Universiteit Antwerpen

The Ubiquity of Test Smells: An Empirical Study

Andrés Carrasco

Principal Adviser: Serge Demeyer

Assistant Adviser: Brent van Bladel

Dissertation Submitted in June 2018 to the
Department of Mathematics and Computer Science
of the Faculty of Sciences, University of Antwerp,
in Partial Fulfillment of the Requirements
for the Degree of Master of Science.

Contents

List of Figures

List of Tables

Acknowledgements

To Professor Serge Demeyer and Brent van Bladel for their invaluable feedback and support. The complete Ansymo research group for allowing me to perform research under their guidance, especially Ali Parsai for his constructive feedback and criticism.

To my friends old and new, especially Lucas and Dominique, with whom I shared the journey of undertaking this Master's degree.

A warm thank you!

With the permission of the reader, I will continue the acknowledgements in Spanish.

A mi pareja de vida Núria, que sin su cariño, incansable apoyo y paciencia nada de esto sería posible.

A mis papás Félix e Inge, hermanos Isolde, Christoph, Ariadne y Marianne, cuñados Alejandro, Isis, Alfonso y Dominik, y sobrinos Aurora, Denise y Christoph, que a pesar de la distancia me dan su cariño y apoyo.

A mis suegros Meme y Pedro, y cuñada Anna, que también a pesar de la distancia me dan su apoyo y cariño.

A Lope, por su inestimable ayuda y apoyo, que fue indispensable en los momentos más vulnerables de mi carrera.

A todos los que me apoyaron y creyeron en mí.

¡Gracias!

Abstract

Modern software development life cycles allow the fast adaptation of constantly changing requirements. This flexibility raises concerns for the maintainability of code, including test code. For this experts have defined bad smells specific to test code or *test smells* alongside several refactoring strategies for removing them.

The initial definition of test smells lead to efforts for their detection, consolidating in multiple detection tools. Even though three empirical studies have shown the presence of test smells in both industrial and open-source projects, many tools' maintenance and development have stopped. This is possibly due to the minor attention test smells have received.

We present a novel approach for detecting and studying test smells with the largest dataset of software projects available using the Boa language and infrastructure. With this approach we perform an empirical study investigating seven test smells for their ubiquity in the available dataset, as well as their evolution.

Our study contributes seven scripts for detecting test smells in Boa. Moreover, with these scripts we investigated 282.577 software projects and conclude that *Assertion Roulette* is the most commonly detected test smell, followed by *Test Code Duplication*, *Assertionless* and *Verbose Test*. We also show a correlation between project size and the number of smells, the larger a project is the more likely it contains test smells. Moreover, $99,17\%$ of the detected test smells were introduced when creating the test smell, and $99,58\%$ were never fixed.

CHAPTER 1

Introduction

1.1 Problem Description

Fowler and Beck introduced the concept of refactoring, paving the way for new source code maintenance techniques [FB99]. *Refactoring* is described as "the process of changing a software system in such a way that it does not alter the external behavior of the code yet improves its internal structure." [FB99]. Ever since, refactoring has become part of the Software Engineering Body of Knowledge (SWEBOK) and is utilized in multiple modern development processes such as Test Driven Development [Bec03].

In their quest for improving code quality, experts have defined many symptoms that indicate candidate code for refactoring. Such symptoms are referred to as *code smells* or *bad smells*. Typically, a bad smell indicates a deeper problem in the system's design [FB99].

In the course of the development and evolution of software projects, code smells may inadvertently be introduced. Time pressure, among other things, might sway developers' preference on quick and easy solutions over better time-consuming solutions. Such quick and easy solutions might make future changes more difficult, implying a cost of rework every time a change is needed. Ward

Cunningham named this accumulation of implied cost *technical debt* [Cun92].

Developers have to deal with technical debt and bad smells by means of refactoring [Cun92, Fow97, KNO12, All12]. Failing to address bad smells leads to long-term issues, such as reduced code maintainability [SYA+13, YM12, YM13].

Kim et. al. showed that developers fear the risk of introducing bugs when refactoring code, hampering their mitigation [KZN14]. This raises the need for a guarantee that code can be refactored safely. Fowler and Beck propose utilizing a compiler, test suite, and code reviews for catching errors while refactoring [FB99].

Automated unit testing [HK07] has become essential as the default safety net for software projects, helping mitigate the developers fear of refactoring [Bec99]. When applied correctly, unit testing can protect against regressions when refactoring [DDN02]. However, in practice faults can still be introduced if the test suite is not good enough [KCK11].

Ideally, unit tests ought to be written once, yet modern software development cycles account for constantly changing requirements [Bec99, BBVB+01]. Changes in requirements often lead to a different desired functionality, culminating in modifications on test code as well. This raises concerns for its maintainability [BQO+12].

The concern for the maintainability of test code, directed experts to define a new set of bad smells specifically aimed at test code. These smells are inherently different from the smells in production code [vDMvdBK01]. Many distinct code smells in test code, or *test smells*, have been proposed by experts, alongside several refactoring strategies for removing test smells.

Even though efforts for detecting test smells have consolidated in multiple detection tools, their current state make the detection of test smells problematic. Moreover, many tools' maintenance and development have stopped, possibly due to the minor attention test smells have received. On the contrary, three empirical studies have shown the presence of test smells in both industrial and open-source projects [BQO+12, TPB+16, PDNP+16]. However, these findings are constrained by their size or the lack of diversification in the dataset.

1.2 Scope and Purpose

In this thesis, we demonstrate the feasibility of detecting test smells at a large-scale. Conducting a large-scale study has many challenges, perhaps the most important challenge is getting access to data. However, with the increasing popularity of online open source repository management services, such as GitHub or SourceForge, access to such data is becoming simpler. More so, with the help of *Boa*. Boa is a domain-specific language and infrastructure for analyzing popular software repositories like GitHub and SourceForge, making large scale-studies on their data reproducible and easy [DNRN13].

Through Boa, the possibly for conducting large-scale test smells studies backed with the largest open and diverse dataset of software projects has become a reality. We demonstrate this reality through a proof of concept. Next, we perform an empirical study utilizing Boa for investigating the most common test smells in projects using JUnit as their test suite. Lastly, with all the commit history of software projects provided by Boa, we investigate the evolution of test smells throughout the software's development.

1.3 Research Questions

In this section, we enumerate research questions (RQ) that are answered in this thesis.

RQ1 *detection* — Is it possible to detect test smells on a large scale?

RQ2 *ubiquity* — What are the most common test smells?

RQ3 *evolution* — How do the amount test smells evolve during software development?

1.4 Contributions

In this thesis, we contribute to the problem domain as follows:

- We demonstrate with a proof-of-concept that detecting test smells at a large-scale using the Boa language and infrastructure is possible.
- We contribute a repository with 7 scripts written in the Boa domain-specific language for detecting test smells and gathering data for their investigation. These scripts are available in the Appendix A.
- A dataset with all the test smells detected utilizing the aforementioned scripts. This dataset includes sufficient information to uniquely identify an instance of a test smell, alongside information on their evolution.
- We show that *Assertion Roulette* is the most common test smell in our findings, in line with previously performed empirical studies.
- We show that the larger a project is, the more likely it contains an instance of a test smell.
- We confirm that most test smells are introduced when creating the test case, as shown in previously performed empirical studies.
- We confirm that most test smells are not fixed throughout the observed timespan of the projects in the dataset, as shown in previously performed empirical studies.

1.5 Outline

In this thesis, we present the related work to our investigation in Chapter 2. Next, in Chapter 3 we give the necessary theoretical background for understanding the utilized tools and concepts related to our investigation. In Chapter 4 we present our case study setup, alongside the dataset provided by Boa. Next, in Chapter 5 we present the results from our study. In Chapter 6 we discuss the threats to the validity of our study. Finally, in Chapter 7 we present our conclusions.

CHAPTER 2

Related Work

While refactoring their Extreme Programming (XP) project, van Deursen et al. realized that refactoring test code is inherently different to production code [vDMvdBK01]. They contributed the initial eleven test smells and proposed solutions in the form of refactoring techniques. This list was extended by Meszaros et al., introducing as well a distinct group of behavior test smells, which manifest themselves while executing test code [MSA03]. Moreover, they contributed as well a set of principles defining the qualities an effective test should express. Several other researchers contributed to the definition of test smells, such as Greiler et al. by adding five new fixture-related test smells [GvDS13].

Although refactoring solutions for the test smells are presented, there are concerns that the correctness of test code can be affected while refactoring. Therefore, Guerra and Fernandes pointed out that the behavior of tests while refactoring must be preserved in order to consider the refactoring safe [GF07]. They proposed a graphical notation for representing test suites, with the goal of simplifying their comprehensibility and manipulability of test code.

After the formalization of multiple test smells, the academia shifted to establish different approaches for their detection. Van Rompaey et al. proposed the formalization of an heuristical approach for detecting two test smells, specifically the *General Fixture* and *Eager Test* [VRDBD06, VRDBDR07]. Both of which were initially proposed by van Deursen et al. [vDMvdBK01].

Their approach was a first step towards the quantitative analysis of test code, demonstrating the feasibility to detect test smells. Next, Reichhart et al. contributed on translating abstract descriptions of test smells in the literature into a set of rules, allowing the rule-based detection of multiple test smells [RGD07]. However, they did not propose a rule for detecting test code clones. Fang and Lam proposed a technique for identifying similar test cases utilizing *assertion fingerprints* [FL15]. Such assertion fingerprints utilize five attributes for detecting *Test Code Duplication*. Other techniques have appeared in the literature, such as a technique for detecting the *Dead Fields* test smell. Satter et al. developed the *Dead Field Identifier* technique [SAS16]. Their findings outperform the detection of *Dead Fields* when compared with the initial tool developed for its detection, *TestHound*.

In hopes for finding more relevant metrics for the detection of test smells, Tahir et al. investigated the metrics *Cyclomatic Complexity* (CC), *Weighted Methods per Class* (WMC), and *Lack of Cohesion of Methods* (LCOM) for strong links with their incidence [TCM16]. They concluded that high LCOM and WMC values appear to indicate the presence of the *Eager Test* and *Test Code Duplication* test smells. In contrast to this, some researchers focused on the relationships between test smells. Mathew and Foegen developed a conceptual model for analyzing the relation between the test smells *General Fixture*, *Eager Test*, and *Obscure Test*, which utilizes metrics and indicators of their presence [MF16].

With the definition of techniques and metrics for the detection of test smells, multiple tools were developed for implementing them. Breugelmans and Van Rompaey published *TestQ*, a tool for statically detecting twelve test smells [BVR08]. Alongside the tool, they included the formal definitions for the detection of twelve test smells based on Van Rompaey et al.'s previous work [VRDBD06, VRDBDR07]. Greiler et al. implemented a tool named *TestHound*, with which the proposed techniques to detect fixture-related test smells were implemented, including the *General Fixture* smell. However, their approach focused on detecting specific problems rather than a general heuristic-based metric proposed by Van Rompaey et al. [VRDBD06]. Another published tool *PMD* developed by Ramler et al. allowed the static analysis of JUnit test code with 42 implemented rules [RMP16]. The rules address issues and misuse of the xUnit framework, by inspecting assert statements, naming conventions, logic detection, and setup and teardown routines. However, no specific test smells are detected by the tool.

With all the effort placed on defining and detecting test smell, some researchers decided to pose the question whether test smells are present in test code. Lanubile and Mallardo conducted a empirical study inspecting auto-

mated test code [LM07]. For this study, the researchers tasked Master students in Computer Science to migrate a legacy web application into a new technology, in which test automation was to be added. After manually inspecting the resulting code, they found that most common problems included: *Manual Intervention*, *Assertion Roulette*, conditional test logic or *Indented Test*, and *Test Code Duplication*. However, their results are not conclusive due to the small number of test subjects in an academic environment.

In 2011, Qusef et al. reported the number of assert statements in 3 software projects totaling 124 unit test cases [QBO+11]. They found that generally unit test cases contain more than two assert statements, concluding that *Assertion Roulette* can pose a problem. However, these findings are based on a small number of projects. Moreover, the usage of an explanation message in the assertions was not reported.

In 2012, Bavota et al. conducted an empirical study analyzing 18 software projects [BQO+12]. Their aim was to reveal the frequency in which test smells occur in software projects. They investigated the following test smells: *Mystery Guest*, *Resource Optimism*, *Test Run War*, *General Fixture*, *Eager Test*, *Lazy Test*, *Assertion Roulette*, *Indirect Testing*, *For Testers Only*, *Sensitive Equality*, and *Test Code Duplication*. To detect these smells they developed a simple tool, which was not made available in the publication. The decision of not using available detection tools, was due to their detection rules being too restrictive and possibly miss test smell instances. They analyzed 16 open source and 2 industrial Java software projects utilizing the JUnit framework. They concluded that 82% of the analyzed JUnit classes were affected by at least one test smell. The most frequent smells included *Assertion Roulette* in 62% of the projects, Eager Test with 32%, and *Test Code Duplication* with 23%. However, this evidence is based on a rather small collection of 637 JUnit classes from 18 software projects. Moreover, with 20 Master Students they investigated the impact of these test smells on the maintainability of test code. They concluded that test smells have a negative impact on maintainability.

Having an answer for the diffusion of test smells, Tufano et al. conducted a large-scale empirical study for understanding when test smells are introduced in software projects, as well as, their longevity, and the potential relationship with code smells [TPB+16]. Moreover, they conducted a survey for analyzing the perception of developers towards test smells as design issues. Their study consists of 152 open source software projects containing JUnit test code from the Apache and Eclipse ecosystems, and 19 surveyed developers. For both studies they analyzed the following test smells: *Assertion Roulette*, *General Fixture*, *Eager Test*, *Mystery Guest*, and *Sensitive Equality*. These test smells were detected using their own implemented detection rules proposed by Bavota

et al. in their empirical study [BQO+12]. The survey concluded that developers do not perceive test smells as design problems and could not detect them in their code. Moreover, in the inspection of software projects, they found that most test are affected by test smells since their creation. They also concluded that these smells have a high survivability with a 50% chance of test being affected by a test smell after 2000 commits from its creation. These findings stress the need for preventive measures, such as quality checks at commit time. Lastly, they assessed a possible relation with code smells in production code, concluding they have a bidirectional relationship. The presence of test smells could signal the presence of code smells and the other way around. However, these results are constrained by the two software ecosystems investigated which may not be representative of the average practitioners.

To the best of our knowledge, the last documented empirical study was conducted in 2016 by Palomba et al. consisting of 110 open source software projects [PDNP+16]. All of these projects had automatically generated JUnit tests by EvoSuite [FA11]. They investigated the following test smells: *Mystery Guest*, *Resource Optimism*, *Eager Test*, *Assertion Roulette*, *Indirect Testing*, *For Testers Only*, *Sensitive Equality*, and *Test Code Duplication*. For the detection of test smells they relied on a detection tool utilized in previous empirical studies not publicly available. They found that 83% of the generated JUnit classes were affected by at least one test smell. Where, *Assertion Roulette* was the most frequent affecting 54% of JUnit classes, followed by *Test Code Duplication* (33%) and *Eager Test* (29%). The co-occurence of test smells was also investigated, concluding that all smells co-occur with *Assertion Roulette*. Moreover, the pairs *Mystery Guest* and *Resource Optimism*, *Mystery Guest* and *Indirect Testing*, and *Indirect Testing* and *Test Code Duplication* tend to co-occur frequently. They concluded that implementations of automatic test code generation introduce several design flaws, as well as the lack of test fixtures introduces code clones when generating tests. However, these findings are concluded from automatically generated tests and do not represent the practice.

CHAPTER 3

Theoretical Background

In this Chapter, we discuss the theoretical background for understanding the performed research. Specifically, in Section 3.1 we discuss the JUnit testing framework. Next in Section 3.2 we discuss Boa, its infrastructure, language, and dataset. Lastly, in Section 3.3 we introduce and exemplify the relevant test smells.

3.1 JUnit

JUnit is a unit testing framework of the xUnit family for Java. This framework allows the creation of test methods for production code, serving as a test suite for software development. In the following sections give a brief introduction to the JUnit4 framework.

3.1.1 A Minimal Test

```
1  import org.junit.Test;
2  import static org.junit.Assert.assertEquals;
3
4  public class MultiplierTest {
5    @Test
6    public void testMultiplyMethod() {
7      Multiplier m = new Multiplier()
8      result = m.multiply(3, 3)
9      assertEquals(9, result)
10   }
11 }
```

Figure 3.1: Example of a JUnit Test Class

In Figure 3.1 an example of a JUnit test class is shown, as well as an example test method. In a test class, multiple related test methods are contained. These test methods are meant to test the production class counterpart. For instance, at line 7, a `Multiplier` object is instantiated. The `Multiplier` class is the production class counterpart of `MultiplierTest`, as their names suggest. Specifically, in this example we test the functionality of the `multiply` instance method. More test methods can be included in the test class, as much as needed for ensuring the correct functionality of the production class.

For testing the production class, a test runner is needed. JUnit includes its own runner, however, there are many test runners available depending on the specific needs of the software. Once the test runner is instantiated for a class, all of the defined test methods in that test class are executed. A defined test method is not simply a method in the test class, it must be annotated with the `@Test` annotation for it to be executed.

The basic functionality of a test method is to define the criteria that determines a passed or failed test. In JUnit the developer can explicitly write statements that probe for correct functionality, these statements are called assertions. However, a test method does not need an assertion to be valid. In this case, a test method that does not raise any exceptions is considered as passed. When the test method contains one or more assertions, if any of them do not meet the criteria the test is considered a failure.

3.1.2 Assertion Methods

There are multiple assertion methods that available in the JUnit framework. Most of these JUnit assertion methods can be found in the `org.junit.Assert` class. However, these methods are available with different signatures. We discuss only the representative characteristics of each assert method without regard to their signature.

assertTrue and assertFalse

Both assertion methods have a boolean parameter. This parameter is compared to `True` or `False`, depending on which of the two assertion methods is utilized. If the parameter is not equal to the expected value, the test is deemed as failed. Otherwise, the test is considered a pass.

assertNull and assertNotNull

Similar to the latter, these assertion methods compare their parameter to an expected value. The expected value in this case checked for its equality to `null` or not `null`. If they are equal, the test has passed, otherwise it has failed.

assertEquals, assertNotEquals and assertArrayEquals

These assertion methods require two parameters, the *expected* and the *actual* parameters. Here the rules for equality of Java primitives apply. However, for self-defined classes the `equals` method should be overridden to define the equality of two objects. Otherwise, by default Java checks the for equality in the object's reference, i.e., they are considered equal if they have the same reference. Moreover, in the `assertArrayEquals` both parameters are a arrays, in which the equality is checked in order. Depending on the method utilized the test passes or fails.

assertSame and assertNotSame

Both of these assertion methods are quite similar to the previously mentioned `assertEquals` and `assertNotEquals` methods. The difference between them is that `assertSame` and `assertNotSame` only checks for the object's reference without regard to the `equals` method. Moreover, these methods do not accept primitives as parameters.

assertThat

This method utilizes two parameters in its signature, as well as a bounded type parameter `T`. Through the bounded type parameter a method can be

defined in a generic manner, allowing the method to accept any type of object. The first parameter is an object of type `T` named *actual*. The second parameter is a `Matcher` object, which allows the implementation of a `matches` method. Whenever the assertion method is ran, the implemented `matches` method is called with the *actual* object as a parameter. The `matches` method then evaluates the *actual* object parameter depending on the implementation. If `matches` returns `True`, the test case is considered a pass, otherwise a fail. For simplifying the usage of the `assertThat` method, JUnit includes the Hamcrest framework. This framework allows the declarative definition of `Matcher` objects.

@Test(expected=Exception.class)

Next to all the previously mentioned assertion methods, there is another type of assertion that can be utilized. In a test method that is expected to throw an exception, e.g. an error message, the expected exception type can be passed as a parameter of the `@Test` annotation. When using this parameter, if the test method does not throw the expected exception the test case is considered as fail. Otherwise, it is considered passed.

@Test(timeout=100)

Next to the previous, there is also the possibility to use the *timeout* parameter. Through this parameter, the developer can specify a threshold of time in milliseconds. Whenever the test method crosses this threshold, i.e. takes longer than the specified amount of time, the test case is considered failed.

3.1.3 Mocking Frameworks

Popular mocking frameworks include PowerMock (`org.powermock`), EasyMock (`org.easymock`), Mockito (`org.mockito`), and JMockit (`org.jmockit`). These mocking frameworks allow the creation of placeholder instance objects that take place of real objects. This is particularly useful when isolating the code to be tested by removing the external dependencies with mock objects. Mock objects can replace their real counterparts, however, the functionality remains to be defined in the test code. For example, the tester can specify in a mock object which method calls to expect, and what values to return for each specific call.

3.2 The Boa Infrastructure and Domain-Specific Language

Boa is a domain-specific language and infrastructure for analyzing popular software repositories. Utilizing the domain-specific language provided, scripts can be composed for extracting information from a dataset. In Section 3.2.1, we discuss the dataset provided by Boa. Next, in Section 3.2.2, we discuss the domain-specific language and lastly, in Section 3.2.3 the visitor pattern utilized for analyzing the dataset.

3.2.1 Dataset

Projects	7.830.023
Code Repositories	380.125
Revisions	23.229.406
Unique Files	146.398.339
JUnit Projects	282.577

Table 3.1: Composition of September 2015 full GitHub dataset

Within the infrastructure of Boa, multiple datasets of different sizes are available for mining. The latest dataset, *September 2015 full GitHub* has been selected for our analysis. The Table 3.1 shows the composition of the dataset: 7,8 million projects, 23 million revisions and 146 million unique files. Out of the 7,8 million projects, 282.577 projects include JUnit test classes without the usage of mocking frameworks. These projects are suitable candidates for analyzing the presence of test smells, as they are comparable to previously performed empirical studies. Moreover, for every revision of each file, the Abstract Syntax Tree (AST) is available. An AST is a representation of the syntactic structure of source code in the form of a tree data structure, as exemplified in Figure 3.2. Through the AST all the information regarding the source code is available, thus enabling the detection of smells in test code.

3.2.2 Domain-Specific Language

A domain-specific language is a tailored made computer language for a specific domain, in contrast to General Purpose Languages, like Java, that are made for a variety of domains. Boa has created their own domain-specific language

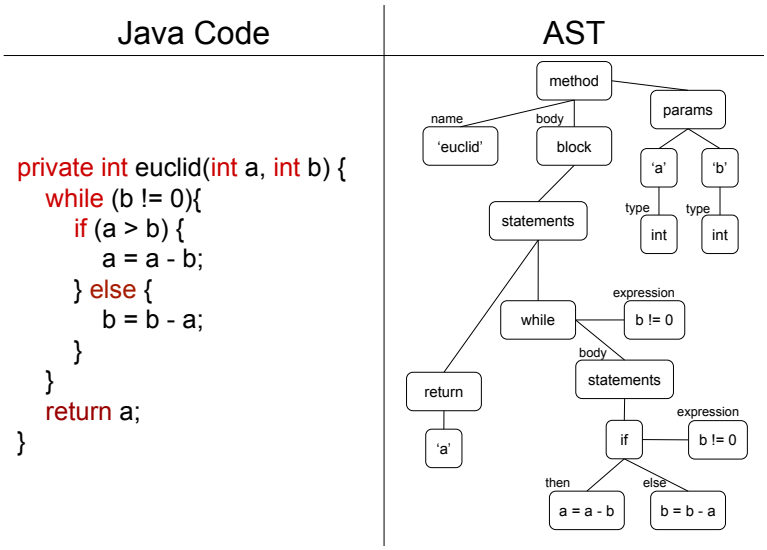


Figure 3.2: A Simplified AST representation of Java Code

bottom(n)	outputs at most n values with the lowest weights
top(n)	outputs at most n values with the highest weights
collection	outputs all the values, i.e. no aggregation
set(n)	outputs the set of values with a maximum of n elements
maximum(n)	outputs the n values with the highest weights
minimum(n)	outputs the n values with the lowest weights
mean	outputs the arithmetic mean of the values
sum	outputs the arithmetic sum of the values

Table 3.2: Output aggregations available in Boa

for analyzing the data collected from software repositories. Scripts composed in this language can be submitted, after which they are parsed and executed by the infrastructure. After the execution an output file is provided as a download link.

The first requirement of the language is that at least one output variable must be defined in a script for ensuring their execution produces an output. Output variables in Boa are aggregators. Aggregators, as their name suggest, collect values and perform an aggregation strategy on them. Once the aggregation of values has taken place, the variable then produces an output. This is rather helpful for consolidating data in meaningful ways. Moreover, multiple aggregators can be defined in any given script, favoring the extraction of compacted data. The aggregations available in the Boa domain-specific language can be seen in Table 3.2. It is also worth noting that groupings can be added to all aggregators by means of indices. These indices allow aggregations to be

separated by groups.

Some aggregators require a parameter, indicated by the `(n)` next to their names in Table 3.2. These parameters are for specifying the number of output values the aggregator should output. Taking the minimum aggregator as an example for defining a variable appropriately named `min`, the syntax is as follows:

```
min : output minimum(2)[string] of string weight int;
```

The variable `min` is defined for outputting the two lowest ranking values by weight. Moreover, the variable has one index of strings, and a string value. Adding more indices can be achieved by simply concatenating more `[type]` keywords in the definition of the variable. Similarly, for an aggregator without a parameter the syntax is as follows:

```
all : output collection[string] of string;
```

In this example, a variable `all` was defined with an index of strings, and a string value. With the collection aggregator, no aggregation is performed and all the values with the different indices are output.

Whenever a value ought to be processed by an aggregator, the value is emitted by using the `<<` operator. In the `min` example, values would be emitted as follows:

```
min["numbers"] << "two" weight 3;
min["numbers"] << "three" weight 1;
min["numbers"] << "one" weight 5;
```

Here the values are emitted on the same `"numbers"` grouping with different weights. Once the execution is finished, the output aggregation outputs the following:

```
min["numbers"] = "three", 1.0
min["numbers"] = "two", 3.0
```

The value `"one"` is excluded from the output, since from the three values emitted `"one"` has the largest weight.

3.2.3 Visitor Pattern

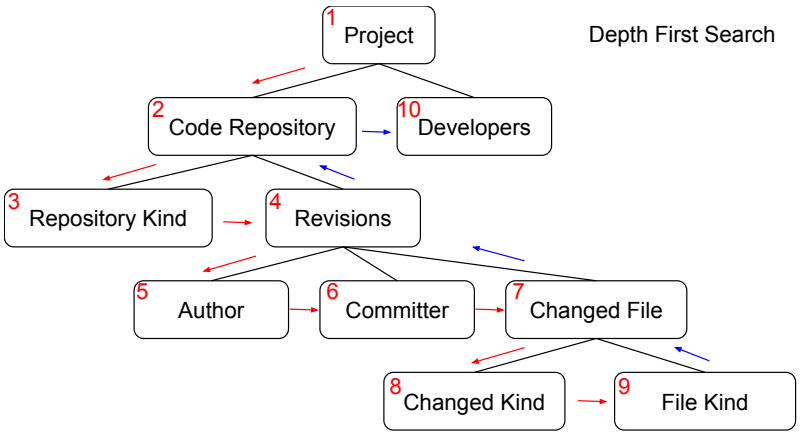


Figure 3.3: DFS Visiting of the Boa Domain Tree

The Boa script starts with a sequential input of all the available projects through the global variable `input`. Each project can be thought of as a tree in which nodes contain different types of information. A tree representation of a project can be seen in Figure 3.3.

For processing each input project, a *visitor* must be defined. A visitor in Boa is a syntactic structure inspired by the object-oriented visitor design pattern. This design pattern is utilized for abstracting an algorithm from the object's structure. In this way, a script can be composed for executing whenever a specified input is given. For instance, in the example shown in Figure 3.3, whenever a new project is inputted, the visitor traverses that project in the order of a Depth First Search (DFS).

Each node in the tree has a different type, therefore, a *clause* can be defined for processing that specific type. A clause has a type parameter, which can be used similarly to Java's polymorphism. Moreover, a clause can be of two classes: `before` and `after`. As their names suggest, the clause executes before or after the node is visited. An example of a script that visits only the Changed File node can be seen in Figure 3.4. In this particular example, the visitor visits nodes 1 to 10 from the previous example in Figure 3.3, but the clause containing the `ChangedFile` type only executes when visiting a node of that type, i.e., node 7. This particular example script shows all the names of the changed files visited in the final output file for every revision in the dataset.

The visitor in Boa can also be stopped by utilizing the `stop` keyword. This allows to stop the default depth-first search traversal strategy at any clause. This is particularly useful for avoiding the complete traversal of sub-trees,

specially when they are not of interest. Whenever the `stop` keyword is used, the visitors stops and returns to the parent node. Moreover, any defined `after` clause of the currently visited node does not execute when stopping the visitor.

Another thing to note is that within the domain tree of a project, there is no direct access to the AST of a file. This has to be specially requested with the built-in function `getast(file: ChangedFile)` that returns an `ASTRoot`. As the name suggests, this is the `ASTRoot` of the file, with which you can continue visiting calling the visit() function.[1]

```
1 p: Project = input;
2 out: output collection of string;
3
4 visit(p, visitor {
5   before n: ChangedFile -> {
6     out << n.name;
7   }
8 });
```

Figure 3.4: Example Visitor for Outputting Changed Files Names

3.2.4 Alternatives to Boa

	Dataset Available	Query Infrastructure	AST Access
Elasticsearch	✗	✔	✗
Sourcerer	(✔)	✔	✔
scrrepo	✗	(✔)	✔
Boa	✔	✔	✔

Table 3.3: Alternatives to Boa

Elasticsearch

Elasticsearch[2] provides a domain-specific language for querying data. This language allows to query data using complex constructs. However, this data must be provided by the user. Moreover, the query language is not specific for source code and as a consequence the ASTs are not available, which are necessary for the detection of test smells.

[1] All the available nodes for an ASTRoot can be found in Boa's webpage `http://boa.cs.iastate.edu/docs/dsl-types.php`.

[2] `https://www.elastic.co/products/elasticsearch`

Sourcerer

Sourcerer is an infrastructure for collecting and analyzing open source code at a large scale [BOL14]. The infrastructure is an SQL database of Java projects for querying the the data. The data, specifically, is the source code's AST translated into a relational database. Unfortunately, the available dataset does not include the commit history of the projects.

scrrepo

The scrrepo system mines software repositories for Eclipse project metadata with the goal of extracting ASTs throughout the version control history [SF14]. Moreover, it provides a query language for extracting information from the collected data. However, the system does not provide a public available infrastructure nor a dataset.

Why Boa?

Boa provides a large dataset including the commit history of over 7,8 million projects. Moreover, a language and infrastructure is provided for querying such large data collection without the need for a special setup. Lastly, for each of the available Java source files (excl. Java 8) their AST is available, thus enabling our test smell detection strategies. Therefore, as Boa is the best suited tool for our research, we utilize their infrastructure and language in this thesis.

3.3 Test Smells

Test smells are indicators of bad design specifically in test code. Many experts have defined test smells in the literature of which a collection of them can be seen in Table 3.4. Test smells 1-11 were defined by van Deursen et al. [vDMvdBK01]. The following 12-18, by Meszaros et al. [MSA03]. Next, 19-23 by Greiler et al. [GvDS13]. Finally, 24-31 by Reichhart et al. [RGD07]. This collection of test smells is in no way a complete list of defined test smells, rather an extract of the test smells available in the literature.

	Smell	**Description**
1	Mystery Guest	Using external resources in test cases introduces hidden dependencies and reduces its understandability.

2	Resource Optimism	Making optimistic assumptions on the absence or state of an external resource causes non-deterministic behavior in the test.
3	Test Run War	Sharing resources with other testers may cause a test to fail, except when the test is not simultaneously run with other instances.
4	General Fixture	Having a too general test fixture where the different tests utilize only a part of it reduces the understandability of the tests.
5	Eager Test	Asserting multiple methods of the tested object reduces the understandability and maintainability of the test suite.
6	Lazy Test	Asserting a method multiple times with the same fixture in different tests may affect the consistency of tests when maintaining them.
7	Assertion Roulette (*)	Asserting several methods in the same test without an explanation message leads to reduced traceability when the test fails, as it is not clear which assertion was not met.
8	Indirect Testing	Interacting with the object under test indirectly through another object leads to reduced maintenance of the test.
9	For Testers Only	Including methods in production classes to be used only by test methods reduces the maintainability and understandability of production code.
10	Sensitive Equality (*)	Utilizing equality checks with the `toString` method introduces dependencies on irrelevant details such as formatting.
11	Test Code Duplication (*)	Having tests with similar test code reduces their maintainability.
12	Obscure Test	Not having the expected behavior of the object under test explicit in the test method leads to reduced understandability and maintenance.
13	Conditional Test Logic or Indented Test (*)	Having conditional logic in a test method has as a consequence code that may or may not be executed leading to nondeterministic behavior.
14	Hard-to-Test Code	When writing new tests to a code base is complicated may indicate the need for refactoring the code base to achieve a more testable code base.
15	Fragile Test	Changes made to a system under test results in the failure of unrelated tests, resulting in reduced understandability.

16	Frequent Debugging	When debugging is required to ascertain the reason of most test failures, resulting in reduced maintainability.
17	Manual Intervention	Whenever the test requires manual intervention, such as user input or manual configuration, resulting in reduced maintainability.
18	Slow Tests	The test takes too long to run, resulting in less often automatic verification of the system.
19	Test Mavericks	Having implicit setup on the test method where the test method is independent of such setup, resulting in reduced understandability.
20	Dead Fields	Having class fields that are not used by any test method, possibly indicating conflicts with the single responsibility principle or a non-optimal inheritance structure.
21	Lack of Cohesion of Test Methods	Having a group of test methods in one class that have low cohesion, i.e. are unrelated to each other, leading to reduced understandability and maintainability.
22	Obscure In-line Setup	Having a group of essential but irrelevant steps for understanding in an in-line setup of a test method, reduces the understandability of the test.
23	Vague Header Setup	Having the initialization of fields in the header of a class instead of an implicit setup, reduces the understandability and maintainability of the test.
24	Overreferencing	Having multiple method calls to production code in the test method, reduces the understandability of the test.
25	Assertionless Test (*)	Having a test without an assertion serves no purpose.
26	Long Test or Verbose Test (*)	Having a high number of statements in a test method reduces its understandability.
27	Likely ineffective Object-Comparison	Object comparisons that are tautologies are unnecessary possibly reducing the understandability of the test.
28	Early Returning Test	Presenting a return statement in a test method resulting in assertions on dead code.
29	Under-the-carpet failing Assertion	Having commented out failing assertions in a test method.
30	Empty Shared-Fixture	Specifying an empty `setUp` method.
31	Empty Test (*)	Having a test method without a body nor assertions of any kind.

Table 3.4: Test Smells

The test smells denoted by a (*) are the test smells selected for our research. The selection of these test smells is guided by the available detection strategies in the literature, as well as their presence in previous empirical studies. All of the selected test smells have their detection strategies defined in the literature by Breugelmans and Van Rompaey [BVR08]. Moreover, the possibility of detecting the test smells statically is also considered as an important characteristic, resulting in this selection.

In the following sections, we discuss these selected test smells. For each of them, their definition and examples are presented.

3.3.1 Assertion Roulette

```
1  @Test
2  public void testGetReservedTicketsByName_CheckNumOfTickets() {
3    String customerName = "%Alvin%";
4    String performance = "%Performance 1%";
5    List<Object[]> reservation =
     ↪  dao.findReservedTicketIdWithName(customerName, performance);
6
7    assertNotNull(reservation);
8    assertThat(reservation.numOfTickets(), is(2));
9  }
```

Figure 3.5: Example of a Test Case with the Assertion Roulette Smell

Definition

The *Assertion Roulette* smell is a test method where multiple assertions exist without an explanation message. This results in a guessing game, where the developer does not know which of the assertions made the test case fail. A test case containing an *Assertion Roulette* is harder to read, requiring multiple reruns or debugging for determining the reason for the test failure. If multiple assertions are needed in a test case, the explanation message provided by the JUnit framework should be utilized. This results in a message being output when an `AssertionError` is thrown, allowing to identify the failing assertion without the need for a rerun or debugging.

Example

In Figure 3.5, the test case includes four assertions in lines 7 and 8. None of them include an explanation message. If there is no reservation with the name `Alvin` in the database, the assertion in line 7 would fail. Moreover, if that

reservation does not include two tickets, as asserted in line 8, the test case would also fail. In both cases, the developer trying to fix the problem would have to check which of the two assertions failed.

3.3.2 Assertionless

```
1  @Test
2  public void testJsonToFile() {
3    Json j = customerList.toJson();
4    File f = new File("~/test_files/json_to_file.txt")
5    j.saveToFile(f)
6  }
```

Figure 3.6: Example of a Test Case with the Assertionless Smell

Definition

The *Assertionless* smell occurs when a test does not contain any assertion method. Such test can either throw an error or succeed but never assert any state or condition, the test only executes the source code. If the goal of a given test is to check that no error is thrown, the intent should be explicitly included with the `try-catch` blocks and a call to the `Assert.fail()` method in the `catch` block. Otherwise, if the goal is to ensure that an error is thrown, the intent should be also explicitly included in the test annotation, e.g. with the following JUnit construct:

```
@Test(except=IllegalArgumentException.class)
```

Making the intention of the test case explicit also helps developers, other than the author of the test case, to understand what the test case is testing.

Example

In Figure 3.6, a test case is shown where a `JSON` object is persisted into a file. However, this test method does not include an assertion to ensure the object was persisted. The test case only executes the underlying implementation for persisting the object. The outcome of the test case must be manually inspected, removing the advantages of having automated testing.

3.3.3 Empty Test

```
1  @Test
2  public void test() {
3    //TODO
4  }
```

Figure 3.7: Example of a Test Case with the Empty Test Smell

Definition

An *Empty Test* is a test case without any statements in it. Any test case that is empty serves no purpose.

Example

An empty test case may have been left as a stub that was never filled, as exemplified in Figure 3.7. As such, it does not test any functionality and does not add any value to the test suite.

3.3.4 Indented Test

```
1   @Test
2   public void test() {
3     switch(Platform.getOsFamily()){
4       case WINDOWS:
5       if (Platform.getArchName().equals("x86")){
6         //Code for testing in x86 Windows
7       } else if (Platform.getArchName().equals("amd64")){
8         //Code for testing in x64 Windows
9       } else {
10        Assert.fail("Not supported architecture");
11      }
12    case LINUX:
13      if (Platform.getArchName().equals("i386")){
14        //Code for testing in i386 Linux
15      } else {
16        Assert.fail("Architecture not supported ");
17      }
18    default:
19      Assert.fail("OS not Supported")
20    }
21  }
```

Figure 3.8: Example of a Test Case with the Indented Test Smell

Definition

An *Indented Test* is a test case that includes loops or conditionals in its body. This results in a more complex test case than necessary. Moreover, the linearity of the test is compromised, possibly leading to untested functionalities due to the branching conditionals. Therefore, it is advised to break such tests into multiple tests and deal with them accordingly.

Example

In Figure 3.8, the test case has been conditioned with a `switch` statement, for ensuring the proper test code is utilized for different architectures. The statements to be executed depends on the architecture where the test case is running. Not all of the functionality is being tested, unless all three platforms and architectures run the test suite. The test is now unnecessarily complex. The test case should be split in three individual test cases for reducing the complexity. Moreover, the test suite runner should be extended for identifying which tests to run depending on the architecture.

3.3.5 Sensitive Equality

```
1  @Test
2  public void test() {
3    Map<String, Integer> map = new HashMap<>();
4    map.put("a", 1);
5    Json jsonObj = new Json().fromMap(map);
6    assertEquals("{\"a\":\"1\"}", jsonObj.toString())
7  }
```

Figure 3.9: Example of a Test Case with the Sensitive Equality Test Smell

Definition

A test with the *Sensitive Equality* smell is verifying an object's characteristics through its `toString()` method. This results in an implicit dependency in irrelevant details, such as formatting.

Example

For example, in Figure 3.9 the functionality of creating a `JSON` object from a `Map` is being tested. However, to assert that the functionality is working correctly, the assert statement compares the `jsonObj` object's state to a hard-coded string. Whenever any of the formatting of the `toString()` method changes, this particular test case fails and consequently need maintenance, even though the functionality is correct.

3.3.6 Test Code Duplication

```
1   @Test
2   public void test1() {
3     basket.add(item1);
4     basket.add(item2);
5     basket.add(item3);
6
7     assertEquals(15.30, basket.calculate());
8   }
9
10  @Test
11  public void test2() {
12    basket.add(item1);
13    basket.add(item2);
14    basket.add(item3);
15    basket.remove(item2);
16
17    assertEquals(10.30, basket.calculate());
18  }
```

Figure 3.10: Example of a Test Case with the Test Code Duplication Smell

Definition

A test with code duplication has a bad impact on maintainability, as any change that breaks a test case can require changes on all duplicates. Therefore, any code duplication should be refactored to helper, setup, or teardown methods.

Example

For example, in Figure 3.10 two test cases share the same three lines of code. By introducing a method using the `@Before` annotation provided by JUnit, we can create a fixture where a basket already contains 3 items. Reducing thus the duplicates and adding a single method for simpler maintenance and better reusability.

3.3.7 Verbose Test

```
1  @Test
2  public void testCreateCustomer() {
3    String firstname = "Mickey";
4    String lastname = "Maus";
5    String ticketcardNumer = "1001";
6    String title = "Dr.";
7    Date validThru = GregorianCalendar
8    .from(ZonedDateTime.now().minusDays(10))
9    .getTime();
10   Address address = new Address();
11   address.setCountry("Austria");
12   address.setCity("Salzburg");
13   address.setPostalCode("5020");
14   address.setStreet("Domplatz 1");
15   Customer c = new Customer(firstname, lastname, title, ticketcardNumer,
     ↪  validThru);
16   c.setAddress(address_1);
17   c.setGender(Gender.MALE);
18   c.setCustomerGroup(CustomerGroup.GOLD);
19   c.setCustomerStatus(CustomerStatus.VALID);
20   dao.save(c);
21
22   Customer saved = dao.save(c);
23   assertThat("Check customer count - should be 13", dao.count(),
     ↪  is(13L));
24 }
```

Figure 3.11: Example of a Test Case with the Verbose Test Smell

Definition

A *Verbose Test* is a test where there are too many lines of code, making it harder to understand and more complex overall. Typically a *Verbose Test* includes too many assertions for ensuring preconditions and postconditions, moreover, they often include fixtures. Therefore, splitting up a *Verbose Test* enables a better maintainability, reusability, and readability.

Example

For example, in Figure 3.11 a test case is shown, where a new customer is being persisted in the database. From line 3 to 17, the customer object is being populated. If the population of the customer object would have been done in a setup method, instead of having 22 lines of code, this method would have 7 lines of code.

```
1    for (i = 0; i < 100; i++)
2    {
3      printf("hello");
4    }
```

(a) 4 LOC, 2 LLOC

```
1    for (i = 0; i < 100; i++)
     ↪  printf("hello");
```

(b) 1 LOC, 2 LLOC

Figure 3.12: Physical Lines of Code (LOC) vs Logical Lines of Code (LLOC)

However, There is no consensus on how many lines of code a test method should have before being considered a *Verbose Test*. Moreover, there is a distinction between logical lines of code (LLOC) and physical lines of code (LOC), as exemplified by Figure 3.12.

CHAPTER 4

Case Study Setup

4.1 Dataset

4.1.1 Selecting Relevant Projects

In order to answer our RQs, we selected the latest dataset in Boa, which is the *September 2015 full Github* dataset. The dataset includes 7,8 million projects, however, not all projects in the dataset are relevant for our investigation. Although GitHub hosts a plethora of programming languages, our research is specifically aimed for Java. All of the empirical studies on test smells known to the authors were performed on Java projects. By maintaining our investigation in-line with previous studies, a comparison of the results with established findings remains possible. Moreover, these studies investigated the usage of the JUnit test framework, as test smells stem from xUnit test patterns. However, they do not specify as to which version of JUnit is utilized in the probed projects.

Previous versions to JUnit4 relied on naming conventions for detecting and executing test cases, whereas JUnit4 relies on Java Annotations. In contrast to previously performed empirical studies, our dataset is not conformed of selected projects known to contain test methods. Therefore, we define a set of

rules for determining whether a Java class is a JUnit test class.

Utilizing naming conventions for detecting test classes may exclude newer test classes, as these conventions are not a requirement for the proper functioning of the framework in later versions. Contrastingly, utilizing the `@Test` annotation for detecting a JUnit test class, excludes projects utilizing older versions of the framework. In this thesis we detect the latter, as projects utilizing JUnit4 are more likely to be in active development and can better help us understand the current diffusion of test smells.

In a typical software project the usage of mocking frameworks can be found. These frameworks allow to replace dependencies for avoiding the calling of complex objects and isolate the system under test. However, due to their nature, mocking framework are typically used in integration tests rather than unit tests. To the best of our knowledge, integration test smells have not been defined in the literature. Therefore, projects including these frameworks are excluded from our selection.

As stated before, GitHub has a vast amount of projects with different programming languages, even projects with multiple programming languages. Our investigation is concerned with JUnit test classes, which can appear in projects with multiple programming languages. Therefore, our selection includes these projects, as long as they contain JUnit test classes.

In our final selection, out of the 7,8 million projects in the Boa dataset 554.864 are projects that contain Java source code. For each of the Java source code files an AST is included, except for Java 8 projects. This is due to Boa not including their ASTs in the dataset. Moreover, in this selection empty Java projects without source code may be included. Therefore, projects that do not include an AST or JUnit test classes are excluded as they cannot be analyzed. When further filtering irrelevant and empty projects, we found 282.577 projects that include JUnit test classes without the usage of mocking frameworks. For each of these projects their commit history is included in the dataset, as well as the AST for each of the files at each revision. This allows us to investigate the smells throughout the development time and see their evolution.

4.1.2 Data Gathering

For each of the test smells, enough data for answering our research questions must be gathered. In order to uniquely identify a detected smell the following data is aggregated into a tuple:

```
(Project ID, File, Method, Smell)
```

Next, for measuring the evolution on the test smells, temporal information must be gathered. In Figure 4.1, a time lapse of the life stages of a test smell is represented.

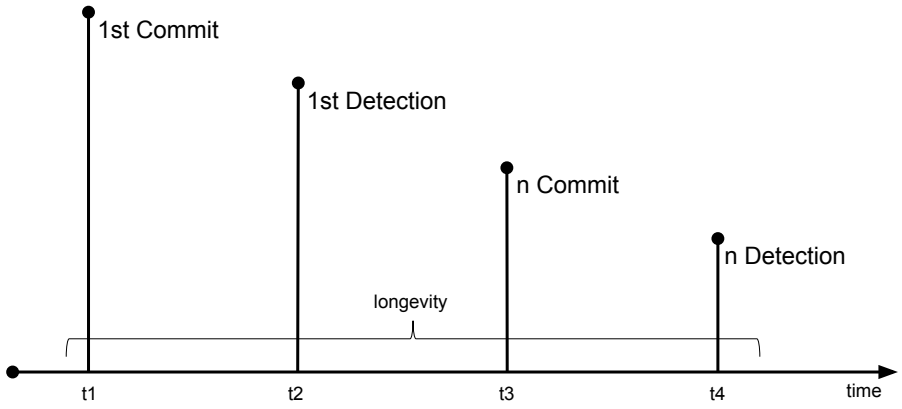


Figure 4.1: Life Stages of a Test Smell

t1 the first commit where the test method T is detected.

t2 a test smell is present in the test method T.

t3 the last commit where the test method T was detected.

t4 the last moment the test smell is present in the test method T.

Therefore, for each test smell detected, we have the following data:

```
(Project ID, File, Method, Smell, t1, t2 , t3, t4, longevity)
```

Where each of the `t`'s are have represented by tuple:

```
(Timestamp, Commit ID)
```

The `Timestamp` is the exact time where the change was committed, and the `Commit ID` is the 40 characters that uniquely identify the commit.

t1-t4 are temporal points and as such they may overlap. Whenever *t1* equals *t4*, the test method has been detected for the first time and no further changes to the class have been made. Similarly, when *t2* equals *t3*, the test smell has been introduced for the first time and no further changes to the class have been made.

Moreover, if *t1* and *t2* are equal, the test smell was introduced when creating the test method. Otherwise, the test smell was introduced at commit time *t2*. Similarly, if *t3* and *t4* are equal, the test smell was detected until the end of the available dataset. This suggest the test smell was not fixed. Otherwise, if $t1 \neq t4$ the test smell was fixed at time *t3*.

Lastly, the `longevity` is the number of commits between the first and the last change, starting from 1.

With this data gathered for each test smell, we answer our proposed research questions in Section 1.3.

4.2 Test Smell Detection

As Boa inputs all available projects in the dataset to the visitor, therefore, ensuring that the visitors visit only the desired projects for analysis is necessary. To achieve this, the visitor must be stopped whenever a project is not of interest. In Figure 4.2 the filtering clauses for identifying a project of interest are shown. First, in line 2, the existence of Java is investigated in the input project. This is achieved by utilizing the `ifall` quantifier. With this quantifier checking all of the programming languages in the project is possible. Whenever all of the programming languages in the project do not match the keyword `java` the visitor is stopped. This means that if at least one of the programming languages is Java, the visiting of the project is continued. Through this filter multi-language projects are also visited, however, the following clauses ensure that only Java files are analyzed.

Next, the existence of a mocking framework is investigated. For this, the `exists` quantifier is utilized in line 8. This statement checks all of the import statements in the AST of the currently visited `ChangedFile` node. Therefore ensuring that the class does not include any import statement for the following Mocking frameworks: PowerMock (`org.powermock`), EasyMock (`org.easymock`), Mockito (`org.mockito`), and JMockit (`org.jmockit`).

Subsequently, in line 10, the existence of an import statement for the `@Test` annotation, i.e. `org.junit.Test` is ensured. At this point, any file that does not include an AST is excluded. The visitor can only visit the `Method` node if an AST exists for the `ChangedFile`, which is only the case for Java files. This is due to the fact that the Boa dataset only includes AST's of Java files, unfortunately excluding Java 8.

Lastly, only tests cases annotated with `@Test` are analyzed. This is achieved with the `ifall` quantifier in line 15. If the `@Test` annotation is not present in the method, the visitor is stopped. However, utilizing this filtering clause excludes any test cases written with a JUnit version prior to JUnit4.

```
1  before n: Project -> {
2    ifall (i: int; !match(`^java$`, lowercase(n.programming_languages[i])))
3      stop;
4  }
5
6  before node: ChangedFile -> {
7    currentAst = getast(node);
8    exists (i: int; match((`org.powermock`), currentAst.imports[i]) ||
     ↪  match((`org.easymock`), currentAst.imports[i]) ||
     ↪  match((`org.mockito`), currentAst.imports[i]) ||
     ↪  match((`org.jmockit`), currentAst.imports[i]))
9      stop;
10   ifall (i: int; !match((`org.junit.Test`), currentAst.imports[i]))
11     stop;
12 }
13
14 before node: Method -> {
15   ifall (i: int; ! match((`Test`), node.modifiers[i].annotation_name))
16       stop;
17 }
```

Figure 4.2: Filtering Clauses for pure JUnit Test Cases in Java Projects

In the following sections, we discuss the technique utilized for the detection of each test smell. All of the presented test smell detection scripts utilize the previously discussed clauses for selecting projects of interest.

4.2.1 Assertion Roulette

Detection

The detection metric for the *Assertion Roulette* is based on the metric published by Breugelmans and Van Rompaey [BVR08]. The metric is adapted to be specific for the JUnit framework.

This smell can be detected by defining the set of all assertion methods from the JUnit framework, denoted as JAM. For a given test case, T is the set of all the statements within that test case. From the set T, all of the method calls that are assertion methods from JAM are selected as follows:

$$AM = \{\alpha | \alpha \in T \wedge \alpha \in JAM\} \tag{4.1}$$

The selected assertion methods are separated in a set of those containing an explanation message, and another set for those without one.

$$AM = AM_{noexplanation} \cup AM_{withexplanation} \tag{4.2}$$

An `Assertion Roulette` smell, denoted as $AROU$, is present when the Equation 4.4 holds.

$$AROU = \{T \mid |AM| \geq 2 \wedge |AM_{noexplanation}| \geq 1\} \tag{4.3}$$

$$AROU \not\equiv \emptyset \tag{4.4}$$

Detection Script

Regular expressions are utilized for identifying all assertion methods made available by the JUnit framework. The identified assertions can be grouped into three categories by the number of parameters needed to make an assertion: binary, unary, and methods without a parameter requirement. The latter refers to the `fail` statement. Next, unary methods can be detected with the expressions `assertFalse`, `assertTrue`, and `assert.*Null`. Lastly, binary methods can be detected with the expressions `assert.*Equals`, `assert.*Same`, and `assertThat`. The *expected* and *timeout* parameters in the `@Test` annotation are excluded, as they are clearly notified by the JUnit framework when their assertion is not met.

```
1  hasAssertionMessage := function(e: Expression) : bool {
2    if(match((`assert.*Equals`), e.method) && len(e.method_args) == 3){
3      return true;
4    } else if(match((`assertFalse`), e.method) && len(e.method_args) == 2){
5      return true;
6    } else if(match((`assertTrue`), e.method) && len(e.method_args) == 2){
7      return true;
8    } else if(match((`assert.*Null`), e.method) && len(e.method_args) ==
     ↪  2){
9      return true;
10   } else if(match((`assert.*Same`), e.method) && len(e.method_args) ==
     ↪  3){
11     return true;
12   } else if(match((`assertThat`), e.method) && len(e.method_args) == 3){
13     return true;
14   } else if(match((`fail`), e.method) && len(e.method_args) == 1){
15     return true;
16   }
17     return false;
18 };
```

Figure 4.3: Function for Detecting Assertion Explanation Message

Next to the number of parameters needed to make an assertion, JUnit provides an extra `String message` parameter. This parameter is utilized for explaining the reason for the assertion. Considering the number of parameters, the `Assertion Roulette` smell can be detected by counting the number of parameters present in those assertion methods using their AST. For methods without a parameter requirement, there should be one parameter. Similarly, two parameters for unary methods and three for binary methods. Should any of the investigated assertion methods lack the extra parameter, it is assumed they lack an explanation message, as shown in Figure 4.3.

All the instances of assertions with and without an explanation message are counted per method basis. After the method subtree from the AST is finished visiting, the Equation 4.3 is calculated.

The full detection script can be found in Appendix A.1.

4.2.2 Assertionless

Detection

This detection metric is also adapted from the metric published by Breugelmans and Van Rompaey [BVR08]. Similar to the previous, this metric is adapted to be specific for the JUnit framework.

This smell can be detected by re-utilizing the sets defined in Equation 4.1. With the set of all assertion methods AM from the set of all statements in a test case T, we can identify whether the *Assertionless* smell, denoted by $ALESS$, is present. When the Equation 4.6 holds, the test case presents the *Assertionless* smell.

$$ALESS = \{T \mid AM \equiv \emptyset\} \tag{4.5}$$

$$ALESS \not\equiv \emptyset \tag{4.6}$$

Detection Script

```
1  hasExpectedAssert := function(m : Method): bool {
2    exists(i: int; match((`Test`), m.modifiers[i].annotation_name)){
3      exists(j : int; match(`expected`,
       ↪  m.modifiers[i].annotation_members[j])
4      || match(`timeout`, m.modifiers[i].annotation_members[j])
5      ){
6        return true;
7      }
8    }
9    return false;
10 };
```

Figure 4.4: Function for Detecting the Expected and Timeout Parameter

The goal of the script, is to detect whether there is any assertion in a given test method. For this, most of the assertion methods provided by JUnit can be detected with the regular expression `assert`. With the exception of the `fail` method. However, the lack of an assertion in the form of a method does not mean that there are no assertions present in the test method. JUnit also provides the possibility to include an *expected* `Exception` or a specific *timeout* where the test should have ended. The function shown in Figure 4.4 checks for the previously mentioned elements. Finally, the script considers all different possible assertions and counts them. If the count is 0 we have found an *Assertionless* test.

The full detection script can be found in Appendix A.2.

4.2.3 Empty Test

Detection

This detection metric is based on the textual description by Breugelmans and Van Rompaey [BVR08]. As no formal description of the metric is provided, the textual description is adapted into a formal description.

The metric states that by utilizing the set of all statements T for a given test case, the *Empty Test* smell can detected when the Equation 4.8 holds.

$$EMPTY = \{T \mid T \equiv \emptyset\} \tag{4.7}$$

$$EMPTY \not\equiv \emptyset \tag{4.8}$$

Detection Script

```
1  before node : Statement -> {
2    if(def(node.kind) && node.kind == StatementKind.EXPRESSION &&
     ↪  def(node.expression) && def(node.expression.kind) &&
     ↪  node.expression.kind == ExpressionKind.LITERAL){
3      # Comments do not count!
4    } else {
5      num_of_statements++;
6    }
7  }
```

Figure 4.5: Visitor for Statements in the *Empty Test* Smell

The script for the detection of an *Empty Test* smell is rather simple. For each of the visited methods' AST, all of the statements are counted. However, comments should not count and must be excluded from the number of statements. This is achieved with the visitor shown in Figure 4.5.

The full detection script can be found in Appendix A.3.

4.2.4 Indented Test

Detection

The detection metric for *Indented Test* is also described by Breugelmans and Van Rompaey [BVR08]. We detect this smell by identifying all the loops and conditionals present in the sets $LOOP$ and $COND$ respectively.

$$LOOP = \{DO, FOR, WHILE\} \tag{4.9}$$

$$COND = \{IF, SWITCH\} \tag{4.10}$$

Next, given a set T with all the statements of a test case, we check whether any of the elements of $LOOP$ or $COND$ are in T. If the Equation 4.12 holds, then we have detected a test case with the *Indented Test* smell.

$$INDENT = \{T \mid \exists \alpha, \alpha \in T \wedge \alpha \in LOOP \wedge \alpha \in COND\} \tag{4.11}$$

$$INDENT \not\equiv \emptyset \tag{4.12}$$

Detection Script

```
1  isLoopOrCond := function(m : StatementKind): bool {
2    if(m == StatementKind.DO || m == StatementKind.FOR || m ==
     ↪  StatementKind.WHILE || m == StatementKind.IF || m ==
     ↪  StatementKind.SWITCH
3    ){
4      return true;
5    }
6    return false;
7  };
```

Figure 4.6: Function for Detecting Loops or Conditionals

For detecting the *Indented Test* smell, the Loop and Conditional Statements must be detected. This is achieved by inspecting all of the statements within a test method and inspect their kind, as shown in Figure 4.6. If any of the loop or conditional statements is present in the test method, an *Indented Test* smell is present.

The full detection script can be found in Appendix A.4.

4.2.5 Sensitive Equality

Detection

The detection metric for *Sensitive Equality* is also described by Breugelmans and Van Rompaey [BVR08]. For detecting the *Sensitive Equality* smell in a given test method, we utilize the aforementioned set of all assertion methods AM. In this set, we look for a call to the `toString()` method. If we find any, we conclude that the test includes the smell as shown below.

$$SEQ = \{T \mid \exists \alpha, \alpha \in T \wedge \alpha \in AM \wedge \alpha \equiv TOSTRING\} \tag{4.13}$$

$$SEQ \not\equiv \emptyset \tag{4.14}$$

Detection Script

```
1  before node : Statement -> {
2    if(def(node.expression.method) && ( match((`assert`),
     ↪  node.expression.method))){
3      exists (i: int; match(`^toString$`,
       ↪  node.expression.method_args[i].method))
4        has_sensitive_equality = true;
5    }
6  }
```

Figure 4.7: Function for Detecting toString Method Calls

Detecting this particular smell is rather simple and straight forward. As shown in Figure 4.7, if on any of JUnit's assertion methods parameters a method call to a `toString` method is placed, the smell has been detected.

The full detection script can be found in Appendix A.5.

4.2.6 Test Code Duplication

Detection

The detection metric for this smell is textually described by Breugelmans and Van Rompaey [BVR08]. However, for detecting code clones, van Bladel et al. propose a tree- and token-based approach for detecting Type-1 and Type-2 code clones, which are syntactically identical code fragments [vBMD17]. The algorithm transforms the AST into a string representation utilizing character tokens, allowing the comparison of different methods and the subsequent detection of duplication. Moreover, this approach is optimized for detecting code clones in Boa. Therefore, by adapting this algorithm to perform the code clones detection only in test code, the detection of *Test Code Duplication* smells is possible.

Detection Script

After the transformation of the AST into tokens, the detection script takes all the functions of a projects as input for creating a permutation of function pairs. For example, a project with n functions, the algorithm creates the permutations: $[f_1, f_2]; ...; [f_1, f_n]; ...; [f_{n-1}, f_n]$.. Next, for each of the permutated pairs a sliding window loops over one of the functions comparing it to the other. If the sliding window matches a portion of the other function, this means a piece of duplicated code has been found.

For more details, the full detection script can be found in Appendix A.6.

4.2.7 Verbose Test

Detection

The detection metric for this smell is textually described by Breugelmans and Van Rompaey [BVR08]. For detecting the *Verbose Test* smell we utilize the aforementioned set of all statements T of a given test case. Moreover, we define a threshold n. This threshold is then compared to the LLOC of the test method using the cardinality of the set T as shown below. If the Equation 4.16 holds, we conclude that the test method contains a *Verbose Test* smell.

$$VERBOSE(n) = \{T \mid |T| > n, n \in \mathbb{N}\} \tag{4.15}$$

$$VERBOSE(n) \not\equiv \emptyset \tag{4.16}$$

Detection Script

Detecting the *Verbose Test* smell is rather simple as well. By counting all the statements belonging to a `Method` and comparing it to the specified threshold, instances of the test smell can be easily detected.

For more details, the full detection script can be found in Appendix A.7.

4.3 Test Smell Detection Parameters

The Boa scripts are designed to gather the data discussed in Section 4.1.2 on the same run. For performing our experiment most of the scripts can be submitted as is. However, some of the presented scripts need certain configuration parameters to be set before the detection. Namely, the *Assertionless*, and *Verbose Test*.

4.3.1 Assertionless and Empty Test

Per definition, an *Assertionless* test case is one that does not contain any assertion method. Theoretically, this would encompass the definition of an *Empty Test* as well. An *Empty Test* does not contain a body and thus no assertion methods. Therefore, for measuring the degree in which an *Assertionless* smell

encompasses an *Empty Test*, a configuration parameter in the *Assertionless* detection script is defined for counting or excluding empty tests. Two analysis with different parameters must be performed: once including empty tests, and another excluding them. By looking at the difference between the results, we can ascertain the degree that an *Assertionless* smell includes the *Empty Test* smell.

4.3.2 Lines of Code Threshold in Verbose Test

To the best of our knowledge, a proper threshold for determining a test case as a *Verbose Test* has not been defined in the literature. However, there has been an educated guess by Van Rompaey et al. setting this threshold at 15 LLOC [VRDBDR07]. A simple Boa script can extract common sizes of test methods. With this information the threshold can be further investigated for its validity.

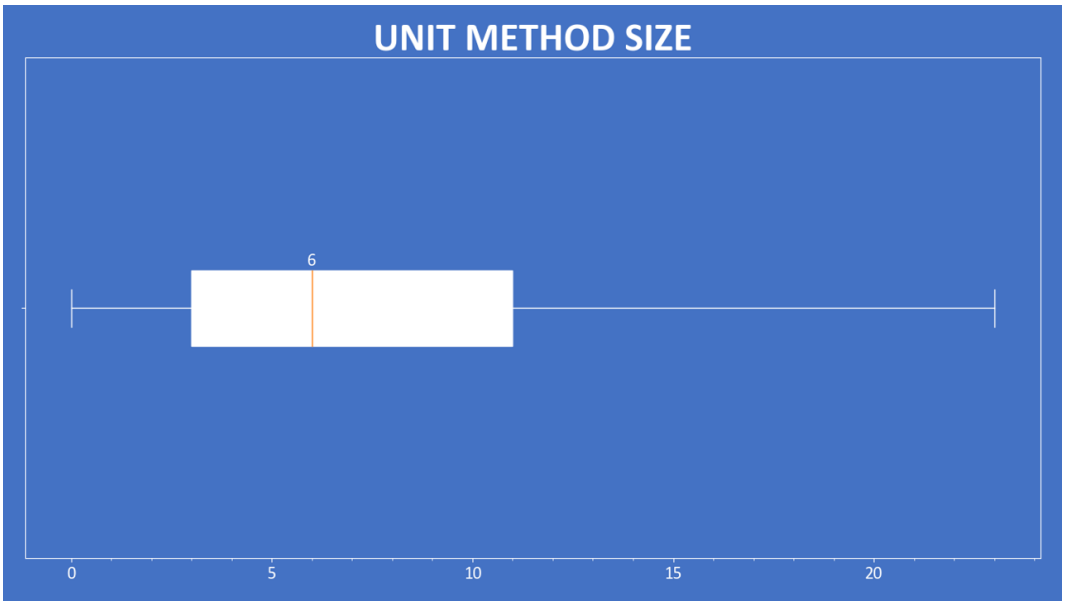


Figure 4.8: Unit Method Size Boxplot

The Figure 4.8, shows a box-plot of the unit method sizes found in the dataset. In this distribution, the aforementioned threshold is within the inner fence of the size distribution. This suggests that the threshold is valid for our investigation, although perhaps high for our dataset. However, as determining an appropriate threshold is out of scope in this thesis, we utilize the threshold defined in the literature.

The utilized script can be found in the Appendix B.2.

4.4 Verification

Initially, our findings were to be validated utilizing detection tools published in the literature. We tested the tools CodeCover, TestHound, TestLint, and TestQ. However, we found that these tools available to us are ill-suited for our purpose. Moreover, their maintenance and development has been abandoned, rekindling the need for a tool for detecting test smells. Therefore, the correctness of the detection strategies has been manually verified. This, however, adds a threat to our validity, as manually verifying all the instances of detected smells is infeasible. False-positives and false-negatives can still occur, and may skew our findings. Great caution in the creation of the scripts and random verification of some results was performed, granting us an acceptable degree of confidence in the detection.

4.5 Detection Scripts Output

```
1  z1_first_occurrence : output minimum(1)[string] of string weight int;
2  z2_last_occurrence : output top(1)[string] of string weight int;
3  z3_longevity : output sum[string] of int;
4  z4_first_change : output minimum(1)[string] of string weight int;
5  z5_last_change : output top(1)[string] of string weight int;
```

Figure 4.9: Output Variables Defined in the Detection Scripts

In the Boa language, output variables are defined for aggregating information, these in turn are automatically output in a file after the analysis is finished. All the detection scripts have five output variables as shown in Figure 4.9. For each instance of a detected smell, five output lines of the form below are printed.

```
1  variable[project_id;filename;method_name] =
   ↪  revision_id;revision_timestamp
```

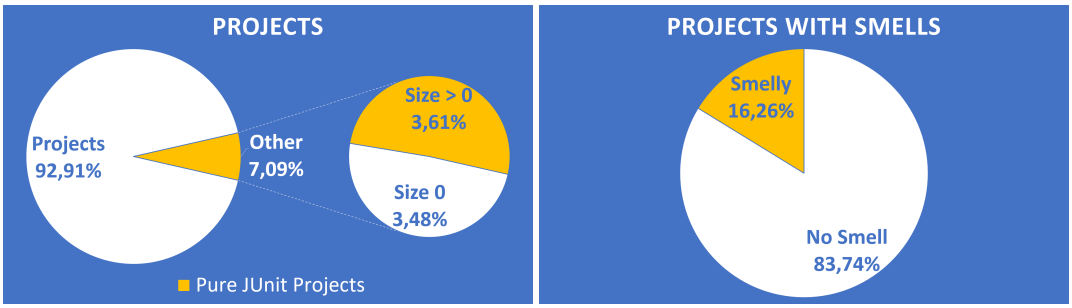
Having millions of smell instances, i.e., on a large scale, a file with the standard Boa output is not easily queried. Therefore, we have transformed

the Boa output files into a CSV file and a SQLite database. This allows us to store the CSV file in a condensed usable dataset. Whereas the SQL database, allows us to perform queries on the dataset. The generated CSV-file can be downloaded here: `http://dx.doi.org/10.6084/m9.figshare.6455666`.

CHAPTER 5

Results

5.1 Test Smells Detection



(a) Selected Projects (b) Projects Presenting Test Smells

Figure 5.1: Selection of Relevant Projects

Out of the 7.830.023 projects in the dataset, only $7,09\%$ (554.863) of the projects are Java projects that solely utilize JUnit in their test suites. However, we found that not all projects in this selection presented an AST. Once the projects without an AST are removed, our selection of projects is reduced to merely $3,61\%$ (282.577) of the dataset, as shown in Figure 5.1a. However,

our selection is the largest selection of projects with which an empirical study on test smells has been performed so far in the literature. The selection composed of 282.577 projects are referred to as dataset for the rest of this chapter. Moreover, when mentioning test smells, we refer to the seven test smells discussed in Section 3.3: *Assertion Roulette*, *Assertionless*, *Empty Test*, *Indented Test*, *Sensitive Equality*, *Test Code Duplication*, and *Verbose Test*.

Out of the 282.577 projects in our dataset, we only detected test smells in $16,26\%$ (45.947) of the projects, as shown in Figure 5.1b. However, projects that do not present any test smells are significantly smaller in terms of AST nodes, than those projects with test smells, as shown in Figure 5.2. This suggests that larger projects are more likely to contain smells. For validating this hypothesis, we investigated the correlation between project size and number of detected smells. A correlation between project size and number of smells was found, with a p-value of $1,8636e^{-128}$ using t-test, indicating strong evidence for our alternative hypothesis.

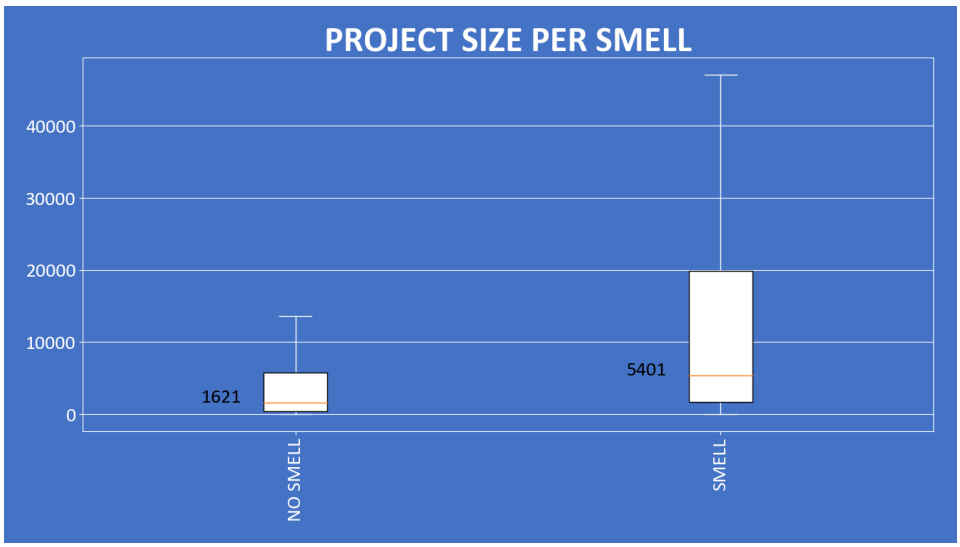
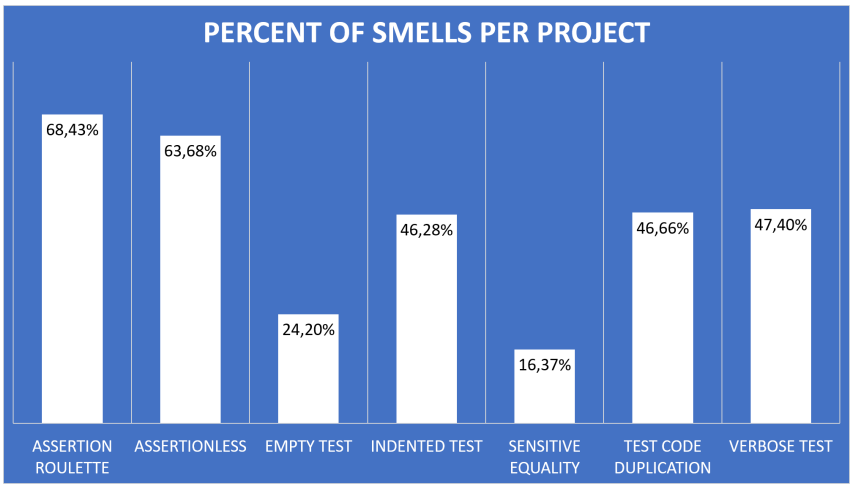


Figure 5.2: Project Sizes With and Without Test Smells

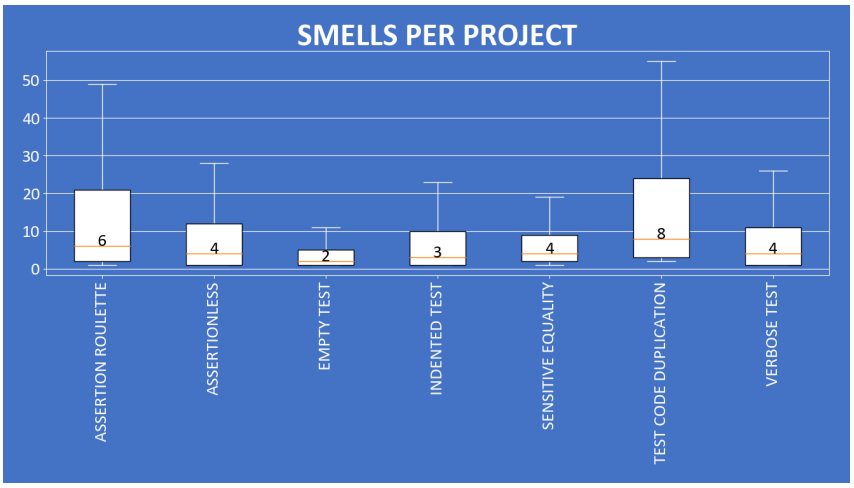
For answering the *ubiquity* RQ2, we investigated our dataset for the most common test smell at three different scopes. First the project scope, aggregating the results per project, allows us to understand what test smell are the most common throughout different projects and development teams, as they typically vary between projects. Next, the class scope, where the results are aggregated per class, allows us to understand how common are the investigated test smells in any given JUnit class file. Finally, the method scope, where the results are not aggregated, allows us to understand how often does these smells appear throughout the dataset.

At the project scope, we found that the most common smell was the *Assertion Roulette* with 68, 43% of the projects presenting the smell, followed by the *Assertionless* (63, 68%) and *Verbose Test* (47, 40%), as shown in Figure 5.3a. The first two smells are present in more than half of the projects in the dataset. Moreover, *Verbose Test*, *Indented Test* (46, 28%) and *Test Code Duplication* (46, 66%) were close to half the dataset, while *Empty Test* (24, 2%) and *Sensitive Equality* (16, 37%) are fairly less common at this scope. However, these result shows the presence of a smell without regard to their frequency.

The box-plot in Figure 5.3b, the distribution of the test smells appearance frequencies is shown. Here the most common smell is *Test Code Duplication* having the highest median, upper quartile and upper fence, closely followed by *Assertion Roulette*.

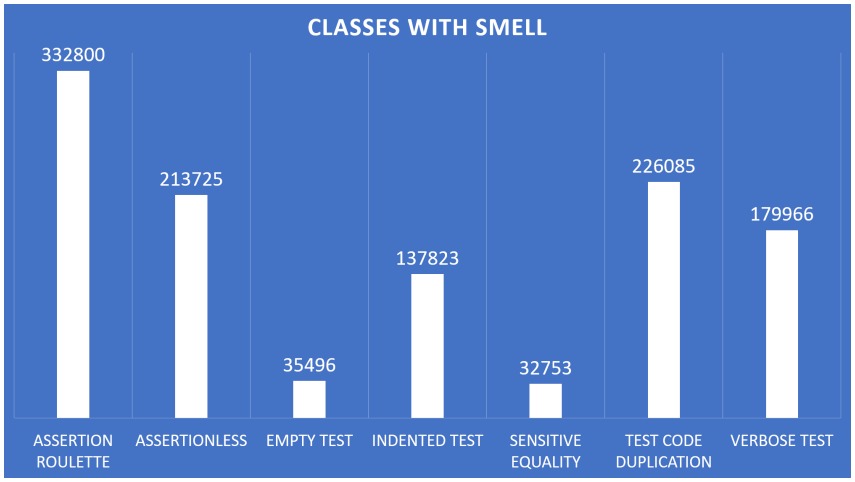


(a) Distribution of Smells in Projects

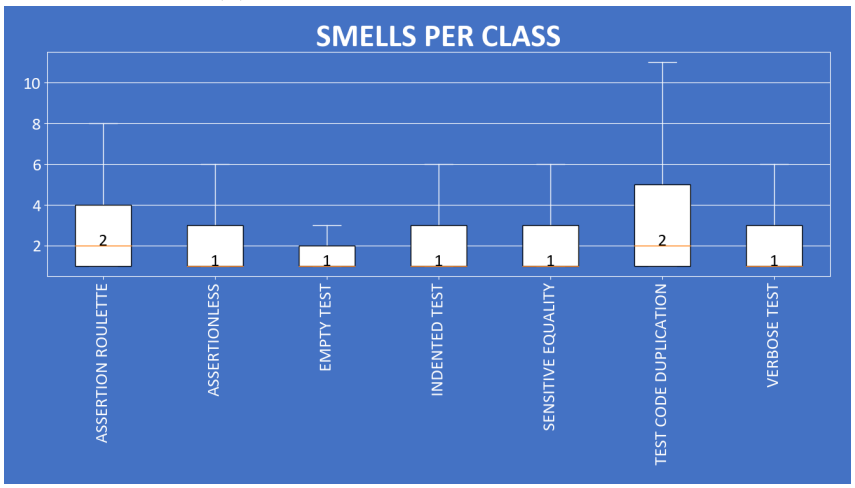


(b) Box-plot of Smells in Projects

At the class scope, we found that the most common smell in terms of unique instances is the *Assertion Roulette*, followed by *Test Code Duplication*, and *Assertionless*, as shown in Figure 5.4a. Moreover, when looking at their distribution with the box-plots shown in Figure 5.4b, the most frequent smells at this scope are *Test Code Duplication*, and *Assertion Roulette*.



(a) Number of Classes with Smell



(b) Box-plot of Smells in Classes

At the method scope, as shown in Figure 5.5, the smell with most appearances in our dataset is the smell *Assertion Roulette* with 1.378.250 detections, followed by the *Test Code Duplication* smell with 1.071.168, and the *Asseritonless* smell with 796.578 detections. Next, the *Verbose Test* (583.077), the *Indented Test* (408.167), the *Sensitive Equality* (93.073), and finally *Empty Test* (79.886).

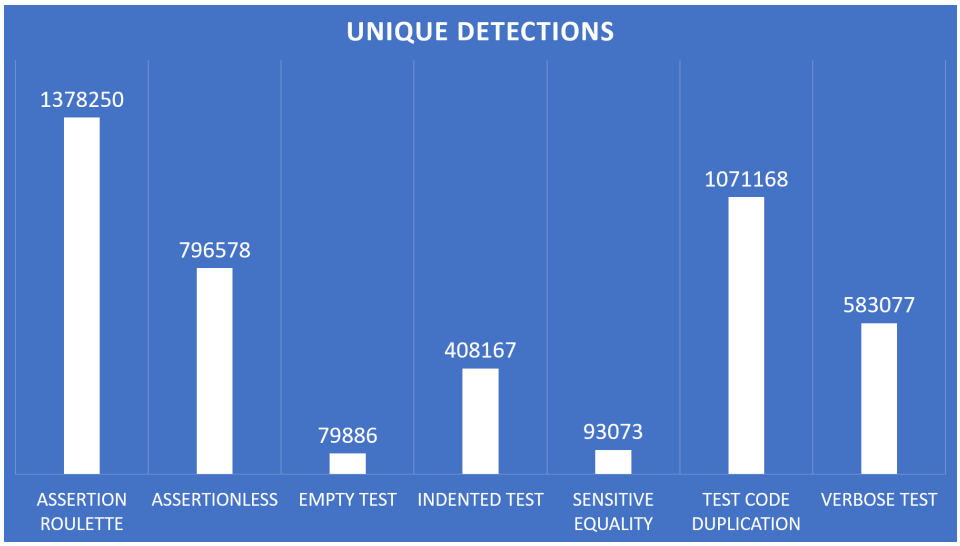


Figure 5.5: Number of Instances Detected by Smell

Assertionless and Empty Test

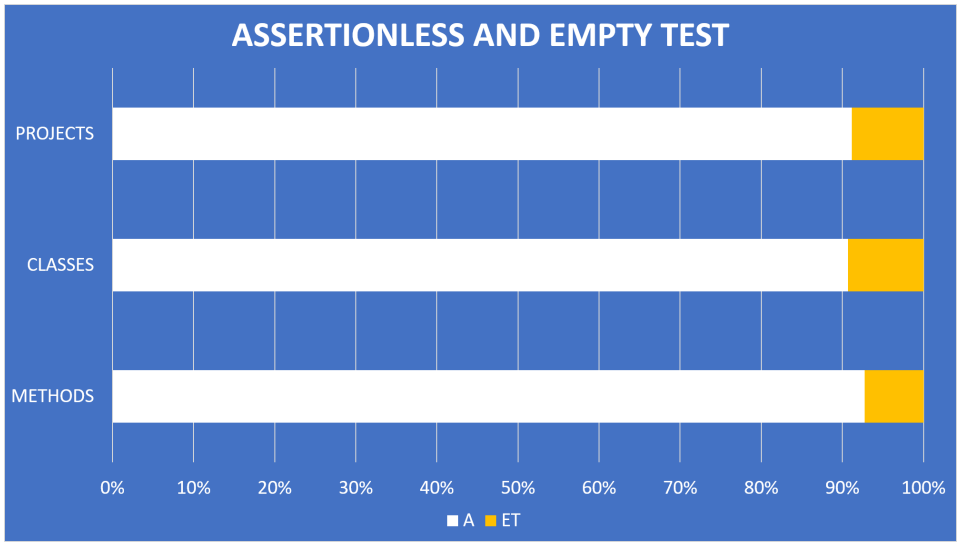


Figure 5.6: Percentage of Empty Test in *Assertionless* Results

As discussed in Section 4.3.1, an *Assertionless* smell may contain instances of *Empty Test* smells. Therefore, we measured the extent in which *Empty Test* are present in *Assertionless* at each scope, as shown in Figure 5.6. In all three scopes, the instances of *Empty Test* never surpasses 10% of the total *Assertionless* instances detected.

5.2 Test Smells Evolution

For answering our *evolution* RQ3, we gathered the data discussed in Section 4.1.2. By utilizing the time of the first detection of a test method, as well as the time of the first detection of a test smell in the same test method, we can determine whether the smell was included on the test method at creation time. As shown in Figure 5.7, most of the investigated test smells instances are introduced when creating the test case. Smells introduced after the creation account to $1,247\%$ of the total detected instances.

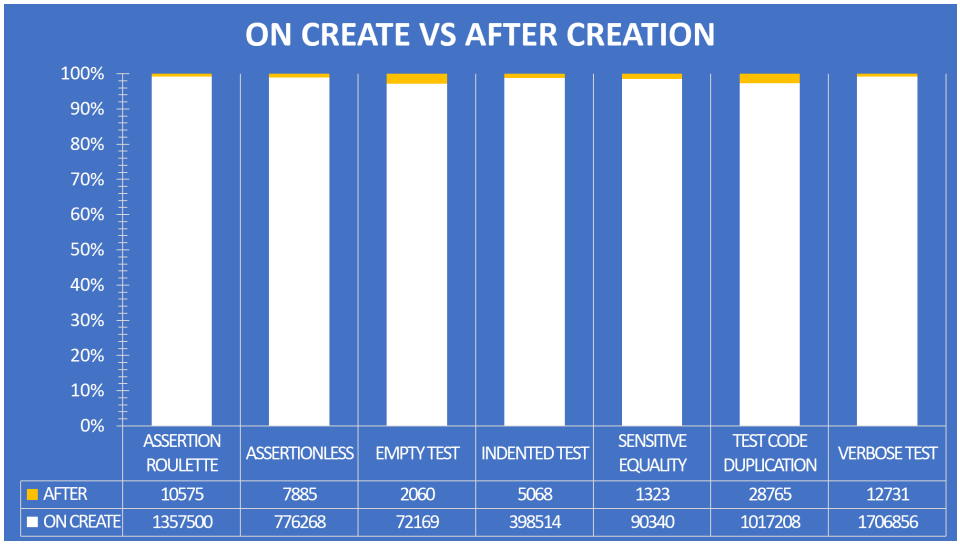


Figure 5.7: Percentage of Smells Introduced at Creation Time or After

Next, we investigated the extent that these test smells are fixed. The results of this investigation are summarized in Figure 5.8. Our results show that most of the instances of test smells detected, were not fixed. This means that until the end date of the data collection of Boa's dataset, we detected the smell and thus conclude it was not fixed. However, a project in active development may remove the test smell in the time after the data collection.

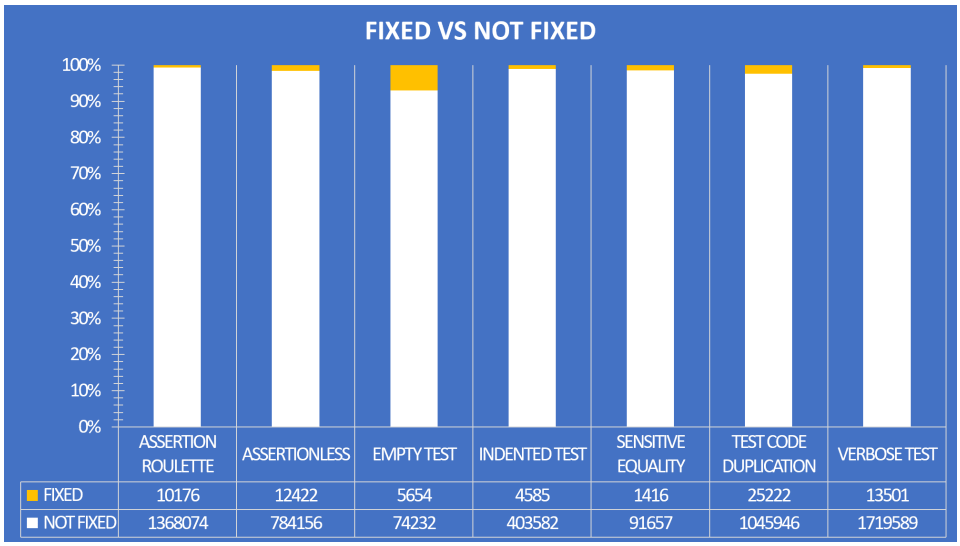


Figure 5.8: Percentage of Smells Fixed or Not Fixed

For understanding how long does it take to fix a test smell, we investigated the test smells that were fixed. Specifically, we investigated their longevity. In Figure 5.9, we can see the results of this investigation. Most smells have a very similar longevity, except for *Assertionless* and *Empty Test*, who have a lower longevity. This suggests they might be more likely to be fixed faster than other smells.

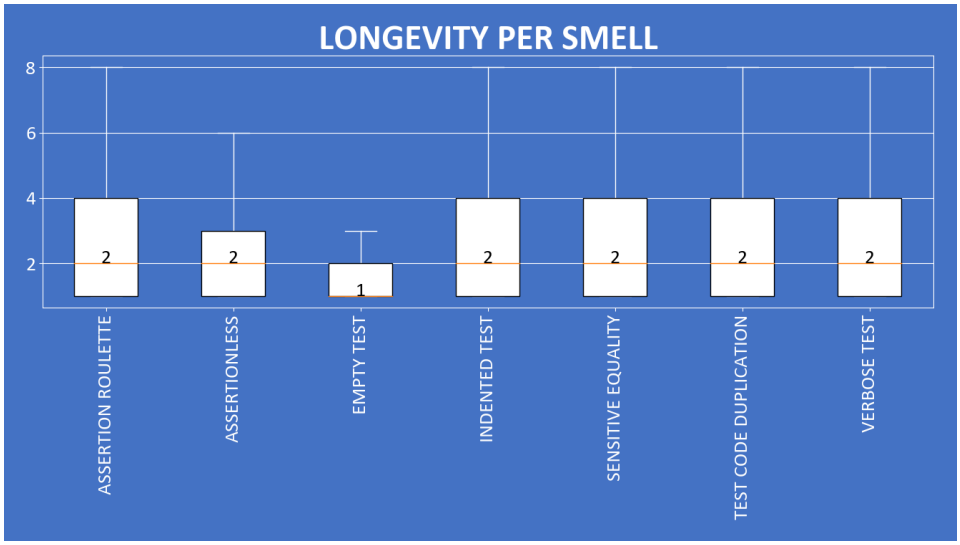


Figure 5.9: Longevity per Smell

CHAPTER 6

Threats to Validity

Conclusion Validity

Conclusion validity concerns itself with issues affecting the drawing of a correct conclusion. A threat to the validity of our conclusions is the manual validation of our test smell detection approach, rather than validating with an established tool. Unfortunately, this was not possible due to the scale of the study and the current state of the investigated tools. However, to minimize this we relied in established test smell detection strategies and implemented these strategies with great care.

Another threat to this validity is the requirement of rule-based decisions for the inclusion or exclusion of software projects due to the scale of our research. Irrelevant projects may have been included in our investigation, as well as relevant projects excluded from our investigation. However, the design of the decision rules for inclusion or exclusion were carefully planned and tested for minimizing this threat.

Construct Validity

Construct validity concerns itself with generalizing the observations to the theory behind the experiment. A threat to the construct validity concerns our conclusions in the *evolution* RQ3. When investigating the number of fixed test smells in our gathered dataset, we conclude that if these smells were detected until the last available snapshot, they were not fixed. Although, these smells

can still be removed in the future outside from our observable data. This may affect the conclusion in the *evolution* RQ3 regarding the number of fixed test smells and their longevity. However, given the size of our investigation we consider this threat to be minimal.

External Validity

External validity concerns itself with the extent in which the results can be generalized to industrial practice. A threat to this validity concerns the nature of the dataset provided by Boa. Although it is one of the largest publicly available software repositories, the available projects are restricted to open source projects. This adds a threat to the generalization of our findings to the industrial practice.

Another threat to this validity concerns our investigation of only projects including JUnit test classes. The test smells are not specific to a xUnit framework, rather they are generalized to all frameworks. Therefore, by including only JUnit projects, we exclude other sources where test smells may be present. However, since the goal of this thesis is to investigate the ubiquity of test smells and Java is currently one of the most popular programming languages, we have a reasonable degree of confidence in the generalization our conclusions.

CHAPTER 7

Conclusions

In this thesis, we conducted an empirical study on the existence of test smells and their evolution utilizing the Boa domain-specific language and infrastructure. To achieve the latter, we proposed to answer the three RQs presented in Section 1.3. In this chapter, we briefly discuss our methodology and our findings for each of the RQs.

RQ1 *detection* — Is it possible to detect test smells on a large scale?

To the best of our knowledge, the largest empirical study available on test smells was conducted by Tufano et al. on 152 software projects [TPB+16]. However, their findings are constrained by the lack of diversity in ecosystems where the projects are being developed. The dataset provided by Boa is not constrained by any ecosystem, as the projects developed in GitHub come from many different sources and development teams. Moreover, none of the tools utilized for large-scale empirical studies are publicly available. Therefore, by detecting test smells through the Boa language and infrastructure, we can conduct the largest empirical study with the largest software projects dataset available. For answering this *detection* RQ, we developed 7 test smell detection scripts as proof-of-concept. Namely, *Assertion Roulette*, *Assertionless*, *Empty Test*, *Indented Test*, *Sensitive Equality*, *Test Code Duplication*, and *Verbose Test*. We selected these smells for their relative ease of implementation using the Boa language, as unfortunately, we could only achieve the detection of

statically detectable test smells. All in all, we have showed that it is possible to detect test smells at a large scale using Boa.

RQ2 *ubiquity* — What are the most common test smells?

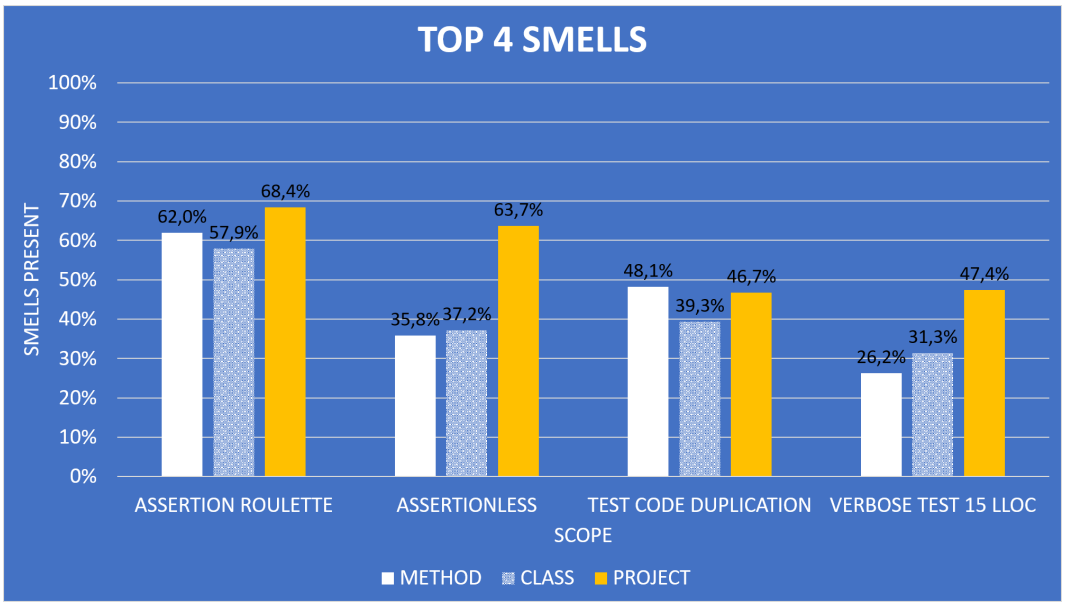


Figure 7.1: Test Smells Detection Percentage by Scope

For answering this *ubiquity* RQ, we utilized the detection scripts we developed in *detection* RQ1. Out of the 282.577 software projects investigated, only $16,26\%$ (45.947) presented test smells. However, a strong correlation with p-value $< 0,05$ between the project size and number of test smells detected was found. Projects presenting test smells are larger in size compared to those who did not. This result suggests that test smells are relative to the project size, i.e., as the project size and complexity increases, so does the number of test smells.

Next, we investigated how the detected test smells were distributed throughout the projects. Out of the 45.947 projects with test smells, *Assertion Roulette* was present in $68,43\%$ of the projects, followed by *Assertionless* $(63,68\%)$, *Verbose Test* $(47,40\%)$ and *Test Code Duplication* $(46,66\%)$, as shown in Figure 7.1. At this scope the most common smell detected was *Assertion Roulette*, suggesting that regardless of team or project this smell is likely to be present.

At the file or class level, we inspected a total of 574.573 unique JUnit classes. $57,92\%$ of the inspected JUnit classes present the *Assertion Roulette* smell, followed by *Test Code Duplication* $(39,35\%)$, *Assertionless* $(37,20\%)$,

and *Verbose Test* (31, 32%). Again, at this scope, the most common smell detected was *Assertion Roulette*, suggesting that this smell is the most commonly introduced by developers in test code.

Lastly, at the method scope, we inspected a total of 2.748.366 unique test methods. The most common smell at this scope, was *Assertion Roulette* with 50, 15%, followed by *Test Code Duplication* (38, 97%), *Assertionless* (28, 98%), and *Verbose Test* (21, 22%). This suggests that the smell most likely to be introduced multiple times is the *Assertion Roulette*.

Overall, the most common smell in all three inspected scopes was the *Assertion Roulette* smell. This result is in line with the previously performed empirical studies, where the *Assertion Roulette* smell was reported as the most common test smell. However, the diffusion of the *Assertionless* smell is also worth noting, as this smell indicates a test that is not testing any functionality. This suggests that tests are being introduced in software projects without an explicit purpose.

RQ3 *evolution* — How do the amount test smells evolve during software development?

Our last *evolution* RQ concerns itself with the evolution of the investigated test smells. We investigated *when* test smells are being introduced. We achieved this by investigating the first moment where the test method was detected and comparing it with the moment where the test smell was detected on that same test method. Our findings show 99, 17% of the smells being detected where introduced when the test method was created. There was no significant different found in this trend between the investigated smells. Our findings therefore suggest that test smells are the result of initial bad design or lack of bad practices, rather than the result of careless maintenance. This suggests that efforts in preventing test smells, such as code checks for smells at commit time, have a worthwhile potential for reducing the diffusion of test smells.

Next, we investigated whether test smells are fixed during their lifespan. Our findings show that 99, 58% of the detected smells, were detected throughout the entire timespan of the dataset, suggesting that they were not fixed. The reason behind this may be lack of maintenance for test code, fear of introducing faults, or lack of knowledge of the existence of test smells. Moreover, there was a slight difference in the number of fixes between the test smells. *Empty Test* was the most fixed smell with 7, 62% of the detected instances being fixed at some point.

Lastly, we investigated the *longevity* of test smells, i.e., how many commits until the test smell was fixed. Our findings show that most test smells are fixed within 2 commits of their existence, whereas *Empty Test* is generally fixed within the next commit. Despite this, most test smells remain unfixed throughout their existence.

In conclusion, our initial findings suggest test smells are not widely common. However, as a software application grows in complexity, the most likely it contains a test smell. Our investigation included 7 selected test smells, of which *Assertion Roulette* was the most common. However, the diffusion of the *Test Code Duplication* and *Assertionless* smells is also worth noting. Moreover, our findings suggest that investing more resources in preventing test smells might be more effective, as most are introduced when designing the test cases. Measures such as test code checks for smells at commit time or similar can help reduce the presence of test smells and raise awareness at the same time. Lastly, we hope the dataset gathered during our investigation is of use to the research community for further research.

As for future work, extending the list of detectable test smells using the Boa infrastructure can help further understand the prevalence of test smells and their evolution. Piling up empirical evidence of their existence may help raise the awareness both in practitioners and the academia of their existence and lay the groundwork for new best practices regarding test code. Moreover, mocking frameworks have also changed the way unit testing is being performed, investigating how these affect test smells is also important for understanding the way test code is evolving.

Bibliography

[All12] Eric Allman. Managing technical debt. *Communications of the ACM*, 55(5):50–55, 2012.

[BBVB+01] Kent Beck, Mike Beedle, Arie Van Bennekum, Alistair Cockburn, Ward Cunningham, Martin Fowler, James Grenning, Jim Highsmith, Andrew Hunt, Ron Jeffries, et al. Manifesto for agile software development. *Available at https://www.agilealliance.org/*, 2001.

[Bec99] Kent Beck. Embracing change with extreme programming. *Computer*, 32(10):70–77, oct 1999.

[Bec03] Kent Beck. *Test-driven development: by example*. Addison-Wesley Professional, 2003.

[BOL14] Sushil Bajracharya, Joel Ossher, and Cristina Lopes. Sourcerer: An infrastructure for large-scale collection and analysis of open-source code. *Science of Computer Programming*, 79:241 – 259, 2014. Experimental Software and Toolkits (EST 4): A special issue of the Workshop on Academic Software Development Tools and Techniques (WASDeTT-3 2010).

[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.

[BVR08] Manuel Breugelmans and Bart Van Rompaey. Testq: Exploring structural and maintenance characteristics of unit test suites. In *Proceedings of the 1st International Workshop on Advanced Software Development Tools and Techniques (WASDeTT 2008)*, 2008.

[Cun92] Ward Cunningham. The wycash portfolio management system. *SIGPLAN OOPS Mess.*, 4(2):29–30, December 1992.

[DDN02] Serge Demeyer, Stéphane Ducasse, and Oscar Nierstrasz. *Object-oriented reengineering patterns*. Elsevier, 2002.

[DNRN13] Robert Dyer, Hoan Anh Nguyen, Hridesh Rajan, and Tien N Nguyen. Boa: A language and infrastructure for analyzing ultra-large-scale software repositories. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 422–431. IEEE Press, 2013.

[FA11] Gordon Fraser and Andrea Arcuri. Evosuite: Automatic test suite generation for object-oriented software. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, pages 416–419. ACM, 2011.

[FB99] Martin Fowler and Kent Beck. *Refactoring: improving the design of existing code*. Addison-Wesley Professional, 1999.

[FL15] Zheng Felix Fang and Patrick Lam. Identifying test refactoring candidates with assertion fingerprints. In *Proceedings of the Principles and Practices of Programming on the Java Platform*, pages 125–137. ACM, 2015.

[Fow97] Martin Fowler. Refactoring: Improving the design of existing code. In *11th European Conference. Jyväskylä, Finland*, 1997.

[GF07] Eduardo Martins Guerra and Clovis Torres Fernandes. Refactoring test code safely. In *International Conference on Software Engineering Advances (ICSEA 2007)*, pages 44–. IEEE Computer Society, 2007.

[GvDS13] Michaela Greiler, Arie van Deursen, and Margaret-Anne Storey. Automated detection of test fixture strategies and smells. In *Proceedings of the 6th IEEE International Conference on Software Testing, Verification and Validation (ICST 2013)*, pages 322–331. IEEE Computer Society, 2013.

[HK07] Dorota Huizinga and Adam Kolawa. *Automated defect prevention: best practices in software management*. John Wiley & Sons, 2007.

[KCK11] Miryung Kim, Dongxiang Cai, and Sunghun Kim. An empirical investigation into the role of api-level refactorings during software evolution. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 151–160, New York, NY, USA, 2011. ACM.

[KNO12] Philippe Kruchten, Robert L Nord, and Ipek Ozkaya. Technical debt: From metaphor to theory and practice. *IEEE Software*, 29(6):18–21, 2012.

[KZN14] Miryung Kim, Thomas Zimmermann, and Nachiappan Nagappan. An empirical study of refactoring challenges and benefits at microsoft. *IEEE Transactions on Software Engineering*, 40(7):633–649, 2014.

[LM07] Filippo Lanubile and Teresa Mallardo. Inspecting automated test code: A preliminary study. In *International Conference on Extreme Programming and Agile Processes in Software Engineering*, pages 115–122. Springer, 2007.

[Mes07] Gerard Meszaros. *xUnit test patterns: Refactoring test code*. Pearson Education, 2007.

[MF16] Delin Mathew and Konrad Foegen. An analysis of information needs to detect test smells. *Full-scale Software Engineering/Current Trends in Release Engineering*, page 19, 2016.

[MSA03] Gerard Meszaros, Shaun M. Smith, and Jennitta Andrea. The test automation manifesto. In *Conference on Extreme Programming and Agile Methods*, pages 73–81. Springer, 2003.

[PDNP+16] Fabio Palomba, Dario Di Nucci, Annibale Panichella, Rocco Oliveto, and Andrea De Lucia. On the diffusion of test smells in automatically generated test code: An empirical study. In *Proceedings of the 9th International Workshop on Search-Based Software Testing*, SBST '16, pages 5–14, New York, NY, USA, 2016. ACM.

[QBO+11] Abdallah Qusef, Gabriele Bavota, Rocco Oliveto, Andrea De Lucia, and David Binkley. Scotch: Test-to-code traceability using slicing and conceptual coupling. In *Software Maintenance*

(ICSM), 2011 27th IEEE International Conference on, pages 63–72. IEEE, 2011.

[RGD07] Stefan Reichhart, Tudor Gîrba, and Stéphane Ducasse. Rule-based assessment of test quality. *Journal of Object Technology*, 6(9):231–251, 2007.

[RMP16] Rudolf Ramler, Michael Moser, and Josef Pichler. Automated static analysis of unit test code. In *Software Analysis, Evolution, and Reengineering (SANER), 2016 IEEE 23rd International Conference on*, volume 2, pages 25–28. IEEE, 2016.

[SAS16] Abdus Satter, Amit Seal Ami, and Kazi Sakib. A static code search technique to identify dead fields by analyzing usage of setup fields and field dependency in test code. In *Proceedings of the Third International Workshop on Concept Discovery in Unstructured Data (CDUD 2016)*, pages 60–71, 2016.

[SF14] Markus Scheidgen and Joachim Fischer. Model-based mining of source code repositories. In Daniel Amyot, Pau Fonseca i Casas, and Gunter Mussbacher, editors, *System Analysis and Modeling: Models and Reusability*, pages 239–254, Cham, 2014. Springer International Publishing.

[SYA+13] Dag IK Sjøberg, Aiko Yamashita, Bente CD Anda, Audris Mockus, and Tore Dybå. Quantifying the effect of code smells on maintenance effort. *IEEE Transactions on Software Engineering*, 39(8):1144–1156, 2013.

[TCM16] Amjed Tahir, Steve Counsell, and Stephen G MacDonell. An empirical study into the relationship between class features and test smells. In *Proceedings of the 23rd Asia-Pacific Software Engineering Conference (APSEC 2016)*, pages 137–144. IEEE, IEEE Computer Society, 2016.

[TPB+16] Michele Tufano, Fabio Palomba, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, Andrea De Lucia, and Denys Poshyvanyk. An empirical investigation into the nature of test smells. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, pages 4–15. ACM, 2016.

[vBMD17] Brent van Bladel, Alessandro Murgia, and Serge Demeyer. An empirical study of clone density evolution and developer cloning tendency. In *2017 IEEE 24th International Conference*

on Software Analysis, Evolution and Reengineering (SANER), pages 551–552, Feb 2017.

[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 (XP 2001)*, pages 92–95, 2001.

[VRDBD06] Bart Van Rompaey, Bart Du Bois, and Serge Demeyer. Characterizing the relative significance of a test smell. In *Proceedings of the 22nd IEEE International Conference on Software Maintenance*, pages 391–400. IEEE Computer Society, 2006.

[VRDBDR07] Bart Van Rompaey, Bart Du Bois, Serge Demeyer, and Matthias Rieger. On the detection of test smells: A metrics-based approach for general fixture and eager test. *IEEE Transactions on Software Engineering*, 33(12):800–817, Dec 2007.

[YM12] Aiko Yamashita and Leon Moonen. Do code smells reflect important maintainability aspects? In *Software Maintenance (ICSM), 2012 28th IEEE International Conference on*, pages 306–315. IEEE, 2012.

[YM13] Aiko Yamashita and Leon Moonen. Exploring the impact of inter-smell relations on software maintainability: An empirical study. In *Software Engineering (ICSE), 2013 35th International Conference on*, pages 682–691. IEEE, 2013.

Appendices

APPENDIX A

Test Smell Detection Scripts

A.1 Assertion Roulette

```
==========================
#
#Copyright (C) 2018 Andres Carrasco
#
#  This program is free software: you can redistribute it and/or
#  modify it under the terms of the GNU General Public License
#  as published by the Free Software Foundation, either version 3
#  of the License, or (at your option) any later version.
#
#  This program is distributed in the hope that it will be useful,
#  but WITHOUT ANY WARRANTY; without even the implied warranty of
#  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
#  GNU General Public License for more details.
#
#You should have received a copy of the GNU General Public License
#along with this program. If not, see http://www.gnu.org/licenses/
#
==========================
p: Project = input;

a_run: output top(1) of string weight int;
a_run << ("[" + string(now()) + "] Assertion Roulette.") weight 1;

currentProjectId : string = "";
currentProject : string = "";
currentRevision : string = "";
currentFile : string = "";
revisionTimestamp : time = now();
currentAst : ASTRoot;
num_of_messageless := 0;
num_of_assertions := 0;
assertion_roulette_project: bool = false;
total_num_of_projects: output sum of int;
num_of_projects: output sum of int;
num_of_assertion_roulette_projects: output sum of int;

z1_first_occurrence : output minimum(1)[string] of string weight int;
z2_last_occurrence : output top(1)[string] of string weight int;
z3_longevity : output sum[string] of int;
z4_first_change : output minimum(1)[string] of string weight int;
z5_last_change : output top(1)[string] of string weight int;

hasAssertionMessage := function(e: Expression) : bool {
  if(match((`assert.*Equals`), e.method) && len(e.method_args) == 3){
    return true;
```

```
  } else if(match((`assertFalse`), e.method) && len(e.method_args) == 2){
    return true;
  } else if(match((`assertTrue`), e.method) && len(e.method_args) == 2){
    return true;
  } else if(match((`assert.*Null`), e.method) && len(e.method_args) == 2){
    return true;
  } else if(match((`assert.*Same`), e.method) && len(e.method_args) == 3){
    return true;
  } else if(match((`assertThat`), e.method) && len(e.method_args) == 3){
    return true;
  } else if(match((`fail`), e.method) && len(e.method_args) == 1){
    return true;
  }
  return false;
};

time_to_int := function(d: time) : int {
  s_year := (yearof(d) - 1970) * 31557600;
  s_days := (dayofyear(d) * 86400);
  s_hours := (hourof(d) * 3600);
  s_min := (minuteof(d) * 60);
  s := secondof(d);
  return s_year + s_days + s_hours + s;
};

visit(p, visitor {
  before n: Project -> {
    currentProjectId = n.id;
    currentProject = n.project_url;
    total_num_of_projects << 1;
    assertion_roulette_project = false;
    ifall (i: int; ! match(`^java$`,
    ↪  lowercase(n.programming_languages[i])))
      stop;
  }

  before node : Revision -> {
    currentRevision = node.id;
    revisionTimestamp = node.commit_date;
  }

  before node: ChangedFile -> {
    currentAst = getast(node);
    currentFile = string(node.name);
    exists (i: int; match((`org.powermock`), currentAst.imports[i])
      || match((`org.easymock`), currentAst.imports[i])
      || match((`org.mockito`), currentAst.imports[i])
      || match((`org.jmockit`), currentAst.imports[i])
    ) {
      stop;
```

```
95      }
96      ifall (i: int; !match((`org.junit.Test`), currentAst.imports[i]))
97        stop;
98    }
99
100   before node: Method -> {
101     ifall (i: int; ! match((`Test`), node.modifiers[i].annotation_name))
102       stop;
103   }
104
105   before node: Statement -> {
106     if (def(node.expression.method) && (match((`assert`),
      ↪  node.expression.method) || match((`fail`),
      ↪  node.expression.method))) {
107       num_of_assertions++;
108       if(!hasAssertionMessage(node.expression)){
109         num_of_messageless++;
110       }
111     }
112   }
113
114   after node: Method -> {
115     k := currentProjectId + ";" + currentFile + ";" + node.name;
116     if (num_of_messageless > 0 && num_of_assertions > 1) {
117       z1_first_occurrence[k] << currentRevision + ";" +
        ↪  string(time_to_int(revisionTimestamp)) weight 1;
118       z3_longevity[k] << 1;
119       z2_last_occurrence[k] << currentRevision + ";" +
        ↪  string(time_to_int(revisionTimestamp)) weight 1;
120       assertion_roulette_project = true;
121     }
122     z4_first_change[k] << currentRevision + ";" +
      ↪  string(time_to_int(revisionTimestamp)) weight 1;
123     z5_last_change[k] << currentRevision + ";" +
      ↪  string(time_to_int(revisionTimestamp)) weight 1;
124     num_of_messageless = 0;
125     num_of_assertions = 0;
126   }
127
128   after node: Project -> {
129     num_of_projects << 1;
130     if (assertion_roulette_project) {
131       num_of_assertion_roulette_projects << 1;
132     }
133   }
134 });
```

A.2 Assertionless

```
# Assertionless Detection Script
# ==========================
#
# Copyright (C) 2018 Andres Carrasco
#
#   This program is free software: you can redistribute it and/or modify
#   it under the terms of the GNU General Public License as published by
#   the Free Software Foundation, either version 3 of the License, or
#   (at your option) any later version.
#
#   This program is distributed in the hope that it will be useful,
#   but WITHOUT ANY WARRANTY; without even the implied warranty of
#   MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
#   GNU General Public License for more details.
#
# You should have received a copy of the GNU General Public License
# along with this program.  If not, see <http://www.gnu.org/licenses/>.
#
# ==========================
####################################OPTIONS#########################
# When true, it includes empty test methods in the detection.
include_empty_tests : bool = false;
####################################################################
p: Project = input;

a_run: output top(1) of string weight int;
a_run << ("[" + string(now()) + "] Assertionless. " +
", IncludeEmptyTest: " + string(include_empty_tests))  weight 1;

currentProjectId : string = "";
currentProject : string = "";
currentRevision : string = "";
currentFile : string = "";
revisionTimestamp : time = now();
currentAst : ASTRoot;
num_of_statements : int = 0;
has_assertion : bool = false;
total_num_of_projects: output sum of int;
num_of_projects: output sum of int;
num_of_assertionless_projects: output sum of int;
assertionless_project : bool = false;

z1_first_occurrence : output minimum(1)[string] of string weight int;
z2_last_occurrence : output top(1)[string] of string weight int;
z3_longevity : output sum[string] of int;
```

```
z4_first_change : output minimum(1)[string] of string weight int;
z5_last_change : output top(1)[string] of string weight int;

hasExpectedAssert := function(m : Method): bool {
  exists(i: int; match((`Test`), m.modifiers[i].annotation_name)){
    exists(j : int; match(`expected`, m.modifiers[i].annotation_members[j])
      || match(`timeout`, m.modifiers[i].annotation_members[j])
    ){
      return true;
    }
  }
  return false;
};

time_to_int := function(d: time) : int {
  s_year := (yearof(d) - 1970) * 31557600;
  s_days := (dayofyear(d) * 86400);
  s_hours := (hourof(d) * 3600);
  s_min := (minuteof(d) * 60);
  s := secondof(d);
  return s_year + s_days + s_hours + s;
};

visit(p, visitor {
  before n : Project -> {
    currentProjectId = n.id;
    currentProject = n.project_url;
    total_num_of_projects << 1;
    assertionless_project = false;
    ifall (i: int; !match(`^java$`, lowercase(n.programming_languages[i])))
      stop;
  }

  before node : Revision -> {
    currentRevision = node.id;
    revisionTimestamp = node.commit_date;
  }

  before node: ChangedFile -> {
    currentAst = getast(node);
    currentFile = string(node.name);
    exists (i: int; match((`org.powermock`), currentAst.imports[i])
      || match((`org.easymock`), currentAst.imports[i])
      || match((`org.mockito`), currentAst.imports[i])
      || match((`org.jmockit`), currentAst.imports[i])
    ) {
      stop;
    }
    ifall (i: int; !match((`org.junit.Test`), currentAst.imports[i]))
      stop;
```

```
96    }
97
98    before node : Method -> {
99      num_of_statements = 0;
100     ifall (i: int; !match((`Test`), node.modifiers[i].annotation_name))
101       stop;
102   }
103
104   before node : Statement -> {
105     if(def(node.expression.method) && ( match((`assert`),
      ↪  node.expression.method) || match((`fail`),
      ↪  node.expression.method))){
106       has_assertion = true;
107     }
108     if(def(node.kind) && node.kind == StatementKind.EXPRESSION &&
      ↪  def(node.expression) && def(node.expression.kind) &&
      ↪  node.expression.kind == ExpressionKind.LITERAL){
109       # Comments do not count!
110     } else {
111       num_of_statements++;
112     }
113   }
114
115   after node : Method -> {
116     k := currentProjectId + ";" + currentFile + ";" + node.name;
117     empty : bool = num_of_statements == 1; # 1 because there is always at
      ↪  least 1 Block Statement!
118     include : bool = !empty || include_empty_tests;
119     if(!has_assertion && !hasExpectedAssert(node) && include){
120       z1_first_occurrence[k] << currentRevision + ";" +
        ↪  string(time_to_int(revisionTimestamp)) weight 1;
121       z3_longevity[k] << 1;
122       z2_last_occurrence[k] << currentRevision + ";" +
        ↪  string(time_to_int(revisionTimestamp)) weight 1;
123       assertionless_project = true;
124     }
125     z4_first_change[k] << currentRevision + ";" +
      ↪  string(time_to_int(revisionTimestamp)) weight 1;
126     z5_last_change[k] << currentRevision + ";" +
      ↪  string(time_to_int(revisionTimestamp)) weight 1;
127     has_assertion = false;
128   }
129
130   after node : Project -> {
131     num_of_projects << 1;
132     if(assertionless_project){
133       num_of_assertionless_projects << 1;
134     }
135   }
136 });
```

A.3 Empty Test

```
# Empty Test Detection Script
# ==========================
#
# Copyright (C) 2018 Andres Carrasco
#
#   This program is free software: you can redistribute it and/or modify
#   it under the terms of the GNU General Public License as published by
#   the Free Software Foundation, either version 3 of the License, or
#   (at your option) any later version.
#
#   This program is distributed in the hope that it will be useful,
#   but WITHOUT ANY WARRANTY; without even the implied warranty of
#   MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
#   GNU General Public License for more details.
#
# You should have received a copy of the GNU General Public License
# along with this program.  If not, see <http://www.gnu.org/licenses/>.
#
# ==========================
p: Project = input;

a_run: output top(1) of string weight int;
a_run << ("[" + string(now()) + "] Empty Test. ")  weight 1;

currentProjectId : string = "";
currentProject : string = "";
currentRevision : string = "";
currentFile : string = "";
revisionTimestamp : time = now();
currentAst : ASTRoot;
num_of_statements : int = 0;
total_num_of_projects: output sum of int;
num_of_projects: output sum of int;
num_of_empty_test_projects: output sum of int;
empty_test_project : bool = false;

z1_first_occurrence : output minimum(1)[string] of string weight int;
z2_last_occurrence : output top(1)[string] of string weight int;
z3_longevity : output sum[string] of int;
z4_first_change : output minimum(1)[string] of string weight int;
z5_last_change : output top(1)[string] of string weight int;

time_to_int := function(d: time) : int {
  s_year := (yearof(d) - 1970) * 31557600;
  s_days := (dayofyear(d) * 86400);
```

```
  s_hours := (hourof(d) * 3600);
  s_min := (minuteof(d) * 60);
  s := secondof(d);
  return s_year + s_days + s_hours + s;
};

visit(p, visitor {
  before n : Project -> {
    currentProjectId = n.id;
    currentProject = n.project_url;
    total_num_of_projects << 1;
    empty_test_project = false;
    ifall (i: int; !match(`^java$`, lowercase(n.programming_languages[i])))
      stop;
  }

  before node : Revision -> {
    currentRevision = node.id;
    revisionTimestamp = node.commit_date;
  }

  before node: ChangedFile -> {
    currentAst = getast(node);
    currentFile = string(node.name);
    exists (i: int; match((`org.powermock`), currentAst.imports[i])
      || match((`org.easymock`), currentAst.imports[i])
      || match((`org.mockito`), currentAst.imports[i])
      || match((`org.jmockit`), currentAst.imports[i])
    ) {
      stop;
    }
    ifall (i: int; !match((`org.junit.Test`), currentAst.imports[i]))
      stop;
  }

  before node : Method -> {
    num_of_statements = 0;
    ifall (i: int; !match((`Test`), node.modifiers[i].annotation_name))
      stop;
  }

  before node : Statement -> {
    if(def(node.kind) && node.kind == StatementKind.EXPRESSION &&
    ↪  def(node.expression) && def(node.expression.kind) &&
    ↪  node.expression.kind == ExpressionKind.LITERAL){
      # Comments do not count!
    } else {
      num_of_statements++;
    }
  }
```

```
94
95    after node : Method -> {
96      k := currentProjectId + ";" + currentFile + ";" + node.name;
97      if(num_of_statements == 1) { # 1 because there is always at least 1
        ↪  Block Statement!
98        z1_first_occurrence[k] << currentRevision + ";" +
          ↪  string(time_to_int(revisionTimestamp)) weight 1;
99        z3_longevity[k] << 1;
100       z2_last_occurrence[k] << currentRevision + ";" +
          ↪  string(time_to_int(revisionTimestamp)) weight 1;
101       empty_test_project = true;
102     }
103     z4_first_change[k] << currentRevision + ";" +
        ↪  string(time_to_int(revisionTimestamp)) weight 1;
104     z5_last_change[k] << currentRevision + ";" +
        ↪  string(time_to_int(revisionTimestamp)) weight 1;
105     num_of_statements = 0;
106   }
107
108   after node : Project -> {
109     num_of_projects << 1;
110     if(empty_test_project){
111       num_of_empty_test_projects << 1;
112     }
113   }
114 });
```

A.4 Indented Test

```
# Indented Test Detection Script
# ==========================
#
# Copyright (C) 2018 Andres Carrasco
#
#   This program is free software: you can redistribute it and/or modify
#   it under the terms of the GNU General Public License as published by
#   the Free Software Foundation, either version 3 of the License, or
#   (at your option) any later version.
#
#   This program is distributed in the hope that it will be useful,
#   but WITHOUT ANY WARRANTY; without even the implied warranty of
#   MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
#   GNU General Public License for more details.
#
# You should have received a copy of the GNU General Public License
# along with this program.  If not, see <http://www.gnu.org/licenses/>.
#
# ==========================
p: Project = input;

a_run: output top(1) of string weight int;
a_run << ("[" + string(now()) + "] Indented Test. ")  weight 1;

currentProjectId : string = "";
currentProject : string = "";
currentRevision : string = "";
currentFile : string = "";
revisionTimestamp : time = now();
currentAst : ASTRoot;
has_indents : bool = false;
total_num_of_projects: output sum of int;
num_of_projects: output sum of int;
num_of_indented_projects: output sum of int;
indented_project : bool = false;

z1_first_occurrence : output minimum(1)[string] of string weight int;
z2_last_occurrence : output top(1)[string] of string weight int;
z3_longevity : output sum[string] of int;
z4_first_change : output minimum(1)[string] of string weight int;
z5_last_change : output top(1)[string] of string weight int;

time_to_int := function(d: time) : int {
  s_year := (yearof(d) - 1970) * 31557600;
  s_days := (dayofyear(d) * 86400);
```

```
  s_hours := (hourof(d) * 3600);
  s_min := (minuteof(d) * 60);
  s := secondof(d);
  return s_year + s_days + s_hours + s;
};

isLoopOrCond := function(m : StatementKind): bool {
  if(m == StatementKind.DO ||
    m == StatementKind.FOR ||
    m == StatementKind.WHILE ||
    m == StatementKind.IF ||
    m == StatementKind.SWITCH
  ){
    return true;
  }
  return false;
};

visit(p, visitor {
  before n : Project -> {
    total_num_of_projects << 1;
    currentProjectId = n.id;
    currentProject = n.project_url;
    indented_project = false;
    ifall (i: int; !match(`^java$`, lowercase(n.programming_languages[i])))
      stop;
  }

  before node : Revision -> {
    currentRevision = node.id;
    revisionTimestamp = node.commit_date;
  }

  before node: ChangedFile -> {
    currentAst = getast(node);
    currentFile = string(node.name);
    exists (i: int; match((`org.powermock`), currentAst.imports[i])
      || match((`org.easymock`), currentAst.imports[i])
      || match((`org.mockito`), currentAst.imports[i])
      || match((`org.jmockit`), currentAst.imports[i])
    ) {
      stop;
    }
    ifall (i: int; !match((`org.junit.Test`), currentAst.imports[i]))
      stop;
  }

  before node : Method -> {
    ifall (i: int; !match((`Test`), node.modifiers[i].annotation_name))
      stop;
```

```
96    }
97
98    before node : Statement -> {
99      if(def(node.kind) && isLoopOrCond(node.kind)){
100       has_indents = true;
101     }
102   }
103
104   after node : Method -> {
105     k := currentProjectId + ";" + currentFile + ";" + node.name;
106     if(has_indents){
107       z1_first_occurrence[k] << currentRevision + ";" +
          ↪  string(time_to_int(revisionTimestamp)) weight 1;
108       z3_longevity[k] << 1;
109       z2_last_occurrence[k] << currentRevision + ";" +
          ↪  string(time_to_int(revisionTimestamp)) weight 1;
110       indented_project = true;
111     }
112     z4_first_change[k] << currentRevision + ";" +
        ↪  string(time_to_int(revisionTimestamp)) weight 1;
113     z5_last_change[k] << currentRevision + ";" +
        ↪  string(time_to_int(revisionTimestamp)) weight 1;
114     has_indents = false;
115   }
116
117   after node : Project -> {
118     num_of_projects << 1;
119     if(indented_project){
120       num_of_indented_projects << 1;
121     }
122   }
123 });
```

A.5 Sensitive Equality

```
# Sensitive Equality Detection Script
# ===========================
#
# Copyright (C) 2018 Andres Carrasco
#
#   This program is free software: you can redistribute it and/or modify
#   it under the terms of the GNU General Public License as published by
#   the Free Software Foundation, either version 3 of the License, or
#   (at your option) any later version.
#
#   This program is distributed in the hope that it will be useful,
#   but WITHOUT ANY WARRANTY; without even the implied warranty of
#   MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
#   GNU General Public License for more details.
#
# You should have received a copy of the GNU General Public License
# along with this program.  If not, see <http://www.gnu.org/licenses/>.
#
# ===========================
p: Project = input;

a_run: output top(1) of string weight int;
a_run << ("[" + string(now()) + "] Sensitive Equality. ")  weight 1;

currentProjectId : string = "";
currentProject : string = "";
currentRevision : string = "";
currentFile : string = "";
revisionTimestamp : time = now();
currentAst : ASTRoot;
has_sensitive_equality : bool = false;
total_num_of_projects: output sum of int;
num_of_projects: output sum of int;
num_of_sensitive_equality_projects: output sum of int;
sensitive_equality_project : bool = false;

z1_first_occurrence : output minimum(1)[string] of string weight int;
z2_last_occurrence : output top(1)[string] of string weight int;
z3_longevity : output sum[string] of int;
z4_first_change : output minimum(1)[string] of string weight int;
z5_last_change : output top(1)[string] of string weight int;

time_to_int := function(d: time) : int {
  s_year := (yearof(d) - 1970) * 31557600;
  s_days := (dayofyear(d) * 86400);
```

```
  s_hours := (hourof(d) * 3600);
  s_min := (minuteof(d) * 60);
  s := secondof(d);
  return s_year + s_days + s_hours + s;
};

visit(p, visitor {
  before n : Project -> {
    total_num_of_projects << 1;
    currentProjectId = n.id;
    currentProject = n.project_url;
    sensitive_equality_project = false;
    ifall (i: int; !match(`^java$`, lowercase(n.programming_languages[i])))
      stop;
  }

  before node : Revision -> {
    currentRevision = node.id;
    revisionTimestamp = node.commit_date;
  }

  before node: ChangedFile -> {
    currentAst = getast(node);
    currentFile = string(node.name);
    exists (i: int; match((`org.powermock`), currentAst.imports[i])
      || match((`org.easymock`), currentAst.imports[i])
      || match((`org.mockito`), currentAst.imports[i])
      || match((`org.jmockit`), currentAst.imports[i])
    ) {
      stop;
    }
    ifall (i: int; !match((`org.junit.Test`), currentAst.imports[i]))
      stop;
  }

  before node : Method -> {
    ifall (i: int; !match((`Test`), node.modifiers[i].annotation_name))
      stop;
  }

  before node : Statement -> {
    if(def(node.expression.method) && ( match((`assert`),
    ↪  node.expression.method))){
      exists (i: int; match(`^toString$`,
      ↪  node.expression.method_args[i].method))
        has_sensitive_equality = true;
    }
  }

  after node : Method -> {
```

```
94      k := currentProjectId + ";" + currentFile + ";" + node.name;
95      if(has_sensitive_equality){
96        z1_first_occurrence[k] << currentRevision + ";" +
          ↪  string(time_to_int(revisionTimestamp)) weight 1;
97        z3_longevity[k] << 1;
98        z2_last_occurrence[k] << currentRevision + ";" +
          ↪  string(time_to_int(revisionTimestamp)) weight 1;
99        sensitive_equality_project = true;
100     }
101     z4_first_change[k] << currentRevision + ";" +
        ↪  string(time_to_int(revisionTimestamp)) weight 1;
102     z5_last_change[k] << currentRevision + ";" +
        ↪  string(time_to_int(revisionTimestamp)) weight 1;
103     has_sensitive_equality = false;
104   }
105
106   after node : Project -> {
107     num_of_projects << 1;
108     if(sensitive_equality_project){
109       num_of_sensitive_equality_projects << 1;
110     }
111   }
112 });
```

A.6 Test Code Duplication

```
# Test Code Duplicates Detection Script
# ==========================
#
# Copyright (C) 2018 Andres Carrasco
#
#   This program is free software: you can redistribute it and/or modify
#   it under the terms of the GNU General Public License as published by
#   the Free Software Foundation, either version 3 of the License, or
#   (at your option) any later version.
#
#   This program is distributed in the hope that it will be useful,
#   but WITHOUT ANY WARRANTY; without even the implied warranty of
#   MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
#   GNU General Public License for more details.
#
# You should have received a copy of the GNU General Public License
# along with this program.  If not, see <http://www.gnu.org/licenses/>.
#
# Based on the 'Clone Detection Algorithm' work of Brent van Bladel,
# distributed with the same license. See below.
# ==========================
# Clone Detection Algorithm
# ==========================
#
# Copyright (C) 2015  Brent van Bladel
#
#   This program is free software: you can redistribute it and/or modify
#   it under the terms of the GNU General Public License as published by
#   the Free Software Foundation, either version 3 of the License, or
#   (at your option) any later version.
#
#   This program is distributed in the hope that it will be useful,
#   but WITHOUT ANY WARRANTY; without even the implied warranty of
#   MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
#   GNU General Public License for more details.
#
# You should have received a copy of the GNU General Public License
# along with this program.  If not, see <http://www.gnu.org/licenses/>.
#
# ==========================
p: Project = input;

a_run: output top(1) of string weight int;
a_run << ("[" + string(now()) + "] Test Code Duplication")  weight 1;

```

```
currentProjectId : string = "";
currentProject : string = "";
currentRevision : string = "";
currentFile : string = "";
currentAst : ASTRoot;
revisionTimestamp : time = now();
total_num_of_projects: output sum of int;
num_of_projects: output sum of int;
num_of_code_duplicate_projects: output sum of int;
code_duplicate_project : bool = false;

z1_first_occurrence : output minimum(1)[string] of string weight int;
z2_last_occurrence : output top(1)[string] of string weight int;
z3_longevity : output sum[string] of int;
z4_first_change : output minimum(1)[string] of string weight int;
z5_last_change : output top(1)[string] of string weight int;

time_to_int := function(d: time) : int {
  s_year := (yearof(d) - 1970) * 31557600;
  s_days := (dayofyear(d) * 86400);
  s_hours := (hourof(d) * 3600);
  s_min := (minuteof(d) * 60);
  s := secondof(d);
  return s_year + s_days + s_hours + s;
};

mapOfFunctions: map[string] of string;
setOfClonedFunctions: set of string;

MIN_CLONE_SIZE := 25;

##
# Returns string representation of the code in a method (AST)
##
SerializeAST := function(ast: Method) : string {
  serializedAST := "";
  visit(ast, visitor {
    before node: Statement -> {
      switch (node.kind) {
        case StatementKind.ASSERT:
          serializedAST = serializedAST + "A";
          break;
        case StatementKind.BLOCK:
          serializedAST = serializedAST + "B";
          break;
        case StatementKind.BREAK:
          serializedAST = serializedAST + "K";
          break;
        case StatementKind.CASE:
          serializedAST = serializedAST + "C";
```

```
96            break;
97          case StatementKind.CATCH:
98            serializedAST = serializedAST + "G";
99            break;
100         case StatementKind.CONTINUE:
101           serializedAST = serializedAST + "N";
102           break;
103         case StatementKind.DO:
104           serializedAST = serializedAST + "D";
105           break;
106         case StatementKind.EMPTY:
107           serializedAST = serializedAST + "E";
108           break;
109         case StatementKind.EXPRESSION:
110           # handled when expression is visited
111           break;
112         case StatementKind.FOR:
113           serializedAST = serializedAST + "F";
114           break;
115         case StatementKind.IF:
116           serializedAST = serializedAST + "I";
117           break;
118         case StatementKind.RETURN:
119           serializedAST = serializedAST + "R";
120           break;
121         case StatementKind.SWITCH:
122           serializedAST = serializedAST + "S";
123           break;
124         case StatementKind.THROW:
125           serializedAST = serializedAST + "H";
126           break;
127         case StatementKind.TRY:
128           serializedAST = serializedAST + "Y";
129           break;
130         case StatementKind.WHILE:
131           serializedAST = serializedAST + "W";
132           break;
133         default:
134           serializedAST = serializedAST + "O";
135           break;
136     }
137   }
138
139   before node: Expression -> {
140     switch (node.kind) {
141       case ExpressionKind.ANNOTATION:
142         serializedAST = serializedAST + "x";
143         break;
144       case ExpressionKind.ARRAYINIT:
145         serializedAST = serializedAST + "y";
```

```
146             break;
147           case ExpressionKind.ARRAYINDEX:
148             serializedAST = serializedAST + "i";
149           case ExpressionKind.ASSIGN:
150             serializedAST = serializedAST + "a";
151             break;
152           case ExpressionKind.ASSIGN_ADD, ExpressionKind.ASSIGN_SUB:
153             serializedAST = serializedAST + "a+";
154             break;
155           case ExpressionKind.ASSIGN_BITAND, ExpressionKind.ASSIGN_BITOR,
              ↪  ExpressionKind.ASSIGN_BITXOR, ExpressionKind.ASSIGN_LSHIFT,
              ↪  ExpressionKind.ASSIGN_RSHIFT,
              ↪  ExpressionKind.ASSIGN_UNSIGNEDRSHIFT:
156             serializedAST = serializedAST + "ab";
157             break;
158           case ExpressionKind.ASSIGN_DIV, ExpressionKind.ASSIGN_MULT:
159             serializedAST = serializedAST + "a*";
160             break;
161           case ExpressionKind.ASSIGN_MOD:
162             serializedAST = serializedAST + "a%";
163             break;
164           case ExpressionKind.BIT_AND, ExpressionKind.BIT_LSHIFT,
              ↪  ExpressionKind.BIT_NOT, ExpressionKind.BIT_OR,
              ↪  ExpressionKind.BIT_RSHIFT, ExpressionKind.BIT_UNSIGNEDRSHIFT,
              ↪  ExpressionKind.BIT_XOR:
165             serializedAST = serializedAST + "b";
166             break;
167           case ExpressionKind.CAST:
168             serializedAST = serializedAST + "c";
169             break;
170           case ExpressionKind.CONDITIONAL, ExpressionKind.NULLCOALESCE:
171             serializedAST = serializedAST + "d";
172             break;
173           case ExpressionKind.EQ, ExpressionKind.NEQ:
174             serializedAST = serializedAST + "=";
175             break;
176           case ExpressionKind.GT, ExpressionKind.GTEQ:
177             serializedAST = serializedAST + ">";
178             break;
179           case ExpressionKind.LT, ExpressionKind.LTEQ:
180             serializedAST = serializedAST + "<";
181             break;
182           case ExpressionKind.LITERAL:
183             serializedAST = serializedAST + "l";
184             break;
185           case ExpressionKind.LOGICAL_AND:
186             serializedAST = serializedAST + "&";
187             break;
188           case ExpressionKind.LOGICAL_NOT:
189             serializedAST = serializedAST + "!";
```

```
            break;
          case ExpressionKind.LOGICAL_OR:
            serializedAST = serializedAST + "|";
            break;
          case ExpressionKind.METHODCALL:
            serializedAST = serializedAST + "m";
            break;
          case ExpressionKind.NEW, ExpressionKind.NEWARRAY:
            serializedAST = serializedAST + "n";
            break;
          case ExpressionKind.OP_ADD, ExpressionKind.OP_INC,
          ↪  ExpressionKind.OP_DEC, ExpressionKind.OP_SUB:
            serializedAST = serializedAST + "+";
            break;
          case ExpressionKind.OP_DIV, ExpressionKind.OP_MULT:
            serializedAST = serializedAST + "*";
            break;
          case ExpressionKind.OP_MOD:
            serializedAST = serializedAST + "%";
            break;
          case ExpressionKind.TYPECOMPARE:
            serializedAST = serializedAST + "t";
            break;
          case ExpressionKind.VARACCESS:
            serializedAST = serializedAST + "v";
            break;
          case ExpressionKind.VARDECL:
            serializedAST = serializedAST + "w";
            break;
          default:
            serializedAST = serializedAST + "o";
            break;
        }
      }
    });
    return serializedAST;
};

##
# Find all function clones given a mapping from function name to string
↪  representation.
# Note: If there are multiple clones between the same 2 functions,
#       it is counted as one larger type-3 clone.
##
FunctionClonesDetection := function(project : map[string] of string) {
    functions := keys(project);
    # for each function i
    foreach(i : int; def(functions[i])){
      current := lookup(project, functions[i], ""); # serialized AST of
      ↪  function i
```

```
237       slidingWindowSize := MIN_CLONE_SIZE; # minimum clone size (on average:
          ↪  5 == 1 LOC)
238       # for each other function j
239       # Note: each combination of functions i and j is only checked once,
          ↪  e.g. (i,j) == (j,i)
240       foreach(j : int; (def(functions[j]) && j > i)){
241         other := lookup(project, functions[j], ""); # serialized AST of
            ↪  function j
242         slidingWindowPos := 0;
243         slidingWindow := "";
244         # move a sliding window over function i
245         while (slidingWindowPos + slidingWindowSize <= len(current)){
246           # get new window
247           slidingWindow = substring(current, slidingWindowPos,
              ↪  slidingWindowPos + slidingWindowSize);
248           # check if window appears in file j
249           foundResult := strfind(slidingWindow, other);
250           # content of sliding window found in file j
251           if (foundResult > -1){
252           add(setOfClonedFunctions, functions[i]);
253           add(setOfClonedFunctions, functions[j]);
254           break;
255           }
256         # move sliding window
257         slidingWindowPos++;
258         }
259       }
260     }
261   };
262
263   ##
264   # Visitor for project management (Project level)
265   ##
266   visit(p, visitor {
267     before n: Project -> {
268       currentProjectId = n.id;
269       currentProject = n.project_url;
270       total_num_of_projects << 1;
271       code_duplicate_project = false;
272       ifall (i: int; !match(`^java$`, lowercase(n.programming_languages[i])))
273         stop;
274     }
275
276     before node : Revision -> {
277       currentRevision = node.id;
278       revisionTimestamp = node.commit_date;
279       clear(mapOfFunctions);
280       clear(setOfClonedFunctions);
281     }
282
```

```
283   before node: ChangedFile -> {
284     currentAst = getast(node);
285     currentFile = string(node.name);
286     exists (i: int; match((`org.powermock`), currentAst.imports[i])
287       || match((`org.easymock`), currentAst.imports[i])
288       || match((`org.mockito`), currentAst.imports[i])
289       || match((`org.jmockit`), currentAst.imports[i])
290     ) {
291       stop;
292     }
293     ifall (i: int; !match((`org.junit.Test`), currentAst.imports[i]))
294       stop;
295   }
296
297   before node: Method -> {
298       isTest := false;
299       foreach (i: int; def(node.modifiers[i])){
300         if (match((`Test`), node.modifiers[i].annotation_name)){
301           isTest = true;
302           break;
303         }
304       }
305       if (isTest){
306       serializedMethod := SerializeAST(node);
307       if (len(serializedMethod) >= MIN_CLONE_SIZE){
308         mapOfFunctions[currentFile + ";" + node.name] = serializedMethod;
309       }
310         z4_first_change[currentProjectId + ";" + currentFile + ";" +
          ↪  node.name] <<  currentRevision + ";" +
          ↪  string(time_to_int(revisionTimestamp)) weight 1;
311         z5_last_change[currentProjectId + ";" + currentFile + ";" +
          ↪  node.name] <<  currentRevision + ";" +
          ↪  string(time_to_int(revisionTimestamp)) weight 1;
312       }
313     }
314
315     after node : Revision -> {
316     FunctionClonesDetection(mapOfFunctions);
317     functions := keys(mapOfFunctions);
318     foreach (i: int; def(functions[i])){
319       k := currentProjectId + ";" + functions[i];
320       if(contains(setOfClonedFunctions, functions[i])){
321         z1_first_occurrence[k] <<  currentRevision + ";" +
          ↪  string(time_to_int(revisionTimestamp)) weight 1;
322         z3_longevity[k] << 1;
323         z2_last_occurrence[k] << currentRevision + ";" +
          ↪  string(time_to_int(revisionTimestamp)) weight 1;
324         code_duplicate_project = true;
325       }
326     }
```

```
327     }
328
329     after node : Project -> {
330       num_of_projects << 1;
331
332       if(code_duplicate_project){
333         num_of_code_duplicate_projects << 1;
334       }
335     }
336   });
```

A.7 Verbose Test

```
# Verbose Test Detection Script
# ==========================
#
# Copyright (C) 2018 Andres Carrasco
#
#   This program is free software: you can redistribute it and/or modify
#   it under the terms of the GNU General Public License as published by
#   the Free Software Foundation, either version 3 of the License, or
#   (at your option) any later version.
#
#   This program is distributed in the hope that it will be useful,
#   but WITHOUT ANY WARRANTY; without even the implied warranty of
#   MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
#   GNU General Public License for more details.
#
# You should have received a copy of the GNU General Public License
# along with this program.  If not, see <http://www.gnu.org/licenses/>.
#
# ==========================
####################################OPTIONS#####################
# SLOC to use for threshold
sloc : float = 6.48;
################################################################
p: Project = input;

a_run: output top(1) of string weight int;
a_run << ("[" + string(now()) + "] Verbose Test. " +
" SLOC: " + string(sloc))  weight 1;

currentProjectId : string = "";
currentProject : string = "";
currentRevision : string = "";
currentFile : string = "";
revisionTimestamp : time = now();
currentAst : ASTRoot;
num_of_statements : int = 0;
total_num_of_projects: output sum of int;
num_of_projects: output sum of int;
num_of_verbose_test_projects: output sum of int;
verbose_test_project : bool = false;

z1_first_occurrence : output minimum(1)[string] of string weight int;
z2_last_occurrence : output top(1)[string] of string weight int;
z3_longevity : output sum[string] of int;
z4_first_change : output minimum(1)[string] of string weight int;
```

```
z5_last_change : output top(1)[string] of string weight int;

time_to_int := function(d: time) : int {
  s_year := (yearof(d) - 1970) * 31557600;
  s_days := (dayofyear(d) * 86400);
  s_hours := (hourof(d) * 3600);
  s_min := (minuteof(d) * 60);
  s := secondof(d);
  return s_year + s_days + s_hours + s;
};

visit(p, visitor {
  before n : Project -> {
    currentProjectId = n.id;
    currentProject = n.project_url;
    total_num_of_projects << 1;
    verbose_test_project = false;
    ifall (i: int; !match(`^java$`, lowercase(n.programming_languages[i])))
      stop;
  }

  before node : Revision -> {
    currentRevision = node.id;
    revisionTimestamp = node.commit_date;
  }

  before node: ChangedFile -> {
    currentAst = getast(node);
    currentFile = string(node.name);
    exists (i: int; match((`org.powermock`), currentAst.imports[i])
      || match((`org.easymock`), currentAst.imports[i])
      || match((`org.mockito`), currentAst.imports[i])
      || match((`org.jmockit`), currentAst.imports[i])
    ) {
      stop;
    }
    ifall (i: int; !match((`org.junit.Test`), currentAst.imports[i]))
      stop;
  }

  before node : Method -> {
    num_of_statements = 0;
    ifall (i: int; !match((`Test`), node.modifiers[i].annotation_name))
      stop;
  }

  before node : Statement -> {
    num_of_statements++;
  }

```

```
96    after node : Method -> {
97      k := currentProjectId + ";" + currentFile + ";" + node.name;
98      if(num_of_statements > sloc) {
99        verbose_test_project = true;
100       z1_first_occurrence[k] << currentRevision + ";" +
          ↪  string(time_to_int(revisionTimestamp)) weight 1;
101       z3_longevity[k] << 1;
102       z2_last_occurrence[k] << currentRevision + ";" +
          ↪  string(time_to_int(revisionTimestamp)) weight 1;
103     }
104     z4_first_change[k] << currentRevision + ";" +
        ↪  string(time_to_int(revisionTimestamp)) weight 1;
105     z5_last_change[k] << currentRevision + ";" +
        ↪  string(time_to_int(revisionTimestamp)) weight 1;
106     num_of_statements = 0;
107   }
108
109   after node : Project -> {
110     num_of_projects << 1;
111     if(verbose_test_project){
112       num_of_verbose_test_projects << 1;
113     }
114   }
115 });
```

APPENDIX B

Other Scripts

B.1 Project Size in AST Nodes

```
# Project AST Nodes Count Script
# ==========================
#
# Copyright (C) 2018 Andres Carrasco
#
#   This program is free software: you can redistribute it and/or modify
#   it under the terms of the GNU General Public License as published by
#   the Free Software Foundation, either version 3 of the License, or
#   (at your option) any later version.
#
#   This program is distributed in the hope that it will be useful,
#   but WITHOUT ANY WARRANTY; without even the implied warranty of
#   MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
#   GNU General Public License for more details.
#
# You should have received a copy of the GNU General Public License
# along with this program.  If not, see <http://www.gnu.org/licenses/>.
#
# ==========================
p: Project = input;

sloc: output collection[string] of int;
currentProjectId : string = "";
currentAst : ASTRoot;
astCount := 0;
visited : bool = false;

nodecount := visitor {
  # by default, count all visited nodes
  before _ -> {
    astCount++;
  }
};

visit(p, visitor {
  before n: Project -> {
    astCount = 0;
    currentProjectId = n.id;
    visited = false;
    ifall (i: int; ! match(`^java$`,
    ↪  lowercase(n.programming_languages[i])))
      stop;
  }

  before n: CodeRepository -> {
```

```
45      snapshot := getsnapshot(n);
46      foreach (i: int; def(snapshot[i]))
47        visit(snapshot[i]);
48      stop;
49    }
50
51    before node: ChangedFile -> {
52      currentAst = getast(node);
53      visit(currentAst, nodecount);
54    }
55
56    after node: ChangedFile -> {
57      ifall (i: int; match((`org.junit.Test`), currentAst.imports[i])){
58        visited = true;
59      }
60      exists (i: int; match((`org.powermock`), currentAst.imports[i])
61        || match((`org.easymock`), currentAst.imports[i])
62        || match((`org.mockito`), currentAst.imports[i])
63        || match((`org.jmockit`), currentAst.imports[i])
64      )
65      visited = false;
66    }
67
68    after node: Project -> {
69      if(visited) {
70        sloc[currentProjectId] << astCount;
71      }
72    }
73  });
```

B.2 Unit Method Size

```
# Test Unit Method Size Script
# ==========================
#
# Copyright (C) 2018 Andres Carrasco
#
#   This program is free software: you can redistribute it and/or modify
#   it under the terms of the GNU General Public License as published by
#   the Free Software Foundation, either version 3 of the License, or
#   (at your option) any later version.
#
#   This program is distributed in the hope that it will be useful,
#   but WITHOUT ANY WARRANTY; without even the implied warranty of
#   MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
#   GNU General Public License for more details.
#
# You should have received a copy of the GNU General Public License
# along with this program.  If not, see <http://www.gnu.org/licenses/>.
#
# ==========================
p: Project = input;

currentProject : string= "";
currentFile : string = "";
currentAst : ASTRoot;
num_of_statements : int = 0;
num_of_projects: output sum of int;

mean_method_size: output mean[string] of int;
min_method_size: output minimum(1)[string] of int weight int;
max_method_size: output maximum(1)[string] of int weight int;
method_size: output collection[string][string][string] of int;

visit(p, visitor {
  before n : Project -> {
    currentProject = p.project_url;
    ifall (i: int; !match(`^java$`, lowercase(n.programming_languages[i])))
      stop;
  }

  before node: CodeRepository -> {
    snapshot := getsnapshot(node);
    foreach (i: int; def(snapshot[i]))
      visit(snapshot[i]);
    stop;
  }
```

```
46
47    before node: ChangedFile -> {
48      currentAst = getast(node);
49      currentFile = string(node.name);
50      exists (i: int; match((`org.powermock`), currentAst.imports[i])
51        || match((`org.easymock`), currentAst.imports[i])
52        || match((`org.mockito`), currentAst.imports[i])
53        || match((`org.jmockit`), currentAst.imports[i])
54      ) {
55        stop;
56      }
57      ifall (i: int; !match((`org.junit.Test`), currentAst.imports[i]))
58        stop;
59    }
60
61    before node : Method -> {
62      num_of_statements = 0;
63      ifall (i: int; !match((`Test`), node.modifiers[i].annotation_name))
64        stop;
65    }
66
67    before node : Statement -> {
68      num_of_statements++;
69    }
70
71    after node : Method -> {
72      mean_method_size[currentProject] << num_of_statements;
73      min_method_size[currentProject] << 1 weight num_of_statements;
74      max_method_size[currentProject] << 1 weight num_of_statements;
75      method_size[currentProject][currentFile][node.name] <<
        ↪  num_of_statements;
76      num_of_statements = 0;
77    }
78
79    after node : Project -> {
80      num_of_projects << 1;
81    }
82  });
```