Master research Internship

Internship report

# Adapting Amplified Unit Tests for Human Comprehension

**Domain: Software Engineering — Artificial Intelligence**

*Supervisor:*
Benoit Baudry

*Author:*
Simon Bihel

KTH Royal Institute of Technology

**Abstract:**

With practices such a test-driven development, software projects now come with strong test suites. They embed knowledge of supported inputs and expected behaviour which make it easy to detect bugs such as unwanted changes in behaviour as the code evolves with time. But writing and maintaining many test cases is time-consuming for the developers and as they are humans they can miss some edge cases resulting in sub-optimal testing. To address these problems, automated methods have been developed to generate tests from scratch, optimise test suites, or extend test suites. Because the space of possible tests is so vast, approximate meta-heuristic methods are often used for these problems. In particular, evolutionary methods are particularly suited for amplifying and improving an existing test suite by generating random variants of test cases. Tools have been developed to amplify test suites but optimising the generated tests is still an open problem. Because developers are still involved in the loop, by reviewing and merging new tests, this thesis focuses on reducing the amount of work the developer has to provide by making them easier to understand for a human.

# Contents

# Introduction

The adoption of test-driven and agile methods for software development and with the more recent emergence of DevOps, developers have serious incentives to write strong test suites. They contain large number of test cases, which embed rich knowledge about relevant input and expected behaviour. The key incentive for developers is that these test suites can be automatically executed on demand to minimise regression bugs as the code continuously evolves. However, the production and maintenance of large test suites to detect these regressions is time-consuming, and, consequently, developers tend to focus on nominal paths when writing test cases, missing the corner, rare cases. The key challenge we address in this thesis is as follows: automatically analyse existing test cases and exploit the knowledge that developers have embedded there in order to generate variant test cases that target new behaviour. This is called *test amplification* [1].

Automatic test generation is a traditional area in software testing [2]: the goal is to generate test cases according to a specific test criterion. However, test generation techniques tend to ignore the existence of manually written test suites: they assume that no tests exist at all, or that they are either too few or of too low quality for being considered useful in the generation process. The emerging field of test amplification considers a different approach and explicitly aims at exploiting knowledge from existing test cases. For this thesis, we focus on search-based techniques to exploit this knowledge.

Search-based software engineering [3] is the discipline that investigates meta-heuristic search (e.g., genetic algorithms or simulated annealing) to automate software engineering tasks. In the case of automatic test generation, this consists in iteratively generating test cases and selecting the ones that improve an objective, expressed as a fitness function (e.g., increase code coverage). Automatically generated tests are challenging for the developers who wish to minimise the time spent on test execution and who want to understand these new tests before integrating them in their code base. Meta-heuristics can serve to minimise the number of relevant test cases, or generate summaries for developers to understand the new test cases.

This thesis aims at increasing the adoption of test suite amplification tools by developing a generator of messages that present and explain amplified tests in a human-friendly way. Such generator has been implemented for the DSpot [4] tool.

Section 1 provides the necessary background on software testing and Section 2 explains the concepts of test suite amplification. Section 3 lies down the challenges of integrating test cases generators in developers' workflow and Section 4 presents the works done on this topic. Section 5 and Section 6 present the contribution of this thesis and how it was evaluated.

# 1 Background

In this section, we present the landscape this thesis fits into. We define testing (Section 1.1) as well as go over its use in the industry. In particular, to understand the needs of practitioners, we present how they assess the quality of their code (Sections 1.2 and 1.3) and what it takes for a new tool to be incorporated in their workflow (Sections 1.4).

## 1.1 Software Testing

In this section we give a definition for the test activities and their actors, the different abstraction levels of tests, and our precise object of study, unit tests.

### 1.1.1 Test Activities

Testing is about verifying that a system, for a given scenario, follows a certain behaviour that was previously defined. The *System-Under-Test* (SUT), in our domain a software system, has a set of components $C$. A scenario is sequence of stimuli that target a subset of $C$, and trigger responses: (i) feedbacks from the SUT; and (ii) changes in the state of its components. From [5], we use the following definition for to combination of stimuli and responses:

**Definition 1.1** (Test Activities)**.** For the SUT $p$, $S$ is the set of stimuli that trigger or constrain $p$'s computation and $R$ is the set of observable responses to a stimulus of $p$. $S$ and $R$ are disjoint. Test activities form the set $A = S \uplus R$. The disjoint union is used to label the elements of $A$.

The use of the terms "stimulus" and "observations" fits various test scenarios, functional and non-functional. As depicted in Figure 1, a stimulus can be either an explicit *input* from the tester, or an environmental factor that can affect the SUT. Stimuli include, among others, the platform, the language version, the resources available, interaction with an interface, etc. Observations encompass anything that can be discerned and measured like the content of a database, the visual responses on a computer screen, the time passed during the execution, etc.

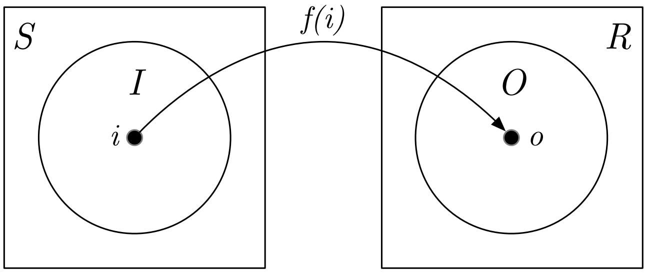


Figure 1: Stimuli and observations: $S$ is anything that can change the observable behaviour of the SUT $f$; $R$ is anything that can be observed about the system's behaviour; $I$ includes $f$'s explicit inputs; $O$ is its explicit outputs; everything not in $S \cup R$ neither affects nor is affected by $f$.

Testing is about stimulation and observation, so we talk about *test activity sequences* that are comprised of at least one stimulus and one observation. But testing is also about verifying that the observed responses, the behaviour, match to ones that were previously defined. This checking part is done by another actor:

**Definition 1.2** (Test Oracle)**.** A test oracle $D : T_A \mapsto \mathbb{B}$ is a partial function from a test activity sequence to true or false.

An oracle can be defined using different methods: a set of specifications an expected value for a variable after the execution of a particular test activity sequence, the expectation of an absence of crash, etc.

A test oracle is a partial function because it can leave certain elements of the SUT's behaviour unspecified. This part of uncertainty is fundamental in the real world because systems have grown to be so complex that humans cannot anticipate all the reactions of the SUT in all environments. It is useful nonetheless to have a name for a theoretical oracle that would have an answer for every possible question:

**Definition 1.3** (Ground Truth)**.** The ground truth oracle, $G$, is a total test oracle that always gives an answer.

This allows us to define two properties for all oracles:

**Definition 1.4** (Soundness)**.** The test oracle $D$ is sound for test activity $a$ iff $D(a) \rightarrow G(a)$

**Definition 1.5** (Completeness)**.** The test oracle $D$ is complete for test activity $a$ iff $G(a) \rightarrow D(a)$

A test oracle cannot, in general, be complete as it would require handling every possible test activity sequence. It is not rare for an oracle to be unsound as humans can make errors writting specifications.

In practice, following pattern designs of modularity, the act of testing is composed of multiple test activity sequences that target specific elements of the SUT's behaviour:

**Definition 1.6** (Test Case)**.** A test case $T_C$ is a test activities sequence that contains at least one stimulus and one observation with a test oracle that verifies each observation.

**Definition 1.7** (Test Suite)**.** A test suite is a collection of tests cases.

In other words [6], a test case is a subset of inputs, and a subset of specifications from the ground truth that are matched against the trace of the test's execution.

Another concept that we will use when talking about software evolution is regression testing [7]:

**Definition 1.8** (Regression Testing)**.** Regression testing is performed between two different versions of the SUT in order to provide confidence that newly introduced features do not interfere with the existing features.

It is a way to avoid having a human write formal specification but the problem remains of trying every possible scenario and observing any useful response.

### 1.1.2 Levels of Testing

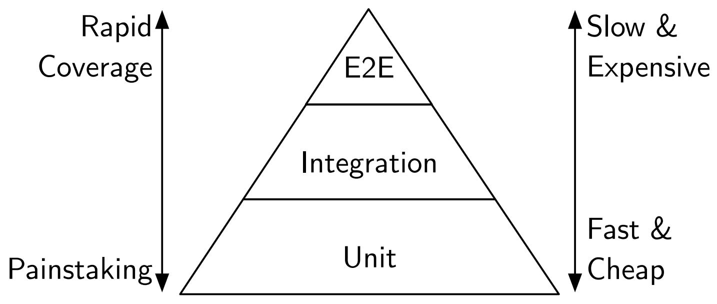


Figure 2: A view of the test pyramid.

In the case of software testing, the SUT can be different elements of a program, i.e. it can belong to different levels of abstraction. One can decide to test a whole program, by manually giving inputs and verifying that the output is correct. Or one could test individual functions, having a more control over the behaviour of each element.

These levels of test abstraction can be visualised with a test pyramid, as shown in Figure 2. Tests are generally separated into three distinct categories. The top category is for the End-to-End tests, which means testing the program as a whole. It can mean interacting with the UI and evaluating the visual responses. Whilst it allows for directly evaluating the well behaviour of the program, each test is costly in time and resources because the whole program has to be run. Not only will each component be run, whether it requires testing or not, but time will also be spent on call to external libraries that are already tested. At the other end of the test spectrum are the unit tests. The goal is to test each and every component of the program (e.g. every function), as an individual and isolated from the rest of the program. Running a single component is faster than running the whole program, but require much more work from the person writing tests, as every component needs its set of tests. In between the two testing practices are the integration tests, which aim at ensuring that the components work together as expected, like verifying that gears rotate in the right way. Examples of integration tests include API testing, or simply the fact of calling a function using the result from another function call. This last example gives an intuition on how fuzzy the distinctions between all kinds of tests are, with definitions often relying on personal opinions[1][2]. More details about the loose definition of unit tests are given Section 1.1.3.

The test hierarchy is often represented by a pyramid because it is generally advised to have many more unit tests than end-to-end tests. Practitioners often recommend putting an emphasis on unit test, with a distribution of 70% unit tests, 20% integration tests, and 10% end-to-end tests for example[3]. More details on the strengths and weaknesses of unit tests are given in the next section, but we can convey some general truths here. End-to-end tests require more maintenance as the software evolves, because test cases rely on many modules and their behaviour eventually change with time. Also, with a broad coverage, it is difficult to identify components at fault in the case of a failing test. Having unit tests for each component gives confidence that they behave accordingly and that if an integration test fails, the error lies in the interactions between the components. Again, the broader the test, the greater the number of interactions which makes it difficult identifying the one at fault. Because of this, the number of test levels has to increase alongside the software's growth — e.g. grouping components into modules, modules into services, and services as the system.

Programs can be tested manually by a human (e.g. by interacting with the UI), but tests are generally automated to save time. Automation often mean to programmatically describe a scenario, observations and their associated specifications. Different levels can require different tools and frameworks[4]. For example, unit tests can be written in the same language as the main program, interacting with it like any other module (more details in the next section). But to test a graphical user interface, one might need to run a window system and control the cursor's movements and clicks.

### 1.1.3 Unit Testing

A unit test is supposed to target a precise component of the SUT, but defining the level of granularity is not trivial, even for experienced programmers[5][6] [8]. In the case of a program written with an Object-Oriented language, such as Java — we will put our focus on Java for the rest of this thesis — the targeted component can be a class or just a method of a class.

An example of unit test is given in Listing 1. Tests are usually sorted in classes, and each test case corresponds to a method (which is why test cases are sometimes refered to as test methods, as we will encounter later on). Specifications of expected values for observations are written using *assertions*. When the condition of an assertion is not met — we call it an assertion *failure* — it raises an exception and causes the test case to fail. Assertions are a feature of test frameworks (i.e. internal DSL [9]), and allow test cases to be regular pieces of code. This has advantages and disadvantages, it is easy to write test cases but might not be the most natural way to express a property on the software artefact targeted. Test methods often rely on naming conventions to help the reader understand what is the target, and what behaviour it is checking. Another pitfall to avoid is to test properties for the programming language used.

Unit tests are usually written by software engineers, as they are the one with the knowledge of the expected behaviour for the piece of software they have produced. Large companies will also have test engineers, focusing on broader tests and commercially important features,

[1]https://testing.googleblog.com/2009/07/software-testing-categorization.html
[2]https://martinfowler.com/bliki/IntegrationTest.html
[3]https://testing.googleblog.com/2015/04/just-say-no-to-more-end-to-end-tests.html
[4]https://martinfowler.com/articles/practical-test-pyramid.html
[5]https://martinfowler.com/bliki/TestPyramid.html
[6]https://martinfowler.com/bliki/UnitTest.html
[7]https://commons.apache.org/proper/commons-collections/xref-test/org/apache/commons/collections4/list/TreeListTest.html#L271

```
 1 public class TreeListTest {
 2     public void testIterationOrder() {
 3         TreeList tl = new TreeList(10);
 4         for (int i = 0; i < 10; i++) {
 5             tl.add(i);
 6         }
 7         ListIterator it = tl.listIterator();

 8         int i = 0;
 9         while (it.hasNext()) {
10             Integer val = it.next();
11             assertEquals(i++, val.intValue());
12         }
13     }
14 }
```

Listing 1: Example of an object-oriented unit test (taken, and adapted for readability, from the Apache Commons Collections, in the class TreeListTest, line 270[7]): it consists of test inputs (lines 3–7) that manipulate the SUT; and assertions (line 11). Methods `.hasNext()` and `.next()` (lines 9 & 10) are expected, implicitly, to behave correctly.

as well as software engineers working on test infrastructure and tooling[8]. Unit testing has becoming an integral part of modern software development with practices such as Test-driven Development (TDD)[9] [10] that consists in writing test cases first, and then writing code that complies. Integrated development environments (IDEs) have been extended to make writing and reviewing tests easier. Test automation in general is also a pillar of modern software delivery cycles like DevOps[10], with Continuous Integration (CI) tools that run test suites each time that code changes are pushed. Tests can then be refined using exploratory testing[11] [11], by trying to manually find flaws while documenting the process to then implement the results as test cases. Manual testing is particularly useful to rapidly detect issues like slow response time, misleading error messages and other design concerns.

Because unit test are written with the same programming language as their target, certain features of the language — that are supposed to help developers write clean code — can become a hindrance. A perfect example with Java are the private methods. They exist because they enforce the separation of the public interface and the inner workings of a class. But when one want to test a private method to make sure it does the right thing, it is not possible. The alternative is to extensively test the public methods, making sure they trigger all sorts of behaviour in the private part of the class. In this situation, we end testing a component that interacts with other components, as depicted in Figure 3, but then it becomes a small integration test[12]. The granularity of tests does not only depend on the needs and desires of the developer, but also on the limitations of the language. Object-Oriented are particularly victim to this, as they are suited for composition design patterns [12], leaving no other option to the developer to try to limit interactions. Ways to avoid writing tests that trigger long chains of

[8] https://testing.googleblog.com/2016/03/from-qa-to-engineering-productivity.html
[9] https://en.wikipedia.org/wiki/Test-driven_development
[10] https://en.wikipedia.org/wiki/DevOps
[11] https://testing.googleblog.com/2008/05/exploratory-testing-on-chat.html
[12] https://martinfowler.com/bliki/UnitTest.html

```java
import org.junit.Test;
import static org.junit.Assert.assertEquals;

public class MyTests {
    @Test
    public void multiplicationOfZeroIntegersShouldReturnZero() {
        MyClass tester = new MyClass();
        assertEquals(0, tester.multiply(10, 0), "10 x 0 must be 0");
    }
}
```

Listing 2: Example of a JUnit test class. A test class is simply a group of test methods (i.e. test cases). Each test case is identified by the annotation `@Test`.

reactions is to use doubles (or mocks) which are hand written clones of classes instances that return predefined values when called. Dependencies are also present in the test's code, as seen in Listing 1 with the lines 9 and 10, as other methods from the class under test can be used for the set-up or to retrieve certain observations.

Another common problem with testing are flaky tests. A flaky test is a test which result is not deterministic. There can be various causes for a flaky test, e.g. a direct usage of RNG function, or a crash happening only in scenario with a certain scheduling in the case of parallel applications. The advantage of small tests with controlled interaction with the environment is that they are less likely to be flaky. This a hard problem[13] as flaky tests can be (i) hard to identify, (ii) hard to refactor, and (iii) hard to avoid in the first place.

An example of test framework, for Java, is JUnit [14]. The basic structure of a JUnit test class is given in Listing 2.

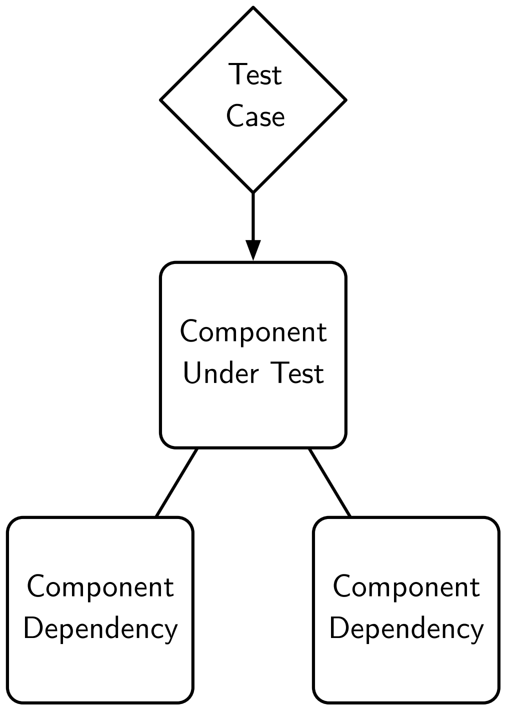


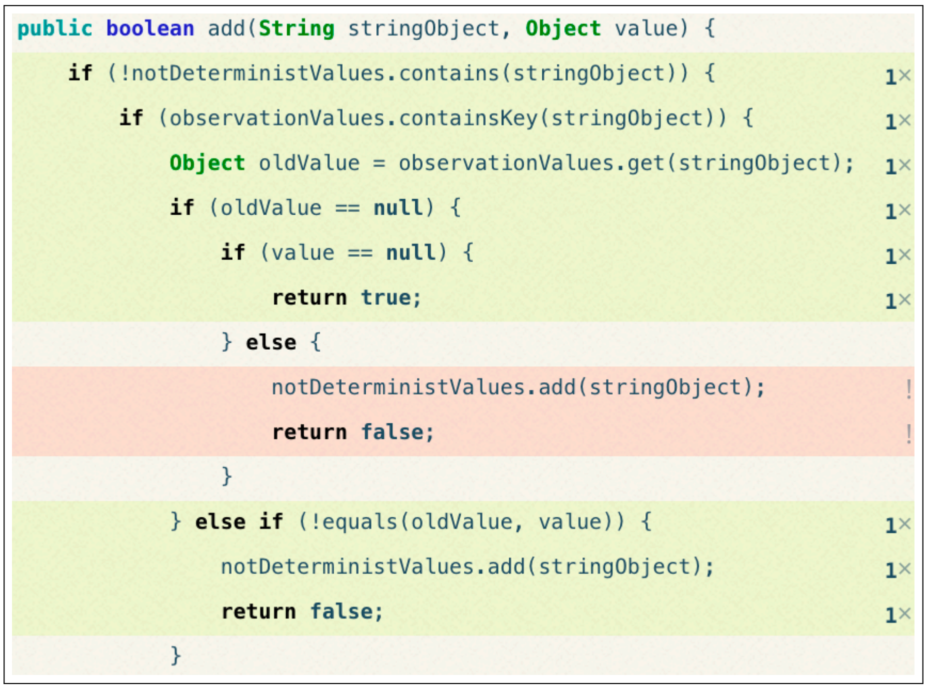
Figure 3: Test target's dependencies.

## 1.2 Elementary Metrics

Metrics have been developed to help developers automatically assess the quality of their code. Each metric measures one specific characteristic, such as the need for refactoring, the size of the program, or — and we are particularly interested in this one — the likelihood of bugs.

The most basic metrics to measure how thoroughly tested a system is, are coverage based metrics. For example, during the test suite execution, one can keep track of all statements that were executed. At the end, you have a percentage of executed statements for the total number of statements. Instead of statements, one can also

[13]https://testing.googleblog.com/2017/04/where-do-our-flaky-tests-come-from.html
[14]https://junit.org/
[15]https://coveralls.io/



```
public boolean add(String stringObject, Object value) {
    if (!notDeterministValues.contains(stringObject)) {                1×
        if (observationValues.containsKey(stringObject)) {             1×
            Object oldValue = observationValues.get(stringObject);     1×
            if (oldValue == null) {                                    1×
                if (value == null) {                                   1×
                    return true;                                       1×
                } else {
                    notDeterministValues.add(stringObject);            !
                    return false;                                      !
                }
            } else if (!equals(oldValue, value)) {                     1×
                notDeterministValues.add(stringObject);                1×
                return false;                                          1×
            }
```

Figure 4: Example of coverage reporting from the Coveralls[15] tool. We can see that the statements in the first `else` branch have not been executed a single time. The numbers on the right represent the number of times each line has been executed.

keep track of control flow branches explored, or functions called. These metrics, especially the statement coverage, are wide spread in the industry — with coverage tools integrated out-of-the-box with source code hosting services and continuous integration services. An example of coverage visual report is shown in Figure 4.

It is generally acknowledged that a system with high coverage means that the system is less likely to have bugs — but it is not foolproof [13, 14]. Such simple metrics are not good at pointing out corner cases[16]. If we take the following example `if` $(C_1$ `AND` $(C_2$ `OR` $C_3))$ where $C_3$ makes the program crash, then it is possible to explore both branches without ever executing $C_3$.

A drawback of metrics in general, is that in practice, testers will change behaviour once they know how the measurement works[17]. They will write tests that will perform well for a given metric and ignoring other important factors (e.g. diversity[18] [15, 16]). In these cases the metric has become the goal instead of a progress measurement.

## 1.3 Mutation Testing

Mutation testing is a method used to assess the quality of a test suite, giving it a grade called *mutation score*. The idea is to insert bugs in the software and see if these faulty versions still pass the tests or not. We call the derived versions *mutants*, hence the name mutation testing. When a mutant goes through the test suite without making a single test fail, it is considered *live*, and the opposite case it is considered *dead*. The global process is depicted in Figure 7. The main appeal is that, a test suite with a good mutation score, in addition to ensure the well behaviour

[16]https://martinfowler.com/bliki/TestCoverage.html
[17]https://testing.googleblog.com/2009/09/7th-plague-and-beyond.html
[18]https://testing.googleblog.com/2009/06/by-james.html

of the SUT in useful scenarios, can detect divergent behaviours — which is particularly useful for regression testing. There is a correlation between mutation score and coverage score as test suites with high mutation score tend to have good coverage [17]. In general, mutation coverage can be seen as a line coverage with levels of coverage depending on the number of mutants present in this line, similarly to the number of times the line is executed. It has been found that correlations between mutation scores and real fault detection are weak [18, 19]. Mutation testing has been extended in various directions (e.g. specification mutation) but we are focusing on the primary usage, the evaluation of a test suite for a given implementation.

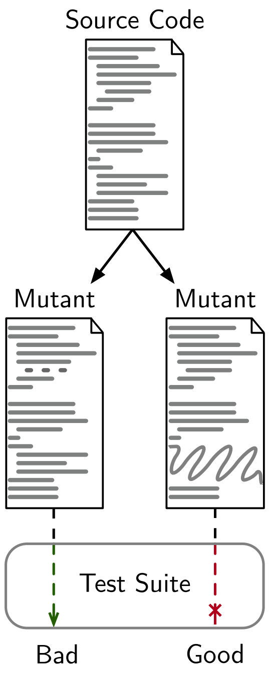


Figure 5: Mutation testing process.

Mutants are obtained by applying *mutators* on the original version. These operators can be varied, from fine-grained like changing a condition (e.g. substituting $\leq$ with $>$), to the granularity of classes [20] (e.g. delete the body of a method[19]). Running first mutation testing with large operators allows to rapidly detect gaps in the coverage of the test suite for unit testing but also for integration testing. Fine grained mutation can then be employed to meticulously evaluate the functional testing capabilities of the test suite [21]. Most of the time, we generate 1-*order* mutants, that means that a mutant is the result of a mutator applied once. A mutator will usually generate several mutants. For example, a mutator that replaces a `while` statement into a `do-while` statement will generate a mutant for each `while` in the artifact. In the case of $k$-order mutants, which can be thought of as the result of $k$ succesive 1-order mutants[22], the number of derived programs starts to blow up.

Mutation testing suffers from drawbacks that have limited its use outside of the research world even though it has been an active research domain for many years [23]. The first obvious flaw is that it is slow. For compiled languages, all mutants have to be compiled, and that process often takes a lot of time for large programs. Then the whole test suite has to be executed for each mutant, which again is a long process. However, given certain trade-offs in terms of quality of the mutation analysis process (e.g. not generating all possible mutants), the computational complexity can be dodged [24, 25]. Optimised techniques for regression testing also exist [7], including minimisation, selection and prioritisation, which can reduce the execution time of the test suite. The mutant generation is also full of traps. The major pitfall that cannot be avoided is the equivalent mutant problem. Some mutants, although they have a different program than the original, have the same semantics. Which means that they will always pass the test suite and will not bring any insight on the SUT. The generation of mutants that are simply syntactically valid is also a non-trivial problem that requires pattern replacement instead of simple text replacement [26].

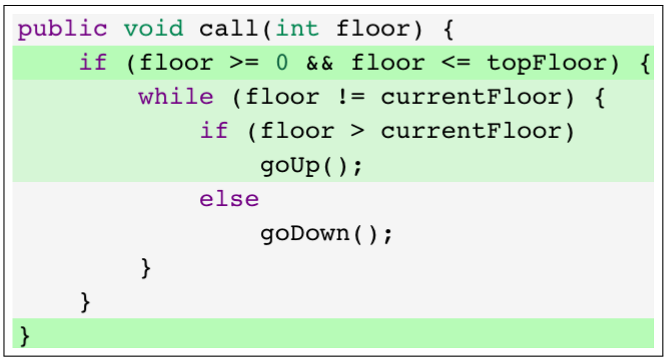
Another obstacle for mutation testing to take on in the industry is the lack of understanding by practitioners. The process of mutation and elimination can be confusing. But also, when a mutant is live, it is not always clear what actions should be taken in order to kill it.

Multiple mutation tools exist, with differences on the level of abstraction they handle, different performances and different correlation with real fault detection. Many of them are robust, thanks to the efforts from the research community to reach the industry, and the need to have reliable implementations to conduct empirical studies related to mutation testing. One of these tools is PIT[20] [27], a tool for the Java bytecode. The research version (i.e. state-of-the-art) of PIT is known to perform better than its competitors overall [28]. Two technical details are

[19]https://github.com/STAMP-project/pitest-descartes
[20]http://pitest.org/

interesting for us. The first one is the way it stores the mutants generated. The location of a mutant is specified by (i) the name of the method and class, (ii) the method signature and (iii) the instruction on which the mutation occurs. This little information is sufficient to recreate each mutant while having a small storage footprint. The other important detail is that PIT is deterministic. That is, for the same program, the same mutants will always be generated, and thus the mutation score will always be the same.

The way mutation testing tool report their result is interesting. Instead of trying to show each statement that has been mutated, and inspired by reports of line coverage, they usually display a regular line coverage report with shades of colour. The lightest means that the line has been executed, and the darker it gets, the higher the number of mutants located in this line. An example of mutation coverage visual report is shown in Figure 6.



```java
public void call(int floor) {
    if (floor >= 0 && floor <= topFloor) {
        while (floor != currentFloor) {
            if (floor > currentFloor)
                goUp();
            else
                goDown();
        }
    }
}
```

Figure 6: Example of mutation coverage reporting from Code Defenders[21] [29]. Shades of green are used to denote the amount of testing for a line.

## 1.4 The Need for Easy-to-use Tools

Software Engineering, as a research field, being tightly coupled with practices in the industry, it is important to understand the attempts at transfering research knowledge to the industry. There are many reasons to help the industry, for example: reducing the risks of bugs in critical systems, automating the development process, etc. One way researcher often go, is to develop a tool and then try to introduce it in the workflow of practictioners. But even with good intentions, the process of acceptance by practitioners for a research product is often complicated [30].

An issue that is often faced, is the difficulty for practitioners to understand the usage of the tool, or even its purpose. Thus, not only should a researcher consider the amount of bugs their tool can detect, they should also make sure it is accessible and easy to understand. Having an easy-to-use tool can also open new paths for research, reveiling new problems, and making studies with professional developers easy.

Valuable experience has been earned from static analysis researchers' attempts at integrating their tools in the workflow of their engineer colleagues [31, 32, 33]. Each false positive undermines the trust developers have in the tool. Each misunderstood error make developers angry. And each warning that do not *feel* relevant to the developer raise the likelihood that future warnings will be ignored. As humans, we avoid change unless it is necessary or is directly beneficial to us. Developers do not want to be slowed down by additional actors in their workflow, and companies do not want additional cost with little return.

The field of software testing, in which this thesis is set, is no stranger to this problem. Tests are, in most cases, written and maintained by general software engineers. Thus, we must aim at producing tools with little overhead and fully integrated with popular test frameworks — making it easy for developers to try the tool to see if it fits their needs, and, if they incorporate it in their workflow, it brings important data to demonstrate the scientific value of the tool.

[21] http://code-defenders.org/

# 2 Test Suite Amplification

In this Section, we provide more context for this thesis' contribution. We explain how global optimisation techniques, including metaheuristic search techniques (e.g. genetic algorithms) can be applied to software engineering problems — for example, by using software metrics as target functions. Section 2.1 presents how source code can be manipulated for search processes, and Section 2.2 presents the specific problem of improving test suites. Section 2.3 introduces DSpot, a tool that enhances tests for a better coverage, and also the software on which this thesis builds on.

## 2.1 Genetic Improvement

Genetic improvement (GI) [34] uses automated search to find improved versions of existing software. Automated search encompasses optimisation methods that approximate optimums (e.g. genetic algorithms, simulated annealing). Applying search-based techniques for software engineering problems is not new [2] as is the field of genetic programming (GP) [35] (with, at the time of writing this thesis, comprises over 12000 references[22]). But from around 2012, when the name of GI has been coined [36], the reached level of computing power that one can easily have access to has allowed research to achieve impressive results (e.g. transplanting a feature from one program to another [37]).

Search operators are most frequently *delete*, *copy* and *replace* [38]. For example, duplicate a line of code, or delete a method call, or replace an integer by another number.

One thing to remember, is that complex search techniques are not always the most efficient [39], as a simple random search can sometimes produce sufficient results. This is especially true for object-oriented test generator, as they consist of sequences of object constructor and method calls, and it can be hard to know how to combine them to pursue a certain coverage goal. Synergy between method calls is hard to predict because classes are interlinked, methods have side effects, and there are goals that cannot be reached. In addition, having a specific goal in mind can be at the expense of a wider coverage.

## 2.2 Test Amplification

As unit tests are program artefacts like others, GI can be applied on test suites [1] and search techniques have been employed since 2005 [40]. It is relatively recent compared to works tackling the problem of Test Data Generation (i.e. generating tests from scratch for a piece of software), but it is only recent that we can expect software projects, large or modest, to come with strong test suites. Various goals can be pursued to improve a test suite, from automated refactoring to improve a specific metric, to enhancement of existing tests to fill the often partial coverage[23]. The term of test *amplification* is used to denote the process of processing and extending tests to reach further targets. Baudry et al. [40] wrote a tool for C# to explore more of the input space and improve the mutation score. Yoo et al. [41] proposed the Test Data Regeneration (TDR) method for creating equivalent tests with different inputs — particularly useful for fault detection and avoiding overfitting (when the SUT passes the tests but does not behave correctly in the general case). Xuan et al. [42, 43] created a tool to refactor test suite to improve dynamic analysis and fault detection.

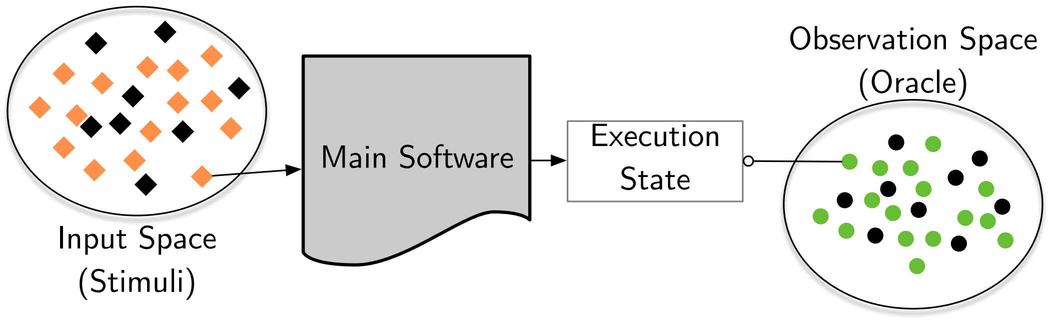


Figure 7: On the left, the testing input space is composed by specified input points (orange diamonds) and unspecified input points (black diamonds). On the right, the observation space over a given program state depicts the assertions of tests. The green circles are values that are already asserted in the existing test suite, the newly added assertions are shown as black circles.

## 2.3 DSpot

DSpot[24][44, 45, 4] is a test amplification tool. Its goal is to amplify tests individually to improve a given metric, e.g. mutation score. Going back to our definition of tests, with the notions of stimuli and observations, DSpot has a search process on both spaces. New inputs are generated, and new observations are made. A visual representation is given in Figure 7. Because of the oracle problem, DSpot focuses on regression testing — assertions (i.e. specifications) are generated from observations of an actual execution of the test. It is a Java tool, it uses Spoon [46] to manipulate the source code, and consists in about 14000 physical lines of code (i.e. non-comment and non-blank).

To explore more of the input space (we also talk about *amplifying* inputs), operators called I-Amplification are applied.

***Amplification of literals*** Similarly to TDR, literals (numeric, boolean, string) are replaced by neighbours. For example, an integer can be multiplied by 2 or a random character can be added to a string.

***Amplification of method calls*** Methods calls can be duplicated, removed, or made-up by picking a random variable in the test and calling one of its methods with random parameters.

***Test objects*** If a new object is required for an amplified method call, one of the appropriate type will be built by using the default constructor.

Like 1-order mutants, only one amplification is applied at a time, to limit the number of generated tests. Because the random amplification can create flaky tests, they are executed three times to verify that they have a consistent behaviour.

As for the amplification of observations (A-Amplification), focus is put on checking more properties to kill more mutants. Assertions are generated as follows: the state of program is collected after execution of the test case to know the actual values and use these values as oracle.

The global amplification process is as follows:

1. remove all assertions, but keeping the method calls used for the assertions' arguments to make sure the test triggers the same behaviour;
2. execute the test with instrumentation to collect the values from getter methods, and then generate assertions with the values from the observations;
3. generate multiple tests by applying input amplifiers;

[22]http://www.cs.bham.ac.uk/~wbl/biblio/
[23]https://testing.googleblog.com/2014/07/measuring-coverage-at-google.html
[24]https://github.com/STAMP-project/dspot

4. create new assertions if possible;
5. compute the mutation score, noting the tests that have a higher score;
6. if the amplified test raises an exception, a `try/catch` statement is added around the whole test to catch this exception, and at the end of the `try` block, a `fail` statement is added to make sure the exception is raised all the time; and
7. repeat a certain amount of time.

Using mutation score as the target metric allows us to see the problem of amplification as a problem of finding bugs. In other words [47], the amplification process consists of improving the code coverage to execute faulty statements, and improving the propagation of faulty program states to an observable output.

A minimisation phase is done on useful amplified tests, to remove noisy amplifications. This make the test easier to understand for a human, but has certain drawbacks. When using general purpose minimisation techniques (e.g. delta-debugging, variable in-lining), the delimitation between original test and amplifications fades away and might produce a test that looks completely different to the original test.

Another to know about DSpot is the way it stores tests. Test methods are Spoon ASTs, and when an amplification is applied, the whole method is cloned. And because of the large number of amplifications, the memory usage ramps up and can quickly reach the memory limit.

DSpot is different from traditional test generator like EvoSuite [48] for multiple reasons. It is more than just using the existing test suite as a good starting population. Building on top of human knowledge allows to cover all possible goals [49]. Evaluating each test individually allows for more focused tests.

The result of DSpot is a set of amplified tests but it should not stop there. Building incrementally on top of humans work is valuable. Adding the generated tests in the main test suite has multiple advantages. First, is the review process. Developers have to review changes before it is accepted in the main code. During the review process developers can tweak the assertions, rearrange the inputs, etc. In that sense, DSpot fits as a software development bot, integrated in the workflow [50], that could be run on code changes, after making sure the test suite is passing. The amplified tests would then be submitted as a Pull Request (a.k.a. Merge Request) in the code version control platform used. Secondly, it is more efficient. Developers will not write tests that are equivalent to generated tests. If a generated test fails they might ignore it or spend a lot of time trying to understand it at a crucial time. And, in general, automated Pull Requests can make developers more keen to maintain a project [51].

A real-life example of amplified is given in Appendix A with the XWiki[25] project.

## 3 Problem Statement

This thesis aim at helping developers understand amplified unit tests. There are two sides for this problem: supporting the developer understand the test (Section 3.1) and cleaning the tests from the noise injected during the amplification process (Section 3.2).

### 3.1 The Need for Unit Test Cases Documentation

The first challenge in rendering generated tests comprehensible, is to bundle them with explanations, like we do by documenting the code we write. Several works have already highlighted the need for test documentation, with developers surveys [52, 53, 54, 55, 56] and experiments [57] to measure the added value of documentation. Even when they are short, tests can be complex and

[25]http://www.xwiki.org/

hard to understand. In all cases, having documentation — which include documentation comments like JavaDoc, but also the naming of the tests and variable — has many advantages[26] [58], especially for generated tests [59, 60]:

- faster to get familiar with the test;
- faster fault localisation (e.g. figuring out if there is a bug in the test or the SUT) when the test fails; and
- helps to build trust in a test generator if it can provide a proof for its result.

Information to understand can be divided into four categories: (i) the component targeted; (ii) the context of the test method; (iii) the actual piece of code (i.e. the logic) that is the test method; and (iv) the properties checked. And we can split them again in two groups: the *what* and *why* informations. As we will see in Section 4, extensive works have been produced to generate automatic documention of the what information. As for the why information, it is difficult to automatically extract the motivation of the developper that wrote the test. But in the case of generated test, we have the reason why they were kept: they bring something to the metric. Thus, if we focus on amplification for mutation score, we would need to explain this score and what new mutants are killed. A lack of works on assisting humans to get the grasp of a mutation score makes a good case for us to tackle the problem of explain a killed mutant.

In source code management platforms, pull requests (through which amplified are submitted for review and merge in the main branch) are accompanied by textual descriptions. Given that amplified tests should, in most cases, extend an existing test — that we can suppose have documentation already — the explanation of the added value of the amplification would fit in the pull request description. It is important to restrict the scope to simple text, as it would not require a change in developers workflow.

### 3.2 The Generated Random Noise

Through the search process, noise is injected in the test case. This problem of ugly generated tests code is a shared problem between all kinds of generators [59]. Generated tests are equally helpful than hand-written tests (with randomized names) for software maintenance tasks [60], even less readable [61], and don't seem to find more bugs [62], based on the limited data available. So the goal is to make a similar job than humans, and because documentation is not sufficient to understand and review the code, tests need to be clean and logical to speedup the comprehension process. Even though it has been known for a long time that the amount of time needed by developers to locate and understand code is frequently greater than the amount of time that they spend making modifications [63], existing works have largely focused on best efforts.

On a side note, in many modest projects, unit test are used as examples of the program's usage, and sometimes they replace traditional documentation. In those cases, developers put a lot of efforts in making tests easy to understand. Doing part of that chore automatically would remove part of the burden for humans.

## 4 Related Works

In this section we review the large body of related works from the domain of software maintenance [64]. We are interested in tools that automatically generate documentation — and there are many of them, as it is well-known that software documentation is important [65] (even for modern agile methods) but it is time-consuming, often times poor and incomplete [66], and is

[26] https://testing.googleblog.com/2014/10/testing-on-toilet-writing-descriptive.html

| Structural | | Collaborational | Creational |
|---|---|---|---|
| Accessor | Mutator | | |
| Get<br>Predicate<br>Property | Set<br>Command | Collaborator-Accessor<br>Collaborator-Mutator | Constructor<br>Copy-Constructor<br>Destructor<br>Factory |

Table 1: A taxonomy of method stereotypes.

hard to maintain [67]. They range from documenting general purpose source code (Section 4.1), to commit message generation (Section 4.2), and documenting test cases (Section 4.3).

## 4.1 Source Code Documentation

Works on source code documentation — also called software artifact summarisation — often consist of retrieving information from the source code to then arrange this information in a way that is easier and quicker to grasp for a human. This section does not aim at providing a survey of the field (for which there are already, including automatic summarisation [68, 69], natural language paraphrasing [70] and automatic comment generation [71]), but to present the general techniques used as well as the kind of results those tools produce.

As said above, such tools will rearrange information, stripping irrelevant pieces, summarising chucks on code, and put central information on the front. In other words, they render the what information in a human comprehensible way — they *paraphrase* the code. This is the main reason why, even automatic summarisation of natural language documents has been attempted by researchers for more than half a century [68], finding a link with our problem of explaining the why information is not trivial.

One of the most influential works was produced by Dragan et al. [72, 73], which is about method stereotypes. They proposed a taxonomy of C++ methods, given in Table 1, and a way to automatically assign stereotypes. This way of annotating methods lead to other works — using stereotype distribution as a signature descriptor for systems [74], extending the taxonomy for Java methods and classes [75].

Buse and Weimer [76] wrote a tool to automatically document the exceptions a method can raise. Exceptions are dugged up from the call stack and the conditions for them to be raised are translated in natural language. They compared their generated documentation with hand-written documention in a survey with developers.

Hill [77, 78] has developed a model, the Software Word Usage Model (SWUM), that captures the occurrences of words in code, and also their linguistic and structural relationships. "SWUM captures the conceptual knowledge of a programmer expressed through both natural language information and programming language structure and semantics, in the form of phrasal concepts. In contrast to existing lexical approaches, we take a transformative step in automatic analysis for software engineering tools by accounting for how words occur together in code, rather than just counting their frequencies." Because programmers tend to choose descriptive and meaning names for code units [79], natural language descriptions of code artefacts can be made, without necessarily having to understand the code.

Sridhara et al. [80, 81] worked on automatically generating summary comments for Java methods. For that they wrote an algorithm to extract important code statements and used SWUM to paraphrase those statements. They did a survey with developers to evaluate the accuracy and usefulness of their contribution.

McBurney and McMillan [82] wrote a tool to generate natural language descriptions of the context of Java methods. What they call context is how the methods are invoked. They use the SWUM to describe the method, describe the method's caller, what the return value is used for, what methods it calls, and how the return value can be used. They did two user studies.

Sridhara et al. [83] worked on describing methods from a high level point of view. They detect high level actions (i.e. statements grouped by a loop or a condition) and use the SWUM to describe them. They did a survey with developers to judge the descriptions synthesised.

Moreno et al. [84] developed a tool to automatically generate summaries of Java classes. A summary is composed of the class' stereotype [75], a description of its interface, structure and behaviour (by enumerating the most important methods). Summaries are in natural language and generated using templates. They did a survey with developers to judge how readable and understandable the summaries are.

Rastkar et al. [85] worked on summarising bug reports, including involved discussions. They label sentences and extract individual information. Works related to bug reports are important for us as a killed mutant is a bug in a certain way — but the knowledge we can extract from these works is not obvious because part of the job of a bug report is to help localise the bug, and in the case of mutation testing we already know the source. Nevertheless, we can learn from these works how to explain the repercussions of a mutant. Except that in this case, it is summarising a human discussion.

## 4.2 Source Code Change Documentation

Software evolves, iteratively, commit by commit. Past the phase of assimilation when a developer first start to work on a project, a significant part of the job consists of managing the growth of the project, making small changes to the code base without unwanted side effects, and reviewing commits. Commits bundle a cohesive set of changes with a textual description, in which it is customary to provide some context, the motivation and possibly a digest of the changes — part what and part why information. Although developers view accurate and concise descriptions as useful, few put the efforts in writing such comments. In consequence, researchers have written tools to automatically generate commit messages.

Dragan et al. [86] proposed commit stereotypes. Those stereotypes are composed of the modified classes and methods' stereotypes. For example, a commit where 75% of the added/deleted methods are factory methods denotes an "object creation modifier".

Cortés-Coy et al. [87, 88] developed *ChangeScribe*, a commit message generator. From Git[27] changes they identify the responsibilities of the modified classes and methods, attribute stereotypes to changes [86], keep the most impactful (i.e. the most called modified methods or classes) changes, and generate natural language descriptions. Messages are generated using templates. The evaluated their tool with survey, comparing real-life commit messages and generated messages, asking the developers which one they prefered. They also provide a table listing many approaches for generating descriptions of source code changes and software artifacts, providing more references than this section.

Jiang and McMillan [89] worked on generating one-phrase summaries of commits. They used a classifier trained on real commits to label a commit with a verb (e.g. add, fix), identify the most important words occurring in the `diff`, and then use common phrase structure to combine the verb and the subject.

Jiang et al. [90] used neural machine translation to generate a commit message from a `diff`. Neural networks were also used by Loyola et al. [91].

[27] https://git-scm.com/

| Categories of commits | Description |
|---|---|
| Implementation | New requirements |
| Corrective | Processing failure; performance failure; implementation failure |
| Adaptive | Change in data environment |
| Perfective | Processing inefficiency; performance enhancement; maintainability |
| Non-functional | Code clean-up; legal; source control system management |

Table 2: Categories of commits in terms of maintenance task and corresponding description.

Shen et al. [92] worked on summarisation of what and why information. The what information is generated by paraphrasing the `diff`, dropping some information or adding some depending on how important the modified unit is. The why information is generated using categories proposed by Swanson [64], given in Table 2, and classify commits using the rules proposed by Moreno et al. [75]. The sentences were generated using templates.

Buse and Weimer [93] produced a technique for documenting program changes, using symbolic execution They use a minimalist programmatic syntax to express changes (e.g. change in condition) and paraphrase changes at the statement level, instead of line granularity. Messages are generated as a pattern-matching and replace. They also studied some open-source projects and showed that what information 12% more common than why information in commit messages. They conducted a human study to compare their tool against existing human-written messages.

Buckley et al. [94] also proposed a taxonomy of software changes. Their taxonomy focuses exclusively on the what information of a change, the mechanisms of change and the factors that influence these mechanisms. Mechanisms include, for example, the change type or the change history, while the influencing factors include, for example, the change frequency, the granularity or the change propagation. In other words, they focus on the *when*, *where*, *what*, and *how* aspects.

Rastkar et al. [95] produced a generator of feature summary for Java projects, to help programmers be aware of, and understand, the ramifications of a feature. A summary provides the number of methods implementing the feature, objects supported, dependencies, and names of methods involved. Relationships between code elements are extracted, including calls between methods, field declarations, class extensions, etc. NL sentences are generated with templates. They evaluated their tool with a lab study, asking developers to modify or add a feature using the summaries, observing their usage and surveyed the developers at the end.

## 4.3 Automatic Test Case Documentation

Finally, the piece of work that is most related to this thesis is about documentation for test cases. Because of test cases are one precise kind of software artefact, the works that we have covered in the previous two sections can be cumbersome here (e.g. stereotypes are not suited).

As unit tests validate a specific component, in object-oriented languages like Java, they often consist in calling a few methods, multiple times, that will affect the state of an object and then verify that the object is in the right state. This kind of method is called a *focal* method, and Ghafari et al. [96] have developed a way to automatically identify them. They use static data-flow analysis to examine the call graph and extract the last method that has a side effect on the assertion. This work is useful to summarise test cases, helping to identify the key stimulus, but it is not directly useful for us as we want to explain a corner case (i.e. an amplification that kills a new mutant).

Cornelissen et al. [97] propose the use of sequence diagrams[28] to visualise Java test cases. Because test suites are sometimes the entry points for new contributors to understand the scope and the inner workings of a project, visualising the actors and their interactions can be more useful than a simple class diagram. Diagrams can be used for complex test cases as they hide certain call (e.g. constructors) and have limited call stack depth.

According to Meszaros [98], a unit test case typically has four phases: setup, exercise, verify and teardown. The setup phase instantiates the classes that the test uses. The exercise phase is the stimulation. The verify phase is the use of the oracle with assertions. The teardown phase is for the cleanup (i.e. bring the SUT back to a neutral state).

Kamimura et al. [99] have developed a generator of textual summaries for Java test cases. A summary is composed of NL paraphrases of unique method invocations and the assertions. There are some interesting details, for example, when a variable's name is too short (e.g. "a1") they make-up a new name with the variable's type. Sentences are built using pre-defined templates.

Li et al. [56] have written a C# tool that generates descriptions for test methods. The output, a GUI window, is composed of 4 pieces of information:

1. a general sentence describing the purpose of the test method (based on class, method and arguments signatures using the SWUM approach [100]);
2. descriptions of the focal methods (i.e. the line number and the assertion it is linked to);
3. descriptions of the assertions (i.e. natural language (NL) paraphrasing); and
4. the slicing paths (backward slicing [101]) for the variables validated with assertions (i.e. NL paraphrasing of each step).

NL sentences are generated with templates.

Li [102] also proposed 21 stereotypes for methods in Java unit tests, based on the JUnit API (14 stereotypes) but also using the data and control flow (7 stereotypes). The API-based stereotypes allow the extraction of fine-grained what information — for example, with `fail` (which raises a failure if the test cases reaches that statement) you can deduce that the test is a "utility verifier" or that a method with the annotation `@Before` is a "test initializer". On the other hand, data/control flow analyses can capture the context of the API calls. The purpose of the test can be deduced, for example, if the number of assertions within branch conditions is greater than 0, it means that the test is a "branch verifier". It is also possible to get insights in the context of the test case in relation to the test class, for example, a test case is a "public field verifier" if values in assertions are from public field accesses.

Beck et al. [103] wrote a framework that generates method execution reports, mostly textual but with a few embedded visualisations. The report includes a summary (number of calls, recursion), summaries for each method calls (number of calls, recursion depth), time consumption tear down, and the code. Embedded visualisations consists of fills bars for relative quantities and histograms for recursion stack. The text generation involves sentence templates and decision graphs (e.g. if there is recursion then present recursion information). All data are collected dynamically.

Delamaro and Maldonado [104] developed a fully fledged GUI for mutation testing, providing test cases execution and management, mutants creation and management, and mutation score reports. A report includes general information like the test cases and the killed mutant's names, the compilation command used, etc.

Code Defenders[29] [29] is a game[30] that aims at teaching [105] software testing concepts, especially mutation testing. In this game, players are either attackers or defenders. Attackers

[28] https://en.wikipedia.org/wiki/Sequence_diagram

[29] http://code-defenders.org/

[30] https://en.wikipedia.org/wiki/Gamification_of_learning

```java
public void test() {
  servlet = new BarcodeServlet();
  params.put("height", "200");
  params.put("width", "3");
  params.put("resolution", "72");
  req.setParameters(params);
  servlet.doGet(req, res);
  Barcode barcode = servlet.getBarcode();
  assertEquals(barcode.getResolution(), 72);
}
```

| | |
|---|---|
| **small** | `testDoGet` |
| **medium** | `testDoGetResolutionIs72` |
| **full** | `testDoGetResolutionIs72WhenParamsResolutionIs72AndSettingParameters` |

Listing 3: Example [109] of a unit test and the generated names.

write mutants, and defenders write tests cases. If there are more killed mutants than live ones the defenders are the winner, and conversely if there are more live mutants. The way a mutant is presented is with a simple `diff` (i.e. the modified statement is highlighted). This representation makes sense in a GUI with mutants on one side and the tests on the other side. Another motivation is that the game aims at teaching the current state of mutation testing. Because of these two choices, it is hard for us to draw inspiration.

Qusef et al. [106] worked on keeping track of traceability between Java unit tests and tested class, to maintain consistency in test classes and give some hints on the target of a test case. They use dynamic slicing [107] to identify all the classes that affect the result of the last assert statement in each unit-test method. And, because the slicing is an overestimate, they use conceptual coupling [108] (analysing names and comments) to filter out superfluous classes (e.g. mock objects). This problem of traceability is useful for us as we would like the amplified tests to remain in the same scope as the original, aiming for a more through coverage of the target instead of just random mutants.

Zhang et al. [109] wrote NameAssist, a tool that generates names for JUnit test cases. They differentiate three different pieces of information contained in the body of a test method: the action, the expected outcome, and the scenario under test. The generated names are made up using these but only the action part, or the action and expected outcome, can be used to make up shorter names. The action consists of the invocation of the method under test — identifying the action requires an extensive set of rules to account for cases where the standard naming convention is not followed, 7 different patterns to identify the class under test, 2 conditions to consider only expressions related to the class under test, and then 5 rules to choose the action. The expected outcome is a paraphrase of the assertion (they only consider tests with a single assertion), so it consists of the name of a getter and the expected value. Finally, the scenario (the part that set-ups the environment) is put into words using the action dependency graph and indicate that parameters are set or that a method is used. An example is given in Listing 3. The text is generated using test name templates [110] (from a high level, test names often consist of an action phrase and a predicate phrase) filled with custom translations of specific expressions (e.g. translating an `assertTrue` as “is true”).

Daka et al. [58] proposed a technique to generate descriptive names for automatically generated unit tests. Their work is based on EvoSuite [48], a test suite generator (from scratch) that

```
public void test0 / testAddPriceReturningFalse() {
  ShoppingCart cart0 = new ShoppingCart();
  boolean boolean0 = cart0.addPrice(2298);
  assertEquals(0, cart0.getTotal());
  assertFalse(boolean0);
}
```

| | |
|---|---|
| **test0** | Method createsShoppingCart |
| | Method addPrice |
| | Output addPriceReturningFalse |
| | Input addPriceWithPositive |

Listing 4: Example [58] of a unit test, its coverage goals, and a generated test.

produces test methods with names with the only characteristic of being unique (`test0`, `test1`, `test2`, etc.). Their naming approach is valuable for us as they consider the fitness function guiding the generation: code coverage. They consider coarse-grained coverage with (i) method coverage; (ii) exception coverage; and (iii) output coverage. For each test they collect the goals reached (methods called, inputs and outputs used in an assertion, and exceptions caught), and then rank them to keep only the most important goals. The tests names are then generated by concatenating (almost as is) the goals kept (the names of the methods, exception, etc.). An example is given in Listing 4.

Most of the papers related to human assistance reviewed here were evaluated with developers surveys. Proteum/IM 2.0 [104] used a case study, and Code Defenders conducted a lab study.

As we can see, a large corpus of works already exists for automatic test documentation. But, two main aspects in our case have not been studied: the documentation of test amplification (i.e. code changes specifically about tests) and the explanation of mutation scores.

## 5 Contribution

In this section, we present this thesis' contribution which mainly consists in developing a generator of messages to present amplified tests produced by DSpot. First, a fork of DSpot[31] was made to implement a logging process for amplifications (more details in Section 5.1). It was made with about 250 new lines of code. Then, a generator of explanations, called PR_GEN[32], was written to use the information logged to automatically generate messages that present the amplified tests (Section 5.2), like a human developer would add a descriptive message when submitting a Pull Request. It is written in Python, with 655 physical lines of code, and generates messages in Markdown. As a side effect of the internship, 6 Pull Requests were pushed to the main repository[33], fixing bugs and contributing to the documentation, consisting in about 350 new lines of code.

### 5.1 Collecting Amplifications

Before explaining how amplifications are logged, we present our categorisation of amplifications. We use four categories: one for all a-amplifications and three for the i-amplifications.

[31] https://github.com/sbihel/dspot/tree/collect_amp
[32] https://github.com/sbihel/internship_amplification/tree/master/pr_message_gen
[33] https://github.com/STAMP-project/dspot/commits?author=sbihel

`ASSERT` encompasses new assertions, `try/catch` blocks, and new variables used for assertions.

`ADD` is for new method calls.

`DEL` is for deleted method calls.

`MODIFY` is for all modified elements (i.e. nodes in the AST) so it can be a modified literal, a modified variable assignment, etc.

Then, the central element of the amplifications logging system is a `map` that links every amplified test to a list of all the amplifications applied. An amplification is characterised by the following attributes:

`ampCategory` the amplification category;

`parent` the parent node of the modified node (e.g. the block that contains a new statement, or the method invocation that has a modified argument);

`role` the role of the modified node for its parent (e.g. the new statement is a *child* of the parent block, or the modified argument is an *argument* of the method invocation);

`oldValue` the value before the amplification, can be `null` if the amplification is an addition; and

`newValue` the current value.

Even if we talk about nodes in the AST, `parent`, `oldValue` and `newValue` are the textual representation.

Having an external list of amplification means that when minimisation actions are taken, the list should be updated. It is easy to forget to implement the update when writing a new minimisation action, which means that the logging system is not really scalable if we wanted DSpot to be modular. DSpot might implement a different way of storing amplified tests, using patches instead of cloning whole methods, which could render our logging system useless if it is not too costly in time.

## 5.2 Natural Language Description

We chose to focus on mutation testing, so the goal of PR_GEN is to explain: (i) the amplifications that constitute the amplified test (Section 5.2.1); and (ii) the reasons why this amplified test was kept, i.e. the improved mutation score (Section 5.2.2).

We wrote a tool, PR_GEN, in Python, that formats the information in the Markdown language. This is because popular source code management services such as GitHub[34] or GitLab[35] use a Markdown syntax for users' messages. Screenshots are given in Appendix B to show how the text generated by PR_GEN is rendered on GitHub.

Natural language sentences are generated using simple templates. A Natural Language Generator was not needed as each sentence carry a single simple piece of information, and the intricacies of details are display using a hierarchy of titles and sections.

In our description of amplified tests, we assume a certain knowledge of the original test of the developer's side. This is in part to really differentiate our work from the related works, but also to leave an automatic *understanding* of the original test out of the scope of our study.

Several iterations of amplifications are applied, and not all intermediate test brings something for the mutation score. So, when we describe an amplified test, we include the amplifications of its (useless) parents.

[34]https://github.com/

[35]https://gitlab.com/

### 5.2.1 Amplifications Overview

There are two sections to present the amplifications: one for the new inputs and one for the new assertions related instructions.

Inputs are presented by simply listing them. For each, their role and the name of the type (e.g. `CtBlock` is the Spoon type of a block) of the parent is given — unless it is so generic that it becomes irrelevant, e.g. “New *child* of *CtBlock*”. Then a `diff` is generated to display the old and new values.

For a-amplifications, there are different scenarios. When a `try/catch` is generated, we only say that an exception handler has been added (around the whole test) for the specifically targeted exception. When a variable is generated, we group every new assertion using that variable. If a new variable is not used by any assertion, we move it to the input category. The remaining assertions are listed at the end.

### 5.2.2 Mutation Score Details

The mutation testing tool that DSpot uses is PIT (presented in Section 1.3). PIT applies mutations at the bytecode level, which makes it nearly impossible to get the Java code of the mutant, but we still have several pieces of information at our disposal.

| | |
|---|---|
| **`mutantOperator`** | The mutator applied. |
| **`mutatedMethod`** | The method that contains the mutation. |
| **`lineNumber`** | Line number where the mutation is. |
| **`description`** | A natural language description of the mutant. |

Because mutation testing is not well understood, we try not to speak directly about mutants, but we still want to provide details about the added value of the new test in case it is not obvious. For this reason, the name of the mutator is not useful. We decided to focus on regression testing, and talk about detectable changes. Thus, we say that changes at the line `lineNumber` in the method `mutatedMethod` can be detected. Details about each mutant are given by giving directly the natural language description of PIT.

We also mention whether it is because of a better oracle or thanks to new behaviours that mutants are detected. For that, we do not track the direct effect of each new assertion or new input, and simply assume that if there are new assertion we have a better oracle, and if we have new inputs we have new behaviours.

# 6 Evaluation

In this section we go through the evaluation of our contribution. First, we give a few words on the limits of our evaluation in Section 6.1. Then, in Section 6.2 we show the performance overhead of our contribution.

## 6.1 Threat to Validity

This thesis’ contribution was only tested on a few projects, and we did not collect developers’ feedback. A large scale study would be difficult to conduct as DSpot is not yet a robust and established tool, but thorough case studies with active developers would give hints at the usefulness of our work.

A concern to be aware of, which arises from PIT and DSpot, is that we do not know if a new mutant was really killed by one particular amplified test. When running a batch of test, PIT does not report which test killed which mutant, which means that the mutation score is not accurate for individual test. This bug could be fixed in the future.

| | **Main** | **Fork** |
|---|---|---|
| Run without monitoring | 19 min 52 sec | 20 min 27 sec |
| Run with monitoring | 21 min 56 sec; 834 MB max used heap | 21 min 40 sec; 884 MB max used heap |

Table 3: Performance evaluation on XWiki's `cipher` module[36].

### 6.2 Performances

As the contribution consists in two parts, the amplification logging system and the generator of messages, both should be evaluated on a performance criterion. For the former, in addition to the obvious overhead in execution time, because DSpot can easily reach an Out of memory (OOM) state it is essential to assess that the amplification log does not precipitate such error. For the later, because of its programming triviality, only the execution time was measured. Comparisons were made between our modified version and the main repository at the last change that was included in our fork.

The results of the evaluation of the logging system are given in Table 3. Standard runs showed no real discrepancy between the two versions' execution time. A run with monitoring (using VisualVM 1.4.1) was made to measure the peak of memory usage. It is not an ideal way to measure the memory overhead of the logging system, because measurements are done by intervals so the can miss the true maximum. Our modified version shows a higher memory usage of 6%. The logging system could be improved by regularly removing discarded amplified tests with only discarded children, but that would probably come as a trade-off for execution time.

The message generator completes its task in a 5 seconds for 11 amplified tests.

## Conclusion

Automatic feedback and automatic improvement of software projects are part of a blooming economy and research domain with the global adoption of the Continuous Integration and Continuous Deployment. Automatic test generation has been studied for decades now, but has never really been widely accepted in developers workflow. On the other hand, automatic test improvement is promising as it fit the iterative process of modern software development, and can use hand-written tests to have a better oracle. But in order to integrate the workflows of developers, much has to be done to ease the integration between the humans and the machine-generated tests. To tackle this problem we developed a generator of textual explanations that help the developers understand the changes and why these changes are useful. This work will really be proven useful once the code of the generated tests is cleaner and once the user base of test enhancement tools grows. Additional information could be added to the messages, for example explaining the relation between an assertion and the mutant it kills, or an explanation of the repercussions of a mutant on the Class Under Test.

## Acknowledgments

Thanks to Benoit Baudry and Martin Monperrus for their guidance. Thanks to Benjamin Danglot for his collaboration and all his work on DSpot. Thanks to Zimin Chen, Thomas

[36]https://github.com/xwiki/xwiki-commons/tree/master/xwiki-commons-core/xwiki-commons-crypto/xwiki-commons-crypto-cipher

Durieux, Nicolas Harrand, He Ye and Long Zhang for making daily life enjoyable. Thanks to Vincent Massol for his feedbacks.

This internship was supported by the Fondation Rennes 1 and its patrons. Many thanks to KTH for hosting me and providing housing.

## References

[1] B. Danglot, O. Vera-Perez, Z. Yu, M. Monperrus, and B. Baudry, "The emerging field of test amplification: A survey," *arXiv preprint arXiv:1705.10692*, 2017.

[2] P. McMinn, "Search-based software testing: Past, present and future," in *Software testing, verification and validation workshops (icstw), 2011 ieee fourth international conference on*, pp. 153–163, IEEE, 2011.

[3] M. Harman and B. F. Jones, "Search-based software engineering," *Information and software Technology*, vol. 43, no. 14, pp. 833–839, 2001.

[4] B. Baudry, S. Allier, M. Rodriguez-Cancio, and M. Monperrus, "Dspot: Test amplification for automatic assessment of computational diversity," *arXiv preprint arXiv:1503.05807*, 2015.

[5] E. T. Barr, M. Harman, P. McMinn, M. Shahbaz, and S. Yoo, "The oracle problem in software testing: A survey," *IEEE transactions on software engineering*, vol. 41, no. 5, pp. 507–525, 2015.

[6] G. Bernot, M. C. Gaudel, and B. Marre, "Software testing based on formal specifications: a theory and a tool," *Software Engineering Journal*, vol. 6, no. 6, pp. 387–405, 1991.

[7] S. Yoo and M. Harman, "Regression testing minimization, selection and prioritization: a survey," *Software Testing, Verification and Reliability*, vol. 22, no. 2, pp. 67–120, 2012.

[8] P. Runeson, "A survey of unit testing practices," *IEEE software*, vol. 23, no. 4, pp. 22–29, 2006.

[9] M. Fowler, *Domain-specific languages.* Pearson Education, 2010.

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

[11] C. Kaner, J. Falk, and H. Q. Nguyen, *Testing Computer Software Second Edition.* Dreamtech Press, 2000.

[12] P. Wolfgang, "Design patterns for object-oriented software development," *Reading Mass*, p. 15, 1994.

[13] D. Hovemeyer and W. Pugh, "Finding bugs is easy," *Acm sigplan notices*, vol. 39, no. 12, pp. 92–106, 2004.

[14] L. Inozemtseva and R. Holmes, "Coverage is not strongly correlated with test suite effectiveness," in *Proceedings of the 36th International Conference on Software Engineering*, pp. 435–445, ACM, 2014.

[15] B. Baudry, M. Monperrus, C. Mony, F. Chauvel, F. Fleurey, and S. Clarke, "Diversify: Ecology-inspired software evolution for diversity emergence," in *Software Maintenance, Reengineering and Reverse Engineering (CSMR-WCRE), 2014 Software Evolution Week-IEEE Conference on*, pp. 395–398, IEEE, 2014.

[16] B. Baudry and M. Monperrus, "The multiple facets of software diversity: Recent developments in year 2000 and beyond," *ACM Computing Surveys (CSUR)*, vol. 48, no. 1, p. 16, 2015.

[17] B. Assylbekov, E. Gaspar, N. Uddin, and P. Egan, "Investigating the correlation between mutation score and coverage score," in *Computer Modelling and Simulation (UKSim), 2013 UKSim 15th International Conference on*, pp. 347–352, IEEE, 2013.

[18] M. Papadakis, D. Shin, S. Yoo, and D.-H. Bae, "Are mutation scores correlated with real fault detection? a large scale empirical study on the relationship between mutants and real faults," in *40th International Conference on Software Engineering, May 27-3 June 2018, Gothenburg, Sweden*, 2018.

[19] R. Just, D. Jalali, L. Inozemtseva, M. D. Ernst, R. Holmes, and G. Fraser, "Are mutants a valid substitute for real faults in software testing?," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pp. 654–665, ACM, 2014.

[20] S. Segura, R. M. Hierons, D. Benavides, and A. Ruiz-Cortés, "Mutation testing on an object-oriented framework: An experience report," *Information and Software Technology*, vol. 53, no. 10, pp. 1124–1136, 2011.

[21] W. E. Howden, "Weak mutation testing and completeness of test sets," *IEEE Transactions on Software Engineering*, no. 4, pp. 371–379, 1982.

[22] K. Wah, "A theoretical study of fault coupling," *Software testing, verification and reliability*, vol. 10, no. 1, pp. 3–45, 2000.

[23] Y. Jia and M. Harman, "An analysis and survey of the development of mutation testing," *IEEE transactions on software engineering*, vol. 37, no. 5, pp. 649–678, 2011.

[24] A. J. Offutt, G. Rothermel, and C. Zapf, "An experimental evaluation of selective mutation," in *Proceedings of the 15th international conference on Software Engineering*, pp. 100–107, IEEE Computer Society Press, 1993.

[25] J. Možucha and B. Rossi, "Is mutation testing ready to be adopted industry-wide?," in *International Conference on Product-Focused Software Process Improvement*, pp. 217–232, Springer, 2016.

[26] A. Simão, J. C. Maldonado, and R. da Silva Bigonha, "A transformational language for mutant description," *Computer Languages, Systems & Structures*, vol. 35, no. 3, pp. 322–339, 2009.

[27] H. Coles, T. Laurent, C. Henard, M. Papadakis, and A. Ventresque, "Pit: a practical mutation testing tool for java," in *Proceedings of the 25th International Symposium on Software Testing and Analysis*, pp. 449–452, ACM, 2016.

[28] M. Kintis, M. Papadakis, A. Papadopoulos, E. Valvis, N. Malevris, and Y. Le Traon, "How effective are mutation testing tools? an empirical analysis of java mutation testing tools with manual analysis and real faults," *Empirical Software Engineering*, pp. 1–38, 2017.

[29] J. M. Rojas, T. D. White, B. S. Clegg, and G. Fraser, "Code defenders: Crowdsourcing effective tests and subtle mutants with a mutation testing game," in *Proceedings of the 39th International Conference on Software Engineering*, pp. 677–688, IEEE Press, 2017.

[30] M. Delahaye and L. Bousquet, "Selecting a software engineering tool: lessons learnt from mutation analysis," *Software: Practice and Experience*, vol. 45, no. 7, pp. 875–891, 2015.

[31] A. Bessey, K. Block, B. Chelf, A. Chou, B. Fulton, S. Hallem, C. Henri-Gros, A. Kamsky, S. McPeak, and D. Engler, "A few billion lines of code later: using static analysis to find bugs in the real world," *Communications of the ACM*, vol. 53, no. 2, pp. 66–75, 2010.

[32] C. Sadowski, J. Van Gogh, C. Jaspan, E. Söderberg, and C. Winter, "Tricorder: Building a program analysis ecosystem," in *Proceedings of the 37th International Conference on Software Engineering-Volume 1*, pp. 598–608, IEEE Press, 2015.

[33] C. Sadowski, E. Aftandilian, A. Eagle, L. Miller-Cushon, and C. Jaspan, "Lessons from building static analysis tools at google," *Communications of the ACM*, vol. 61, no. 4, pp. 58–66, 2018.

[34] J. Petke, S. Haraldsson, M. Harman, D. White, J. Woodward, *et al.*, "Genetic improvement of software: a comprehensive survey," *IEEE Transactions on Evolutionary Computation*, 2017.

[35] J. R. Koza, "Genetic programming as a means for programming computers by natural selection," *Statistics and computing*, vol. 4, no. 2, pp. 87–112, 1994.

[36] M. Harman, W. B. Langdon, Y. Jia, D. R. White, A. Arcuri, and J. A. Clark, "The gismoe challenge: Constructing the pareto program surface using genetic programming to find better programs (keynote paper)," in *Automated Software Engineering (ASE), 2012 Proceedings of the 27th IEEE/ACM International Conference on*, pp. 1–14, IEEE, 2012.

[37] E. T. Barr, M. Harman, Y. Jia, A. Marginean, and J. Petke, "Automated software transplantation," in *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, pp. 257–269, ACM, 2015.

[38] J. Petke, "New operators for non-functional genetic improvement," in *Proceedings of the Genetic and Evolutionary Computation Conference Companion*, pp. 1541–1542, ACM, 2017.

[39] S. Shamshiri, J. M. Rojas, L. Gazzola, G. Fraser, P. McMinn, L. Mariani, and A. Arcuri, "Random or evolutionary search for object-oriented test suite generation?," *Software Testing, Verification and Reliability*, 2017.

[40] B. Baudry, F. Fleurey, J.-M. Jézéquel, and Y. Le Traon, "Automatic test case optimization: A bacteriologic algorithm," *ieee Software*, vol. 22, no. 2, pp. 76–82, 2005.

[41] S. Yoo and M. Harman, "Test data regeneration: generating new test data from existing test data," *Software Testing, Verification and Reliability*, vol. 22, no. 3, pp. 171–201, 2012.

[42] J. Xuan, B. Cornu, M. Martinez, B. Baudry, L. Seinturier, and M. Monperrus, "Dynamic analysis can be improved with automatic test suite refactoring," *arXiv preprint arXiv:1506.01883*, 2015.

[43] J. Xuan, B. Cornu, M. Martinez, B. Baudry, L. Seinturier, and M. Monperrus, "B-refactoring: Automatic test code refactoring to improve dynamic analysis," *Information and Software Technology*, vol. 76, pp. 65–80, 2016.

[44] B. Baudry, S. Allier, M. Rodriguez-Cancio, and M. Monperrus, "Automatic software diversity in the light of test suites," *arXiv preprint arXiv:1509.00144*, 2015.

[45] B. Baudry, S. Allier, and M. Monperrus, "Tailored source code transformations to synthesize computationally diverse program variants," in *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, pp. 149–159, ACM, 2014.

[46] R. Pawlak, M. Monperrus, N. Petitprez, C. Noguera, and L. Seinturier, "Spoon: A Library for Implementing Analyses and Transformations of Java Source Code," *Software: Practice and Experience*, vol. 46, pp. 1155–1179, 2015.

[47] S. Shamshiri, R. Just, J. M. Rojas, G. Fraser, P. McMinn, and A. Arcuri, "Do automatically generated unit tests find real faults? an empirical study of effectiveness and challenges (t)," in *Automated software engineering (ASE), 2015 30th IEEE/ACM international conference on*, pp. 201–211, IEEE, 2015.

[48] G. Fraser and A. 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*, pp. 416–419, ACM, 2011.

[49] J. M. Rojas, M. Vivanti, A. Arcuri, and G. Fraser, "A detailed investigation of the effectiveness of whole test suite generation," *Empirical Software Engineering*, vol. 22, no. 2, pp. 852–893, 2017.

[50] S. Urli, Z. Yu, L. Seinturier, and M. Monperrus, "How to design a program repair bot? insights from the repairnator project," in *ICSE 2018-40th International Conference on Software Engineering, Track Software Engineering in Practice (SEIP)*, pp. 1–10, 2018.

[51] S. Mirhosseini and C. Parnin, "Can automated pull requests encourage software developers to upgrade out-of-date dependencies?," in *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*, pp. 84–94, IEEE Press, 2017.

[52] E. Daka and G. Fraser, "A survey on unit testing practices and problems," in *Software Reliability Engineering (ISSRE), 2014 IEEE 25th International Symposium on*, pp. 201–211, IEEE, 2014.

[53] M. P. Prado, E. Verbeek, M.-A. Storey, and A. M. Vincenzi, "Wap: Cognitive aspects in unit testing: The hunting game and the hunter's perspective," in *Software Reliability Engineering (ISSRE), 2015 IEEE 26th International Symposium on*, pp. 387–392, IEEE, 2015.

[54] M. P. Prado and A. M. Vincenzi, "Advances in the characterization of cognitive support for unit testing: The bug-hunting game and the visualization arsenal," in *Software Reliability Engineering Workshops (ISSREW), 2016 IEEE International Symposium on*, pp. 213–220, IEEE, 2016.

[55] M. P. Prado and A. M. R. Vincenzi, "Towards cognitive support for unit testing: a qualitative study with practitioners," *Journal of Systems and Software*, 2018.

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

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

[58] E. Daka, J. M. Rojas, and G. Fraser, "Generating unit tests with descriptive names or: Would you name your children thing1 and thing2?," in *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pp. 57–67, ACM, 2017.

[59] J. M. Rojas and G. Fraser, "Is search-based unit test generation research stuck in a local optimum?," in *Proceedings of the 10th International Workshop on Search-Based Software Testing*, pp. 51–52, IEEE Press, 2017.

[60] S. Shamshiri, J. M. Rojas, J. P. Galeotti, N. Walkinshaw, and G. Fraser, "How do automatically generated unit tests influence software maintenance?," in *Software Testing, Verification and Validation (ICST), 2018 IEEE International Conference on*, IEEE, 2018.

[61] G. Grano, S. Scalabrino, H. C. Gall, and R. Oliveto, "An empirical investigation on the readability of manual and generated test cases," 2018.

[62] G. Fraser, M. Staats, P. McMinn, A. Arcuri, and F. Padberg, "Does automated unit test generation really help software testers? a controlled empirical study," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 24, no. 4, p. 23, 2015.

[63] A. J. Ko, B. A. Myers, M. J. Coblenz, and H. H. Aung, "An exploratory study of how developers seek, relate, and collect relevant information during software maintenance tasks," *IEEE Transactions on software engineering*, vol. 32, no. 12, pp. 971–987, 2006.

[64] E. B. Swanson, "The dimensions of maintenance," in *Proceedings of the 2nd international conference on Software engineering*, pp. 492–497, IEEE Computer Society Press, 1976.

[65] S. C. B. de Souza, N. Anquetil, and K. M. de Oliveira, "A study of the documentation essential to software maintenance," in *Proceedings of the 23rd annual international conference on Design of communication: documenting & designing for pervasive information*, pp. 68–75, ACM, 2005.

[66] L. C. Briand, "Software documentation: how much is enough?," in *Software Maintenance and Reengineering, 2003. Proceedings. Seventh European Conference on*, pp. 13–15, IEEE, 2003.

[67] A. Forward and T. C. Lethbridge, "The relevance of software documentation, tools and technologies: a survey," in *Proceedings of the 2002 ACM symposium on Document engineering*, pp. 26–33, ACM, 2002.

[68] K. S. Jones, "Automatic summarising: The state of the art," *Information Processing & Management*, vol. 43, no. 6, pp. 1449–1481, 2007.

[69] N. Nazar, Y. Hu, and H. Jiang, "Summarizing software artifacts: A literature review," *Journal of Computer Science and Technology*, vol. 31, no. 5, pp. 883–909, 2016.

[70] G. Neubig, "Survey of methods to generate natural language from source code," 2016.

[71] X. Wang, Y. Peng, and B. Zhang, "Comment generation for source code: State of the art, challenges and opportunities," *arXiv preprint arXiv:1802.02971*, 2018.

[72] N. Dragan, M. L. Collard, and J. I. Maletic, "Reverse engineering method stereotypes," in *Software Maintenance, 2006. ICSM'06. 22nd IEEE International Conference on*, pp. 24–34, IEEE, 2006.

[73] N. Dragan, "Emergent laws of method and class stereotypes in object oriented software," in *Software Maintenance (ICSM), 2011 27th IEEE International Conference on*, pp. 550–555, IEEE, 2011.

[74] N. Dragan, M. L. Collard, and J. I. Maletic, "Using method stereotype distribution as a signature descriptor for software systems," in *Software Maintenance, 2009. ICSM 2009. IEEE International Conference on*, pp. 567–570, IEEE, 2009.

[75] L. Moreno and A. Marcus, "Jstereocode: automatically identifying method and class stereotypes in java code," in *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, pp. 358–361, ACM, 2012.

[76] R. P. Buse and W. R. Weimer, "Automatic documentation inference for exceptions," in *Proceedings of the 2008 international symposium on Software testing and analysis*, pp. 273–282, ACM, 2008.

[77] E. Hill, L. Pollock, and K. Vijay-Shanker, "Automatically capturing source code context of nl-queries for software maintenance and reuse," in *Software Engineering, 2009. ICSE 2009. IEEE 31st International Conference on*, pp. 232–242, IEEE, 2009.

[78] E. Hill, *Integrating natural language and program structure information to improve software search and exploration.* University of Delaware, 2010.

[79] B. Liblit, A. Begel, and E. Sweetser, "Cognitive perspectives on the role of naming in computer programs," in *Proceedings of the 18th annual psychology of programming workshop*, pp. 53–67, Citeseer, 2006.

[80] G. Sridhara, E. Hill, D. Muppaneni, L. Pollock, and K. Vijay-Shanker, "Towards automatically generating summary comments for java methods," in *Proceedings of the IEEE/ACM international conference on Automated software engineering*, pp. 43–52, ACM, 2010.

[81] G. Sridhara, *Automatic generation of descriptive summary comments for methods in object-oriented programs.* University of Delaware, 2012.

[82] P. W. McBurney and C. McMillan, "Automatic source code summarization of context for java methods," *IEEE Transactions on Software Engineering*, vol. 42, no. 2, pp. 103–119, 2016.

[83] G. Sridhara, L. Pollock, and K. Vijay-Shanker, "Automatically detecting and describing high level actions within methods," in *Proceedings of the 33rd International Conference on Software Engineering*, pp. 101–110, ACM, 2011.

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

[85] S. Rastkar, G. C. Murphy, and G. Murray, "Summarizing software artifacts: a case study of bug reports," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*, pp. 505–514, ACM, 2010.

[86] N. Dragan, M. L. Collard, M. Hammad, and J. I. Maletic, "Using stereotypes to help characterize commits," in *Software Maintenance (ICSM), 2011 27th IEEE International Conference on*, pp. 520–523, IEEE, 2011.

[87] L. F. Cortés-Coy, M. Linares-Vásquez, J. Aponte, and D. Poshyvanyk, "On automatically generating commit messages via summarization of source code changes," in *Source Code Analysis and Manipulation (SCAM), 2014 IEEE 14th International Working Conference on*, pp. 275–284, IEEE, 2014.

[88] M. Linares-Vásquez, L. F. Cortés-Coy, J. Aponte, and D. Poshyvanyk, "Changescribe: A tool for automatically generating commit messages," in *Software Engineering (ICSE), 2015 IEEE/ACM 37th IEEE International Conference on*, vol. 2, pp. 709–712, IEEE, 2015.

[89] S. Jiang and C. McMillan, "Towards automatic generation of short summaries of commits," in *Proceedings of the 25th International Conference on Program Comprehension*, pp. 320–323, IEEE Press, 2017.

[90] S. Jiang, A. Armaly, and C. McMillan, "Automatically generating commit messages from diffs using neural machine translation," in *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*, pp. 135–146, IEEE Press, 2017.

[91] P. Loyola, E. Marrese-Taylor, and Y. Matsuo, "A neural architecture for generating natural language descriptions from source code changes," *arXiv preprint arXiv:1704.04856*, 2017.

[92] J. Shen, X. Sun, B. Li, H. Yang, and J. Hu, "On automatic summarization of what and why information in source code changes," in *Computer Software and Applications Conference (COMPSAC), 2016 IEEE 40th Annual*, vol. 1, pp. 103–112, IEEE, 2016.

[93] R. P. Buse and W. R. Weimer, "Automatically documenting program changes," in *Proceedings of the IEEE/ACM international conference on Automated software engineering*, pp. 33–42, ACM, 2010.

[94] J. Buckley, T. Mens, M. Zenger, A. Rashid, and G. Kniesel, "Towards a taxonomy of software change," *Journal of Software: Evolution and Process*, vol. 17, no. 5, pp. 309–332, 2005.

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

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

[97] B. Cornelissen, A. Van Deursen, L. Moonen, and A. Zaidman, "Visualizing testsuites to aid in software understanding," in *Software Maintenance and Reengineering, 2007. CSMR'07. 11th European Conference on*, pp. 213–222, IEEE, 2007.

[98] G. Meszaros, *xUnit test patterns: Refactoring test code.* Pearson Education, 2007.

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

[100] K. Herbert, J. Goldhamer, and D. Ilvento, "Swummary: Self-documenting code," 2016.

[101] R. Jhala and R. Majumdar, "Path slicing," in *ACM SIGPLAN Notices*, vol. 40, pp. 38–47, ACM, 2005.

[102] B. Li, *Automatically Documenting Software Artifacts.* PhD thesis, College of William & Mary, 2018.

[103] F. Beck, H. A. Siddiqui, A. Bergel, and D. Weiskopf, "Method execution reports: Generating text and visualization to describe program behavior," in *Software Visualization (VISSOFT), 2017 IEEE Working Conference on*, pp. 1–10, IEEE, 2017.

[104] M. E. Delamaro, J. C. Maldonado, and A. M. R. Vincenzi, "Proteum/im 2.0: An integrated mutation testing environment," in *Mutation testing for the new century*, pp. 91–101, Springer, 2001.

[105] B. S. Clegg, J. M. Rojas, and G. Fraser, "Teaching software testing concepts using a mutation testing game," in *Software Engineering: Software Engineering Education and Training Track (ICSE-SEET), 2017 IEEE/ACM 39th International Conference on*, pp. 33–36, IEEE, 2017.

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

[107] B. Korel and J. Laski, "Dynamic program slicing," *Information processing letters*, vol. 29, no. 3, pp. 155–163, 1988.

[108] D. Poshyvanyk, A. Marcus, R. Ferenc, and T. Gyimóthy, "Using information retrieval based coupling measures for impact analysis," *Empirical software engineering*, vol. 14, no. 1, pp. 5–32, 2009.

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

[110] B. Zhang, E. Hill, and J. Clause, "Automatically generating test templates from test names (n)," in *Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on*, pp. 506–511, IEEE, 2015.

# Appendices

## A Example of Amplified Test

```java
@Test
public void testRSAEncryptionDecryptionProgressive() throws Exception
{
    Cipher cipher = factory.getInstance(true, publicKey);
    cipher.update(input, 0, 17);
    cipher.update(input, 17, 1);
    cipher.update(input, 18, input.length - 18);
    byte[] encrypted = cipher.doFinal();
    cipher = factory.getInstance(false, privateKey);
    cipher.update(encrypted, 0, 65);
    cipher.update(encrypted, 65, 1);
    cipher.update(encrypted, 66, encrypted.length - 66);
    assertThat(cipher.doFinal(), equalTo(input));

    cipher = factory.getInstance(true, privateKey);
    cipher.update(input, 0, 15);
    cipher.update(input, 15, 1);
    encrypted = cipher.doFinal(input, 16, input.length - 16);
    cipher = factory.getInstance(false, publicKey);
    cipher.update(encrypted);
    assertThat(cipher.doFinal(), equalTo(input));
}
```

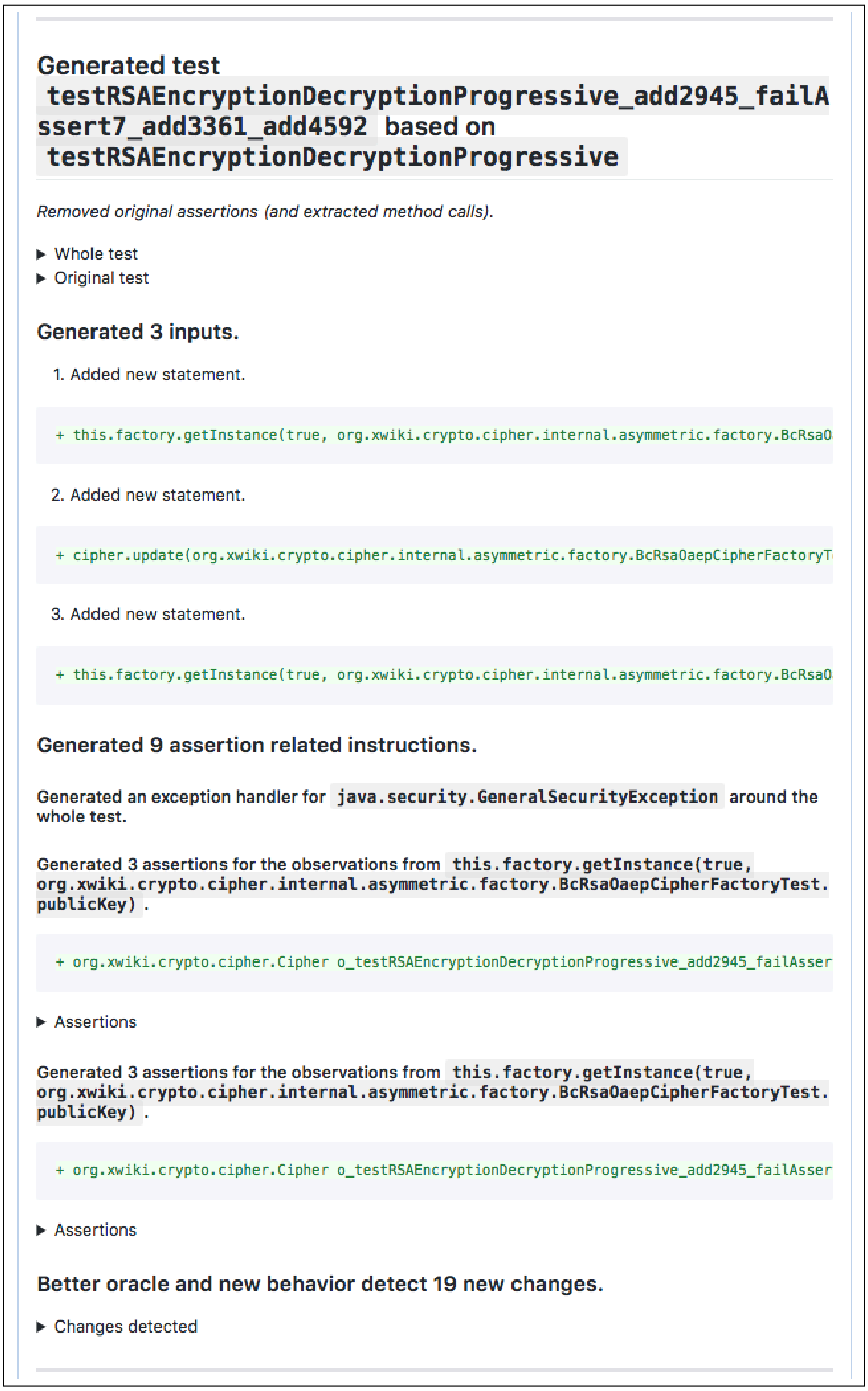
Listing 5: Test from the XWiki project.

```
@Test(timeout = 10000)
public void testRSAEncryptionDecryptionProgressive_add2945_failAssert7_add3361_add4592()
↪  throws Exception {
    try {
        Cipher o_testRSAEncryptionDecryptionProgressive_add2945_failAssert7_add3361__3 =
        ↪  this.factory.getInstance(true, AmplBcRsaOaepCipherFactoryTest.publicKey);
        Cipher o_testRSAEncryptionDecryptionProgressive_add2945_failAssert7_add3361_add4592__6
        ↪  = this.factory.getInstance(true, AmplBcRsaOaepCipherFactoryTest.publicKey);

        ↪  Assert.assertTrue(((org.xwiki.crypto.cipher.internal.asymmetric.BcBufferedAsymmetricC
        Assert.assertEquals("RSA/OAEP",
        ↪  ((org.xwiki.crypto.cipher.internal.asymmetric.BcBufferedAsymmetricCipher)o_testRSAEnc
        Assert.assertEquals(256, ((int)
        ↪  (((org.xwiki.crypto.cipher.internal.asymmetric.BcBufferedAsymmetricCipher)o_testRSAEn
        Assert.assertEquals(214, ((int)
        ↪  (((org.xwiki.crypto.cipher.internal.asymmetric.BcBufferedAsymmetricCipher)o_testRSAEn
        Cipher cipher = this.factory.getInstance(true,
        ↪  AmplBcRsaOaepCipherFactoryTest.publicKey);
        cipher.update(AmplBcRsaOaepCipherFactoryTest.input, 0, 17);
        cipher.update(AmplBcRsaOaepCipherFactoryTest.input, 17, 1);
        cipher.update(AmplBcRsaOaepCipherFactoryTest.input, 18,
        ↪  ((AmplBcRsaOaepCipherFactoryTest.input.length) - 18));
        cipher.update(AmplBcRsaOaepCipherFactoryTest.input, 18,
        ↪  ((AmplBcRsaOaepCipherFactoryTest.input.length) - 18));
        byte[] encrypted = cipher.doFinal();
        cipher = this.factory.getInstance(false, AmplBcRsaOaepCipherFactoryTest.privateKey);
        cipher.update(encrypted, 0, 65);
        cipher.update(encrypted, 65, 1);
        cipher.update(encrypted, 66, ((encrypted.length) - 66));
        cipher.doFinal();
        CoreMatchers.equalTo(AmplBcRsaOaepCipherFactoryTest.input);
        cipher = this.factory.getInstance(true, AmplBcRsaOaepCipherFactoryTest.privateKey);
        cipher.update(AmplBcRsaOaepCipherFactoryTest.input, 0, 15);
        cipher.update(AmplBcRsaOaepCipherFactoryTest.input, 15, 1);
        encrypted = cipher.doFinal(AmplBcRsaOaepCipherFactoryTest.input, 16,
        ↪  ((AmplBcRsaOaepCipherFactoryTest.input.length) - 16));
        cipher = this.factory.getInstance(false, AmplBcRsaOaepCipherFactoryTest.publicKey);
        cipher.update(encrypted);
        cipher.doFinal();
        CoreMatchers.equalTo(AmplBcRsaOaepCipherFactoryTest.input);
        cipher.doFinal();
        CoreMatchers.equalTo(AmplBcRsaOaepCipherFactoryTest.input);
        org.junit.Assert.fail("testRSAEncryptionDecryptionProgressive_add2945 should have
        ↪  thrown GeneralSecurityException");
    } catch (GeneralSecurityException expected) {
        expected.getMessage();
    }
}
```

Listing 6: Amplified test for the test `testRSAEncryptionDecryptionProgressive` in Listing 5.

## B Example of PR Message



### Generated test `testRSAEncryptionDecryptionProgressive_add2945_failAssert7_add3361_add4592` based on `testRSAEncryptionDecryptionProgressive`

*Removed original assertions (and extracted method calls).*

▶ Whole test
▶ Original test

**Generated 3 inputs.**

1. Added new statement.

```
+ this.factory.getInstance(true, org.xwiki.crypto.cipher.internal.asymmetric.factory.BcRsaO
```

2. Added new statement.

```
+ cipher.update(org.xwiki.crypto.cipher.internal.asymmetric.factory.BcRsaOaepCipherFactoryT
```

3. Added new statement.

```
+ this.factory.getInstance(true, org.xwiki.crypto.cipher.internal.asymmetric.factory.BcRsaO
```

**Generated 9 assertion related instructions.**

Generated an exception handler for `java.security.GeneralSecurityException` around the whole test.

Generated 3 assertions for the observations from `this.factory.getInstance(true, org.xwiki.crypto.cipher.internal.asymmetric.factory.BcRsaOaepCipherFactoryTest.publicKey)` .

```
+ org.xwiki.crypto.cipher.Cipher o_testRSAEncryptionDecryptionProgressive_add2945_failAsser
```

▶ Assertions

Generated 3 assertions for the observations from `this.factory.getInstance(true, org.xwiki.crypto.cipher.internal.asymmetric.factory.BcRsaOaepCipherFactoryTest.publicKey)` .

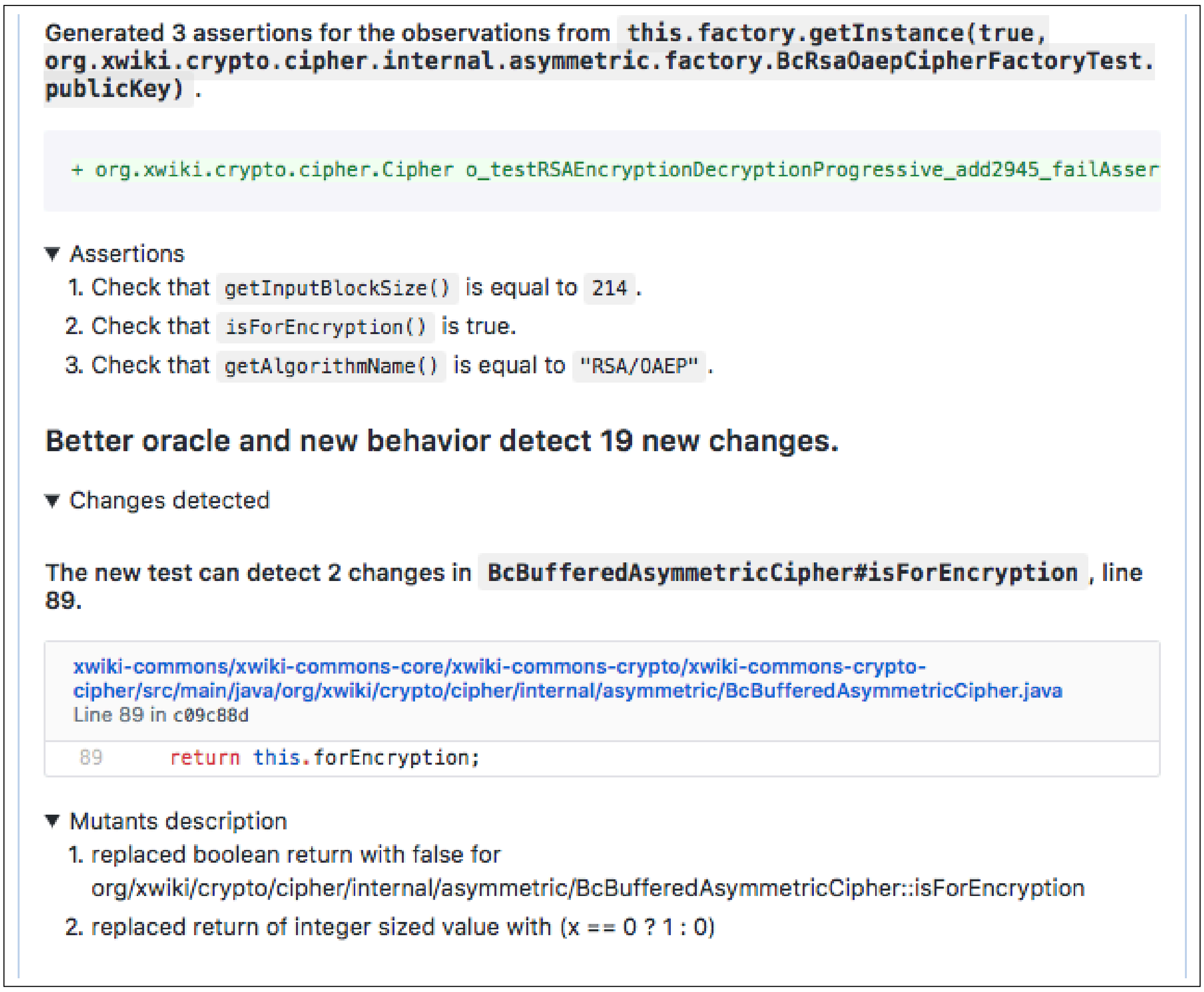
```
+ org.xwiki.crypto.cipher.Cipher o_testRSAEncryptionDecryptionProgressive_add2945_failAsser
```

▶ Assertions

**Better oracle and new behavior detect 19 new changes.**

▶ Changes detected

Figure 8: Example of a message as rendered by GitHub.


Generated 3 assertions for the observations from `this.factory.getInstance(true, org.xwiki.crypto.cipher.internal.asymmetric.factory.BcRsaOaepCipherFactoryTest.publicKey)` .

```
+ org.xwiki.crypto.cipher.Cipher o_testRSAEncryptionDecryptionProgressive_add2945_failAsser
```

▼ Assertions

1. Check that `getInputBlockSize()` is equal to `214` .
2. Check that `isForEncryption()` is true.
3. Check that `getAlgorithmName()` is equal to `"RSA/OAEP"` .

**Better oracle and new behavior detect 19 new changes.**

▼ Changes detected

The new test can detect 2 changes in `BcBufferedAsymmetricCipher#isForEncryption` , line 89.

xwiki-commons/xwiki-commons-core/xwiki-commons-crypto/xwiki-commons-crypto-cipher/src/main/java/org/xwiki/crypto/cipher/internal/asymmetric/BcBufferedAsymmetricCipher.java
Line 89 in c09c88d

```
89        return this.forEncryption;
```

▼ Mutants description

1. replaced boolean return with false for org/xwiki/crypto/cipher/internal/asymmetric/BcBufferedAsymmetricCipher::isForEncryption
2. replaced return of integer sized value with (x == 0 ? 1 : 0)

Figure 9: Example of a message as rendered by GitHub with assertions and mutation score details unfolded blocks.

Figure 8 is a screenshot of a message explaining the amplified tests of the `cipher` module of XWiki, as rendered by GitHub on June 7th 2018. Figure 9 is the same message with unfolded blocks.

GitHub and GitLab allow some HTML in the messages, like `<details>` tags which are blocks that are hidden until the user clicks on the title (indicated by the ▶ symbol). Choosing to add some HTML make some other major services like BitBucket[37]. But having the possibility to add details without overwhelming the reader is great.

Another feature of GitHub is the possibility to a code snippet for links to lines of code. We use the feature to show where a mutant is originating, which is important because the mutant can be far away from the Class Under Test and the developer might not know the method affected.

As said before, mutants descriptions can be hard to generate as PIT acts on the bytecode level. But PIT generates a natural language description for each mutator applied. And even if description can sometimes be vague (e.g. because there was some re-ordering of conditions in the compilation from Java to bytecode) it is still useful to add them.

When a variable is created along with assertions targeting this variable, we can bundle them

[37] https://bitbucket.org/

together and only mention the method called and the expected value. Because DSpot only uses 4 kinds of assertions (`assertTrue`, `assertFalse`, `assertEquals`, `assertNull`) a simple Natural Language template for each of them allow us to generate NL descriptions of assertions.