes	–
90	vert.x	3.5.0	Yes	–
91	Spring Boot	1.5.8	Yes	–
92	GATE	NA	No	Source not available
93	Apache Shindig	2.5.2	Yes	–
94	Hazelcast Jet	0.5	Yes	–
95	Tekir	NA	No	Source not available
96	Android Studio	14.1.2	No	Time limit exceeded
97	Sonar IDE	1.0	No	Build error
98	Eclipse Sapphire	9.1	No	Build error
99	Marauroa	3.9.2	Yes	–
100	Griffon	2.8.0	No	Build error

3.3.3 Tools Assessment Results

Integrability

To analyze the integrability of each tool, results from the literature review (sections section 3.2 & 2.2) were considered. Additionally the source code of the tools were analyzed to find whether the tool has a requirement that it depends on for critical operations. Non critical operations are *e.g.* the drawing of a GUI, critical operations are connected to the parsing and detecting of smells in the code.

JDeodorant. JDeodorant is not a standalone tool, but a plug-in for the IDE Eclipse [TCC08]. JDeodorant depends strongly on Eclipse JDT for the parsing of the to-be-examined code, but more importantly to bind information provided by the compiler after the project is built. To use JDeodorant Eclipse has to be installed and the to-be-examined project imported into the workspace. Efforts were made to make JDeodorant usable outside of the Eclipse GUI based on instructions by the development team[4], but were not successful. Due to this JDeodorant was rejected for this thesis.

DECOR. DECOR comes as part of the Ptidej[5] tool suite and offers a GUI for the detection of code smells. Nonetheless DECOR is also available as a stand alone tool. It does not require any additional environment.

TACO. TACO is a standalone tool and usable without any additional requirements. TACO is therefore integrable into TestSmellDescriber and was further considered for this thesis.

Organic. Organic is a plug-in for the IDE Eclipse. It requires the Eclispe JDT API in order to parse and analyze the syntactical structure of Java classes. Organic was therefore rejected for this thesis.

Applicability on Test Suites

To detect the usability of the accepted tools we selected 67 projects in section 3.3.2. In the next step we examined the projects with the accepted tools TACO and DECOR. To efficiently apply the tools on the projects, we developed the module TestSmellRunner that acts as a common entry point for each project. The module allows for easy integration and extensibility with other detection tools. We additionally modified the tools to only consider code from test classes. For TACO we only consider Java files, which contain the string " extends TestCase" or the annotation "@test". DECOR considers all Java files, that are in a folder whose name contains the string "test" and files, whose names contain the string "test".

TACO. TACO was found to be universally usable on 100% of all selected projects. It was successfully able to parse and examine the test code on smells.

[4] https://users.encs.concordia.ca/ñikolaos/jdeodorant/index.php?option=com_content&view=article&id=87:how-can-i-execute-jdeodorant-in-headless-mode&catid=27:installation-questions&Itemid=41 (last visit: 14.01.18)

[5] http://www.ptidej.net/tools/ (last visit: 14.01.18)

DECOR. DECOR was able to detect smells at least partially in 57 out of 67 projects (85% of all projects). 10 projects were not parsable by DECOR. Upon further investigation it was found that all 10 projects are compiled with the Java SE Development Kit version 9. The current version of DECOR is able to parse bytecode that was compiled with Java 8 or lower. A statistic[6] has shown that in 2017 35% of 1400 different JVMs are using Java 7 or lower. We conclude from this that although Java 8 was released three years before the release of the statistic, older versions are still in use. In 2018, mere months after the release of Java 9, Java 8 or older versions will still be in use and demand for a tool that is able to parse Java 8 compiled code will still be there. DECOR was therefore further used in this thesis.

The list of projects and their applicability with the tools can be found in table 3.6.

Detectable Smells

The tools were further modified to allow a summarization of each detected smell in all 67 projects. The result shows that there are 9 smells that are detectable by TACO and DECOR. Both TACO test smells (Eager Test & General Fixture) were found in the test projects. Out of the code smells that are detectable by DECOR, 6 of them were found in test code: Long Parameter List, Long Method and Message Chains, Lazy Class, Speculative Generality, Large Class and Refused Bequest. The full result along with the number of occurrence can be found in table 3.5, while table 3.7 displays which smell was found in each project. We decided to further concentrate on the smells that have some relevance in test code. A smell has to be detected in at least 10% of all tested projects (7 projects). The for the summarization considered smells are therefore:

- Long Parameter List
- Eager Test
- General Fixture
- Long Method
- Lazy Class
- Refused Parent Bequest

Table 3.5: Occurrence of detected smells in the test projects

ID	Smell	Tool	# of occurences	in # of projects
R1	LongParameterList	DECOR	23	9
R2	EagerTest	TACO	54797	60
R3	MessageChains	DECOR	1	1
R4	GeneralFixture	TACO	926	48
R5	LongMethod	DECOR	71	11
R6	LazyClass	DECOR	16	8
R7	SpeculativeGenerality	DECOR	6	4
R8	LargeClass	DECOR	3	1
R9	RefusedParentBequest	DECOR	44	10

[6] https://plumbr.io/blog/java/java-version-and-vendor-data-analyzed-2017-edition (last visit: 14.01.18)

Table 3.6: Results of applicability. D: DECOR, T: TACO

ID	Project name	D	T
1	Apache Hadoop	Y	Y
2	Apache Cloudstack	N	Y
7	Apache MyFaces Core	N	Y
8	Apache Geronimo	N	Y
9	Apache Hive	N	Y
10	Apache HBase	Y	Y
11	Apache Felix	Y	Y
12	Apache ActiveMQ	Y	Y
13	Apache Camel	Y	Y
14	Apache Struts	Y	Y
15	Apache Directory DS	Y	Y
16	Apache Maven	Y	Y
17	Apache Aries Application	N	Y
18	Apache Spark	Y	Y
19	Apache Jackrabbit	Y	Y
20	Apache ServiceMix	Y	Y
21	Apache Cassandra	Y	Y
22	Apache Qpid	Y	Y
23	Apache OODT	Y	Y
24	Apache OFBiz	Y	Y
25	Apache Tomcat	Y	Y
26	Apache CXF	Y	Y
27	Apache Portals Pluto	Y	Y
28	Apache Apex Core	Y	Y
30	Apache Sling	Y	Y
31	Apache Bahir Flink	Y	Y
32	Apache Chemistry Client	Y	Y
33	Apache Drill	N	Y
35	Apache Tez	Y	Y
37	Apache cTakes	Y	Y
40	Apache POI	Y	Y
41	Apache Storm	Y	Y
44	Apache Synapse	Y	Y
45	Apache Airavata	Y	Y

ID	Project name	D	T
46	Apache Deltaspike	Y	Y
47	Apache Ignite Fabric	N	Y
48	Apache OpenJPA	Y	Y
49	Apache Phoenix HBase	Y	Y
51	toxiclibs	Y	Y
52	jPOS	N	Y
53	Axis2	Y	Y
54	iText7	Y	Y
55	Jackrabbit Oak	N	Y
57	Netty Project	Y	Y
60	Apache Karaf	Y	Y
63	WildFly	N	Y
64	Apache Tika	Y	Y
65	Apache Marmotta	Y	Y
67	Apache Zookeeper	Y	Y
68	OrientDB	Y	Y
70	Apache UIMA	Y	Y
72	WSO2 Carbon	Y	Y
73	Apache Ant	Y	Y
75	Apache ODE	Y	Y
78	DataNucleus	Y	Y
79	Apache jclouds	Y	Y
81	JORAM	Y	Y
82	Apache Cayenne	Y	Y
83	JaCoCo	Y	Y
84	geogson	Y	Y
85	Eclipse Scout	Y	Y
89	google-gson	Y	Y
90	vert.x	Y	Y
91	Spring Boot	Y	Y
93	Apache Shindig	Y	Y
94	Hazelcast Jet	Y	Y
99	Marauroa	Y	Y

Table 3.7: Detected smells in the test projects

ID	Project name	Smell
1	Apache Hadoop	R1, R2, R4, R5, R6, R7, R9
2	Apache Cloudstack	R2, R4
7	Apache MyFaces Core	–
8	Apache Geronimo	R2, R4
9	Apache Hive	R2, R4
10	Apache HBase	R1, R2, R4, R5, R6
11	Apache Felix	R2, R4
12	Apache ActiveMQ	R2, R4
13	Apache Camel	R1, R2, R4, R6, R7, R9
14	Apache Struts	R2, R4
15	Apache Directory DS	R2, R4
16	Apache Maven	R2
17	Apache Aries Application	R2, R4
18	Apache Spark	R1, R2, R4, R6, R9
19	Apache Jackrabbit	R1, R2, R4, R6, R9
20	Apache ServiceMix	R1, R5, R8, R9
21	Apache Cassandra	R2, R4
22	Apache Qpid	–
23	Apache OODT	R1, R2, R4, R5, R9
24	Apache OFBiz	R1, R2, R3, R5, R9
25	Apache Tomcat	R2, R4
26	Apache CXF	R2, R4
27	Apache Portals Pluto	R2
28	Apache Apex Core	R2
30	Apache Sling	R2
31	Apache Bahir Flink	–
32	Apache Chemistry Client	R2
33	Apache Drill	R2, R4
35	Apache Tez	R2, R4
37	Apache cTakes	R2, R4
40	Apache POI	R2, R4
41	Apache Storm	R2, R4, R5, R6, R9
44	Apache Synapse	R2, R4
45	Apache Airavata	R2, R4
46	Apache Deltaspike	R2, R5, R6, R9
47	Apache Ignite Fabric	R2, R4
48	Apache OpenJPA	R2, R4
49	Apache Phoenix HBase	R2, R4
51	toxiclibs	R2, R4
52	jPOS	R2, R4
53	Axis2	R2, R4
54	iText7	R2, R5, R7
55	Jackrabbit Oak	R2, R4
57	Netty Project	R2, R4
60	Apache Karaf	R2, R4
63	WildFly	R2, R4
64	Apache Tika	R2, R4, R5
65	Apache Marmotta	R2, R4
67	Apache Zookeeper	R2, R4
68	OrientDB	R2, R4, R5, R7
70	Apache UIMA	R2, R4
72	WSO2 Carbon	R1, R2, R5, R9
73	Apache Ant	R2, R4, R6
75	Apache ODE	R2, R4
78	DataNucleus	R2, R4
79	Apache jclouds	R2, R4
81	JORAM	R2
82	Apache Cayenne	R2, R4
83	JaCoCo	R2
84	geogson	–
85	Eclipse Scout	R2, R4
89	google-gson	R2
90	vert.x	R2, R4
91	Spring Boot	R2, R4
93	Apache Shindig	R2, R4
94	Hazelcast Jet	R2
99	Marauroa	R2

Chapter 4

TestSmellDescriber

In this chapter we depict the development and structure of our tool TestSmellDescriber. Section 4.1 talks about the architecture of the tool and gives a general picture of its structure. All further sections describe the individual modules and steps that are necessary to construct a description for found smells and are outlined in section 4.1. The source code of TestSmellDescriber can be checked out from our repository[1].

4.1 Architecture

TestSmellDescriber consists of four modules that are used to check an object of interest on smells and generate a textual description. The activity diagram in figure 4.1 shows the structural organization of TestSmellDescriber along with the interaction among its modules, while figure 4.2 shows an overview of the module `TestSmellUtil`. In the following are the steps for the detection of test smells and generation of descriptions explained, which are enumerated in the activity diagram.

(1) `TestSmellRunner` is the interface that is called by a tool to start the process of detecting and generating test smell descriptions. The `TestSmellRunner` delegates the passed input consisting of the root folder of a project and the desired output folder to the smell detection tools DECOR and TACO. The class diagram of `TestSmellRunner` can be viewed in figure 4.3. The module will be further explained in section 4.2.

(2) The smell detection tools TACO and DECOR gather all files that are part of the test suite and examine the code for any smells (3). The inner working of DECOR and TACO along with the detection of the smells will be explained in section 4.3.

(3) Whenever a new smelly entity was found, that information will be forwarded to the *TestSmellUtil*. All detected smelly entities are stored in the `SmellyClassContainer`, which consists of 0–N `SmellyClass` objects. A `SmellyClass` object denotes a class which has a smell. This smell can be specific to the whole class or an attribute, *i.e.*, a method or field. To define this `SmellyClass` objects have a `SmellyAttributeContainer`, which stores 0–N `SmellyAttribute` objects. A `SmellyAttribute` can be a `SmellyField` or a `SmellyClass` object. Before a smell is assigned to a `SmellyClass` the enum `ClassSmell` is used to evaluate, whether a smell is explicit to a class or an attribute. The smell detection tools use different libraries to read and store classes, methods and attributes. To allow the implementation of different smell detection tools into TestSmellDescriber the `EntityAdapter` is

[1] http://bit.ly/2DUiZrC

available to unify the returned entities. The detailed class diagram showing the classes involved in the storage are in figure 4.9. This along with the changes made to DECOR and TACO to gather those information will be detailed in section 4.4.

(4) *TestSmellUtil* then generates a textual description of the problems in a class and their belonging methods along with procedures to remove those problems. This is achieved by calling the `TestSmellDescriptionWrapper`, which acts as the entry point for obtaining a smell description. The `TestSmellDescriptionWrapper` obtains the specified smelly entity from the `SmellyClassContainer`, uses the `SmellyParser` to gather necessary quantitative data about the smelly entity and the project, and finally pases those values to `Description` which generates the textual descriptions. This step will be explained in detail in section 4.5.

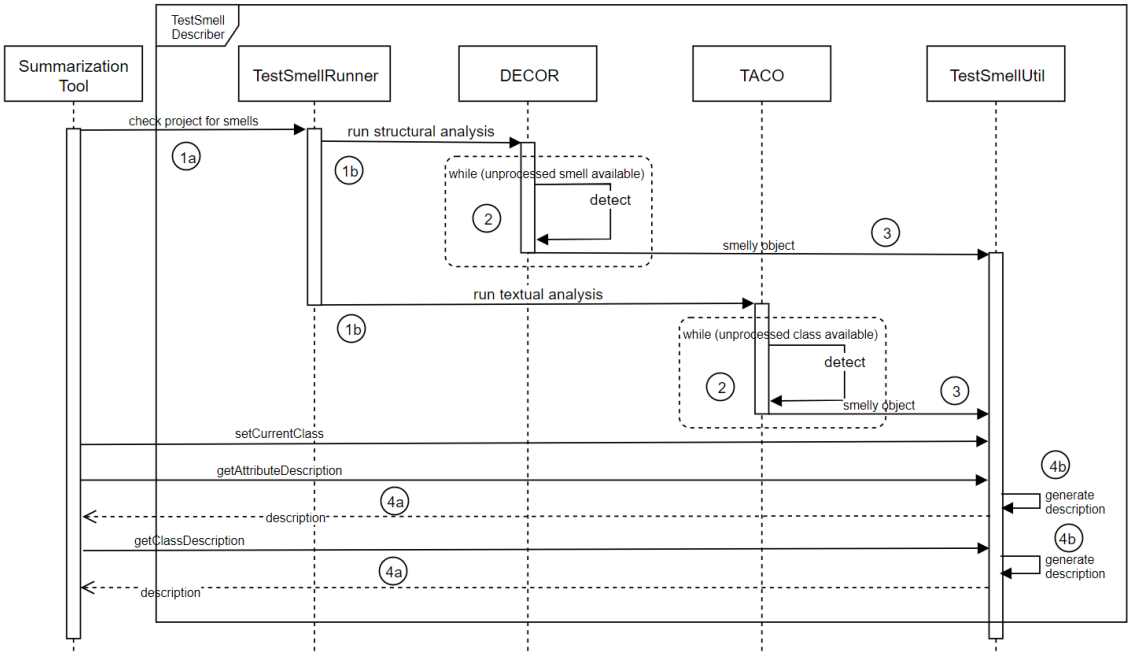

Figure 4.1: Activity Diagram: TestSmellDescriber

4.2 Test Smell Runner

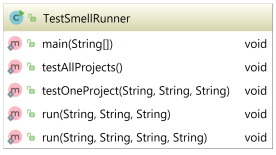

Figure 4.3: Class Diagram: TestSmellRunner

The TestSmellRunner consists of multiple functions that are used to start the smell detection process. The functions `run(String, String, String, String)` and `run(String, String, String)` allow the specification of the smell detection method which will be applied on the project. The methods require the following parameters:

1. The first parameter denotes the type of smell detection, currently this can be either "textual", denoting a smell detection by TACO, or "structural", which starts the smell examination using DECOR.

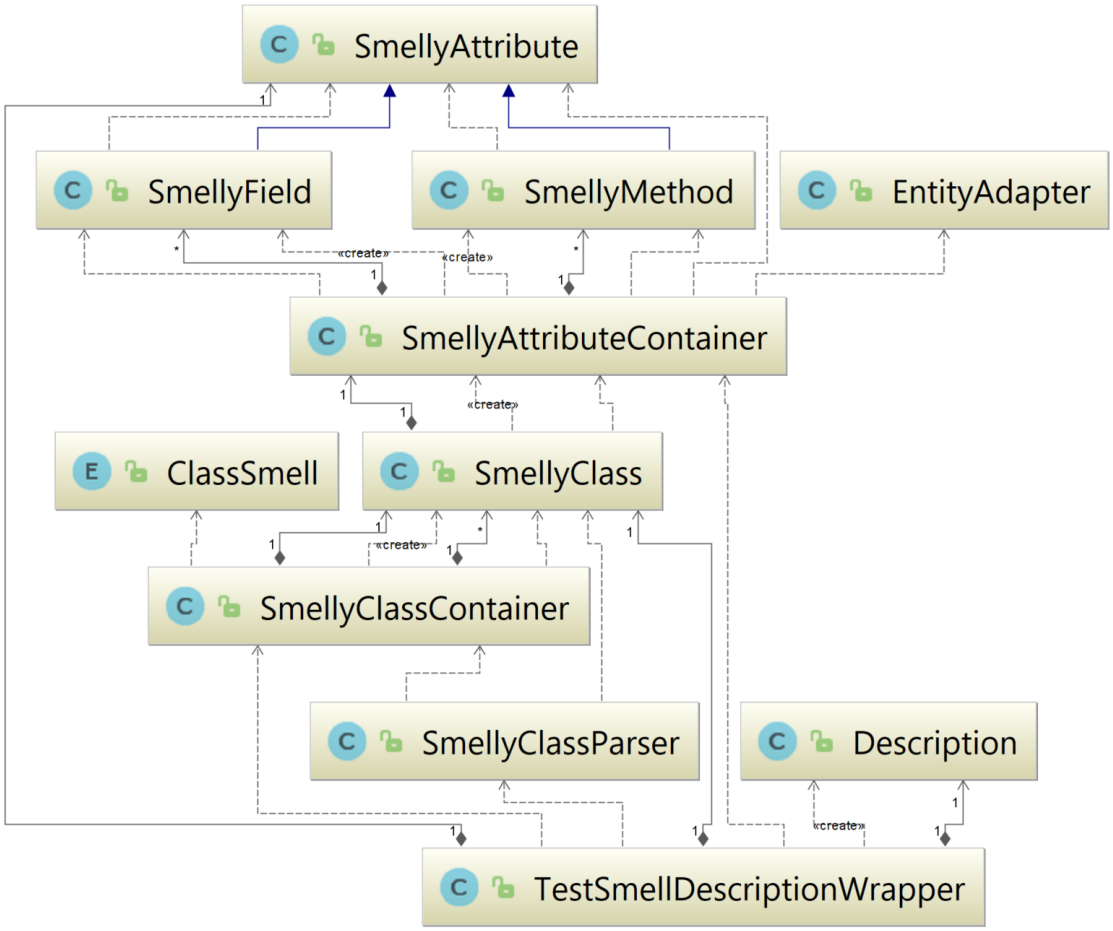

Figure 4.2: Class Diagram: TestSmellDescriber

2. It further requires the path where the root folder of the project is located.
3. The third parameter points to the requested output path for the results generated by TACO and DECOR.
4. As already mentioned is DECOR detecting smells in the code by examining the bytecode. To correctly identify the files either "class" or "jar" needs to be specified, denoting whether the files can be found in class or JAR format. This parameter can be omitted, if only a textual detection is requested, *i.e.*, a detection by TACO.

If a detection with all detection tools is requested, the method `testOneProject(String, String, String)` can be invoked. The method requires three parameters:

1. The first parameter is the path to the project to be parsed. To use both DECOR and TACO for the test smell detection, both binary and source code of the project have to be available. TestSmellRunner delegates the sub folder */src/* of the passed project path to TACO, while it passes the sub folder */bin/* to DECOR.
2. The second parameter is the path to the desired output destination for the code smell detection tools. Here detailed information about the smell detection results can be viewed.

3. The last parameter declares the file type of a project's binary files. This can be either "class", or if all class files are packaged in a JAR file "jar".

To replicate the in section 3.3 conducted analysis of the smell detection tools, the method `testAllProjects()` can be used.

4.3 Smell Detection

This section describes the inner working of the smell detection tools DECOR and TACO in detail. Besides presenting the tools in general, we also present each method and metric used for the detection of the 9 smell that were evaluated in section 3.3.3.

4.3.1 DECOR

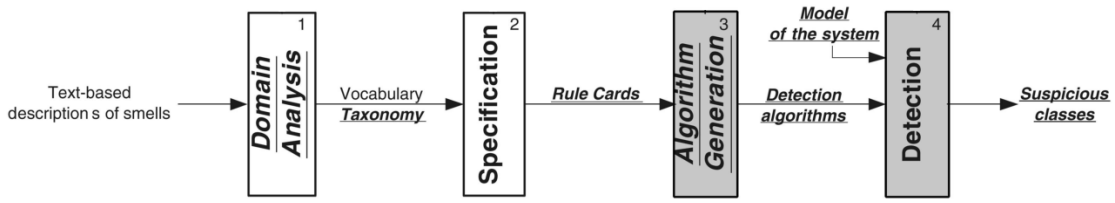

Figure 4.4: DECOR detection technique (Boxes represent steps and arrows connect inputs and outputs of each step. Gray boxes are fully automated steps.) Source: [MGDLM10]

To define the detection of a smell in DECOR a code smell has to be first analyzed (1) and then translated into their own domain specific language (2), coined *POM*. The result is a "Rule Card" that describes the detection of a smell. DECOR then uses those Rule Cards to automatically generate an algorithm (3). The resulting algorithm is then used on the model of the system (4), *i.e.*, the code under test. Unlike other research DECOR performs the translation of code/design smell specifications into detection algorithms transparently. This process can be viewed in figure 4.4. Before starting the detection process, DECOR first finds the list of files that are to be examined. These are either all *JAR* files, or all *Test Class* files that are contained in the root and every subsequent sub folders. The Test Class files are all class files, that have a parent folder with a name that contains the string "test". Here we cover all projects that follow the Standard Directory Layout defined by the Apache Maven Project[2]. To cover projects that do not follow the Maven guidelines we additionally search for files with a name that contains the string "test". After creating a model from the files, DECOR goes through the list of detection classes and examines the model for anti-patterns.
In the following are the metrics and methods explained for the detection of the from DECOR detectable smells as defined in section 3.3.3.

Large Class. The detection of a *Large Class* is done with the combination of two rules: Large Class Only and Low Cohesion Only. The result of the *Large Class* detection is done by performing

[2]https://maven.apache.org/guides/introduction/introduction-to-the-standard-directory-layout.html (last visit: 17.01.18)

an intersection of both intermediate findings ($LC = LClO \cap LCoO$). The Rule Card for the Large Class smell can be viewed in Listing 4.1. The individual rules are applied from bottom to top.

```
RULE_CARD : LargeClass {
   RULE : LargeClass { INTER LargeClassOnly LowCohesionOnly } ;
   RULE : LargeClassOnly { (METRIC: NMD + NAD, VERY_HIGH, 0) } ;
   RULE : LowCohesionOnly { (METRIC: LCOM5, VERY_HIGH, 20) } ;
};
```

Listing 4.1: Rule Card for Large Class Smell taken from DECOR source code

A rule is defined with a metric that is applied on all entities of a system, and values that denote if the rule has been met. The first value is a numerical or ordinal value. While numerical values define thresholds, ordinal values define values that are relative to all the entities of a system [MGLM+10]. The second value is the acceptable degree of fuzziness, "which is the margin acceptable in percentage around the numerical value [...] or around the threshold relative to the ordinal value" [MGLM+10]. To detect a Large Class smell the metric of the first rule *LowCohesionOnly* is applied on all entities of a model. The metric returns a number which identifies the lack of cohesion in methods of an entity. From the set of values only the high outliers with a fuzziness of 20 are considered. The second rule *LargeClassOnly* is defined by a combination of two metrics. `NMD` returns the number of methods declared by an entity, while `NAD` returns the number of attributes declared by an entity. From the resulting sum are again only the high outliers considered. Here with a fuzziness of 0. Both rules then contain a set of classes. By performing an intersection of both sets we obtain a list of classes that contain a Large Class smell.

Long Parameter List. The rule card for a Long Parameter List smell consists of one rule: `METRIC: NOParam, VERY_HIGH, 20`. To detect a LPL smell the metric `NOParam` is first applied on all entities of a model. The metric returns a value denoting the largest number of parameters from all methods of a class. Entities contain a LPL smell if that value is a high outlier (`VERY_HIGH`) with a set fuzziness of 20. The rule card for the LPL smell can be seen in Listing 4.2.

```
RULE_CARD : LongParameterList {
   RULE : LongParameterListClass { (METRIC: NOParam, VERY_HIGH, 20) } ;
};
```

Listing 4.2: Rule Card for Long Parameter List Smell taken from DECOR source code

Message Chains. The rule card for a Message Chains (MC) smell also consists of one rule: `METRIC: NOTI, SUP_EQ, 4, 0`. The metric `NOTI` returns the highest number of transitive invocations among methods of a class. The rule is met if that number is larger than or equal to (`SUP_EQ`) 4.0. The full rule card for the MC smell can be seen in Listing 4.3.

```
RULE_CARD : MessageChains {
   RULE : MessageChainsClass { (METRIC: NOTI, SUP_EQ, 4, 0) } ;
};
```

Listing 4.3: Rule Card for Message Chains taken from DECOR source code

Long Method. The lone rule for the Long Method (LM) smell is `METRIC: METHOD_LOC, HIGH, 8`. Again is the metric first applied on all entities of a class. The metric `METHOD_LOC` calculates the number of line of codes for the largest method of an entity. A method is smelly if it is in the 75th percentile or upper quartile (`HIGH`) with a set fuzziness of 8. Listing 4.4 displays the rule card for the LM smell.

```
RULE_CARD : LongMethod {
   RULE : LongMethodClass { (METRIC: METHOD_LOC, HIGH, 8) };
};
```

Listing 4.4: Rule Card for Long Method Smell taken from DECOR source code

Refused Parent Bequest. Refused Parent Bequest consists of three rules as can be seen in listing 4.5. The metric `IR` for rule `RareOverriding` computes the number of accepted bequests, *i.e.*, how many protected methods and fields of a superclass a subclass has overwritten or called. If the class has no superclass, the metric returns 0. From the computed values only the low outliers are considered (`VERY_LOW`). These are the values that are lower than the minimum bound. The result of the first rule is therefore a list of all classes that have no parents or use only a low amount of the fields and methods of their parents. The second rule `ParentClassProvidesProtected` uses the metric `USELESS`, which returns the value 1.0. By checking if the value is equal to 1.0 (`[...] EQ, 1,0`) a list of all entities in the model is returned. The last rule examines both list of classes on their relationship, specifically if there exists an inheritance between a class in the first list and the classes in the second list, *i.e.*, all other classes. By applying this rule the classes from rule `RareOverriding` are filtered based on whether they extend an entity.

```
RULE_CARD : RefusedParentBequest {
   RULE : RefusedParentBequest {
            INHERIT: inherited FROM: ParentClassProvidesProtected
            ONE TO: RareOverriding ONE } ;
   RULE : ParentClassProvidesProtected { (METRIC: USELESS, EQ, 1,0) } ;
   RULE : RareOverriding { (METRIC: IR, VERY_LOW, 0)};
};
```

Listing 4.5: Rule Card for Refused Parent Bequest Smell taken from DECOR source code

4.3.2 TACO

Unlike DECOR which analyzes a system structurally, TACO detects smells in the code by performing a textual analysis. The detection of test files is also performed textually, as it only considers Java files that contain the strings " extends TestCase" or the annotation "@test". TACO then iterates through the list of test files and examines them for smells. TACO then detects smells by evaluating textual information that are contained in elements of the source code and by computing the textual similarity between code elements. To achieve this TACO computes the cosine similarity (CS) between two textual elements. Before the cosine similarity can be computed the content of those elements are first normalized. For this, terms are first split by underscores, capital letters and digits, then set to lower case letters, and lastly are special characters, common English stop words and programming keywords removed. In a second step are the normalized words then weighted by computing their occurrence within the content. The resulting normalized elements are thereupon modeled as vectors. The textual similarity is measured as the cosine of the angle between those vectors [BDLMO14].

General Fixture. To detect a General Fixture, TACO first searches for a test fixture within the test class, by searching for a method with the name "setUp". A fixture is a General Fixture, if the following conditions hold[3]:

[3]Source: TACO source code.

- The fixture is composed of more than or equal to 10 lines of code.
- The test class of that fixture contains at least two methods that have a cosine similarity compared to the fixture larger than 0.0.
- The cosine similarity between those two methods is smaller than 0.1.

Eager Test. To detect if a test case (TC) is an Eager Test, TACO first gathers the content of all production methods (PM) that are called (*i.e.*, tested) within the test case. Here, only methods with more than 3 lines of code are considered. A test case is an Eager Test[3], if the sum of all cosine similarities between test case and production methods ($CS(TC, PM)$), and the sum of all cosine similarities of the n-permutations of all production methods ($CS(PM!)$) is smaller than $\frac{2N^2-3N}{5}$, where N is the number of called production methods. With a test case that calls two production methods (A & B), the following has to be true, for the test case to be considered an Eager Test: $CS(TC, A) + CS(TC, B) + 2 \cdot CS(A, B) < \frac{2N^2-3N}{5}$.

4.4 Gather Information from Detection Tools

4.4.1 Storing Smell Information

We have constructed several entities to retain smell information. The class diagram of the storage can be found in figure 4.9. The `SmellyClassContainer` stores the smelly entities in `SmellyClass` objects. By invoking `addClass(String)` a new `SmellyClass` object is created and added to the list `smellyClasses`. The smelly part of an entity can either be the whole class (*i.e.*, *Class Smell*) or a smelly attribute (*i.e.*, *Field* or *Method Smell*). *E.g.* a Large Class smell denotes that the class is smelly, whereas a Long Method denotes a smelly attribute. Those attributes are stored in `SmellyMethod` or `SmellyField` objects and are added by invoking `addMethod(String, String)` or `addField(String, String)`, and by passing the header of the attribute, along with the name of the smell. If a class has been found to being guilty of having a Class Smell, the class is set smelly by invoking `setClassSmelly(String)`, while passing the name of the smell. To differentiate between a Class and Attribute Smell, the enum `ClassSmell` is available. The method `isClassSmell(String)` compares the values stored in the enum to the passed smell. To retain the number of occurrences a smell has in a class, the smells are stored as keys in a Map and assigned an Integer denoting the number of appearances. Each `SmellyClassContainer` holds a `SmellyAttributeContainer`, which retains Attribute Smells, *i.e.*, `SmellyMethod` and `SmellyField`. The containers store the `SmellyClass` and `SmellyAttribute` objects and act as the entry point for the generation and obtainment of those objects. The `SmellyClassContainer` is a singleton. This enables us to obtain one class container for all detection tools.
Important to note is that TACO and DECOR use different libraries to store class entities and their attributes. TACO uses the Eclipse JDT API, while DECOR uses the library Toad[4]. This leads to some irregularities regarding method parameters.

1. TACO maps every subclass of `Collection` to its superclass.
2. DECOR maps `I` to `Iterable`.
3. DECOR maps `org.w3c.dom.Element` to `w3Element`.
4. DECOR adds the package path to classes.

[4]http://www.research.ibm.com/haifa/projects/systems/cot/toad/

The class `EntityAdapter` is available to accurately store, obtain and compare methods from different tools. To account for the first three cases listed above, the method `compareMethodNames(String, String)` is used to compare method headers between DECOR, TACO and TestDescriber. The method `equalizeParameterStyle(String)` is used to equalize the method headers from DECOR and TACO, by removing all package paths from the parameters.

4.4.2 TACO

The test smell detection of TACO takes place in the `TestSmellDetectionRunner` (see class diagram in figure 4.5). The class has access to all the information about the class under test and its methods. At the start of the detection, the class under test is relayed to the `SmellyClassContainer` by invoking `addClass(String)`, which preserves the class in a `SmellyClass` object. The `TestSmellDetectionRunner` passes the test class to be examined to the `GeneralFixtureRule` class, and each test case to the `EagerTestRule`. After examining a method for a smell, a boolean is returned reporting the result of the detection. If a smell was detected, the method along with the smell are reported to the `SmellyClassContainer` by calling `addMethod(String, String)`, which in turn adds `SmellyMethod` object to the `SmellyClass` object. If the class was not found to being smelly, the SmellyClass is removed from the SmellyClassContainer. The process can be view in the activity diagram in figure 4.6.

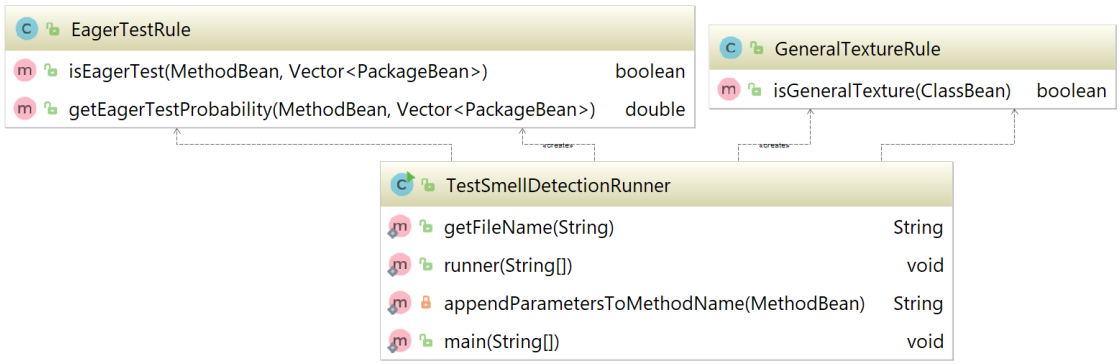

Figure 4.5: Class Diagram: TACO

4.4.3 DECOR

The smell detection in DECOR takes place in many entities. As already mentioned is a smell composed of one or more rules. In DECOR each rule is called a "code smell", whereas a code smell, as defined in this thesis, is called "design smell". DECOR automatically generates detection classes (*i.e.*, `DesignSmell` and `CodeSmell` classes) through the composition of a Rule Card and the execution of `RuleCreator`. By defining a new Rule Card or by editing an existing one a new algorithm is created that dictates the detection process. To allow flexibility in the definition of smell detections, we decided not to gather information in the concrete detection classes, since a redefinition of a smell detection algorithm would remove the collection of smell information. The activity diagram in figure 4.7 shows a simplified version of the detection and forwarding of information to the TestSmellUtil. `IDesignSmellDetection` is the interface of a concrete `DesignSmell`,

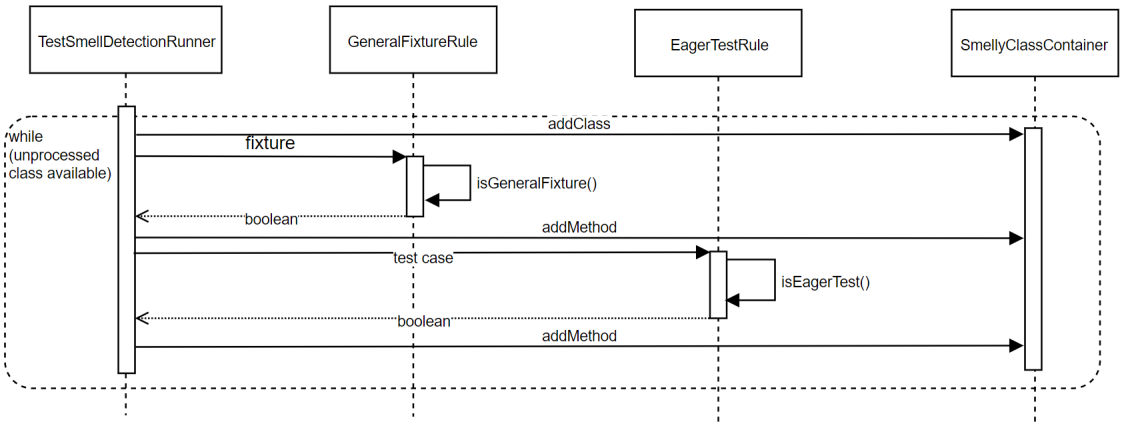

Figure 4.6: Activity Diagram: TACO – smell detection and forwarding of information

while `ICodeSmellDetection` is the interface of a concrete `CodeSmell`. Figure 4.10 displays a class diagram that shows the hierarchy of the detection classes using the example of the Long Method smell detection. In DECOR a smelly class is composed of a `ClassProperty` object, and 0 – N other objects, such as `FieldProperty`, `MethodProperty` and `MetricProperty`. To relay the information to TestSmellUtil we iterate over the collection of ClassProperties and map them to their corresponding entities in the TestSmellUtil, *i.e.*, `FieldProperty` to `SmellyField` and `MethodProperty` to `SmellyMethod`. If a `ClassProperty` has no `FieldProperty` or `MethodProperty` the smell in question is a Class Smell.

4.5 Generation of Descriptions

After detecting a smell in a test class, that information will be displayed to the developer. The goal is to help developers to improve their code, therefore, alongside to bringing attention to the problem itself by displaying a description, we additionally relay a text describing how to eliminate the discovered problem. These descriptions are relayed to the user as comments in the source of his code. To augment a class with descriptions we will use the tool TestDescriber (section 2.4). The tool is able to contribute to the quality of tests by summarizing the code under test as well as delivering information regarding the coverage of each case. We integrate our tool TestSmellDescriber into TestDescriber for two reasons:

- As TestDescriber is able to deliver some qualitative information regarding the test code by making an assessment about its code coverage, we want to further contribute to that fact by delivering more qualitative information. This information is attained by our tool TestSmellDescriber by examining the code for test smells and generating descriptive information to help developers to narrow down the problem and to improve their code.
- TestDescriber is able to read the content of a test code and to define or augment the comments of the class or its methods. We will use this ability to append our descriptions to the code.

The description texts are composed with the smell specifications and categorizations by Fowler, van Deursen, Mäntylä and Meszaros (sections 2.1.1 & 2.1.2). There are short and long version of descriptions available. The long smell descriptions are used at the class level, while short

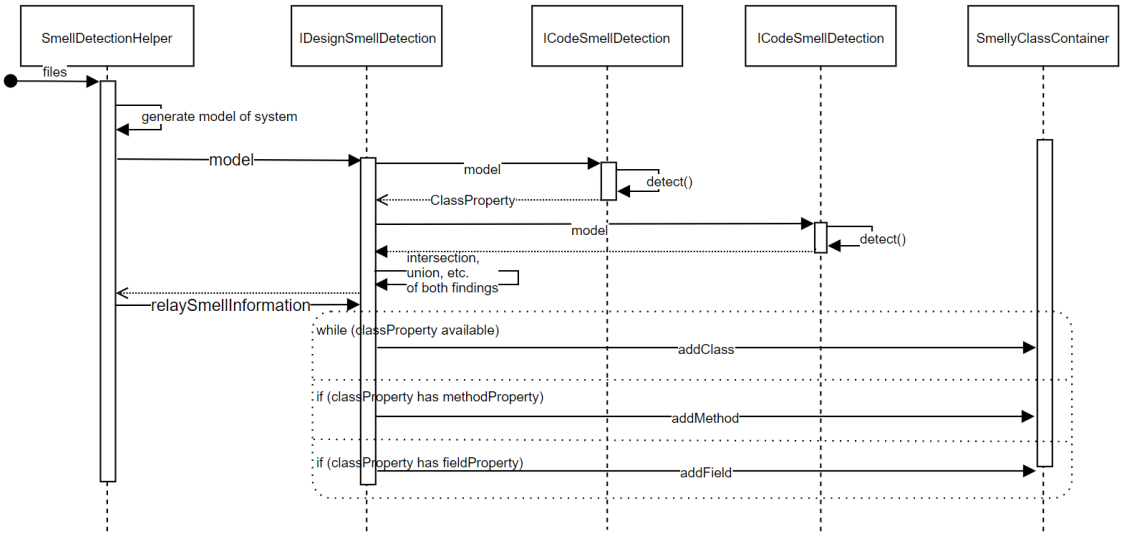

Figure 4.7: Activity Diagram: DECOR – smell detection and forwarding of information

smell descriptions are for method level comments. The smell descriptions serve the purpose of defining the (potential) problem to the developer. By relaying a detailed description of the smell we aim to provide the opportunity for the developer to first assess the situation and the severity of the potential problem that is caused by the smell, and to localize the cause of the smell. If a smell is strong by causing problems and confusions, a refactoring should be performed [Fow99]. The shorter method descriptions further assist in localizing the cause of a smell. Parallel to smell descriptions are refactoring descriptions available in short and long form. After assessing the severity of the problem caused by a smell and localizing its root through the smell description, the refactoring description provides the information to eliminate a smell by performing a refactoring. We number refactoring descriptions to allow users to easily link the shorter method refactoring messages to the detailed descriptions located in the class comment. The texts are parameterized and are adapted using the StringTemplate library[5]. The parameterized descriptions can be found in our repository[6]. These strings can be obtained by invoking the methods `getAttributeDescription()` and `getClassDescription()` from the class `TestSmellDescriptionWrapper`, which gathers information from the `SmellyClassContainer`, and uses the `SmellyClassParser` to calculate required values. The class `Description` then generates the textual description using the library StringTemplate. The class diagram for the obtainment of description can be found in figure 4.8.

4.5.1 Class Descriptions

Class descriptions consist of four elements:

a) a pretext stating that problems were found,

b) a long detailed description of the smell,

c) a long description on how to conduct a refactoring to remove the smell,

[5]http://www.stringtemplate.org/
[6]Folder: */TestDescriberTestSmellUtil/descriptions/*, link to repository: http://bit.ly/2DUiZrC

d) and a quantitative description denoting the frequency of the smell in the class and the whole project.

In addition to describing the smell, the quantitative data helps a developer in evaluating the severeness of the problem in the class and the whole project. By identifying this developers will not only be able to assess the quality of their current tests, but also gain knowledge into their overall test writing performance. Listing 4.6 displays part of the class smell description for the class `UtilCacheTest` from Apache OFBiz. Here we can observe the different elements of the description outlined above.

```
   /**
a) * Some problems were detected in this test class:
b) * - This test contains a method that does too many things at once. This
   *  makes the code hard to understand and maintain.
c) *  (0) Shorten the method using Fowler's Extract Method, by finding parts
   *  of the method that go together and extracting them to new methods.
d) *  This method accounts for 50.00% of all found problems in this test class.
   *  This smell represents 28.85% of all found problems in the project with
   *  6.67% occurring in this test.
   **/
```

Listing 4.6: Class Smell Description for UtilCacheTest

Before obtaining a class description the currently viewed class is passed to `TestSmellDescriptionWrapper` by invoking `setCurrentClass(String)`. This obtains the corresponding `SmellyClass` object from the `SmellyClassContainer`. A class description is then generated by calling `getClassDescription()` on the `TestSmellDescriptionWrapper`. First the pretext is generated by invoking `Description::preClassDescription(int)` and passing the number of found smells, which is provided by the `SmellyClass` object, then all class smell descriptions are generated by invoking `getAllClassDescriptions(Iterator)`, passing the smells iterator that is obtained from the `SmellyClass` object. The method obtains the texts from `Description` by invoking `classDescription(int, String)` to generate the long detailed description of the smell, `longRefactoring(int)` to generate the long refactoring description, and `postClassDescription(int, String, String, String)` to generate the quantitative description.

Class Smell Descriptions

Table 4.1 lists the class smell descriptions that are displayed to the developer.

Table 4.1: Class level smell descriptions

Smell	Description
Eager Test	This test contains a method that checks/<numberOfMethods> number of methods that check too many methods of <classUnderTest> in a single test case, which makes this test difficult to read, understand and maintain.
General Fixture	The fixture in this test is too general. Individual test cases only require part of the provided features. This makes this test hard to read, understand, and may slow down the execution of test cases.
Long Parameter List	This test contains methods/a method that requires a large amount (<amountOfParameters>) of parameters. A long list of parameters is hard to understand, to use and may becomes inconsistent over time.

Continuation on the next page

Continuation of Table 4.1

Smell	Description
Long Method	This test contains methods/a method that does too many things at once. This makes the code hard to understand and maintain.
Lazy Class	This class has been found to do too little, which artificially increases the code size and makes the system hard to maintain.
Refused Bequest	This test class does not require all methods and fields of its superclass. This often points to a wrong hierarchy. The relation hurts the clarity and organization of your system.

Class Refactoring Descriptions

Table 4.2 lists all class refactoring descriptions that are displayed to the developer.

Table 4.2: Class level smell refactoring descriptions

Smell	Refactoring description
Eager Test	(<refactoringNumber>) Consider using Fowler's Extract Method technique on the guilty test case by separating the code into test methods that only test one method. Simultaneously improve the documentative purpose of this test by using meaningful names that describe the goal of the individual test cases.
General Fixture	(<refactoringNumber>) Use the fixture only for code that is shared by all tests. Extract the rest using Fowler's Extract Method technique into the methods that use it. This will speed up the execution of your test.
Long Parameter List	(<refactoringNumber>) If all parameters belong to the same object, replace the individual data with the object itself. Otherwise introduce a new object that contains all the passed data.
Long Method	(<refactoringNumber>) Shorten the method using Fowler's Extract Method, by finding parts of the method that go together and extracting them to new methods.
Lazy Class	Refactor it by eliminating this class.
Refused Bequest	Refactor your test by creating a new sibling and pushing your fields and methods down to the new sibling.

Lazy Class and *Refused Bequest* are Class Smells and are therefore not numbered.

Quantitative Descriptions

The quantitative description relays three information to the user. First it states how dominant the smell is in the class compared to all the smells in the class. The calculation for this is $100 \times \frac{thisSmellOccurrencesInClass}{allSmellOccurencesInClass}$. Secondly, it states how often this smell was found in the project relative to all found smells in the project by calculating $100 \times \frac{thisSmellOccurencesInProject}{allSmellOccurencesInProject}$. And lastly it displays how often this smell has occurred in this class compared to all the smell occurrences in the project with $100 \times \frac{thisSmellOccurrencesInClass}{thisSmellOccurencesInProject}$.
The following shows the string template that is used to display the quantitative description.

> *This method accounts/This accounts for* <smellInClassToAllInClass>*% of all found problems in this test class. This smell represents* <smellInProjectToAllInProject>*% of all found problems in the project with* <smellInClassToAllInProject>*% occurring in this test.*

With the help of the quantitative description, we give the developer the opportunity to assess the problematic areas in the current class, as well as over the whole project. This information indicates reoccurring problems or deficit in the test code. By acknowledging this users are able to assess their overall test writing skills, and to not only improve their current code by eliminating smells, but to improve their tests for the future.

4.5.2 Method Descriptions

Method level comments are used to narrow down the root of the problem. Those comments are generated for Method Smells, *i.e.*, problems whose source of smell is a method. The Method Smells are Eager Test, General Fixture, Long Parameter List and Long Method. Method descriptions consist of two elements:

a) a short description of the smell,

b) a short refactoring message pointing to the long description in the class comment.

These elements can also be observed in listing 4.7, which presents the method description for `UtilCacheTest::asserKey(String, UtilCache, K, V, V, int, Map)`. Here we can additionally perceive the subsequent numbering of the refactoring description that links the method refactoring message (listing 4.7b) to the class refactoring message (listing 4.6c).

```
   /**
a) * - This method requires too many parameters.
b) *  Apply a refactoring suggested in (0) to improve the code.
   */
```

Listing 4.7: Method Description for UtilCacheTest.assertKey

The method descriptions are obtained by calling `getAttributeDescription()` on the `TestSmellDescriptionWrapper`. The method gathers all smells that a method is guilty of, and passes it to `getAllAttributeDescriptions(Iterator)`, which obtains all method descriptions from `Description` by invoking `methodDescription(String)` and `shortRefactoring(int)`.

Method Smell Descriptions

Table 4.3 lists all method smell descriptions that are displayed to the developer. *Lazy Class* and *Refused Bequest* are Class Smells and have therefore no method level descriptions.

Table 4.3: Method level smell descriptions

Smell	Description
Eager Test	This method checks too many methods of `<classUnderTest>` in a single test case.
General Fixture	This fixture is too general. Not everything here is required for all test cases.
Long Parameter List	This method requires too many (`<amountOfParameters>`) parameters.
Long Method	This method does too many things at once.

Method Refactoring Descriptions

Table 4.4 lists all class refactoring descriptions that are displayed to the developer.

Table 4.4: Method level smell refactoring descriptions

Smell	Refactoring description
Eager Test	Apply (`<refactoringNumber>`) to improve it.
General Fixture	Apply (`<refactoringNumber>`) to improve the test.
Long Parameter List	Apply a refactoring suggested in (`<refactoringNumber>`) to improve the code.
Long Method	Improve it by applying (`<refactoringNumber>`).

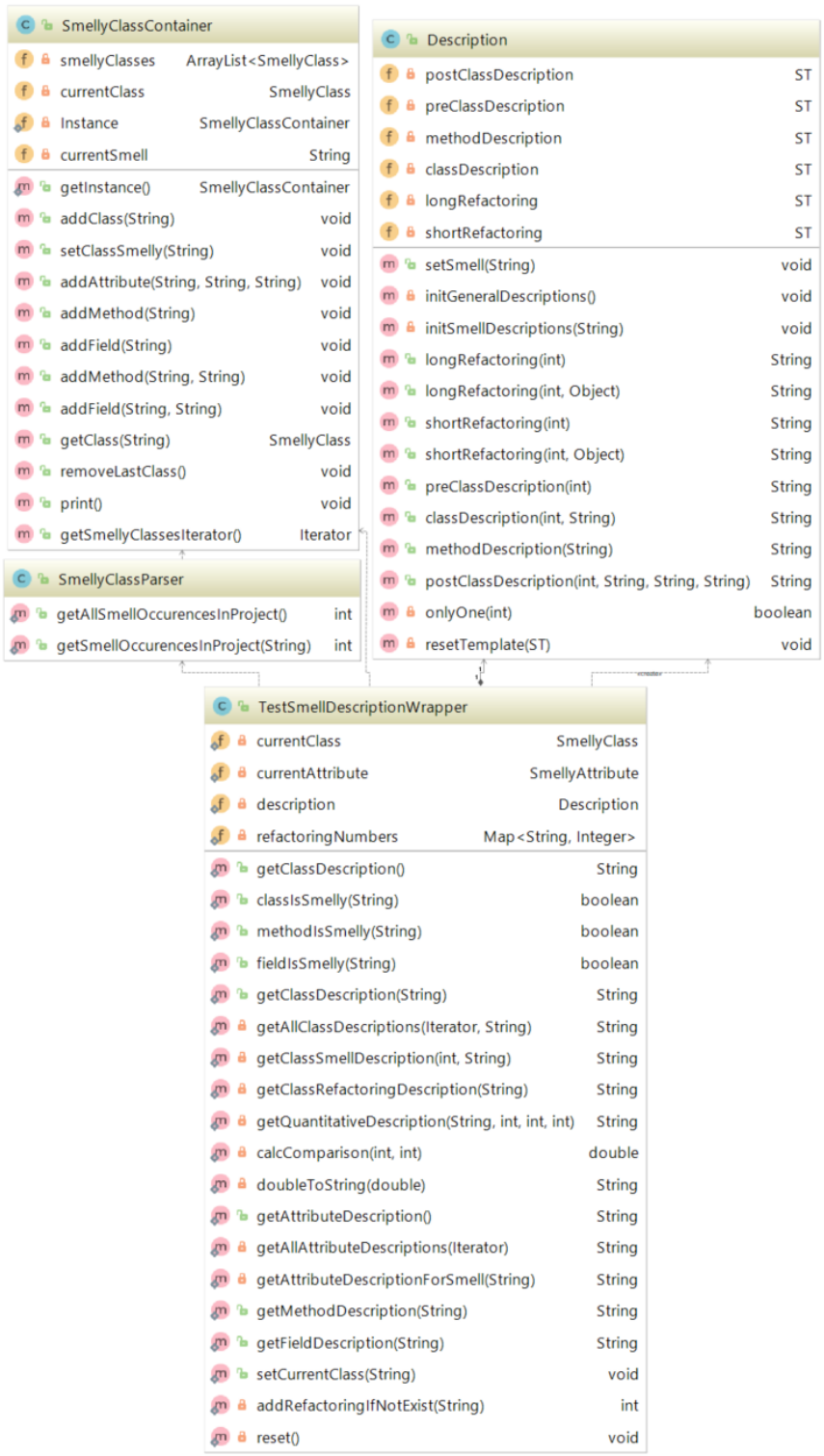

Figure 4.8: Class Diagram: Obtaining smell descriptions

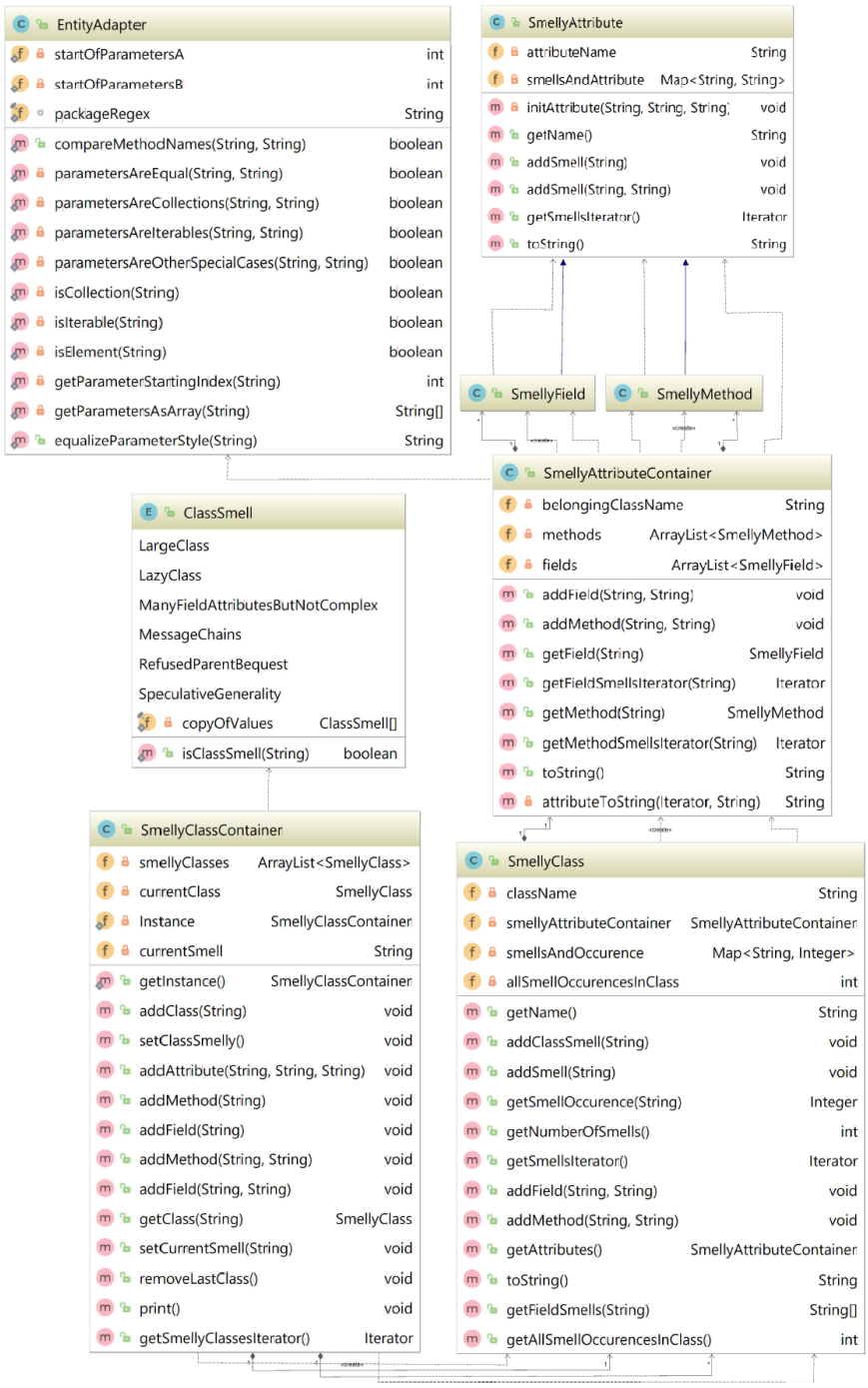

Figure 4.9: Class Diagram: Storing smelly information

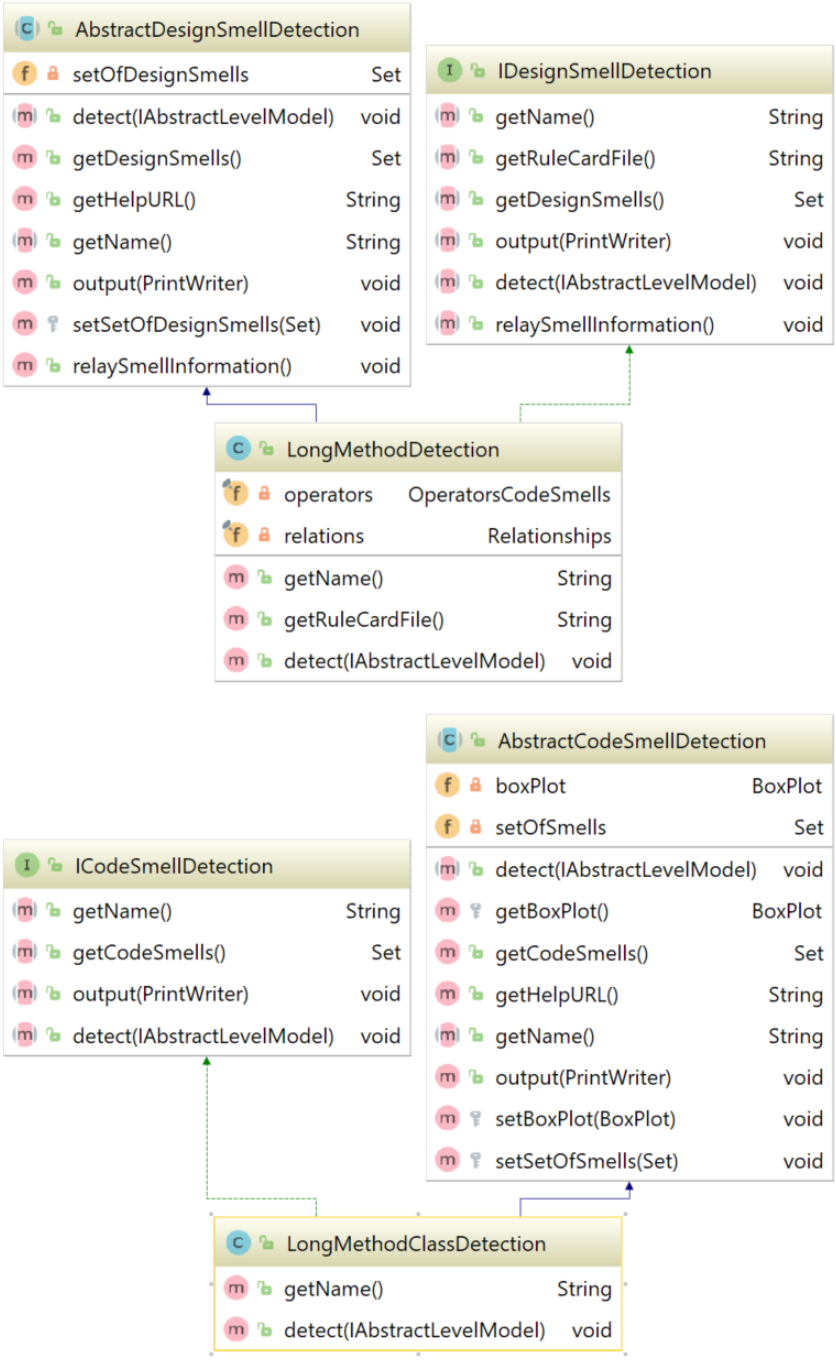

Figure 4.10: Class Diagram: DECOR code and design smell hierarchy

Chapter 5

Prototype

In chapter 4 we have seen a detailed description of TestSmellDescriber (TSD) and the definition of smell description and refactoring templates. This chapter will showcase the utilization of TSD on the tool Apache OFBiz. For this we will first download and compile the to be examined project. In the next step we will download TestSmellDescriber and export it as a library. The library is then imported into TestDescriber. Finally we will run the test smell detection on the downloaded and compiled project. We will highlight the individual steps it requires to run TSD and to augment classes with test smell descriptions.

5.1 Project Under Test

The test smell detection is performed on a system with a test suite. For this prototype we will present TSD on the Apache project OFBiz. First, download the source code of Apache OFBiz[1]. The results of the smell examination will be added as comments in the source code. TACO additionally requires the source code to conduct a smell examination. To examine the project on smells using DECOR a compiled version of the code has to be provided. To compile Apache OFBiz run the Gradle command `gradlew build`[2]. The user guide for the installation of Gradle can be found on the Gradle website[3].

5.2 Smell Description Tools

To augment a class with descriptions using the tool TestDescriber, we will export TSD and then import it as a library into TestDescriber. The following steps describe how to generate a library from TSD. This is additionally presented in figure 5.1.

1. Clone TSD from our GitHub repository[4].
2. To set up the included module DECOR, navigate to the folder containing the recently downloaded repository and execute the following commands:
   - `git submodule init`
   - `git submodule update`

[1] https://ofbiz.apache.org/developers.html
[2] https://github.com/apache/ofbiz/blob/trunk/README.md
[3] https://docs.gradle.org/current/userguide/installation.html
[4] http://bit.ly/2DUiZrC

3. Import the project into your IDE[5].
4. Select the class `testsmell.runner.TestSmellRunner` and run it as a Java Application. This step is required to make `TestSmellRunner` available as a launch configuration for step 7.
5. Open the context menu of `testsmell.runner.TestSmellRunner` and click the item *Export*.
6. Select *Runnable JAR file*.
7. Set the launch configuration to *TestSmellRunner - TestSmell* and select the desired output path in the Export destination field.
8. Select "Copy required libraries into a sub-folder next to the generated JAR".
9. Click Finish.

In the next step, import TestDescriber, or any other tool with the capability to filter method and class names, and the ability to edit the Javadoc comment of a Java file, into your workspace. Include the newly created *TestSmell.jar* as a library into your application. The for TSD required dependencies can be found in the folder *TestSmell_lib*. The folder has to be located in the same location as *TestSmell.jar*.
Finally the folder containing the smell description templates has to be added to the root path of your application. Our repository includes the by us defined templates that are used for the generation of smell descriptions. The folder is located in */TestDescriberTestSmellUtil/descriptions/*.

5.3 Smell Detection

There are two ways to start the smell detection process:

- If the source files of the project under test are contained in the sub folder */src/* and binary files in a sub folder */bin/*, the method `TestSmellRunner.testOneProject(rootPath, toolOuputPath, "class")` can be used to easily start the smell detection. The method then starts the smell detection with TACO, passing it the sub folder */src/*, and upon finishing the task will start DECOR, passing it the sub folder */bin/*.
- If source and binary root paths are in different locations, the paths need to be passed individually to TSD. This can be achieved by invoking the following methods in this order:

  1. `TestSmellRunner.run("textual", sourcePath, toolOutputPath)`
  2. `TestSmellRunner.run("structural", sourcePath, toolOutputPath, "class")`

  The order ensures that the by both smell detection tools detected smell entities are stored in the same `SmellyClassContainer`. Starting a new textual examination of a project will reset the content of your `SmellyClassContainer`. This allows for the subsequent detection of more than one project.

TSD will then evaluate the desired project on test smells with the help of the smell detection tools. Table 5.1 shows the results of the examination detailing the smell occurrences per class, omitting classes with no smells. Out of five runs TACO required approximately 17.5 seconds for the examination of Apache OFBiz, while DECOR required 4.5 seconds.

[5]We used the following IDE: Eclipse Java EE, version Oxygen.1a Release (4.7.1a)

Table 5.1: Test Smell Detection Results for project Apache OFBiz

Class	ET	LM	LPL	MC	RB	Total
AssertTests	0	1	0	0	0	**1**
CCServicesTest	0	0	0	0	1	**1**
ComparableRangeTests	0	0	1	0	0	**1**
EntityQueryTestSuite	0	1	0	0	0	**1**
EntitySaxReaderTests	1	0	0	0	0	**1**
EntityTestSuite	0	1	0	0	0	**1**
EntityXmlAssertTest	0	0	0	0	1	**1**
FinAccountTest	0	0	0	0	1	**1**
FinAccountTests	0	0	0	0	1	**1**
FlexibleMapAccessorTests	0	1	1	0	0	**2**
FlexibleStringExpanderTests	*1*	*1*	*1*	*0*	*0*	***3***
GenericMapTest	0	1	0	0	0	**1**
GenericTestCaseBase	3	0	1	0	0	**4**
InventoryItemTransferTest	0	0	0	0	1	**1**
IssuanceTest	0	1	0	0	1	**2**
LuceneTests	0	0	0	0	1	**1**
MiniLangTests	0	0	0	0	1	**1**
ModelTestSuite	1	1	0	1	0	**3**
ObjectTypeTests	0	0	1	0	0	**1**
OFBizTestCase	0	0	0	0	1	**1**
OrderTestServices	0	1	0	0	0	**1**
PerformFindTests	0	1	0	0	1	**2**
PurchaseOrderTest	0	0	0	0	1	**1**
SalesOrderTest	0	1	0	0	1	**2**
ServiceEngineTests	0	0	0	0	1	**1**
ServiceEntityAutoTests	0	0	0	0	1	**1**
ServiceSOAPTests	0	0	0	0	1	**1**
ServiceTest	0	0	0	0	1	**1**
SimpleMethodTest	0	0	0	0	1	**1**
StockMovesTest	0	0	0	0	1	**1**
StringUtilTests	0	1	0	0	0	**1**
TestJSONConverters	2	0	0	0	0	**2**
TestRunContainer	0	1	0	0	0	**1**
TimeDurationTests	0	1	1	0	0	**2**
UspsServicesTests	0	0	0	0	1	**1**
UtilCacheTests	0	1	1	0	0	**2**
WidgetMacroLibraryTests	0	0	0	0	1	**1**
XmlRpcTests	0	0	0	0	1	**1**
Total	**8**	**15**	**7**	**1**	**21**	**52**

We will further present the results of the smell descriptions for the class `FlexibleStringExpanderTests`. We chose this class to display a class with more than one class description, *i.e.*, a class with more than one smell, and which includes a method with more than one smell occurrence. The smell detection has shown that this class contains three types of smells: Long Parameter List, Long Method and Eager Test.

5.3.1 Long Parameter List

The project has 7 Long Parameter List occurrences. As outlined in section 4.3 is the `NOParam` metric used to calculate how many number of parameters a method has. Applying the metric on the method `doFseTest(String, String, FlexibleStringExpander, Map, TimeZone, Locale, String, Object, boolean)` yields the result 9.0. The ordinal value `VERY_HIGH` with a fuzziness of 20 results in a value relative to all the entities of the system. A method has a Long Parameter List smell if the number of parameters is a high outlier. The max bound for the Long Parameter List smell for the project Apache OFBiz is 6.0. `FlexibleStringExpanderTests::doFseTest` was therefore found to contain a Long Parameter List smell.

5.3.2 Long Method

There are 15 occurrences of the Long Method smell in this project. The metric `METHOD_LOC` calculates the number of lines of code the methods have. The method `doFseTest` was found to have 489 lines of code. The relative upper quartile of the project Apache OFBiz is 280. `FlexibleStringExpanderTests::doFseTest` was consequentially found to contain a Long Method smell.

5.3.3 Eager Test

TACO determines that a method is an Eager Test if the following calculation[6] holds: $CS(TC, PM) + CS(PM!) < \frac{2N^2-3N}{5}$. The test case `doFseTest` in the class `FlexibleStringExpander` calls 6 production methods, therefore, the sum of cosine similarities between TC and PM, and between the n-permutation of all PMs has to be smaller than $\frac{2\times6^2-3\times6}{5} = 10.8$. The sum of all cosine similarities in `doFseTest` is 2.398236703944477. TACO therefore determines that the test case `FlexibleStringExpander::doFseTest` is an Eager Test.

5.4 Generating Comments

To generate a comment using TSD templates have to be provided that contain the textual content of the description. We provide templates for the in section 4.5 defined descriptions. These files can be found in our repository in the sub folder */descriptions/*. To use these descriptions copy the folder into the root path of your application. To then obtain a comment we first have to set the currently observed entity by invoking `TestSmellDescriptionWrapper.setCurrentClass(String)`, and passing the class name along with the full path. The full class name for above class is *org.apache.ofbiz.base.util.string.test.FlexibleStringExpanderTests*. The method will then obtain the `SmellyClass` object from the `SmellyClassContainer`.
TestDescriber first iterates over all methods of the class, *i.e.*, over all test cases. By invoking `TestSmellDescriptionWrapper.methodIsSmelly(String)` we receive a boolean stating if a smell was found in the method. The passed string has to contain the method name as well as its parameters. This is important to differentiate between methods that are overloaded. For the method `FlexibleStringExpanderTests::doFseTest`, which can be found at the end of the chapter in listing 5.3, we pass the following parameter: `doFseTest(String, String, FlexibleStringExpander, Map, TimeZone, Locale, String, Object, boolean)`. The `TestSmellDescriptionWrapper` then tries to obtain the method from the corresponding `SmellyAttributeContainer`, which contains all smelly attributes, such as fields and methods.

[6] PM = Production Method, TC = Test Case, CS = Cosine Similarity, N = Number of Production Methods

To recover the correct method, TSD will use the class `EntityAdapter`, which matches method headers between TACO, DECOR and TestDescriber. As mentioned is this necessary due to the different implementations in the tools.
As outlined in the previous section has the method `doFseTest` been found of being guilty of two types of smells: Long Parameter List & Long Method. Invoking `TestSmellDescriptionWrapper .getAttributeDescription()` will then generate a text describing the smells that were found in this method. It first generates a short description for the smell and then a short description of the refactoring. The refactoring strings are numbered consecutively starting from 0. The returned string for the method `doFseTest` is:

- This method requires too many parameters.
Apply a refactoring suggested in (0) to improve the code.
- This method does too many things at once.
Improve it by applying (1).

Additionally, calling `TestSmellDescriptionWrapper.methodIsSmelly("setUp()")` followed by `TestSmellDescriptionWrapper.getAttributeDescription()` will return the following smell description:

- This method checks too many methods of the class being tested.
Apply (2) Extract Method to improve it.

Finally the class description is obtained by invoking `TestSmellDescriptionWrapper .getClassDescription()`. Here, the pretext is first generated depending on the number of smells that were found in this class:

Some problems were detected in this test class:

It then generates a description for all found smells by first generating a description of the smell, followed by a string describing the refactoring, and finally the quantitative description. The full string for the Long Method smell is:

- This test contains a method that requires a large amount of parameters.
A long list of parameters is hard to understand, to use and may become inconsistent over time.
(0) If all parameters belong to the same object, replace the individual data with the object itself. Otherwise introduce a new object that contains all the passed data.
This method accounts for 33.33% of all found problems in this test class.
This smell represents 13.46% of all found problems in the project with 14.29% occurring in this test.

TestDescriber augments each description returned by TestSmellDescriber as comments into the code.

5.5 Results

In the following are the results of the test smell detection and augmentation of descriptions in the classes `FlexibleStringExpanderTests` and `TestJSONConverters` presented.

5.5.1 FlexibleStringExpanderTests

Listing 5.1 shows the results achieved by running TestSmellDescriber on the class `FlexibleStringExpanderTests` and augmenting the descriptions as comments with the help of TestDescriber. Only the descriptions, along with class header, and method header of smelly methods are displayed. We have marked, with letters along the left side, the different sections of the comment.

```
   /**
a) * Some problems were detected in this test class:
b) * - This test contains a method that requires a large amount of parameters.
   *   A long list of parameters is hard to understand, to use and may become
   *   inconsistent over time.
c) *   (0) If all parameters belong to the same object, replace the individual
   *   data with the object itself. Otherwise introduce a new object that
   *   contains all the passed data.
d) *   This method accounts for 33.33% of all found problems in this test class.
   *   This smell represents 13.46% of all found problems in the project with
   *   14.29% occurring in this test.
e) * - This test contains a method that checks too many methods of
   *   the class being tested at once, which makes this test difficult to read,
   *   understand and maintain.
f) *   (2) Consider using Fowler's Extract Method technique on the guilty test
   *   case by separating the code into test methods that only test one method.
   *   Simultaneously improve the documentative purpose of this test by using
   *   meaningful names that describe the goal of the individual test cases.
g) *   This method accounts for 33.33% of all found problems in this test class.
   *   This smell represents 15.38% of all found problems in the project with
   *   12.50% occurring in this test.
h) * - This test contains a method that does too many things at once. This
   *   makes the code hard to understand and maintain.
i) *   (1) Shorten the method using Fowler's Extract Method, by finding parts
   *   of the method that go together and extracting them to new methods.
j) *   This method accounts for 33.33% of all found problems in this test class.
   *   This smell represents 28.85% of all found problems in the project with
   *   6.67% occurring in this test.
   **/
   public class FlexibleStringExpanderTests extends TestCase {

      [...]

      /**
k)    * - This method requires too many parameters.
      *   Apply a refactoring suggested in (0) to improve the code.
      * - This method does too many things at once.
l)    *   Improve it by applying (1).
      */
      private static void doFseTest(String label, String input,
            FlexibleStringExpander fse, Map<String, Object> context,
            TimeZone timeZone, Locale locale, String compare,
```

```
            Object expand, boolean isEmpty) {
         [...]
      }

      /**
m)    * - This method checks too many methods of the class being tested.
      *   Apply (2) Extract Method to improve it.
      */
      @Override
      public void setUp() {
         [...]
      }
   }
```

Listing 5.1: Smell detection result: FlexibleStringExpanderTests

With a) marked text displays the pretext that states that a smell or multiple smells were found in this class. Letters b) – d) are the parameterized descriptions for the Long Parameter List smell, while you can find the description of the Long Method smell in e) – g), and the descriptions for the Eager Test smell in h) – j). Along with the description of smells in the class comment, are smelly methods also augmented with comments. The method `doFseTest` is guilty of two smells, Long Parameter List (k) and Long Method (l), while `setUp` has been found to be an Eager Test (m). To gain the ability to easily find the longer description of a smelly method in the class comment and its refactoring message, method comments and refactoring messages are numbered. The quantitative descriptions (d, g, & j) relate the number of occurrences each smell has in the class to the occurrence in the project.

5.5.2 TestJSONConverters

Listing 5.2 presents the result of the `TestJSONConverters` class. Here we can observe a smell which occurs twice in a class. In addition to commentating the methods is this also reflected in the class smell description stating that "2 number of methods" of the respective smell were found.

```
/**
* A problem was detected in this test class:
* - This test contains 2 number of methods that check too many methods of
*  the class being tested at once,
*  which makes this test difficult to read, understand and maintain.
*  (0) Consider using Fowler's Extract Method technique on the guilty test case
*  by separating the code into test methods that only test one method.
*  Simultaneously improve the documentative purpose of this test by using
*  meaningful names that describe the goal of the individual test cases.
*  These methods account for 100.00% of all found problems in this test class.
*  This smell represents 18.52% of all found problems in the project with
*  20.00% occurring in this test.
**/
public class TestJSONConverters extends TestCase {

   [...]
```

```
    /**
    * - This method checks too many methods of the class being tested.
    *  Apply (0) Extract Method to improve it.
    */
    public void testJSONToMap() throws Exception {
        [...]
    }

    /**
    * - This method checks too many methods of the class being tested.
    *  Apply (0) Extract Method to improve it.
    */
    public void testJSONToList() throws Exception {
        [...]
    }
}
```

Listing 5.2: Smell detection result: TestJSONConverters

```
private static void doFseTest(String label, String input,
      FlexibleStringExpander fse, Map<String, Object> context,
      TimeZone timeZone, Locale locale, String compare,
      Object expand, boolean isEmpty) {

   assertEquals("isEmpty:" + label, isEmpty, fse.isEmpty());
   if (input == null) {
      assertEquals("getOriginal():" + label, "", fse.getOriginal());
      assertEquals("toString():" + label, "", fse.toString());
      assertEquals("expandString(null):" + label, "", fse.expandString(null));
      assertEquals("expand(null):" + label, null, fse.expand(null));
      if (timeZone == null) {
         assertEquals("expandString(null):" + label, "", fse.expandString(null,
            locale));
         assertEquals("expand(null):" + label, null, fse.expand(null, locale));
      } else {
         assertEquals("expandString(null):" + label, "",
            fse.expandString(null, timeZone, locale));
         assertEquals("expand(null):" + label, null,
            fse.expand(null, timeZone, locale));
      }
   } else {
      assertEquals("getOriginal():" + label, input, fse.getOriginal());
      assertEquals("expandString(null):" + label, input, fse.expandString(null));
      assertEquals("expand(null):" + label, null, fse.expand(null));
      if (timeZone == null) {
         assertEquals("expandString(null):" + label, input,
            fse.expandString(null, locale));
         assertEquals("expand(null):" + label, null, fse.expand(null, locale));
      } else {
         assertEquals("expandString(null):" + label, input,
            fse.expandString(null, timeZone, locale));
         assertEquals("expand(null):" + label, null,
            fse.expand(null, timeZone, locale));
      }
   }
   if (locale == null) {
      assertEquals(label, compare, fse.expandString(context));
      assertEquals("expand:" + label, expand, fse.expand(context));
   } else {
      Locale defaultLocale = Locale.getDefault();
      TimeZone defaultTimeZone = TimeZone.getDefault();
      try {
         Locale.setDefault(locale);
         TimeZone.setDefault(timeZone);
         assertEquals(label, compare, fse.expandString(context, null, null));
         assertEquals(label, expand, fse.expand(context, null, null));
         Locale.setDefault(badLocale);
```

```
            TimeZone.setDefault(badTimeZone);
            assertNotSame(label, compare, fse.expandString(context, null, null));
            if (input != null) {
               assertNotSame(label, expand, fse.expand(context, null, null));
            }
            Map<String, Object> autoUserLogin = new HashMap<String, Object>();
            autoUserLogin.put("lastLocale", locale.toString());
            autoUserLogin.put("lastTimeZone",
               timeZone == null ? null : timeZone.getID());
            context.put("autoUserLogin", autoUserLogin);
            assertEquals(label, compare, fse.expandString(context, null, null));
            assertEquals(label, expand, fse.expand(context, null, null));
            autoUserLogin.put("lastLocale", badLocale.toString());
            autoUserLogin.put("lastTimeZone", badTimeZone.getID());
            assertNotSame(label, compare, fse.expandString(context, null, null));
            if (input != null) {
               assertNotSame(label, expand, fse.expand(context, null, null));
            }
            context.remove("autoUserLogin");
            context.put("locale", locale);
            context.put("timeZone", timeZone);
            assertEquals(label, compare, fse.expandString(context, null, null));
            assertEquals(label, expand, fse.expand(context, null, null));
            context.put("locale", badLocale);
            context.put("timeZone", badTimeZone);
            assertNotSame(label, compare, fse.expandString(context, null, null));
            if (input != null) {
               assertNotSame(label, expand, fse.expand(context, null, null));
            }
            context.remove("locale");
            context.remove("timeZone");
            assertEquals(label, compare,
               fse.expandString(context, timeZone, locale));
            assertEquals(label, expand,
               fse.expand(context, timeZone, locale));
            assertNotSame(label, compare,
               fse.expandString(context, badTimeZone, badLocale));
            if (input != null) {
               assertNotSame(label, expand,
                  fse.expand(context, badTimeZone, badLocale));
            }
         } finally {
            Locale.setDefault(defaultLocale);
            TimeZone.setDefault(defaultTimeZone);
         }
      }
   }
```

Listing 5.3: Smelly Method: doFseTest

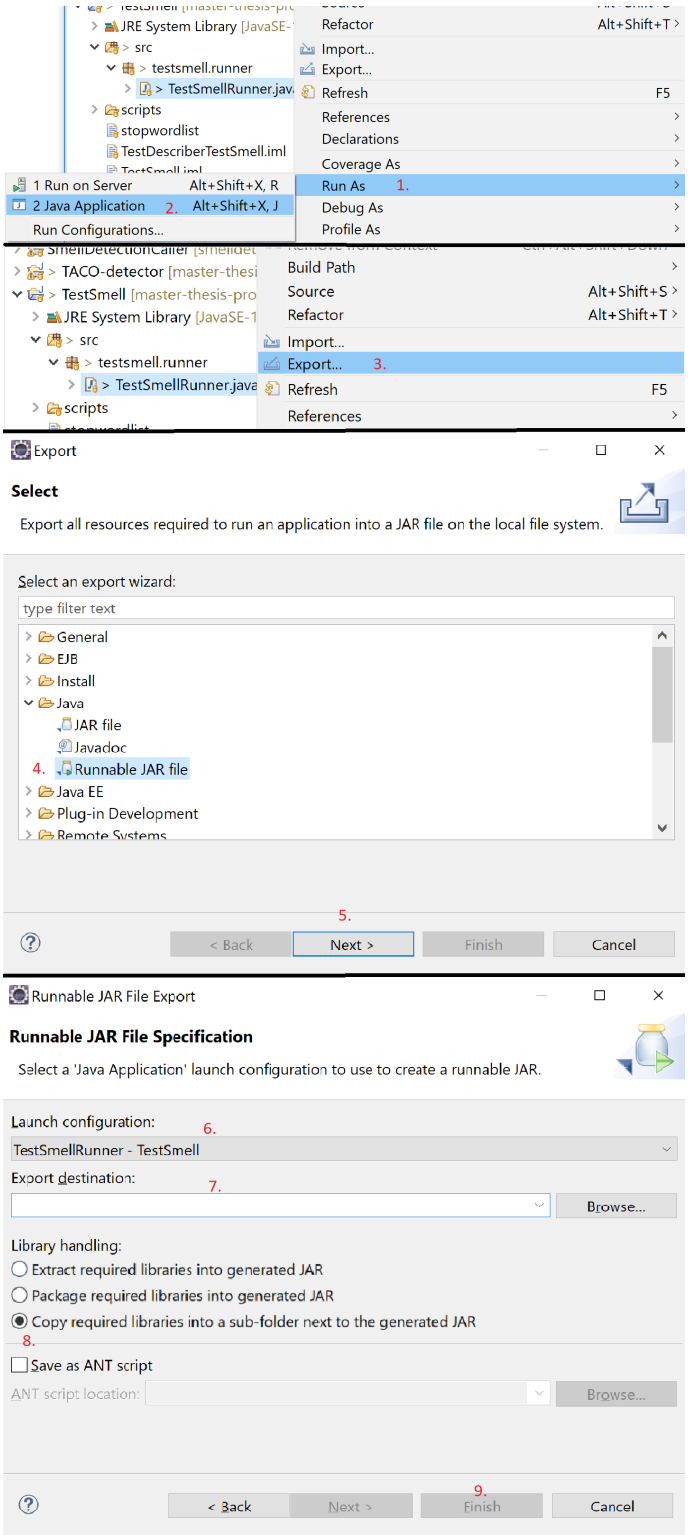

Figure 5.1: Export TestSmellDescriber from Eclipse Jave EE

Chapter 6

Conclusion and Future Work

In this thesis we presented TestSmellDescriber, a tool able to detect smells in test suites and to display them to the developer. The goal of TestSmellDescriber is to bring awareness on the quality of test code, to help developers to improve the quality of test cases, and to improve writing test code in general. This is achieved by detecting and describing potential problems in the code and inserting them as Javadoc comments in the source code. Those areas are detected by statically examining the code for smells. A code smell occurs due to poor design and implementation choices. Smelly code is especially prone to bugs and defects, but more than simply contributing to the incorrectness of code, code smells negatively impact the intentions of software testing by countering the goals of software testing, as well as amplifying its limitations.
The examination of a test suite is done with the help of automatic smell detection tools. We initially selected four tools by conducting a literature study, and then assessed these on their usability on test suites, their integrability into one tool, and if and which of the implemented smell detections metrics are applicable on test cases. The assessment resulted in two tools: DECOR and TACO. We then combined both tools into one tool, by developing a common entry point for both tools. The resulting module TestSmellRunner allows for the integration of additional smell detection tools. We additionally developed a module to unify and collect the results of the by the tools performed code smell detection, taking different implementations into account.
Furthermore, we have conducted a literature study on the definition and categorization of code and test smells. The study provided our basis of knowledge for the definition of smell and refactoring descriptions. Those messages are parameterized and then displayed to the user. Along with the descriptions are quantitative information provided in regards to the overall quality of the examined test suite. The information are intended to help developers in assessing their overall test writing skills by pointing out reoccurring problems or deficits. With the use of TestDescriber are the descriptions then integrated as comments into the source code of the examined code.

There are several opportunities for future work that can be conducted on top of this thesis. TestSmellDescriber currently provides a smell examination with two tools: DECOR and TACO. In the future more tools could be integrated into TestSmellDescriber to expand the number of detectable smells and to increase the thoroughness of the smell examination. Alternative automatic smell detection tools could present the ability to detect more as well as different smells than the currently nine detectable smells. The architecture of TestSmellDescriber enables this as it allows for easy integration of smell detection tools, and unification of different implementations. This expansion would allow users to conduct a better assessment of their tests.
Current smell descriptions are rather general and lack low level information in terms of what the precise problem is per smell. For the smell Test Code Duplication this could be *e.g.*, a set of calls that were found in more than one test case. This information would lead to more concrete and detailed refactoring messages, providing better explanations as to how users could improve their

code, such as: "extract lines 100–105 from the method." To achieve this tools have to be found or changes have to be made to the currently used tools, to obtain data detailing what information lead to the detection of a concrete smell.
This thesis has additionally brought to our attention that while many tools and automatizations were researched on the detection of code smells, not much research was conducted on the automating of test smell detections and the development of test smell detection tools. With the importance of software this is a rather surprising finding, We suggest that research is to be conducted on automating test smell detection, defining test smell metrics and the applicability of currently available code smell metrics on test cases.
In general a study has to be conducted on TestSmellDescriber, if and to what degree the augmented comments, detailing the cause of the smell, the refactoring, and quantitative data, help developers to improve their tests, as well as to help them in improving their test writing skills.

Bibliography

[AKGA11] Marwen Abbes, Foutse Khomh, Yann-Gael Gueheneuc, and Giuliano Antoniol. An empirical study of the impact of two antipatterns, blob and spaghetti code, on program comprehension. In *Software maintenance and reengineering (CSMR), 2011 15th European conference on*, pages 181–190. IEEE, 2011.

[And11] Marc Andreessen. Why software is eating the world. *The Wall Street Journal*, 20.08.2011.

[apa18] Apache: Projects by language. https://projects.apache.org/projects.html?language, 2018. [accessed 03. January 2018].

[BDLDP+15] Gabriele Bavota, Andrea De Lucia, Massimiliano Di Penta, Rocco Oliveto, and Fabio Palomba. An experimental investigation on the innate relationship between quality and refactoring. *Journal of Systems and Software*, 107:1–14, 2015.

[BDLMO14] Gabriele Bavota, Andrea De Lucia, Andrian Marcus, and Rocco Oliveto. Automating extract class refactoring: an improved method and its evaluation. *Empirical Software Engineering*, 19(6):1617–1664, 2014.

[BML07] Andreas Leitner Bertrand Meyer, Ilinca Ciupa and Lisa (Ling) Liu. Automatic testing of object-oriented software. In *SOFSEM 2007: Theory and Practice of Computer Science*, pages 114–129. Springer Berlin Heidelberg, 2007.

[BQO+12] Gabriele Bavota, Abdallah Qusef, Rocco Oliveto, Andrea De Lucia, and David Binkley. An empirical analysis of the distribution of unit test smells and their impact on software maintenance. In *Software Maintenance (ICSM), 2012 28th IEEE International Conference on*, pages 56–65. IEEE, 2012.

[BQO+15] Gabriele Bavota, Abdallah Qusef, Rocco Oliveto, Andrea De Lucia, and Dave Binkley. Are test smells really harmful? an empirical study. *Empirical Software Engineering*, 20(4):1052–1094, 2015.

[CGP+06] Cristian Cadar, Vijay Ganesh, Peter M Pawlowski, David L Dill, and Dawson R Engler. Exe: Automatically generating inputs of death. In *Proceedings of the Conference on Computer and Communications Security (CCS)*, pages 322–335. ACM, 2006.

[che17] checkstyle. http://checkstyle.sourceforge.net/, 2017. [accessed 26. January 2018].

[clo18] Clone digger: discovers duplicate code in python and java. http://clonedigger.sourceforge.net/, 2018. [accessed 26. January 2018].

[con18] Conqat. https://www.cqse.eu/en/products/conqat/overview/, 2018. [accessed 26. January 2018].

[Csa04] Christoph Csallner. Jcrasher: an automatic robustness tester for java. pages 1025–1050, 2004.

[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*, pages 107–118. ACM, 2015.

[ESČP17] Eduard Enoiu, Daniel Sundmark, Adnan Čaušević, and Paul Pettersson. A comparative study of manual and automated testing for industrial control software. In *Software Testing, Verification and Validation (ICST), 2017 IEEE International Conference on*, pages 412–417. IEEE, 2017.

[FA15] Gordon Fraser and Andrea Arcuri. 1600 faults in 100 projects: Automatically finding faults while achieving high coverage with evosuite. *Empirical Software Engineering*, 20(3):611–639, 2015.

[FBZ12] Francesca Arcelli Fontana, Pietro Braione, and Marco Zanoni. Automatic detection of bad smells in code: An experimental assessment. *Journal of Object Technology*, 11(2):5–1, 2012.

[FKG09] M. Di Penta F. Khomh and Y.-G. Guéhéneuc. An exploratory study of the impact of code smells on software change-proneness, June 2009.

[Fow99] Martin Fowler. *Refactoring: Improving the Design of Existing Code*. 1999.

[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*, pages 291–301. ACM, 2013.

[FSM+15] Gordon Fraser, Matt Staats, Phil McMinn, Andrea Arcuri, and Frank Padberg. Does automated unit test generation really help software testers? a controlled empirical study. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 24(4):23, 2015.

[FTC07] Marios Fokaefs, Nikolaos Tsantalis, and Alexander Chatzigeorgiou. Jdeodorant: Identification and removal of feature envy bad smells. In *Software Maintenance, 2007. ICSM 2007. IEEE International Conference on*, pages 519–520. IEEE, 2007.

[jde18] Jdeodorant on github. https://github.com/tsantalis/JDeodorant, 2018. [accessed 05. January 2018].

[KDPGA12] Foutse Khomh, Massimiliano Di Penta, Yann-Gaël Guéhéneuc, and Giuliano Antoniol. An exploratory study of the impact of antipatterns on class change-and fault-proneness. *Empirical Software Engineering*, 17(3):243–275, 2012.

[KS10] Pavan Kumar and Khasim Syed. Software testing–goals, principles, and limitations. *International Journal of Engineering Science & Advanced Technology*, (1):52–56, 2010.

[Mä03] Mika Mäntylä. Bad smells in software: a taxonomy and an empirical study, 2003.

[MAK+15] Mika V. Mäntylä, Bram Adams, Foutse Khomh, Emelie Engström, and Kai Petersen. On rapid releases and software testing: a case study and a semi-systematic literature review. *Empirical Software Engineering*, 20(5):1384–1425, Oct 2015.

[Mar04] Radu Marinescu. Detection strategies: Metrics-based rules for detecting design flaws. In *Software Maintenance, 2004. Proceedings. 20th IEEE International Conference on*, pages 350–359. IEEE, 2004.

[Mar07] Robert C. Martin. Professionalism and test-driven development. *IEEE Software*, 24(3):32–36, May 2007.

[Mar08] Robert C. Martin. *Clean Code: A Handbook of Agile Software Craftsmanship*. 2008.

[Mes07] Gerard Meszaros. *xUnit Test Patterns: Refactoring Test Code*. 2007.

[MGDLM10] Naouel Moha, Yann-Gael Gueheneuc, Laurence Duchien, and Anne-Francoise Le Meur. Decor: A method for the specification and detection of code and design smells. *IEEE Transactions on Software Engineering*, 36(1):20–36, 2010.

[MGLM+10] Naouel Moha, Yann-Gaël Guéhéneuc, Anne-Françoise Le Meur, Laurence Duchien, and Alban Tiberghien. From a domain analysis to the specification and detection of code and design smells. *Formal Aspects of Computing*, 22(3-4):345–361, 2010.

[MHB10] Emerson Murphy-Hill and Andrew P Black. An interactive ambient visualization for code smells. In *Proceedings of the 5th international symposium on Software visualization*, pages 5–14. ACM, 2010.

[MMMW05] Cristina Marinescu, Radu Marinescu, Petru Florin Mihancea, and R Wettel. iplasma: An integrated platform for quality assessment of object-oriented design. In *In ICSM (Industrial and Tool Volume*. Citeseer, 2005.

[MSB11] Glenford J Myers, Corey Sandler, and Tom Badgett. *The art of software testing*. John Wiley & Sons, 2011.

[MTSV16] Davood Mazinanian, Nikolaos Tsantalis, Raphael Stein, and Zackary Valenta. Jdeodorant: clone refactoring. In *Software Engineering Companion (ICSE-C), IEEE/ACM International Conference on*, pages 613–616. IEEE, 2016.

[ope18] Open hub, the open source network. https://www.openhub.net/, 2018. [accessed 03. January 2018].

[org18] Organic on github. https://github.com/opus-research/organic, 2018. [accessed 05. January 2018].

[Pal15] F. Palomba. Textual analysis for code smell detection. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 2, pages 769–771, May 2015.

[PBDP+13] Fabio Palomba, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, Andrea De Lucia, and Denys Poshyvanyk. Detecting bad smells in source code using change history information. In *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering*, pages 268–278. IEEE Press, 2013.

[PDFS17] Thanis Paiva, Amanda Damasceno, Eduardo Figueiredo, and Cláudio Sant'Anna. On the evaluation of code smells and detection tools. *Journal of Software Engineering Research and Development*, 5(1):7, 2017.

[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*, pages 815–816. ACM, 2007.

[PKT15] Annibale Panichella, Fitsum Meshesha Kifetew, and Paolo Tonella. Reformulating branch coverage as a many-objective optimization problem. In *Software Testing, Verification and Validation (ICST), 2015 IEEE 8th International Conference on*, pages 1–10. IEEE, 2015.

[pmd17] Pmd: An extensible cross-language static code analyzer. https://pmd.github.io/, 2017. [accessed 26. January 2018].

[PMGZ13] Fayola Peters, Tim Menzies, Liang Gong, and Hongyu Zhang. Balancing privacy and utility in cross-company defect prediction. *IEEE Transactions on Software Engineering*, 39(8):1054–1068, 2013.

[PPB+16] Sebastiano Panichella, Annibale Panichella, Moritz Beller, Andy Zaidman, and Harald C Gall. The impact of test case summaries on bug fixing performance: An empirical investigation. In *Software Engineering (ICSE), 2016 IEEE/ACM 38th International Conference on*, pages 547–558. IEEE, 2016.

[PPDL+16] Fabio Palomba, Annibale Panichella, Andrea De Lucia, Rocco Oliveto, and Andy Zaidman. A textual-based technique for smell detection. In *Program Comprehension (ICPC), 2016 IEEE 24th International Conference on*, pages 1–10. IEEE, 2016.

[RFA15] José Miguel Rojas, Gordon Fraser, and Andrea Arcuri. Automated unit test generation during software development: A controlled experiment and think-aloud observations. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, pages 338–349. ACM, 2015.

[RMPM12] Dudekula Mohammad Rafi, Katam Reddy Kiran Moses, Kai Petersen, and Mika V Mäntylä. Benefits and limitations of automated software testing: Systematic literature review and practitioner survey. In *Proceedings of the 7th International Workshop on Automation of Software Test*, pages 36–42. IEEE Press, 2012.

[RT01] Filippo Ricca and Paolo Tonella. Analysis and testing of web applications. In *Proceedings of the 23rd international conference on Software engineering*, pages 25–34. IEEE Computer Society, 2001.

[SSS14] Girish Suryanarayana, Ganesh Samarthyam, and Tushar Sharma. *Refactoring for Software Design Smells: Managing Technical Debt*. 2014.

[TC09a] Nikolaos Tsantalis and Alexander Chatzigeorgiou. Identification of extract method refactoring opportunities. In *Software Maintenance and Reengineering, 2009. CSMR'09. 13th European Conference on*, pages 119–128. IEEE, 2009.

[TC09b] Nikolaos Tsantalis and Alexander Chatzigeorgiou. Identification of move method refactoring opportunities. *IEEE Transactions on Software Engineering*, 35(3):347–367, 2009.

[TC11] Nikolaos Tsantalis and Alexander Chatzigeorgiou. Identification of extract method refactoring opportunities for the decomposition of methods. *J. Syst. Softw.*, 84(10):1757–1782, October 2011.

[TCC08] Nikolaos Tsantalis, Theodoros Chaikalis, and Alexander Chatzigeorgiou. Jdeodorant: Identification and removal of type-checking bad smells. In *Software Maintenance and Reengineering, 2008. CSMR 2008. 12th European Conference on*, pages 329–331. IEEE, 2008.

[Ton04] Paolo Tonella. Evolutionary testing of classes. In *ACM SIGSOFT Software Engineering Notes*, volume 29, pages 119–128. ACM, 2004.

[TPB+15] Michele Tufano, Fabio Palomba, Gabriele Bavota, Rocco Oliveto, Massimiliano Di Penta, Andrea De Lucia, and Denys Poshyvanyk. When and why your code starts to smell bad. In *Software Engineering (ICSE), 2015 IEEE/ACM 37th IEEE International Conference on*, volume 1, pages 403–414. IEEE, 2015.

[VDMvdBK01] Arie Van Deursen, Leon Moonen, Alex van den Bergh, and Gerard Kok. Refactoring test code. In *Proceedings of the 2nd international conference on extreme programming and flexible processes in software engineering (XP2001)*, pages 92–95, 2001.

[VEM02] Eva Van Emden and Leon Moonen. Java quality assurance by detecting code smells. In *Reverse Engineering, 2002. Proceedings. Ninth Working Conference on*, pages 97–106. IEEE, 2002.

[VVDP+15] Santiago Vidal, Hernan Vazquez, J Andres Diaz-Pace, Claudia Marcos, Alessandro Garcia, and Willian Oizumi. Jspirit: a flexible tool for the analysis of code smells. In *Chilean Computer Science Society (SCCC), 2015 34th International Conference of the*, pages 1–6. IEEE, 2015.

[WB98] Hays McCormick William Brown, Raphael Malveau. *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis*. 1998.

[ZA10] Nico Zazworka and Christopher Ackermann. Codevizard: a tool to aid the analysis of software evolution. In *Proceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, page 63. ACM, 2010.