FACULTY OF SCIENCES

# Optimising Continuous Integration using Test Case Prioritisation

Pieter De Clercq
Student number: 01503338

Supervisors: Prof. dr. Bruno Volckaert, Prof. dr. ir. Filip De Turck
Counsellors: Jasper Vaneessen, Dwight Kerkhove

Master's dissertation submitted in order to obtain the academic degree of
Master of Science in de informatica

Academic year 2019-2020

GHENT UNIVERSITY

Pieter De Clercq – May 25, 2020.

# Acknowledgements

Completing this thesis would not have been possible without the help and support of many people, some of which I want to thank personally.

First of all, I want to thank prof. dr. Bruno Volckaert and prof. dr. ir. Filip De Turck for allowing me to propose this subject and for their prompt and clear responses to every question I have asked. I especially want to thank you for permitting me to insert a two-week hiatus during the Easter break, so I could help out on the UGent Dodona project.

Secondly, I want to express my gratitude towards my counsellors Jasper Vaneessen and Dwight Kerkhove, for steering me into researching this topic, as well as their guidance, availability, and willingness to review every intermediary version of this thesis.

Furthermore, I want to thank my parents, my brother Stijn and my family for convincing me and giving me the possibility to study at the university, to support me throughout my entire academic career and to provide me with the opportunity to pursue my childhood dreams.

Last, but surely not least, I want to thank my amazing friends, a few of them in particular. My best friend Robbe, for always being there when I need him even when I least expect it. For both supporting my wildest dreams while protecting me against my often unrealistic ideas and ambition to excel. Helena for never leaving my side, for always making me laugh when I don’t want to, and most importantly to remind me that I should relax from time to time. Jana for my daily dose of laughter, fun and inexhaustible positivity. Tobiah for the endless design discussions and for outperforming me in almost every school project, to encourage me to continuously raise the bar and to never give up. Finally, I want to thank Doortje and Freija for answering my mathematical questions, regularly asking about my thesis progression and thereby motivating me to persevere.

*Thank you.*

Pieter – Ghent, 2020

# Summary

Summary in English will come here.

# Samenvatting

Nederlandse samenvatting komt hier.

# Optimising Continuous Integration using Test Case Prioritisation

Pieter De Clercq

Supervisor(s): Prof. dr. B. Volckaert, Prof. dr. ir. F. De Turck, J. Vaneessen, D Kerkhove

*Abstract*—**This abstract is very abstract.**

*Keywords*—**words, will, appear, here, soon**

## I. INTRODUCTIE

Things will appear here. [1]

## REFERENCES

[1] Michael Cusumano, Akindutire Michael, and Stanley Smith, “Beyond the waterfall : software development at microsoft,” 02 1995.

# Optimaliseren van Continue Integratie door middel van Test Prioritering

Pieter De Clercq

Supervisor(s): Prof. dr. B. Volckaert, Prof. dr. ir. F. De Turck, J. Vaneessen, D Kerkhove

*Abstract*—**Dit abstract is super abstract.**

*Trefwoorden*—**woorden, komen, hier**

## I. Introductie

Dingen komen hier. [1]

## Referenties

[1] Michael Cusumano, Akindutire Michael, and Stanley Smith, “Beyond the waterfall : software development at microsoft,” 02 1995.

# Lay summary

Lay summary will come here.

# Contents

# Glossary

**CI** Continuous Integration. 2, 20

**MapReduce** a programming paradigm that allows large amounts of data to be processed in a distributed manner. 45

**TCP** Test Case Prioritisation. 2, 20, 23, 24

**TCS** Test Case Selection. 2, 20, 22, 23

**TSM** Test Suite Minimisation. 2, 20, 21, 23, 24

**VCS** Version Control System. 2

# Chapter 1

# Introduction

Given the complexity and rapid pace at which software is being built today, it is inevitable that sooner or later, bugs will emerge. These bugs can either be introduced by a malfunctioning new feature, or by breaking existing functionality (*a regression*). In order to detect bugs in an application before its users do, we require an adequate *testing infrastructure*.

This testing infrastructure consists of multiple *test cases*, collectively referred to as the *test suite* of the application. The quality of a test suite can be assessed in multiple ways. The first and most commonly used method is to measure which fraction of the source code is tested by at least one test case, a ratio which is indicated as the *coverage* of the application. Another possibility is to apply transformations to the source code and validate whether or not this results in a failed test case, a process indicated as *mutation testing*.

Ideally, this testing process should be automated and performed after every change to the source code. This process is generally very time-consuming, and as such has led to the creation of various automation frameworks and tools, collectively called Continuous Integration (CI). Common examples of CI practices are automatically running the test suite and estimating the code coverage after every pushed change to the Version Control System (VCS).

However, applying these practices and maintaining a qualitative test comes at a cost. Every addition or modification to the source code must be followed by at least one test case to validate its correctness. As a result of the speed at which the source code tends to grow, the test suite suffers from severe scalability issues. While it is desirable and ideally required to execute every single test case in the test suite, there are examples known to literature where this is not possible since this incurs an increasing delay in the development process, which in turn results in economic loss.

We can take three approaches to resolve this issue and reduce the time waiting for the test results: Test Suite Minimisation (TSM), Test Case Selection (TCS) and Test Case Prioritisation (TCP). The main subject of this thesis will be to implement a framework for TCP.

The structure of this thesis is as follows. The next chapter will introduce essential concepts used in modern software engineering. Chapter 3 will elaborate more on the three mentioned approaches and present accompanying algorithms. The implementation details of the new framework will be discussed in chapter 4. Afterwards, chapter 5 will evaluate the performance of this framework and provide insights into the characteristics of a typical test suite. More specifically, this chapter will investigate the probability of (repeated) test failure and the average duration of a test run. Finally, chapter 6 will present additional ideas and improvements to the framework.

# Chapter 2

# Software Engineering [TODO REVISE]

ZEG IETS OVER WHITE BOX/BLACK BOX

The Institute of Electrical and Electronics Engineers [IEEE] defines the practice of Software Engineering as: ”Application of a systematic, disciplined, quantifiable approach to the development, operation and maintenance of software; that is, the application of engineering to software” [24, p. 421]. The word “systematic” in this definition, emphasises the need for a structured process, depicting guidelines and models that describe how software should be developed the most efficient way possible. Such a process does exist and it is often referred to as the Software Development Life Cycle (SDLC) [24, p. 420]. In the absence of a model, i.e. when the developer does what they deem correct without following any rules, the term *Cowboy coding* is used [27, p. 34].

## 2.1 Software Development Life Cycle

An implementation of the SDLC consists of two major components. First, the process is broken down into several smaller phases. Depending on the nature of the software, it is possible to omit steps or add more steps. I have compiled a simple yet generic approach from multiple sources [16, 23], to which most software projects adhere. This approach consists of five phases.

1. **Requirements phase:** This is the initial phase of the development process. During this phase, the developer gets acquainted with the project and compiles a list of the desired functionalities [23]. Using this information, the developer eventually decides on the required hardware specifications and possible external software which will need to be acquired.

2. **Design phase:** After the developer has gained sufficient knowledge about the project requirements, they can use this information to draw an architectural design of the application. This design consists of multiple documents, including user stories and UML-diagrams.

3. **Implementation phase:** During this phase, the developer will write code according to the specifications defined in the architectural designs.

4. **Testing phase:** This is the most important phase. During this phase, the implementation is tested to identify potential bugs before the application is used by

other users.

5. **Operational phase:** In the final phase, the project is fully completed and it is integrated in the existing business environment.

Subsequently, a model is chosen to define how to transition from one phase into another phase. A manifold of models exist [16], each having advantages and disadvantages, but I will consider the basic yet most widely used model, which is the Waterfall model by Benington [5]. The initial Waterfall model required every phase to be executed sequentially and in order, cascading. However, this imposes several issues, the most prevalent being the inability to revise design decisions taken in the second phase, when performing the actual implementation in the third phase. To mitigate this, an improved version of the Waterfall model was proposed by Royce [37]. This version allows a phase to transition back to a previous phase (Figure 2.1).

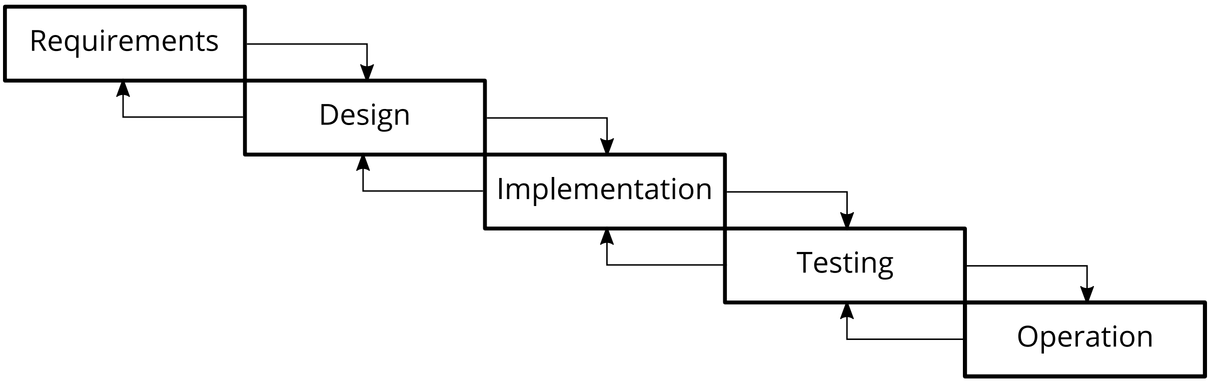


Figure 2.1: Improved Waterfall model by Royce

In this thesis I will solely focus on the implementation and testing phase, as these are the most time-consuming phases of the entire process. The modification to the Waterfall model by Royce is particularly useful when applied to these two phases, in the context of *software regressions*. A regression [34] is a feature that was previously working correctly, but is now malfunctioning. This behaviour can have external causes, such as a change in the system clock because of daylight saving time, but can also be the result of a change to another, seemingly unrelated part of the application code [22].

Software regressions and other functional bugs can ultimately incur disastrous effects, such as severe financial loss or damage to the reputation of the software company. The most famous example in history is without any doubt the explosion of the Ariane 5-rocket, which was caused by an integer overflow [28]. In order to reduce the risk of bugs, malfunctioning components should be detected as soon as possible to proactively defend against potential failures. Because of this reason, the testing phase is to be considered as the most important phase of the entire development process and an application should therefore include sufficient tests. The collection of all tests included in an application, or a smaller chosen subset of certain tests, is referred to as the *test suite*. Tests can be classified in multiple categories, this thesis will consider three distinguishable categories:

1. **Unit test:** This is the most basic kind of test. The purpose of a unit test is to verify the behaviour of an individual component [41]. The scope of a unit test should be limited to a small and isolated piece of code, such as one function. Unit tests are typically implemented as *white-box tests* [22, p. 12]. A white-box test is constructed by manually inspecting the function under test, to identify important *edge values*. The unit test should then feed these values as arguments to the function under test, to observe its behaviour. Common edge cases include zero, negative numbers, empty arrays or array boundaries that might result in an overflow.

2. **Integration test:** A more advanced test, an integration test verifies the interaction between multiple individually tested components [41]. Examples of integration tests include the communication between the front-end and the back-end side of an application. As opposed to unit tests, an integration test is an example of a *black-box* test [22, p. 6], meaning that implementation-specific details should be irrelevant or unknown when writing an integration test.

3. **Regression test:** After a regression has been detected, a regression test [24, p. 372] is added to the test suite. This regression test should replicate the exact conditions and sequence of actions that have caused the regression, to warden the implementation against subsequent failures if the same conditions would reapply in the future.

### 2.1.1 Test Suite Assessment

#### Coverage

The most frequently used metric to measure the quantity and thoroughness of a test suite is the *code coverage* or *test coverage* [24, p. 467]. The test coverage is expressed as a percentage and indicates which fraction of the application code is affected by code in the test suite. Internally, this works by augmenting every statement in the application code using binary instrumentation. A hook is inserted before and after every statement to keep track of which statements are executed during tests. Many different criteria exist to interpret these instrumentation results and thus to express the fraction of covered code [33], the most commonly used ones are *statement coverage* and *branch coverage*.

**Statement coverage** expresses the fraction of code statements that are executed in any test of the test suite [22], out of all executable statements in the application code. Analogously, the fraction of lines covered by a test may be used to calculate the *line coverage* percentage. Since one statement can span multiple lines and one line may also contain more than one statement, both of these criteria implicitly represent the same value. Statement coverage is heavily criticised in literature [33, p. 37], since it is possible to achieve a statement coverage percentage of 100% on a code fragment which can be proven to be incorrect. Consider the code fragment in Listing 2.1. If a test would call the `example`-function with arguments $\{a = 1, b = 2\}$, the test will pass and every statement will be covered, resulting in a statement coverage of 100%. However, it is clear to see that if the function would be called with arguments $\{a = 0, b = 0\}$, a *division-by-zero* error would be raised, resulting in a crash. This very short example already indicates that statement coverage is not trustworthy, yet it may still be useful for other purposes, such as detecting unreachable code which may safely be removed.

```c
int example(int a, int b) {
        if (a == 0 || b != 0) {
                return a / b;
        }
}
```

Listing 2.1: Example of irrelevant statement coverage in C.

**Branch coverage** on the other hand, requires that every branch of a conditional statement is traversed at least once [33, p. 37]. For an `if`-statement, this results in two tests being required, one for every possible outcome of the condition (`true` or `false`). For a `loop`-statement, this requires a test case in which the loop body is never executed and another test case in which the loop body is always executed. Remark that while this criterion is stronger than statement coverage, it is still not sufficiently strong to detect the bug in Listing 2.1. In order to mitigate this, *multiple-condition coverage* [33, p. 40] is used. This criterion requires that for every conditional statement, every possible combination of subexpressions is evaluated at least once. Applied to Listing 2.1, the `if`-statement is only covered if the following four cases are tested, which is sufficient to detect the bug.

- $a = 0, b = 0$
- $a = 0, b \neq 0$
- $a \neq 0, b = 0$
- $a \neq 0, b \neq 0$

It should be self-evident that achieving and maintaining a coverage percentage of 100% at all times is critical. However, this does not necessarily imply that all lines, statements or branches need to be covered explicitly [8]. Some parts of the code might simply be irrelevant or untestable. Examples include wrapper or delegation methods that simply call a library function. All major programming languages have frameworks and libraries available to collect coverage information during test execution, and each of these frameworks allows the developer to exclude parts of the code from the final coverage calculation. As of today, the most popular options are JaCoCo[1] for Java, coverage.py[2] for Python and simplecov[3] for Ruby. These frameworks are able to generate in-depth statistics on which parts of the code are covered and which parts require more tests, as illustrated in Figure 2.3.

### Mutation testing

Whereas code coverage can be used to identify whether or not a part of the code is currently affected by the test suite, *mutation testing* can be used to measure its quality and ability to detect future failures. This technique creates several syntactically different instances of the source code, referred to as *mutants*. A mutant can be created by applying one or more *mutation operators* to the original source code. These mutation operators are aimed at simulating typical mistakes that developers tend to make, such as the introduction of off-by-one errors, removal of statements and replacement of logical connectors [36]. The *mutation order* refers to the amount of mutation operators that have been applied consecutively to an instance of the code. This order is traditionally rather low, as a result of the *Competent Programmer Hypothesis*, which states that programmers develop programs which are near-correct [25].

**Creating and evaluating** the mutant versions of the code is a computationally expensive process and requires human intervention, which is why very few software developers have managed to employ this technique in practice. Figure 2.2 shows how mutation testing is performed. First of all, the mutation system takes the original program $P$ and a set of test cases $T$. Then, several mutation operators are applied to construct a large set of mutants $P'$. The next step is to evaluate every test case $t$ on the original program $P$ to verify its correctness, this is a task that needs to be performed manually. If at least one of these test cases proves incorrect, a bug has been found in the original program, which needs to be resolved before the mutation analysis can continue. When $P$ successfully passes every test case, every test case are evaluated for each of the mutants. A mutant $p'$ is said to be “killed” if its output is different from

[1] `https://www.jacoco.org/jacoco/`

[2] `https://github.com/nedbat/coveragepy`

[3] `https://github.com/colszowka/simplecov`

$P$ for at least one test case, otherwise it is considered “surviving”. After executing all test cases, the set of surviving mutants should be analysed in order to introduce subsequent test cases that can be used to kill them. However, it is also possible that the surviving mutants are functionally equivalent to $P$. This needs to be verified manually, since the detection of program equivalence is impossible [25, 36].

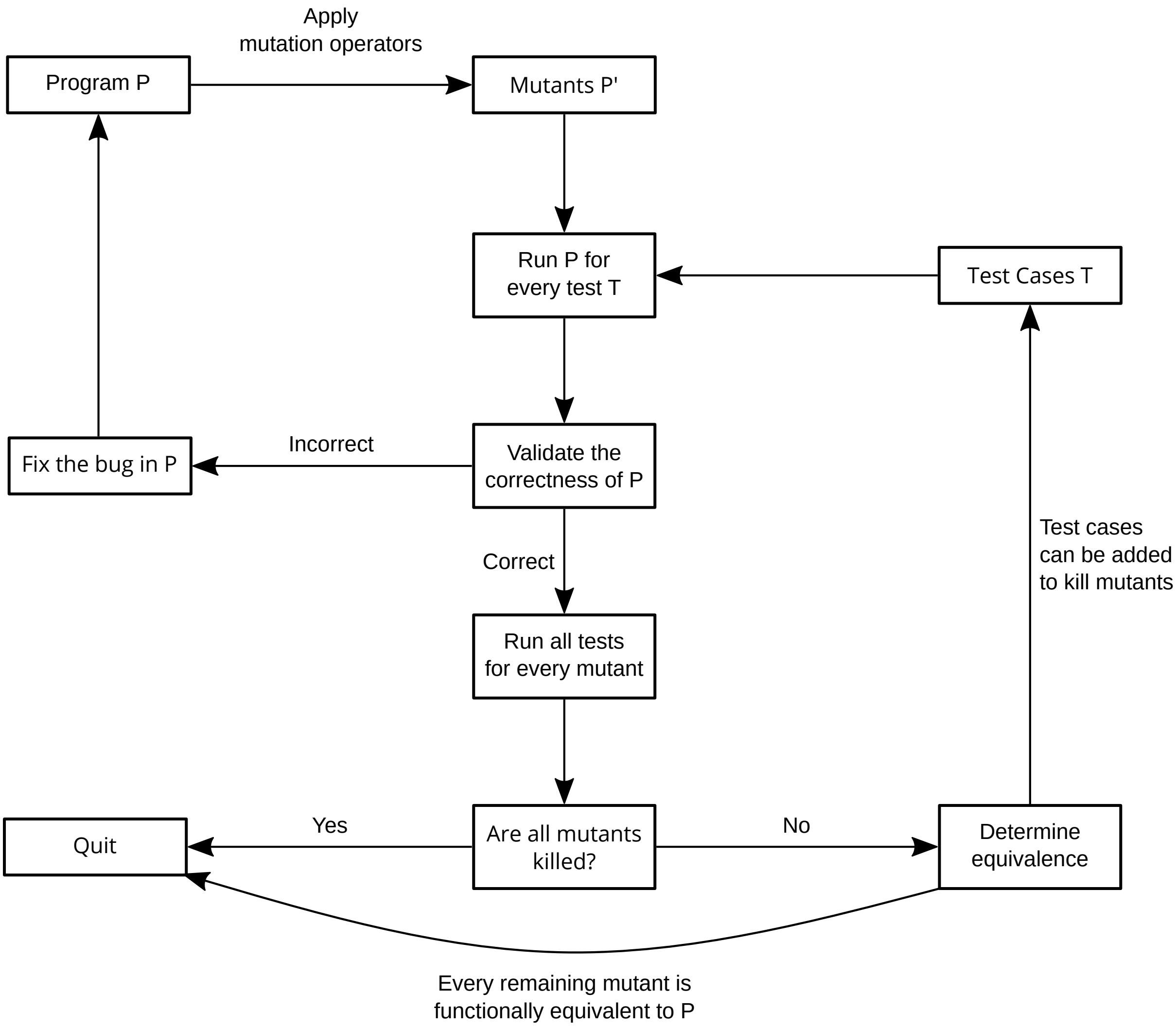
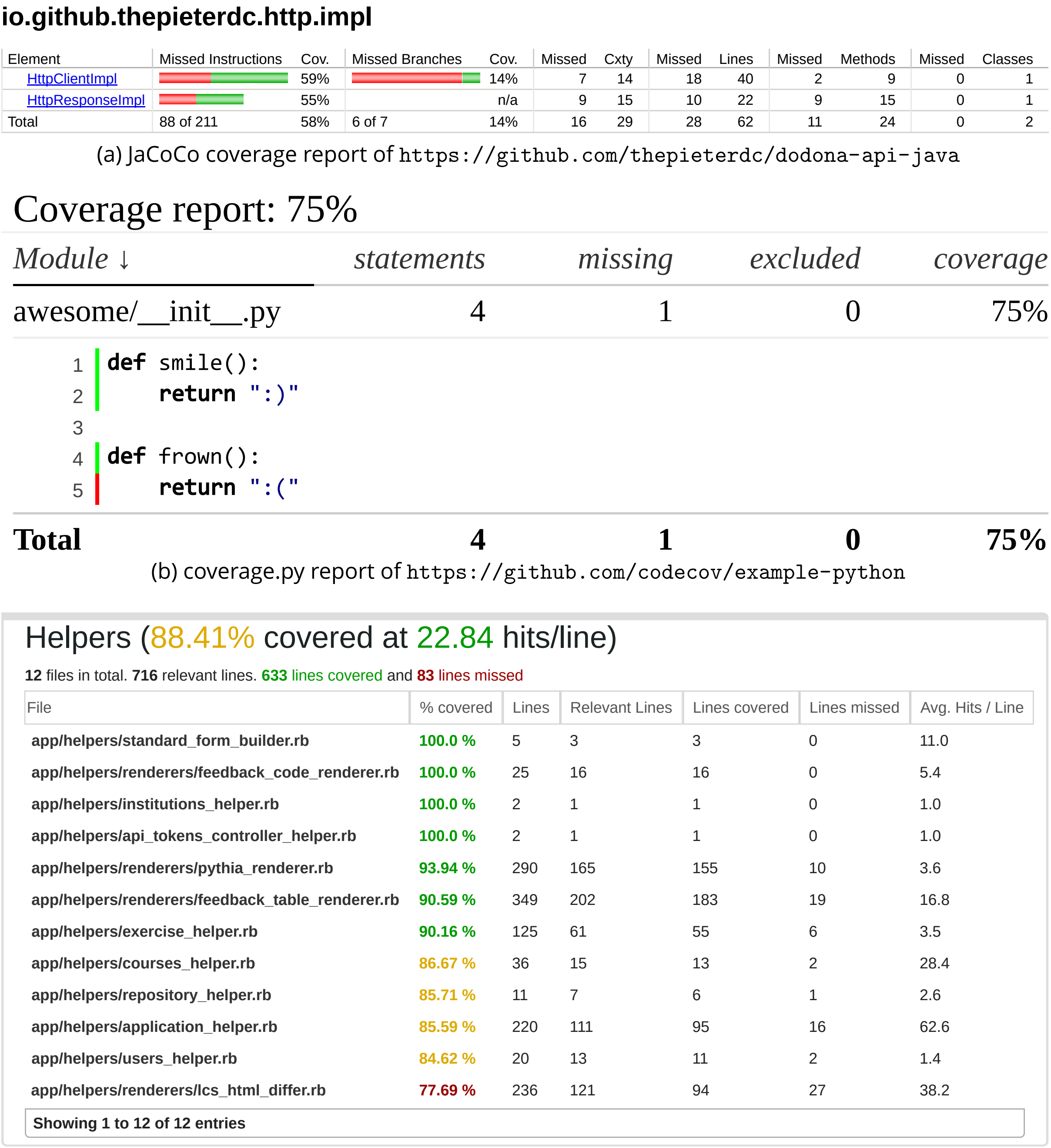


Figure 2.2: Process of Mutation Testing (based on [36])

After every mutant has either been killed or marked equivalent to the original problem, the test suite is assigned a *mutation score* which is calculated using Equation 2.1. In an ideal test suite, this score should be equal to 1, indicating that the test suite was able to detect every mutant.

$$\text{Mutant Score} = \frac{\text{killed mutants}}{\text{non-equivalent mutants}} \tag{2.1}$$



## io.github.thepieterdc.http.impl

| Element | Missed Instructions | Cov. | Missed Branches | Cov. | Missed | Cxty | Missed | Lines | Missed | Methods | Missed | Classes |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| HttpClientImpl | | 59% | | 14% | 7 | 14 | 18 | 40 | 2 | 9 | 0 | 1 |
| HttpResponseImpl | | 55% | | n/a | 9 | 15 | 10 | 22 | 9 | 15 | 0 | 1 |
| Total | 88 of 211 | 58% | 6 of 7 | 14% | 16 | 29 | 28 | 62 | 11 | 24 | 0 | 2 |

(a) JaCoCo coverage report of `https://github.com/thepieterdc/dodona-api-java`

## Coverage report: 75%

| Module ↓ | statements | missing | excluded | coverage |
|---|---|---|---|---|
| awesome/__init__.py | 4 | 1 | 0 | 75% |

```
def smile():
    return ":)"

def frown():
    return ":("
```

| **Total** | **4** | **1** | **0** | **75%** |
|---|---|---|---|---|

(b) coverage.py report of `https://github.com/codecov/example-python`

## Helpers (88.41% covered at 22.84 hits/line)

**12** files in total. **716** relevant lines. **633** lines covered and **83** lines missed

| File | % covered | Lines | Relevant Lines | Lines covered | Lines missed | Avg. Hits / Line |
|---|---|---|---|---|---|---|
| **app/helpers/standard_form_builder.rb** | **100.0 %** | 5 | 3 | 3 | 0 | 11.0 |
| **app/helpers/renderers/feedback_code_renderer.rb** | **100.0 %** | 25 | 16 | 16 | 0 | 5.4 |
| **app/helpers/institutions_helper.rb** | **100.0 %** | 2 | 1 | 1 | 0 | 1.0 |
| **app/helpers/api_tokens_controller_helper.rb** | **100.0 %** | 2 | 1 | 1 | 0 | 1.0 |
| **app/helpers/renderers/pythia_renderer.rb** | **93.94 %** | 290 | 165 | 155 | 10 | 3.6 |
| **app/helpers/renderers/feedback_table_renderer.rb** | **90.59 %** | 349 | 202 | 183 | 19 | 16.8 |
| **app/helpers/exercise_helper.rb** | **90.16 %** | 125 | 61 | 55 | 6 | 3.5 |
| **app/helpers/courses_helper.rb** | **86.67 %** | 36 | 15 | 13 | 2 | 28.4 |
| **app/helpers/repository_helper.rb** | **85.71 %** | 11 | 7 | 6 | 1 | 2.6 |
| **app/helpers/application_helper.rb** | **85.59 %** | 220 | 111 | 95 | 16 | 62.6 |
| **app/helpers/users_helper.rb** | **84.62 %** | 20 | 13 | 11 | 2 | 1.4 |
| **app/helpers/renderers/lcs_html_differ.rb** | **77.69 %** | 236 | 121 | 94 | 27 | 38.2 |

**Showing 1 to 12 of 12 entries**

(c) simplecov report of `https://github.com/dodona-edu/dodona`

Figure 2.3: Statistics from Code coverage tools

## 2.2 Agile Software Development

### 2.2.1 Agile Manifesto

Since the late 1990's, developers have tried to reduce the time occupied by the implementation and testing phases. In order to accomplish this, several new implementations of the SDLC were proposed and evaluated, later collectively referred to as *Agile development methodologies*. The term *Agile development* was coined during a meeting of seventeen prominent software developers, held between February 11-13, 2001, in Snowbird, Utah [20]. As a result of this meeting, the developers defined the four key values and twelve principles that define these new methodologies, called the *Manifesto for Agile Software Development*, also known as the *Agile Manifesto*.

According to the authors, the four key values of Agile software development should be interpreted as follows: "While there is value in the items on the right, we value the items on the left more" [3]. Meyer provides a the following definition for the four values: "general assumptions framing the agile view of the world", while defining the principles as "core agile rules, organizational and technical" [32, p. 2]. Martin identifies the principles as "the characteristics that differentiate a set of agile practices from a heavyweight process" [31, p. 33]. A variety of different programming models, based on the agile ideologies, have arisen since 2001 and each one incorporates these values and principles in their own unique way. I will very briefly explain these values and their corresponding principles, using the mapping proposed by Kiv [26, p. 12].

***Individuals and interactions* over processes and tools**

Instead of meticulously following an outlined development process and utilising the best tools available, the main focus of attention should shift to the people behind the development and how they are interacting with each other. According to Glass, the quality of the programmers and the team is the most influential factor in the successful development of software [15].

**Principle 5: Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done.**
The key to successful software development is ensuring that the people working on the project are both skilled and motivated. Research has shown that, while proficient programmers can cost twice as much as their less-skilled counterparts, their productivity lies between 5 to 30 times higher [15]. Any factor that negatively impacts a healthy environment or decreases motivation should be changed [31, p. 34].

**Principle 6: The most efficient and effective method of conveying information to and within a development team is face-to-face conversation.**
Real-life conversations and human interaction, ideally in an informal setting, should be preferred over forms of digital communication. Direct communication techniques will encourage the developers to raise questions instead of making (possibly) wrong assumptions [11, 15].

**Principle 8: Agile processes promote sustainable development. The sponsors, developers, and users should be able to maintain a constant pace indefinitely.**
The team should aim for a fast, yet sustainable pace instead of rushing to finish the project. This reduces the risk of burnouts and ensures high-quality software will be delivered [31].

**Principle 11: The best architectures, requirements, and designs emerge from self-organizing teams.**
The idea of requiring a hierarchy within a team should be abolished. Every team member must be considered equal and must have input on how to divide the work and the corresponding responsibilities [31]. Subsequently, Fowler and Highsmith state that a minimal amount of process rules and an increase in human interactions has a positive influence on innovation and creativity [11].

**Principle 12: At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behavior accordingly.**
An agile team is versatile and aware that the environment changes continuously, and that they should act accordingly [31]. An important aspect to keep in mind is that the decision on whether to incorporate changes, should be taken by the team itself instead of by an upper hand, since all members share equal responsibilities [15].

#### ***Working software*** **over comprehensive documentation**

The primary goal of software engineering is to deliver a working end product which fulfils the needs of the customer. In order to accomplish this, development should start as soon as possible. Traditional programming models demand a lot of documentation to be written prior to the actual development, which will inevitably lead to inconsistencies between the documentation and the actual application as the project grows and the requirements change [19].

**Principle 1: Our highest priority is to satisfy the customer through early and continuous delivery of valuable software.**
Research has identified a negative correlation between the functionalities of the initial

delivery and the quality of the final release. This implies that the team should strive to deliver a rudimentary version of the project as soon as possible [31], rather than attempting to implement all required features at once.

**Principle 3: Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale.**
In the first principle I have explained the importance of an early, rudimental version of the project. After this initial delivery, new functionality is added in an incremental fashion in subsequent deliveries until all required features are implemented, with the interval between two delivery cycles being as short as possible. An important distinction to make is the difference between a "delivery" and a "release". Deliveries are iterations of the project sent to the customer, while releases are those deliveries that the customer considers suitable for public use [11]. Glass criticises this statement and sees little point in delivering development versions to the customer [15].

**Principle 7: Working software is the primary measure of progress.**
In traditional software development, the progress is measured by the amount of documentation that has been written. This way of measuring is however not representative for the actual completion of the project. Glass gives the example of a team lacking behind on schedule. They can hide their lack of progress by simply writing documentation instead of code, fooling their management [15]. In agile software development, the progress is measured directly by the fraction of completed functionality [31].

**Principle 10: Simplicity –the art of maximizing the amount of work not done– is essential.**
Agile software development tries to realise a minimal working version as soon as possible. In order to achieve this goal, optimal time management is crucial. This imposes two important consequences. First, the developers should only start writing code when the design is thoroughly tested, to avoid having to restart all over again [15]. Secondly, as subsection 2.2.1 will explain, it is possible that the structure of the project can change completely, something which needs to be accounted for when writing the code [31].

#### ***Customer collaboration* over contract negotiation**

In traditional software engineering, the role of the customer is subordinate to the developer. Agile software engineering maintains a different perception of this role, treating both the customer and developers as equal entities. Daily contact between both parties is of vital importance to avoid misunderstandings and a short feedback loop allows the developers to cope with changes in requirements and to ensure that the customer is satisfied with the delivered product [19].

**Principle 4: Business people and developers must work together daily throughout the project.**
Martin: "For a project to be agile, customers, developers and stakeholders must have significant and frequent interaction." [31]. This has already been emphasised before by the principles discussed in subsection 2.2.1. Note that the word "customer" is missing in the definition of this principle. According to Glass, this was done on purpose to make the agile ideas apply to non-business applications as well [15].

#### ***Responding to change*** **over following a plan**

The first step of the aforementioned waterfall model (section 2.1) was to ensure both the customer and the developers have a complete and exhaustive view of the entire application. In reality however, this has proven to be rather difficult and sometimes even impossible. As a result of this, a change in requirements was one of the most common causes of software project failure [15]. Consequently, the agile software development methodologies do not require a complete specification of the final product to be known a priori and stimulate the developers to successfully cope with changes as the application is being developed [19].

**Principle 2: Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage.**
Due to the iterative development approach, agile methodologies are able to implement required changes much earlier in the process, resulting in only a minimal impact on the system [31].

**Principle 9: Continuous attention to technical excellence and good design enhances agility.**
"High quality is the key to high speed", according to Fowler [31]. Code of high quality can only be achieved if the quality of the design is high as well, since this is required to handle changes in requirements. As a consequence, agile programmers should manage a "refactor early, refactor often" approach. While this might not result in a short-term benefit, as no new functionality is added, it definitely has a major impact in the long run and is essential to maintaining agility [11].

### 2.2.2 The need for Agile

In the wake of the world economic crisis, software companies were forced to devote efforts into researching how their overall expenses could be reduced. This research has concluded that in order to reduce financial risks, the *time-to-market* of an application should be as short as possible. In order to accomplish this, further research was conducted, resulting in an increase of attention for agile methodologies in scientific literature [21]. As was previously described in subsection 2.2.1, agile methodologies strive to deliver a minimal version as soon as possible, allowing additional functionality to be added in an incremental fashion. This effectively results in a shorter *time-to-market* and lower costs, since the company can decide to cancel the project much earlier in the process.

In addition to a reduced time-to-market, maintaining an agile workflow has also proven beneficial to the success rate of development. A study performed by The Standish Group revealed that the success rate of agile projects is more than three times higher compared to when traditional methodologies are practised, as illustrated in Figure 2.4.

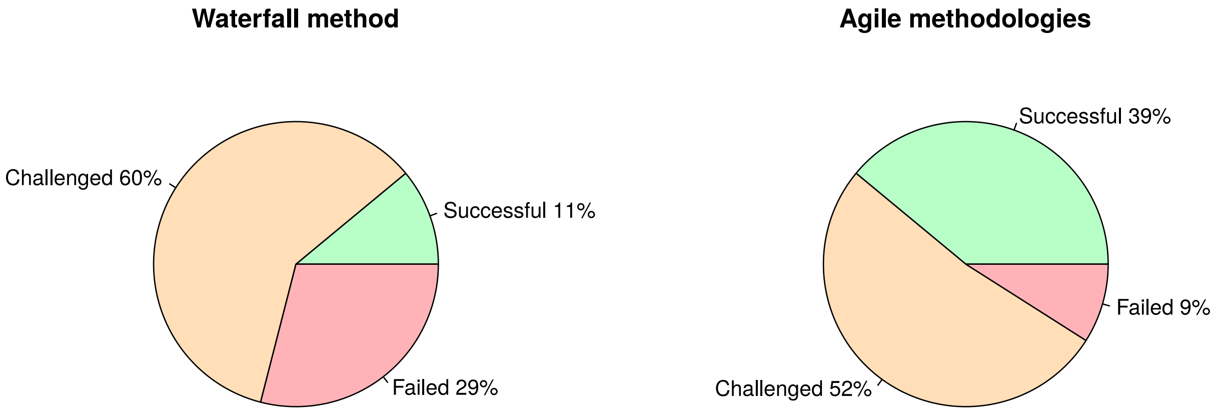


Figure 2.4: Success rate of Agile methodologies [17].

### 2.2.3 Continuous Integration

In traditional software development, the design phase typically leads to a representation of the required functionality in multiple, stand-alone modules. Subsequently, every module is implemented separately by individual developers. Afterwards, an attempt is made to integrate all the modules into the final application, an event to which Meyer refers to as the “Big Bang” [32, p.103]. The name *Big Bang* reflects the complex nature of this operation. This can prove to be a challenging operation, because every developer can take unexpected assumptions at the start of the project, which may ultimately result in mutually incompatible components. Furthermore, since the code was written over a span of several weeks to months, the developers often need to rewrite code that they have not touched in a long time. Eventually this will lead to unanticipated delays and costs [38].

Contrarily, agile development methodologies advocate the idea of frequent, yet small deliveries (subsection 2.2.1). Consequently, this implies that the code is built often and that the modules are integrated multiple times, on a *continuous* basis, rather than just once at the end, thus allowing for early identification of problems [14]. This practice of frequent builds is referred to as *Continuous Integration* [31, 32]. It should be noted that this idea has existed and has been applied before the agile manifesto was written. The first notorious software company that has adopted this practice is Microsoft, already in 1989 [7, p.11]. Cusumano reports that Microsoft typically builds the entire application at least once per day [7, p.12], therefore requiring developers to integrate and test their changes multiple times per day.

The introduction of Continuous Integration `[CI]` in software development has important consequences on the life cycle. Where the waterfall model used a cascading life cycle, Continuous Integration employs a circular, repetitive structure consisting of three phases, as visualised in Figure 2.5.

1. **Implementation:** In the first phase, every developer individually writes code for the module they were assigned to. At a regular interval, the code is committed to the remote repository.

2. **Integration:** When the code is committed, the developer simultaneously fetches the changes to other modules. Afterwards, the developer must integrate the changes with his own module, to ensure compatibility. In case a conflict occurs, the developer is responsible for its resolution [31].

3. **Test:** After the module has successfully been integrated, the test suite is run to ensure no bugs have been introduced.

Adopting Continuous Integration can prove to be a lengthy and repetitive task. Luckily, a variety of tools and frameworks exist to automate this process. Essentially, these tools are typically attached to a version control system (e.g. Git, Mercurial, . . . ), using a `post-receive` hook. Every time a commit is pushed by one of the developers, the CI system is notified, after which the code is automatically built and tests are executed. Optionally, the system can be configured to automatically publish successful runs to the end users, a process referred to as *Continuous Delivery*. I will now proceed by discussing four prominent Continuous Integration systems.

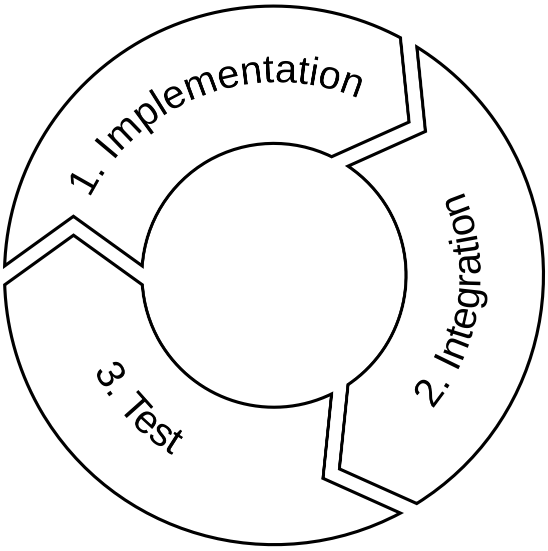


Figure 2.5: Development Life Cycle with Continuous Integration

### Jenkins

Jenkins CI[4] was started as a hobby project in 2004 by Kohsuke Kawaguchi, a former employee of Sun Microsystems. Jenkins is programmed in Java and profiles itself as “The leading open source automation server”. It was initially named Hudson, but after Sun was acquired by Oracle, issues related to the trademark Hudson arose. In response, the developer community decided to migrate the Hudson code to a new repository and rename the project to Jenkins [38]. As of today, Jenkins is still widely used for many reasons. Since it is open source and its source code is located on GitHub, it is free to use and can be self-hosted by the developers in a private environment. Furthermore, Jenkins provides an open ecosystem to support developers into writing new plugins and extending its functionality. A market research conducted by ZeroTurnaround in 2016 revealed that Jenkins is the preferred Continuous Integration tool by 60% of the developers [29].

Figure 2.6: Logo of Jenkins CI (`https://jenkins.io/`)

[4] `https://jenkins.io/`

### GitHub Actions

Following the successful beta of GitHub Actions which had started in August 2019, GitHub launched its own Continuous Integration system later that year in November[5]. GitHub Actions executes builds in the cloud on servers owned by GitHub and can therefore only be used in conjunction with a GitHub repository, support for GitHub Enterprise repositories is not yet available. The developers can define builds using *workflows* that can be configured to run both on Linux, Windows as well as OSX hosts. Private repositories are allowed a fixed amount of free build minutes per month, while builds of public repositories are always free of charges [10]. Similar to Jenkins, GitHub Actions can be extended with custom plugins. These plugins can be created using either a Docker container, or in native JavaScript, which allows faster execution [1]. It should be noted however that due to the recent nature of this system, not many plugins have been created yet.

Figure 2.7: Logo of GitHub Actions (`https://github.com/features/actions`)

### GitLab CI

GitLab, the main competitor of GitHub, announced its own Continuous Integration service in late 2012 named GitLab CI[6]. GitLab CI builds are configured using *pipelines* and are executed by *GitLab Runners*. These runners are operated by developers on their own infrastructure. Additionally, GitLab also offers the possibility to use *shared runners*, which are hosted by themselves [2]. Equivalent to the aforementioned GitHub Actions, shared runners can be used for free by public repositories and are bounded by quota for private repositories [13]. A downside of using GitLab CI is the lack of a community-driven plugin system, however this is a planned feature [7].

Figure 2.8: Logo of GitLab CI (`https://gitlab.com/`)

[5] https://github.blog/2019-08-08-github-actions-now-supports-ci-cd/

[6] https://about.gitlab.com/blog/2012/11/13/continuous-integration-server-from-gitlab/

[7] https://gitlab.com/gitlab-org/gitlab/issues/15067

### Travis CI

The final Continuous Integration platform which I will discuss is Travis CI. This Continuous Integration system was launched in 2011 and can only be used in addition to an existing GitHub repository. Travis CI build tasks can be configured in a similar fashion as GitLab CI, but the builds can exclusively be executed on their servers. Besides running builds after a commit has been pushed to the repository, it is also possible to schedule daily, weekly or monthly builds using cron jobs. Similar to GitHub Actions, open-source projects can be built at zero cost and a paid plan exists for private repositories [9]. It is not possible to create custom plugins, however Travis CI already features built-in support for a variety of programming languages. In 2020, almost 1 million projects are being built using Travis CI [39].

Figure 2.9: Logo of Travis CI (`https://travis-ci.com/`)

# Chapter 3

# Related work

In the previous chapter, we have stressed the paramount importance of frequently integrating one's changes into the upstream repository. This process can prove to be a complex and lengthy operation. As a result, software engineers have sought and found ways to automate this task. These solutions and practices embody Continuous Integration (CI). However, CI is not the golden bullet for software engineering, as there is a flip side to applying this practice. After every integration, we must execute the entire test suite to ensure that we have not introduced any regressions. As the project evolves and the size of the codebase increases, the number of test cases will increase accordingly to preserve a sufficiently high coverage level [35]. Walcott, Soffa and Kapfhammer illustrate the magnitude of this problem by providing an example of a project consisting of 20 000 lines of code, whose test suite requires up to seven weeks to complete [40].

Fortunately, developers and researchers have found multiple techniques to address the scalability issues of ever-growing test suites. We can classify the techniques currently known in literature into three categories [35]. These categories are Test Suite Minimisation (TSM), Test Case Selection (TCS) or Test Case Prioritisation (TCP). We can apply each technique to every test suite, but the outcome will be different. TSM and TCS will have an impact on the execution time of the test suite, at the cost of a reduced test coverage level. In contrast, TCP will have a weaker impact on the execution time but will not affect the test adequacy.

The following sections will discuss these three approaches in more detail and provide accompanying algorithms. Because the techniques are very similar, the corresponding algorithms can (albeit with minor modifications) be used interchangeably for every approach. The final section of this chapter will investigate the adoption and integration of these techniques in modern software testing frameworks.

## 3.1 Classification of approaches

### 3.1.1 Test Suite Minimisation

The first technique is called Test Suite Minimisation (TSM), also referred to as *Test Suite Reduction* in literature. This technique will try to reduce the size of the test suite by permanently removing redundant test cases. This problem has been formally defined by Rothermel [42] in definition 1 and illustrated in Figure 3.1.

**Definition 1** (Test Suite Minimisation)**.**
*Given:*

- $T = \{t_1, \ldots, t_n\}$ *a test suite consisting of test cases* $t_j$*.*
- $R = \{r_1, \ldots, r_m\}$ *a set of requirements that must be satisfied in order to provide the desired "adequate" testing of the program.*
- $\{T_1, \ldots, T_m\}$ *subsets of test cases in* $T$*, one associated with each of the requirements* $r_i$*, such that any one of the test cases* $t_j \in T_i$ *can be used to satisfy requirement* $r_i$*.*

*Subsequently, we can define Test Suite Minimisation as the task of finding a subset* $T'$ *of test cases* $t_j \in T$ *that satisfies every requirement* $r_i$*.*

If we apply the concepts of the previous chapter to the above definition, we can interpret the set of requirements $R$ as source code lines that must be covered. A requirement $r_i$ can subsequently be satisfied by any test case $t_j \in T$ that belongs to the subset $T_i$. Observe that the problem of finding $T'$ is closely related to the *hitting set problem* (definition 2) [42].

**Definition 2** (Hitting Set Problem)**.**
*Given:*

- $S = \{s_1, \ldots, s_n\}$ *a finite set of elements.*
- $C = \{c_1, \ldots, c_n\}$ *a collection of sets, with* $\forall c_i \in C : c_i \subseteq S$*.*
- $K$ *a positive integer,* $K \leq |S|$*.*

*The hitting set is a subset* $S' \subseteq S$ *such that* $S'$ *contains at least one element from each subset in* $C$*.*

In the context of Test Suite Minimisation, $T'$ corresponds to the hitting set of $T_i$s. In order to effectively minimise the amount of tests in the test suite, $T'$ should be the minimal hitting set [42]. Since we can reduce this problem to the NP-complete *Vertex Cover*-problem, we know that this problem is NP-complete as well [12].

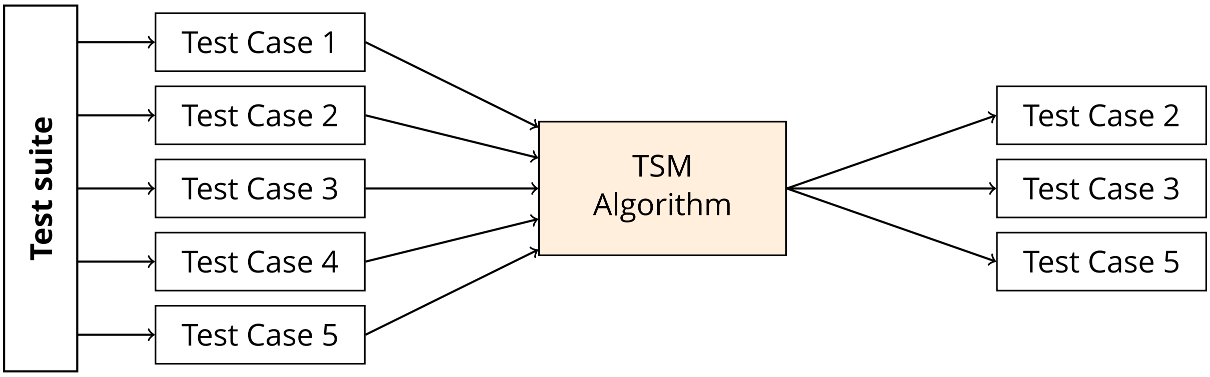


Figure 3.1: Test Suite Minimisation

### 3.1.2 Test Case Selection

The second approach closely resembles the previous one. However, instead of permanently removing redundant test cases, Test Case Selection (TCS) has a notion of context. In this algorithm, we will not calculate the minimal hitting set at runtime, but before executing the test suite, we will perform a *white-box static analysis* of the source code. This analysis identifies which parts of the source code have been changed and executes only the corresponding test cases. Subsequent executions of the test suite will require a new analysis, thus making the selection temporary (Figure 3.2) and modification-aware [42]. Rothermel and Harrold define this formally in definition 3.

**Definition 3** (Test Case Selection)**.**
*Given:*

- $P$ *the previous version of the codebase*
- $P'$ *the current (modified) version of the codebase*
- $T$ *the test suite*

*Test Case Selection aims to find a subset* $T' \subseteq T$ *that is used to test* $P'$*.*

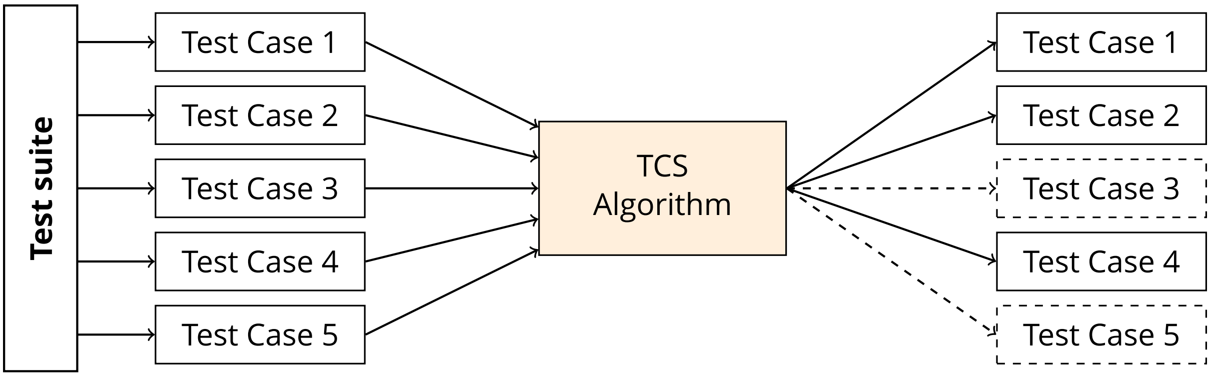


Figure 3.2: Test Case Selection

### 3.1.3 Test Case Prioritisation

Both TSM and TCS attempt to execute as few tests as possible to reduce the execution time of the test suite. Nevertheless, in some cases, we may require to execute every test case to guarantee correctness. In this situation, we can still optimise the test suite. Test Case Prioritisation (TCP) aims to find a permutation of the sequence of test cases, rather than eliminating specific tests from being executed (Figure 3.3). We choose the order of the permutation in such a way that we can complete a predefined objective as soon as possible. Once we have achieved our objective, we can early terminate the execution of the test suite. In the worst-case scenario, we will still execute every test case. Some examples of objectives include covering as many lines of code as fast as possible or executing tests ordered on their probability of failure [42]. Definition 4 provides a formal definition of this approach.

**Definition 4** (Test Case Prioritisation)**.**
*Given:*

- $T$ *the test suite*
- $PT$ *the set of permutations of* $T$
- $f : PT \mapsto \mathbb{R}$ *a function from a subset to a real number, this function is used to compare sequences of test cases to find the optimal permutation.*

*Test Case Prioritisation finds a permutation* $T' \in PT$ *such that* $\forall T'' \in PT : f(T') \geq f(T'') \Rightarrow (T'' \neq T')$

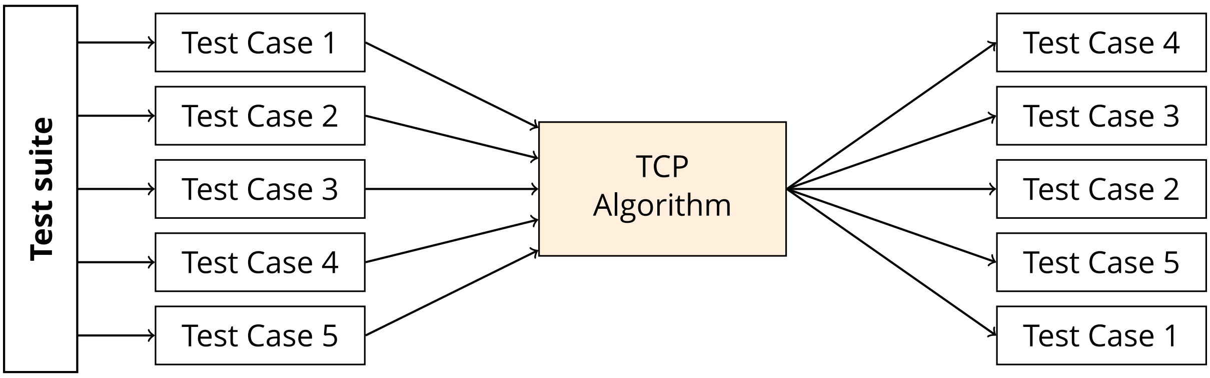


Figure 3.3: Test Case Prioritisation

## 3.2 Algorithms

TCP is essentially an extended version of TSM since we can first execute the minimised test suite and afterwards the remaining test cases. Additionally, section 3.1.1 has explained that TSM is an instance of the minimal hitting set problem, which is an NP-complete problem. Consequently, we know that both TSM and TCP are NP-complete problems as well and therefore, we require the use of *heuristics*. A heuristic is an experience-based method that can be applied to solve a hard to compute problem by finding a fast approximation [22]. However, the found solution will mostly be sub-optimal, or sometimes the algorithm might even fail to find any solution at all. Given the relation between TSM and the minimal hitting set problem, we can implement an optimisation algorithm by modifying any known heuristic that finds the minimal hitting set. This paper will now proceed by discussing a selection of these heuristics. The used terminology and the names of the variables have been changed to ensure mutual consistency between the algorithms. Every algorithm has been adapted to adhere to the conventions provided in definitions 5 and 6.

**Definition 5** (Naming convention)**.**

- $TS = \{T_1, \ldots, T_n\}$*: the set of all test cases* $t$ *in the test suite.*
- $RS = \{T_1, \ldots, T_n\} \subseteq TS$*: the representative set of test cases* $t$ *that have been selected by the algorithm.*
- $C = \{c_1, \ldots, c_m\}$*: the set of all source code lines in the application, that are covered by at least one test case* $T \in TS$*.*
- $CT = \begin{bmatrix} CT_1 & \ldots & CT_m \end{bmatrix}$*: the list of test groups.*
  - $CT_c = \{T_1, \ldots, T_n\} \subseteq TS$*: the test group* $c$*, which corresponds to the set of all test cases* $T \in TS$ *that cover the source code line* $c \in C$*.*
- $TL = \begin{bmatrix} TL_1 & \ldots & TL_n \end{bmatrix}$*: the list of coverage groups.*
  - $TL_t = \{c_1, \ldots, c_m\} \subseteq C$*: the set of all source code lines* $c \in C$ *that are covered by test case* $t \in TS$*.*

**Definition 6** (Cardinality)**.** *For a finite set* $S$*, the cardinality* $|S|$ *is defined as the number of elements in* $S$*. In case of potential confusion, we can use* $Card(S)$ *to denote the cardinality of* $S$*.*

### 3.2.1 Greedy algorithm

The first algorithm is a *greedy* heuristic, which was initially designed by Chvatal to find an approximation for the set-covering problem [35]. A greedy algorithm always makes a locally optimal choice, assuming that this will eventually lead to a globally optimal solution [6]. Algorithm 1 presents the Greedy algorithm for Test Suite Minimisation. The objective of the algorithm is to construct a set of test cases that cover every line in the code, by requiring as few test cases as possible.

Initially, the algorithm starts with an empty representative set $RS$, the set $TS$ of all test cases and the set $C$ of all coverable source code lines. Furthermore, $TL$ denotes the set of coverage groups as specified in the definition. In essence, the algorithm will iteratively select test cases from $TS$ and add them to $RS$. The locally optimal choice is always to select the test case that will contribute the most still uncovered lines, ergo the test case $t$ for which the cardinality of the intersection between $C$ and $TL_t$ is maximal. After every iteration, we remove the code lines $TL_t$ from $C$, since these are now covered. We repeat this selection process until $C$ is empty, which indicates that we have covered every source code line. Afterwards, when we execute the test suite, we only need to execute test cases in $RS$. We can apply this algorithm to Test Case Prioritisation as well, by changing the type of $RS$ to a list instead. We require a list to maintain the insertion order since this is equivalent to the ideal order of execution.

**Algorithm 1** Greedy algorithm for Test Suite Minimisation

**Input:** the test suite $TS$, all coverable lines $C$, the list of coverage groups $TL$
**Output:** representative set $RS \subseteq TS$ of test cases to execute

```
procedure GreedyTSM($TS, C, TL$)
    $RS \leftarrow \emptyset$
    while $C \neq \emptyset$ do
        $t\_max \leftarrow 0$
        $tl\_max \leftarrow \emptyset$
        for all $t \in TS$ do
            $tl\_current \leftarrow C \cap TL[t]$
            if $|tl\_current| > |tl\_max|$ then
                $t\_max \leftarrow t$
                $tl\_max \leftarrow tl\_current$
        $RS \leftarrow RS \cup \{t\_max\}$
        $C \leftarrow C \setminus tl\_max$
    return $RS$
```

### 3.2.2 HGS

The second algorithm is the HGS algorithm. The algorithm was named after its creators Harrold, Gupta and Soffa [18]. Similar to the Greedy algorithm (section 3.2.1), this algorithm will also iteratively construct the minimal hitting set. However, instead of considering the coverage groups $TL$, the algorithm uses the test groups $CT$. More specifically, we will use the distinct test groups, denoted as $CTD$. We consider two test groups $CT_i$ and $CT_j$ as distinct if they differ in at least one test case. The pseudocode for this algorithm is provided in Algorithm 2.

The algorithm consists of two main phases. The first phase begins by constructing an empty representative set $RS$ in which we will store the selected test cases. Subsequently, we iterate over every source code line $c \in C$ to create the corresponding test groups $CT$. As mentioned before, we will reduce this set to $CTD$ for performance reasons and as such, only retain the distinct test groups. Next, we select every test group of which the cardinality is equal to 1 and add these to $RS$. The representative set will now contain every test case that covers precisely one line of code, which is exclusively covered by that single test case. Afterwards, we remove every covered line from $C$. The next phase consists of repeating this process for increasing cardinalities until $C$ is empty. However, since the test groups will now contain more than one test case, we need to make a choice on which test case to select. The authors prefer the test case that covers the most remaining lines. In the event of a tie, we defer the choice until the next iteration.

The authors have provided an accompanying calculation of the computational time complexity of this algorithm [18]. In addition to the naming convention introduced in definition 5, let $n$ denote the number of distinct test groups $CTD$, $nt$ the number of test cases $t \in TS$ and $MAX\_CARD$ the cardinality of the test group with the most test cases. In the HGS algorithm we need to perform two steps repeatedly. The first step involves computing the number of occurrences of every test case $t$ in each test group. Given that there are $n$ distinct test groups and, in the worst-case scenario, each test group can contain $MAX\_CARD$ test cases which we all need to examine once, the computational cost of this step is equal to $O(n * MAX\_CARD)$. For the next step, in order to determine which test case we should include in the representative set $RS$, we need to find all test cases for which the number of occurrences in all test groups is maximal, which requires at most $O(nt * MAX\_CARD)$. Since every repetition of these two steps adds a test case that belongs to at least one out of $n$ test groups to the representative set, the overall runtime of the algorithm is $O(n*(n+nt)*MAX\_CARD)$.

**Algorithm 2** HGS algorithm ([18])

```
Input: distinct test groups $CTD$, total amount of test cases $nt = Card(TS)$
Output: representative set $RS \subseteq TS$ of test cases to execute
function SelectTest($CTD, nt, MAX\_CARD, size, list, marked$)
    $count \leftarrow array[1 \dots nt]$                                    ▷ initially $0$
    for all $t \in list$ do
        for all $group \in CTD$ do
            if $t \in group \wedge \neg marked[group] \wedge Card(group) = size$ then
                $count[t] \leftarrow count[t] + 1$
    $max\_count \leftarrow$ MAX($count$)
    $tests \leftarrow \{t | t \in list \wedge count[t] = max\_count\}$
    if $|tests| = 1$ then return $tests[0]$
    else if $|tests| = MAX\_CARD$ then return $tests[0]$
    else return SelectTest($CTD, nt, MAX\_CARD, size + 1, tests, marked$)
procedure HGSTSM($CTD, nt$)
    $n \leftarrow Card(CTD)$
    $marked \leftarrow array[1 \dots n]$                                  ▷ initially $false$
    $MAX\_CARD \leftarrow$ MAX($\{Card(group) | group \in CTD\}$)
    $RS \leftarrow \bigcup \{singleton | singleton \in CTD \wedge Card(singleton) = 1\}$
    for all $group \in CTD$ do
        if $group \cap RS \neq \emptyset$ then
            $marked[group] \leftarrow true$
    $current \leftarrow 1$
    while $current < MAX\_CARD$ do
        $current \leftarrow current + 1$
        $list \leftarrow \{t | t \in grp \wedge grp \in CTD \wedge Card(grp) = current \wedge \neg marked[grp]\}$
        while $list \neq \emptyset$ do
            $next \leftarrow$ SelectTest($current, list$)
            $reduce \leftarrow false$
            for all $group \in CTD$ do
                if $next \in group$ then
                    $marked[group] = true$
                    if $Card(group) = MAX\_CARD$ then
                        $reduce \leftarrow true$
            if $reduce$ then
                $MAX\_CARD \leftarrow$ MAX($\{Card(grp) | grp \in CTD \wedge \neg marked[grp]\}$)
            $RS \leftarrow RS \cup \{next\}$
            $list \leftarrow \{t | t \in grp \wedge grp \in CTD \wedge Card(grp) = current \wedge \neg marked[grp]\}$
    return $RS$
```

### 3.2.3 ROCKET algorithm

The third and final algorithm is the ROCKET algorithm. This algorithm has been presented by Marijan, Gotlieb and Sen [30] as part of a case study to improve the testing efficiency of industrial video conferencing software. Contrarily to the previous algorithms, which attempted to execute as few test cases as possible, this algorithm does execute the entire test suite. Unlike the previous algorithms that only take code coverage into account, this algorithm also considers historical failure data and test execution time. The objective of this algorithm is twofold: select the test cases with the highest successive failure rate, while also maximising the number of executed test cases in a limited time frame. In the implementation below, we will consider an infinite time frame as this is a domain-specific constraint and irrelevant for this thesis. This algorithm will yield a total ordering of all the test cases in the test suite, ordered using a weighted function.

The modified version of the algorithm (of which the pseudocode is provided in Algorithm 3) takes three inputs:

- $TS = \{T_1, \ldots, T_n\}$: the set of test cases to prioritise.
- $E = \begin{bmatrix} E_1 & \ldots & E_n \end{bmatrix}$: the execution time of each test case.
- $F = \begin{bmatrix} F_1 & \ldots & F_n \end{bmatrix}$: the failure statuses of each test case.
    - $F_t = \begin{bmatrix} f_1 & \ldots & f_m \end{bmatrix}$: the failure status of test case $t$ over the previous $m$ successive executions. $F_{ij} = 1$ if test case $i$ has failed in execution $(current - j)$, $0$ if it has passed.

The algorithm starts by creating an array $P$ of length $n$, which contains the priority of each test case. The priority of each test case is initialised at zero. Next, we construct an $m \times n$ failure matrix $MF$ and fill it using the following formula.

$$MF[i,j] = \begin{cases} 1 & \text{if } F_{ji} = 1 \\ -1 & \text{otherwise} \end{cases}$$

Table 3.1 contains an example of this matrix $MF$. In this table, we consider the hypothetical failure rates of the last two executions of six test cases.

| **run** | $T_1$ | $T_2$ | $T_3$ | $T_4$ | $T_5$ | $T_6$ |
|---|---|---|---|---|---|---|
| $current - 1$ | $1$ | $1$ | $1$ | $1$ | $-1$ | $-1$ |
| $current - 2$ | $-1$ | $1$ | $-1$ | $-1$ | $1$ | $-1$ |

Table 3.1: Example of the failure matrix $MF$.

Afterwards, we fill $P$ with the cumulative priority of each test case. We can calculate the priority of a test case by multiplying its failure rate with a domain-specific weight heuristic $\omega$. This heuristic reflects the probability of repeated failures of a test case, given earlier failures. In their paper [30], the authors apply the following weights:

$$\omega_i = \begin{cases} 0.7 & \text{if } i = 1 \\ 0.2 & \text{if } i = 2 \\ 0.1 & \text{if } i >= 3 \end{cases}$$

$$P_j = \sum_{i=1 \dots m} MF[i,j] * \omega_i$$

Finally, the algorithm groups test cases based on their calculated priority in $P$. Every test case that belongs to the same group is equally relevant for execution in the current test run. However, within every test group, the test cases will differ in execution time $E$. The final step is to reorder test cases that belong to the same group in such a way that test cases with a shorter duration are executed earlier in the group.

**Algorithm 3** ROCKET algorithm

**Input:** the test suite $TS$, the execution times of the test cases $E$, the amount of previous executions to consider $m$, the failure statuses $F$ for each test case over the previous $m$ executions

**Output:** priority $P$ of the test cases

```
procedure ROCKETTCP($TS, E, m, F$)
    $n \leftarrow Card(TS)$
    $P \leftarrow array[1 \dots n]$                                  ▷ initially $0$
    $MF \leftarrow array[1 \dots m]$
    for all $i \in 1 \dots m$ do
        $MF[i] \leftarrow array[1 \dots n]$
        for all $j \in 1 \dots n$ do
            if $F[j][i] = 1$ then $MF[i][j] \leftarrow -1$
            else $MF[i][j] \leftarrow 1$
    for all $j \in 1 \dots n$ do
        for all $i \in 1 \dots m$ do
            if $i = 1$ then $P[j] \leftarrow P[j] + (MF[i][j] * 0.7)$
            else if $i = 2$ then $P[j] \leftarrow P[j] + (MF[i][j] * 0.2)$
            else $P[j] + (MF[i][j] * 0.1)$
    $Q \leftarrow \{P[j] | j \in 1 \dots n\}$                       ▷ distinct priorities
    $G \leftarrow array[1 \dots Card(Q)]$                          ▷ initially empty sets
    for all $j \in 1 \dots n$ do
        $p \leftarrow P[j]$
        $G[p] \leftarrow G[p] \cup \{j\}$
    Sort every group in $G$ based on ascending execution time in $E$.
    Sort $P$ according to which group it belongs and its position within that group.
    return $P$
```

## 3.3 Adoption in testing frameworks

In the final section of this chapter, we will investigate how existing software testing frameworks have implemented these and other optimisation techniques.

### 3.3.1 Gradle and JUnit

Gradle[1] is a dependency manager and development suite for Java, Groovy and Kotlin projects. It supports multiple plugins to automate tedious tasks, such as configuration management, testing and deploying. One of the supported testing integrations is JUnit[2], which is the most widely used testing framework by Java developers. JUnit 5 is the newest version which is still under active development as of today. Several prominent Java libraries and frameworks, such as Android and Spring have integrated JUnit as the preferred testing framework. The testing framework offers mediocre support for features that optimise the execution of the test suite, primarily when used in conjunction with Gradle. The following three key elements are available:

1. **Parallel test execution:** The Gradle implementation of JUnit features multiple *test class processors*. A test class processor is a component which processes Java classes to find all the test cases, and eventually to execute them. One of these processors is the `MaxNParallelTestClassProcessor`, which is capable of running a configurable amount of test cases in parallel. Concurrently executing the test cases results in a significant speed-up of the overall test suite execution.

2. **Prioritise failed test cases:** Gradle provides a second useful test class processor: the `RunPreviousFailedFirstTestClassProcessor`. This processor will prioritise test cases that have failed in the previous run. This practice is similar to the ROCKET-algorithm (section 3.2.3), but the processor does not take into account the duration of the test cases.

3. **Test order specification:** JUnit allows us to specify the sequence in which it will execute the test cases. By default, it uses a random yet deterministic order[3]. The order can be manipulated by annotating the test class with the `@TestMethodOrder`-annotation, or by applying the `@Order(int)`-annotation to an individual test case. However, we can only use this feature to alter the order of test cases within the same test class. JUnit does not support inter-test class reordering. We could use this feature to (locally) sort test cases based on their execution time.

[1] https://gradle.org

[2] https://junit.org

[3] https://junit.org/junit5/docs/current/user-guide/#writing-tests-test-execution-order

Figure 3.4: Logo of Gradle

Figure 3.5: Logo of JUnit 5

### 3.3.2 Maven Surefire

A commonly used alternative to Gradle is Apache Maven[4]. This framework also supports executing JUnit test cases using the Surefire plugin. As opposed to Gradle, Surefire does offer multiple options to specify the order in which the test cases will be executed using the `runOrder` property. Without any configuration, Maven will run the test cases in alphabetical order. By switching the `runOrder` property to `failedFirst`, we can tell Maven to prioritise the previously failed test cases. Another supported value is `balanced`, which orders test cases based on their duration. Finally, we can choose to implement a custom ordering scheme for absolute control.

Figure 3.6: Logo of Maven

### 3.3.3 OpenClover

OpenClover[5] is a code coverage tool for Java and Groovy projects. It was created by Atlassian and open-sourced in 2017. OpenClover profiles itself as “the most sophisticated code coverage tool”, by extracting useful metrics from the coverage results and by providing features that can optimise the test suite. These features include powerful integrations with development software and prominent Continuous Integration systems. Furthermore, OpenClover can automatically analyse the coverage results to detect relations between the application source code and the test cases. This feature allows OpenClover to predict which test cases will have been affected, given a set of modifications to the source code. Subsequently, we can interpret these predictions to implement Test Case Selection and therefore reduce the test suite execution time.

Figure 3.7: Logo of Atlassian Clover

[4] http://maven.apache.org/

[5] https://openclover.org

# Chapter 4

# Proposed framework: VeloCIty [TODO REVISE]

The implementation part of this thesis consists of a framework and a set of tools, tailored at optimising the test suite as well as providing accompanying metrics and insights. The framework was named *VeloCIty* to reflect its purpose of enhancing the speed at which Continuous Integration is practised. This paper will now proceed by describing the design goals of the framework. Afterwards, a high-level schematic overview of the implemented architecture will be provided, followed by a more in-depth explanation of every pipeline step. In the final section of this chapter, the *Alpha* algorithm will be presented.

## 4.1 Design goals

VeloCIty has been implemented with four design goals in mind:

1. **Extensibility:** It should be possible and straightforward to support additional Continuous Integration systems, programming languages and test frameworks. Subsequently, a clear interface should be provided to integrate additional prioritisation algorithms.

2. **Minimally invasive:** Integrating VeloCIty into an existing test suite should not require drastic changes to any of the test cases.

3. **Language agnosticism:** This design goal is related to the framework being extensible. The implemented tools should not need to be aware of the programming language of the source code, nor the used test framework.

4. **Self-improvement:** The prioritisation framework supports all of the algorithms presented in section 3.2. It is possible that the performance of a given algorithm is strongly dependent on the nature of the project it is being applied to. In order to facilitate this behaviour, the framework should be able to measure the performance of every algorithm and “learn” which algorithm offers the best prediction, given a set of source code.

## 4.2 Architecture

The architecture of the VeloCIty framework consists of seven steps that are performed sequentially in a pipeline fashion, as illustrated in the sequence diagram (Figure 4.1). Every step is executed by one of three individual components, which will now be introduced briefly.

### 4.2.1 Agent

The first component that will be discussed is the agent. This is the only component that depends actively on both the programming language, as well as the used test framework, since it must interact directly with the source code and test suite. For every programming language or test framework that needs to be supported, a different implementation of an agent must be provided. This implementations are however strongly related, so much code can be reused or even shared. In this thesis, an agent was implemented in Java, more specifically as a plugin for the widely used Gradle and JUnit test framework. This combination was previously described in subsection 3.3.1. This plugin is responsible for running the test suite in a certain prioritised order, which is obtained by communicating with the controller (subsection 4.2.2). After the test cases have been executed, the plugin sends a feedback report to the controller, where it is analysed.

### 4.2.2 Controller

The second component is the core of the framework, acting as an intermediary between the agent on the left side and the predictor (subsection 4.2.3) on the right side. In order to satisfy the second design goal and allow language agnosticism, the agent communicates with the controller using the `HTTP` protocol by exposing a *REST*-interface. Representational State Transfer `[REST]` is a software architecture used by modern web applications that allows standardised communication using existing HTTP methods. On the right side, the controller does not communicate directly with the predictor, but rather stores prediction requests in a shared database which is periodically polled by the predictor. Besides routing prediction requests from the agent to the predictor, the controller will also update the meta predictor by evaluating the accuracy of earlier predictions of this project.

### 4.2.3 Predictor and Metrics

The final component is twofold. Its main responsibility is to apply the prioritisation algorithms and predict an order in which the test cases should be executed. This order

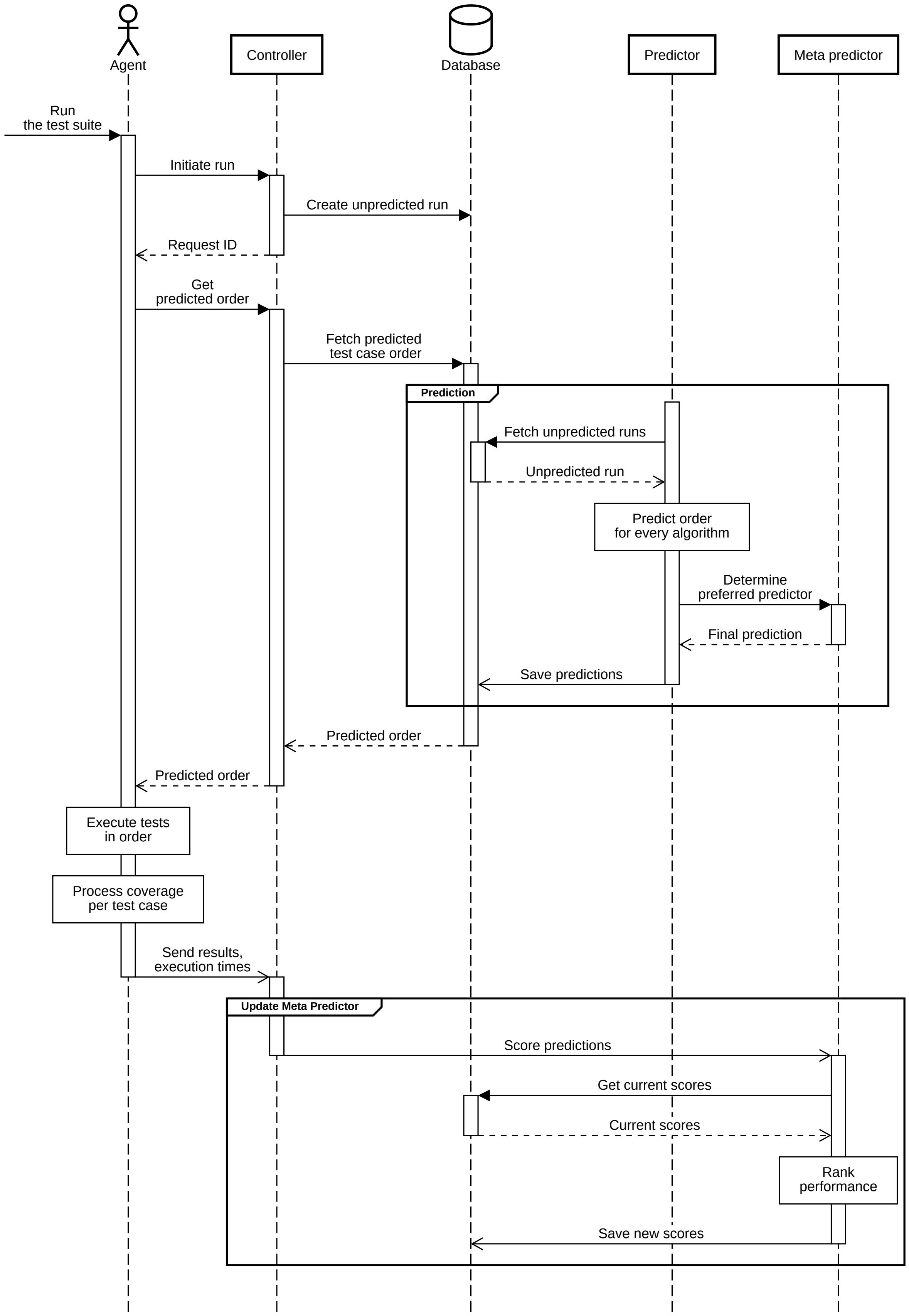


Figure 4.1: Sequence diagram of VeloCIty

is calculated by first executing ten algorithms and subsequently picking the algorithm that has been preferred by the meta predictor. Additionally, this component is able to provide metrics about the test suite, such as identifying superfluous test cases by applying Test Suite Minimisation. More specifically, this redundancy is obtained using the greedy algorithm (subsection 3.2.1). Both of these scripts have been implemented in Python, because of its simplicity and existing libraries for many common operations, such as numerical calculations (NumPy[1]) and machine learning (TensorFlow[2]).

## 4.3 Pipeline

This section will elaborate on the individual steps of the pipeline. The steps will be discussed by manually executing the pipeline that has hypothetically been implemented on a Java project. For the sake of simplicity, this explanation will assume a steady-state situation, ensuring the existence of at least one completed run of this project in the database at the controller side.

### 4.3.1 Initialisation

As was explained before, the provided Java implementation of the agent was designed to be used in conjunction with Gradle. In order to integrate VeloCIty into a Gradle project, the build script (`build.gradle`) should be modified in two places. The first change is to include and apply the plugin in the header of the file. Afterwards, the plugin requires three properties to be configured:

- `base` the path to the Java source files, relative to the location of the build script. This will typically resemble `src/main/java`.
- `repository` the url to the git repository at which the project is hosted. This is required in subsequent steps of the pipeline, to detect which code lines have been changed in the commit currently being analysed.
- `server` the url at which the controller can be reached.

Listing 4.1 contains a minimal integration of the agent in a Gradle build script, applied to a library for generating random numbers[3]. The controller is hosted at the same host as the agent and is accessible at port `8080`.

```
buildscript {
    dependencies {
```

[1] https://numpy.org/

[2] https://www.tensorflow.org/

[3] https://github.com/thepieterdc/random-java

```
        classpath 'io.github.thepieterdc.velocity:velocity-junit:0.0.1-SNAPSHOT'
    }
}

plugins {
    id 'java'
}

apply plugin: 'velocity-junit'

velocity {
    base 'src/main/java/'
    repository 'https://github.com/thepieterdc/random-java'
    server 'http://localhost:8080'
}
```

Listing 4.1: Minimal Gradle buildscript

After the project has been configured, the test suite must be executed. For the Gradle agent, this involves executing the built-in `test` task. This task requires an additional argument to be passed, which is the commit hash of the changeset to prioritise. In every discussed Continuous Integration system, this commit hash is available as an environment variable.

The first step is for the agent to initiate a new test run in the controller. This is accomplished by sending a `POST`-request to the `/runs` endpoint of the controller, which will reply with an identifier. On the controller side, this request will result in a new prioritisation request being enqueued in the database that will asynchronously be processed by the predictor daemon in the next step.

### 4.3.2 Prediction

The prediction of the test execution order is performed by the predictor daemon. This daemon continuously polls the database to fetch new test runs that need to be predicted. When a new test run is detected, the predictor executes every available prediction algorithm in order to obtain multiple prioritised test sequences. The following algorithms are available:

**AllInOrder** The first algorithm will simply prioritise every test case alphabetically and will be used for for benchmarking purposes in **??**.

**AllRandom** The second algorithm has also been implemented for benchmarking purposes. This algorithm will “prioritise” every test case arbitrarily.

**AffectedRandom** This algorithm will only consider the test cases that cover source code lines which have been modified in the current commit. These test cases will be ordered randomly, followed by the other test cases in the test suite in no particular order.

**GreedyCoverAll** The first of three implementations of the Greedy algorithm (subsection 3.2.1) will execute the algorithm to prioritise the entire test suite.

**GreedyCoverAffected** As opposed to the previous greedy algorithm, the second Greedy algorithm will only consider test cases covering changed source code lines to be prioritised. After these test cases, the remaining test cases in the test suite will be ordered randomly.

**GreedyTimeAll** Instead of greedily attempting to cover as many lines of the source code using as few tests as possible, this implementation will attempt to execute as many tests as possible, as soon as possible. In other words, this algorithm will prioritise test cases based on their average execution time.

**HGSAll** This algorithm is an implementation of the algorithm presented by Harrold, Gupta and Soffa (subsection 3.2.2). It is executed for every test case in the test suite.

**HGSAffected** Similar to the *GreedyAffected* algorithm, this algorithm is identical to the previous *HGSAll* algorithm besides that it will only prioritise test cases covering changed source code lines.

**ROCKET** The penultimate algorithm is a straightforward implementation of the pseudocode provided in subsection 3.2.3.

**Alpha** The final algorithm has been inspired by the other implemented algorithms. section 4.4 will further elaborate on the details.

Subsequently, the final prioritisation order is determined by applying the meta predictor. Essentially, the meta predictor can be seen as a table which assigns a score to every algorithm, indicating its performance on this codebase. subsection 4.3.4 will explain later how this score is updated. The predicted order by the algorithm with the highest score is eventually elected by the meta predictor as the final prioritisation order, and saved to the database.

### 4.3.3 Test case execution

Regarding the agent, the identifier obtained in subsection 4.3.1 is used to poll the controller by sending a `GET` request to `/runs/id`, which will reply with the test execution order if this has already been determined. One of the discussed features of Gradle in subsection 3.3.1 was the possibility to execute test cases in a chosen order by adding annotations. However, this feature cannot be used to implement the Java agent, since it only supports ordering test cases within the same test class. In order to facilitate complete control over the order of execution, a custom `TestProcessor` and `TestListener` have been implemented.

The `TestProcessor` is responsible for processing every test class in the classpath and forward it along with configurable options to a delegate processor. The final processor in this chain will eventually perform the actual execution of the test class. Since the delegate processors that are built into Gradle will by default execute every method in the test class, the custom processor needs to work differently. The implemented agent will first store every received test class into a list and load the class to obtain all test cases in the class using reflection. After all classes have been processed, the processor will iterate over the prioritised order. For every test case $t$ in the order, the delegate processor is called with a tuple of the corresponding test class and an options array which excludes every test case except $t$. This will effectively forward the same test class multiple times to the delegate processor, but each time with an option that restricts test execution to the prioritised test case, resulting in the desired behaviour.

Subsequently, the `TestListener` is a method that is called before and after every invocation of a test case. This listener allows the agent to calculate the duration of every test case, as well as collect the intermediary coverage and save this on a per-test case basis.

### 4.3.4 Post-processing and analysis

The final step of the pipeline is to provide feedback to the controller, to evaluate the accuracy of the predictions and thereby implementing the fourth design goal of self-improvement. After executing all test cases, the agent sends the test case results, the execution time and the coverage per test case to the controller by issuing a `POST` request to `/runs/id/test-results` and `/runs/id/coverage`.

Upon receiving this data, the controller will update the meta predictor using the following procedure. The meta predictor is only updated if at least one of the test cases has failed, since the objective of Test Case Prioritisation is to detect failures as fast as possible, thus every prioritised order is equally good if there are no failures at all. If however a test case did fail, the predicted orders are inspected to calculate the duration until the first failed test case for every order. Subsequently, the average of all these durations is calculated. Finally, the score of every algorithm that predicted a below average duration until the first failure is increased, otherwise it is decreased. This will eventually lead to the most accurate algorithms being preferred in subsequent test runs.

## 4.4 Alpha algorithm

Besides the earlier presented Greedy, HGS and ROCKET algorithms (section 3.2), VeloCIty features an additional algorithm. The *Alpha* algorithm has been constructed by examining the individual strengths and weaknesses of the three preceding algorithms and subsequently combining their philosophies into a novel prioritisation algorithm. This paper will now proceed by providing its specification in accordance with the conventions described in definition 5. The corresponding pseudocode is listed in Algorithm 4.

The algorithm consumes the following inputs:

- the set of all $n$ test cases: $TS = \{T_1, \dots, T_n\}$
- the set of $m$ *affected* test cases: $AS = \{T_1, \dots, T_m\} \subseteq TS$. A test case $t$ is considered “affected” if any source code line which is covered by $t$ has been modified or removed in the commit that is being predicted.
- $C$: the set of all lines in the application source code, for which a test case $t \in TS$ exists that covers this line and that has not yet been prioritised. Initially, this set contains every covered source code line.
- the failure status of every test case, for every past execution out of $k$ executions of that test case: $F = \{F_1, \dots, F_n\}$, where $F_i = \{f_1, \dots, f_k\}$. $F_{tj} = 1$ implies that test case $t$ has failed in execution $current - j$.
- the execution time of test case $t \in TS$ for run $r \in [1 \dots k]$, in milliseconds: $D_{tr}$.
- for every test case $t \in TS$, the set $TL_t$ is composed of all source code lines that are covered by test case $t$.

The first step of the algorithm is to determine the execution time $E_t$ of every test case $t$. This execution time is calculated as the average of the durations of every successful (i.e.) execution of $t$, since a test case will be prematurely aborted upon the first failed assertion, which introduces bias in the duration timings. In case $t$ has never been executed successfully, the average is computed over every execution of $t$.

$$E_t = \begin{cases} \overline{\{D_{ti} | i \in [1 \ldots k], F_{ti} = 0\}} & \exists j \in [1 \ldots k], F_{tj} = 0 \\ \overline{\{D_{ti} | i \in [1 \ldots k]\}} & \text{otherwise} \end{cases}$$

Next, the algorithm executes every affected test case that has also failed at least once in its three previous executions. This reflects the behaviour of a developer attempting to resolve the bug that caused the test case to fail. Specifically executing *affected* failing test cases first is required in case multiple test cases are failing and the developer is resolving these one by one, an idea which was extracted from the ROCKET algorithm (subsection 3.2.3). In case there are multiple affected failing test cases, the test cases are prioritised by increasing execution time. After every selected test case, $C$ is updated by subtracting the code lines that have been covered by at least one of these test cases.

Afterwards, the same operation is repeated for every failed but unaffected test case, likewise ordered by increasing execution time. Where the previous step helps developers to get fast feedback about whether or not the specific failing test case they were working on has been resolved, this step ensures that other failing test cases are not forgotten and are executed early in the run as well. Similar to the previous step, $C$ is again updated after every prioritised test case.

Research (subsection 5.4.1) has indicated that on average, only a small fraction (10 $\%-$ 20 $\%$) of all test runs will contain failed tests, resulting in the previous two steps not being executed at all. Therefore, the most time should be dedicated to executing test cases that cover affected code lines. More specifically, the next step of the algorithm executes every affected test case, sorted by decreasing cardinality of the intersection between $C$ and the lines which are covered by the test case. Conforming to the prior two steps, $C$ is also updated to reflect the selected test case. As a consequence of these updates, the cardinalities of these intersections change after every update, which will ultimately lead to affected tests not strictly requiring to be executed. This idea has been adopted from the Greedy algorithm subsection 3.2.1.

In the penultimate step, the previous operation is repeated in an identical fashion for the remaining test cases, similarly ordered by the cardinality of the intersection with the remaining uncovered lines in $C$.

Finally, the algorithm selects every test case which had not yet been prioritised. Notice that these test cases do not contribute to the test coverage, as every test case that would incur additional coverage would have been prioritised already in the previous step. Subsequently, these test cases are actually redundant and are therefore candidates for removal by Test Suite Minimisation. However, since this is a prioritisation algorithm, these tests will still be executed and prioritised by increasing execution time.

**Algorithm 4** Alpha algorithm for Test Case Prioritisation

**Input:** the test suite $TS$, the affected testcases $AS \subseteq TS$, all coverable lines $C$, execution time $E$ of every test case over $k$ runs, failure status $F$
Execution times $D_{tr}$ of every test case $t$, over all $k$ runs $r$ of that test case,
Failure status $FS$ for each test case over the previous $m$ successive iterations,
Sets $TL = \{TL_1, \ldots, TL_n\}$ of all source code lines that are covered by test case $t \in TS$.

**Output:** ordered list $P$, sorted by descending priority

```
procedure AlphaTCP(TS, AS, C)
    $P \leftarrow array[1 \ldots n]$                                          ▷ initially $0$
    $i \leftarrow n$
    $FTS \leftarrow \{t | t \in TS \wedge (F[t][1] = 1 \vee F[t][2] = 1 \vee F[t][3] = 1)\}$
    $AFTS \leftarrow AS \cap FTS$
    for all $t \in AFTS$ do                  ▷ sorted by execution time in $E$ (ascending)
        $C \leftarrow C \setminus TL[t]$
        $P[t] \leftarrow i$
        $i \leftarrow i - 1$
    $FTS \leftarrow FTS \setminus AFTS$
    for all $t \in FTS$ do                   ▷ sorted by execution time in $E$ (ascending)
        $C \leftarrow C \setminus TL[t]$
        $P[t] \leftarrow i$
        $i \leftarrow i - 1$
    $AS \leftarrow AS \setminus FTS$
    while $AS \neq \emptyset$ do
        $t\_max \leftarrow AS[1]$                                ▷ any element from $AS$
        $tl\_max \leftarrow \emptyset$
        for all $t \in AS$ do
            $tl\_current \leftarrow C \cap TL_t$
            if $|tl\_current| > |tl\_max|$ then
                $t\_max \leftarrow t$
                $tl\_max \leftarrow tl\_current$
        $C \leftarrow C \setminus tl\_max$
        $P[t] \leftarrow i$
        $i \leftarrow i - 1$
    $TS \leftarrow TS \setminus (AS \cup FTS)$
    while $TS \neq \emptyset$ do
        $t\_max \leftarrow TS[1]$                                ▷ any element from $TS$
        $tl\_max \leftarrow \emptyset$
        for all $t \in TS$ do
            $tl\_current \leftarrow C \cap TL_t$
            if $|tl\_current| > |tl\_max|$ then
                $t\_max \leftarrow t$
                $tl\_max \leftarrow tl\_current$
        $C \leftarrow C \setminus tl\_max$
        $P[t] \leftarrow i$
        $i \leftarrow i - 1$
    return $P$
```

# Chapter 5

# Evaluation

This chapter will evaluate the performance of the framework presented in the previous chapter. The first section introduces the two test subjects that will be used in subsequent experiments. The next section will restate the research questions formally and extend these. Afterwards, we will elaborate on the procedure of the data collection. The final section will provide answers to the research questions as well as present the results of applying Test Case Prioritisation to the test subjects.

## 5.1 Test subjects

### 5.1.1 Dodona

Dodona[1] is an open-source online learning environment created by Ghent University, which allows students from secondary schools and universities in Belgium and South-Korea to submit solutions to programming exercises and receive instant, automated feedback. The application is built on top of the Ruby-on-Rails web framework. To automate the testing process of the application, Dodona employs GitHub Actions (section 2.2.3) which executes the more than 450 test cases in the test suite and performs static code analysis afterwards. The application is tested using the default `MiniTest` testing framework and `SimpleCov`[2] is used to record the coverage of the test suite. Currently, the coverage ratio is approximately 89 %. This analysis will consider builds between January 1 and May 17, 2020.

### 5.1.2 Stratego

The second test subject has been created for the Software Engineering Lab 2 course at Ghent University in 2018. The application was created for a Belgian gas transmission system operator and consists of two main components: a web frontend and a backend. This thesis will test the backend in particular since it is written in Java using the Spring framework. Furthermore, the application uses Gradle and JUnit to execute the $300 - 400$ test cases in the test suite, allowing the Java agent (section 4.2.1) to be applied directly.

[1] https://dodona.ugent.be/

[2] https://github.com/colszowka/simplecov

## 5.2 Research questions

We will answer the following research questions in the subsequent sections:

**RQ1: What is the probability that a test run will contain at least one failed test case?** The first research question will provide useful insights into whether a typical test run tends to fail or not. The expectancy is that the probability of failure will be rather low, indicating that it is not strictly necessary to execute every test case and therefore making a case for Test Suite Minimisation.

**RQ2: What is the average duration of a test run?** Measuring how long it takes to execute a typical test run is required to estimate the benefit of applying any form of test suite optimisation. We will only consider successful test runs, to reduce bias introduced by prematurely aborting the execution.

**RQ3: Suppose that a test run has failed, what is the probability that the next run will fail as well?** The ROCKET algorithm (section 3.2.3) relies on the assumption that if a test case has failed in a given test run, it is likely to fail in the subsequent run as well. This research question will investigate the correctness of this hypothesis.

**RQ4: How can Test Case Prioritisation be applied to Dodona and what is the resulting performance benefit?** This research question will investigate the possibility to apply the VeloCIty framework to the Dodona project and analyse how quickly the available predictors can discover a failing test case.

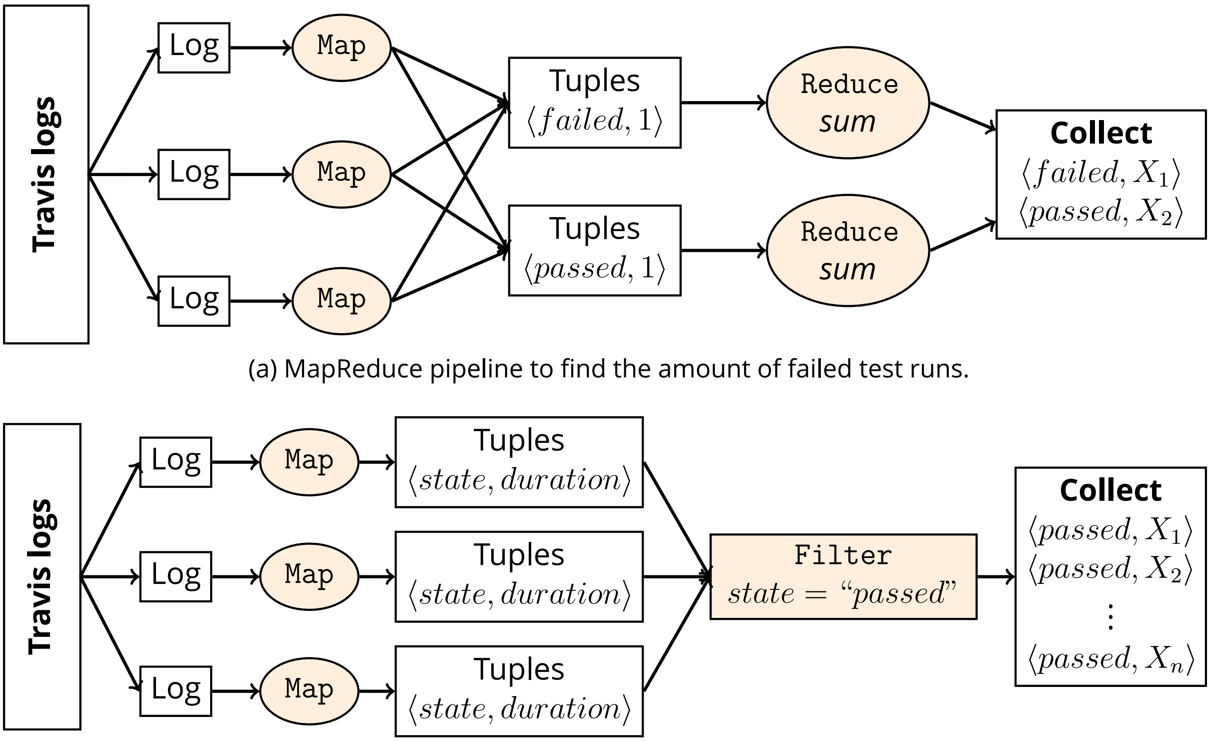
**RQ5: Can the Java agent be applied to Stratego?** Since the testing framework used by Stratego should be supported natively by the Java agent, this research question will verify its compatibility. Furthermore, we will analyse the prediction performance, albeit with a small number of relevant test runs.

## 5.3 Data collection

### 5.3.1 Travis CI build data

We can answer the first three research questions by analysing data from projects hosted on Travis CI (section 2.2.3). This data has been obtained from two sources.

The first source comprises a database [9] of 35 793 144 log files of executed test runs, which has been contributed by Durieux et al. The magnitude of the dataset (61.11 GiB) requires a big data approach to parse these log files. Two straightforward MapReduce pipelines (Figure 5.1) have been created using the Apache Spark[3] engine, to provide an answer to the first and second research question.


Travis logs
Log
Log
Log
Map
Map
Map
Tuples
$\langle failed, 1\rangle$
Tuples
$\langle passed, 1\rangle$
Reduce
sum
Reduce
sum
Collect
$\langle failed, X_1\rangle$
$\langle passed, X_2\rangle$
Travis logs
Log
Log
Log
Map
Map
Map
Tuples
$\langle state, duration\rangle$
Tuples
$\langle state, duration\rangle$
Tuples
$\langle state, duration\rangle$
Filter
$state =$ "$passed$"
Collect
$\langle passed, X_1\rangle$
$\langle passed, X_2\rangle$
$\vdots$
$\langle passed, X_n\rangle$

(a) MapReduce pipeline to find the amount of failed test runs.

(b) MapReduce pipeline to find the average duration of a successful test run.

Figure 5.1: MapReduce pipelines for Travis CI data

In addition to the first source, another 3 702 595 jobs have been analysed from the *TravisTorrent* project [4]. To identify which projects are using Travis CI, the authors have crawled the GitHub API and examined the build status of every commit to retrieve the run identifier. Subsequently, the Travis CI API is used to obtain information about both the build, as well as the project itself. This information includes the programming language, the amount of source code lines and the amount of failed test cases. The latter value provides an accurate answer to the first research question since it indicates why the test run has failed. Without this information, the test suite might have failed to compile as opposed to an actual failure in the test cases. Furthermore, the dataset includes the identifier of the previously executed test run, which we can use to answer the third research question. Additionally, the information contains the build duration. This dataset has been excluded from the second research question however, as the included execution time does not correspond to the actual duration reported on the webpage of Travis CI. The authors have provided a Google BigQuery[4] interface to allow querying the dataset more efficiently. Appendix A contains the executed queries.

[3] https://spark.apache.org/

[4] https://bigquery.cloud.google.com/

### 5.3.2 Dodona data

As mentioned before, Dodona utilises the MiniTest testing framework in conjunction with SimpleCov to calculate the coverage. MiniTest will by default only emit the name of every failed test case, without any further information. Furthermore, SimpleCov can only calculate the coverage for the entire test suite and does not allow us to retrieve the coverage on a per-test basis. To answer the fourth research question and apply the VeloCIty predictors to Dodona, a Python script has been created to reconstruct the conditions of every failed test run. The script first queries the API of GitHub Actions to find which test runs have failed. This thesis will consider 120 failed runs. For every failed commit, the script retrieves the parent commit and calculates the coverage on a per-test basis. This thesis will assume that the coverage of the parent commit resembles the coverage of the failed commit. The coverage is calculated by applying the following two transformations to the parent commits and subsequently rescheduling these in GitHub Actions:

- **Cobertura formatter:** The current SimpleCov reports can only be generated as HTML reports, preventing convenient analysis. We can resolve this problem by using the Cobertura formatter instead, which generates XML reports. The controller already supports the structure of these reports, as this formatter is commonly used by Java testing frameworks as well.
- **Parallel execution:** The Dodona test suite currently executes the test cases by four processes concurrently, to reduce the execution time. Every process individually records the code coverage, and at the end of the test suite, SimpleCov merges these separate reports into one. However, this process is not entirely thread-safe since the test suite requires shared resources. We do not require thread-safety to calculate the total coverage, but we do require this to generate the coverage on a per-test basis. As a result, parallel execution has been disabled in these experiments.

### 5.3.3 Stratego data

To integrate VeloCIty with the existing Stratego codebase, we can use the instructions described in chapter 4. Afterwards, to analyse the prediction performance, we can take an approach similar to the previous test subject. The GitHub API has been used to identify the failed commits and to find their parent (successful) commits. The parent commits have subsequently been modified to use the VeloCIty Java agent and have been executed using GitHub Actions.

## 5.4 Results

### 5.4.1 RQ1: Probability of failure

The two pie charts in Figure 5.2 illustrate the amount of failed and successful test runs. The leftmost chart visualises the failure rate in the dataset [9] by Durieux et al. 4 558 279 test runs out of the 28 882 003 total runs have failed, which corresponds to a failure probability of 18.74 %. The other pie chart uses data from the TravisTorrent [4] project. Since we can infer the cause of failure from this dataset, it is possible to obtain more accurate results. 42.89 % of the failed runs are due to a compilation failure where the test suite did not execute. For the remaining part of the runs, 225 766 out of 2 114 920 executions contain at least one failed test case, corresponding to a failure percentage of 10.67 %.

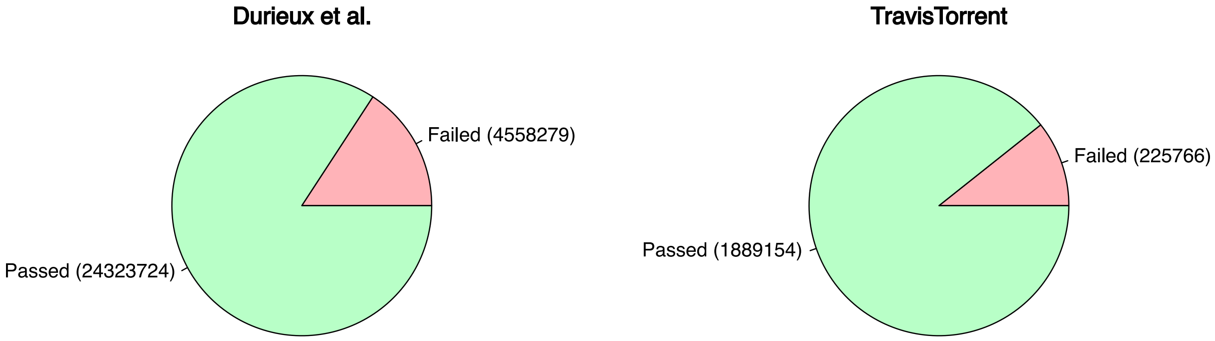


Figure 5.2: Probability of test run failure

### 5.4.2 RQ2: Average duration of a test run

The dataset by Durieux et al. [9] has been refined to only include test runs that did not finish within 10 s. A lower execution time generally indicates that the test suite did not execute and that a compilation failure has occurred instead. Table 5.1 contains the characteristics of the remaining 24 320 504 analysed test runs. The median and average execution times suggest that primarily small projects are Travis CI, yet the maximum value is very high. Figure 5.3 confirms that 71 378 test runs have taken longer than one hour to execute. Further investigation has revealed that these are typically projects which are using mutation testing, such as `plexus/yaks`[5].

| # runs | Minimum | Mean | Median | Maximum |
|---|---|---|---|---|
| 24 320 504 | 10 s | 385 s | 178 s | 26 h11 min26 s |

Table 5.1: Characteristics of the test run durations in [9].

[5] A Ruby library for hypermedia (`https://github.com/plexus/yaks`).

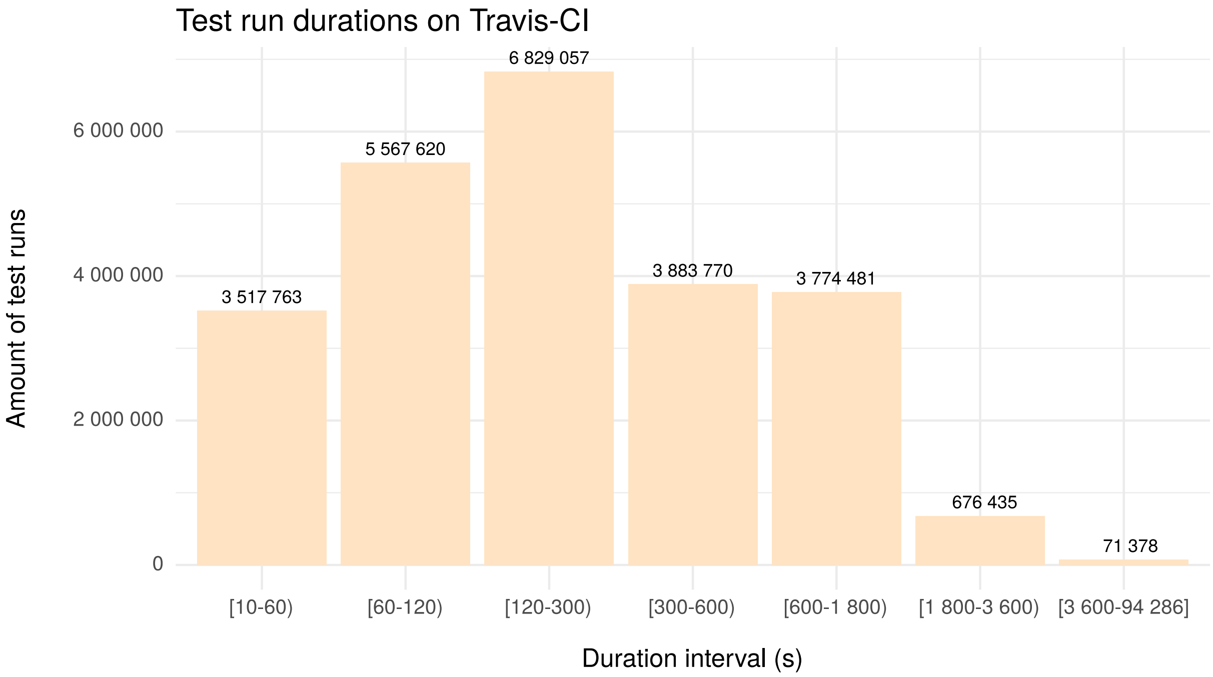


Figure 5.3: Test run durations on Travis CI

### 5.4.3 RQ3: Consecutive failure probability

Because the TravisTorrent project is the only dataset that contains the identifier of the previous run, only runs from this project have been used. This dataset consists of 211 040 test runs, immediately following a failed execution. As illustrated in Figure 5.4, 109 224 of these test runs have failed as well, versus 101 816 successful test runs (51.76 %).

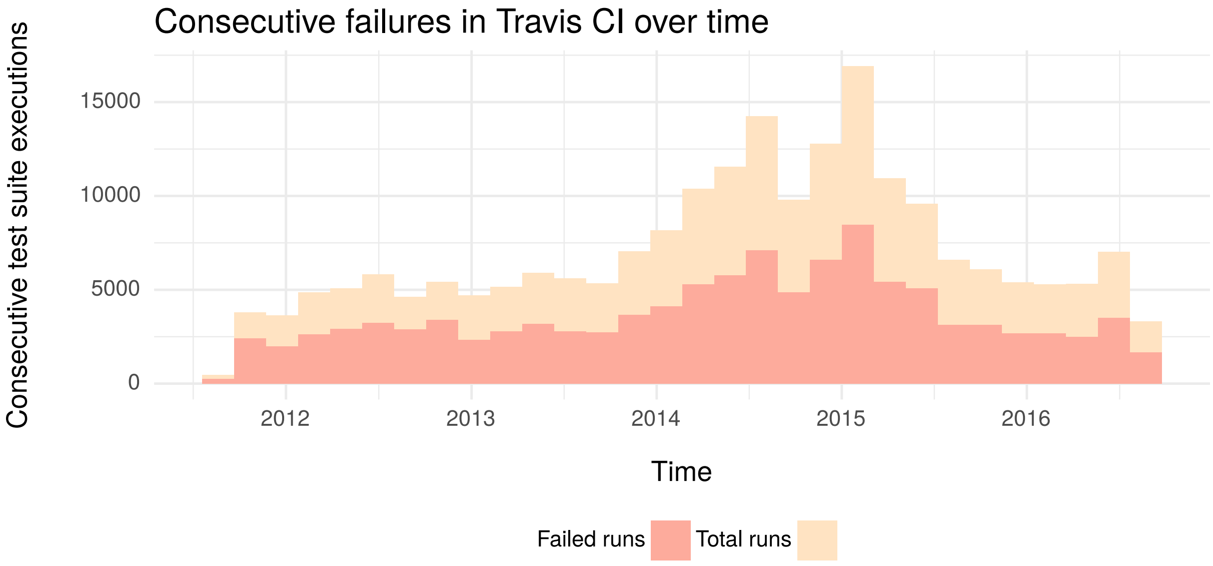


Figure 5.4: Consecutive test run failures on Travis CI

### 5.4.4 RQ4: Applying Test Case Prioritisation to Dodona

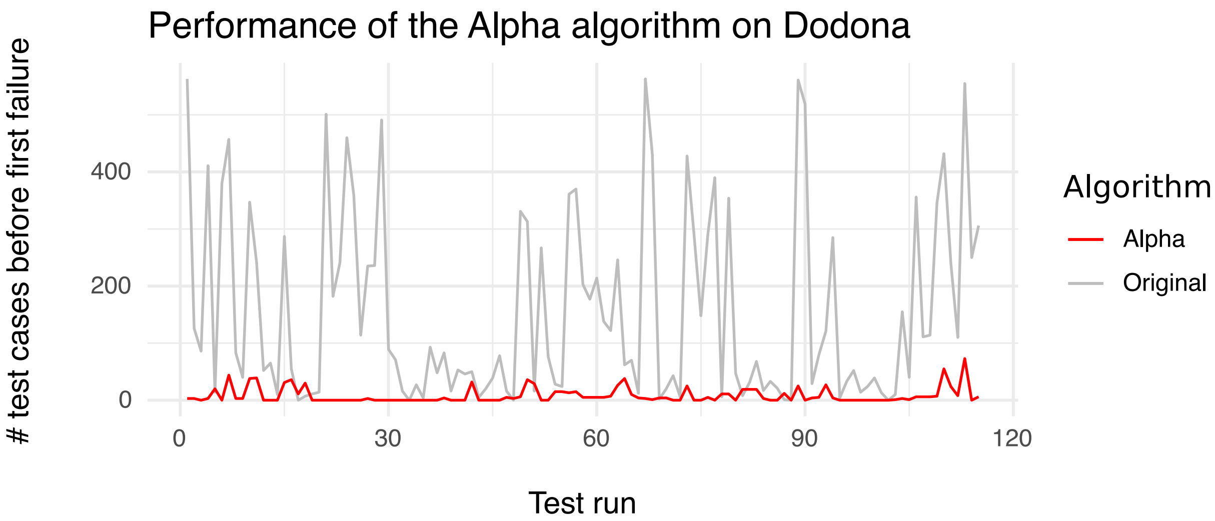
After executing the 120 failed test runs, the log files have been inspected. These log files have revealed that an error in the configuration was the actual culprit of five failed test runs, rather than a failed test case. These test runs have therefore been omitted from the results because the test suite did not execute. Since configuration-related problems require in-depth contextual information about the project, we cannot automatically predict these.

Table 5.2 contains the amount of executed test cases until we observe the first failure. These results indicate that every predictor is capable of performing at least one successful prediction. Furthermore, the maximum amount of executed test cases is lower than the original value, which means that every algorithm is a valid predictor. The data suggests that the Alpha algorithm and the HGS algorithm are the preferred predictors for the Dodona project. In contrast, the performance of the ROCKET algorithm is rather low.

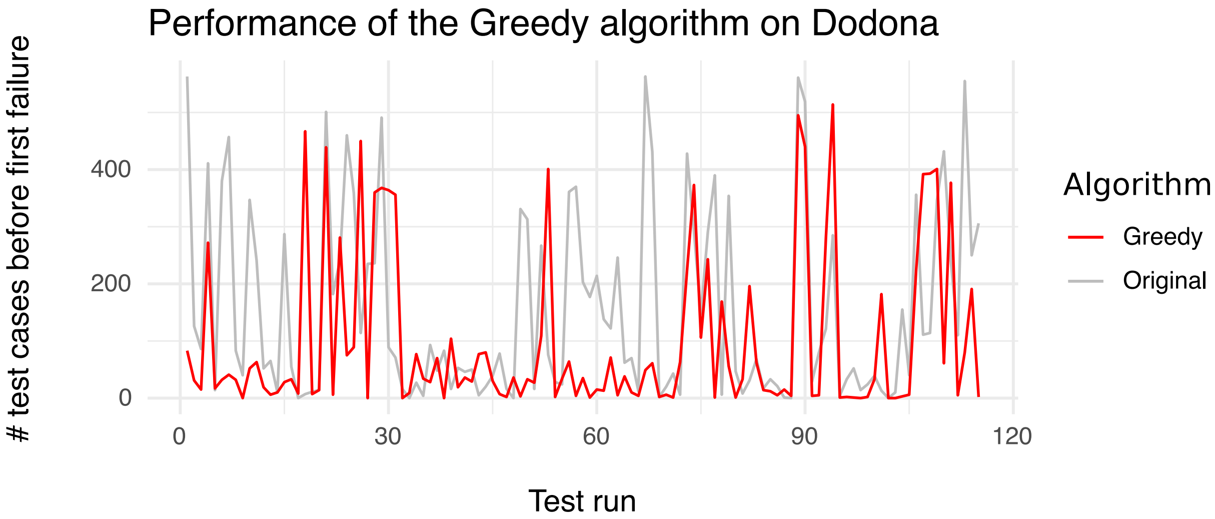
| **Algorithm** | **Minimum** | **Mean** | **Median** | **Maximum** |
|---|---|---|---|---|
| *Original* | 0 | 155 | 78 | 563 |
| Alpha | 0 | 8 | 3 | 73 |
| AffectedRandom | 0 | 54 | 10 | 428 |
| AllInOrder | 0 | 119 | 82 | 460 |
| AllRandom | 0 | 90 | 27 | 473 |
| GreedyCoverAffected | 0 | 227 | 246 | 494 |
| GreedyCoverAll | 0 | 98 | 33 | 514 |
| GreedyTimeAll | 0 | 210 | 172 | 482 |
| HGSAffected | 0 | 61 | 10 | 511 |
| HGSAll | 0 | 124 | 54 | 507 |
| ROCKET | 0 | 210 | 170 | 482 |

Table 5.2: Amount of executed test cases until the first failure.

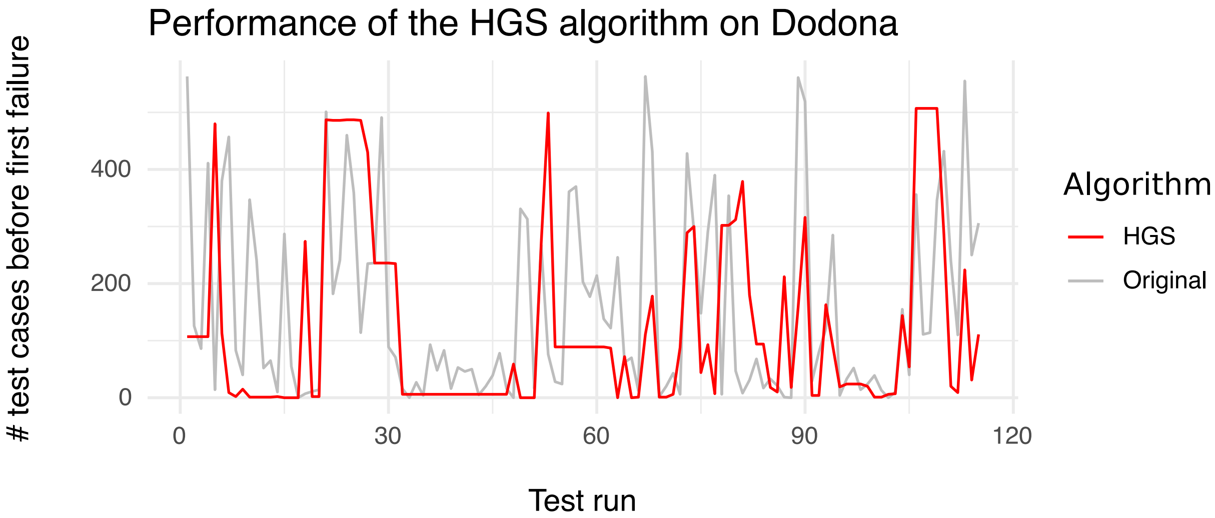
The previous results have been visualised in Figure 5.2. These charts confirm the low accuracy of the ROCKET algorithm. The Alpha algorithm and the HGS algorithm offer the most accurate predictions, with the former algorithm being the most consistent. Notice the chart of the Greedy algorithm, which succeeds in successfully predicting some of the test runs, while failing to predict others. This behaviour is specific to a greedy heuristic.



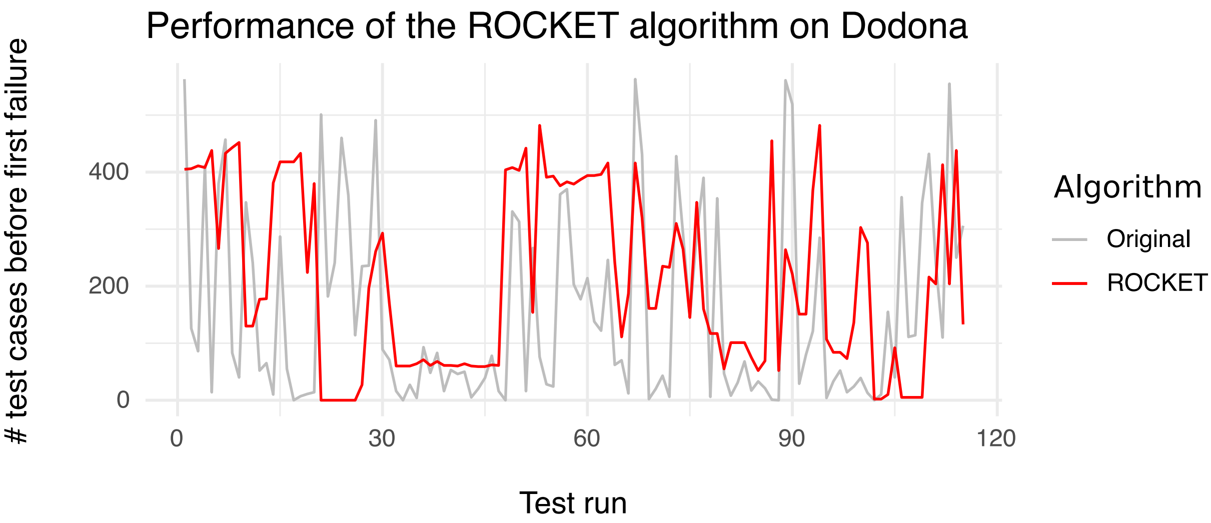
(a) Alpha algorithm



(b) Greedy algorithm



(c) HGS algorithm



(d) ROCKET algorithm

Figure 5.2: Prediction performance on the Dodona project.

The duration until the first observed failure is reported in Table 5.3. Observe that the previous table indicates that the ROCKET algorithm does not perform well, while this table suggests otherwise. We can explain this behaviour by examining the objective function of this algorithm. This function prioritises cases with a low execution time to be executed first.

| **Algorithm** | **Minimum** | **Mean** | **Median** | **Maximum** |
|---|---|---|---|---|
| *Original* | 0 s | 135 s | 123 s | 380 s |
| Alpha | 0 s | 3 s | 1 s | 33 s |
| AffectedRandom | 0 s | 28 s | 5 s | 190 s |
| AllInOrder | 0 s | 82 s | 71 s | 270 s |
| AllRandom | 0 s | 43 s | 11 s | 270 s |
| GreedyCoverAffected | 0 s | 88 s | 86 s | 314 s |
| GreedyCoverAll | 0 s | 46 s | 12 s | 280 s |
| GreedyTimeAll | 0 s | 55 s | 32 s | 175 s |
| HGSAffected | 0 s | 35 s | 6 s | 356 s |
| HGSAll | 0 s | 75 s | 34 s | 377 s |
| ROCKET | 0 s | 54 s | 32 s | 175 s |

Table 5.3: Duration until the first failure for the Dodona project.

### 5.4.5 RQ5: Integrate VeloCIty with Stratego

The data collection phase has already proven that the Java agent is compatible with Stratego. Since VeloCIty is not yet able to predict test cases which have been added in the current commit, we can only use 35 of the 54 failed test runs.

Similar to the previous test subject, Table 5.4 lists how many test cases have been executed before the first observed failure. The table only considers the four main algorithms, since the actual prediction performance was only secondary to this research question, and we have only analysed a small number of test runs. The results suggest that every algorithm except the ROCKET achieves a high prediction accuracy on this project.

| **Algorithm** | **Minimum** | **Mean** | **Median** | **Maximum** |
|---|---|---|---|---|
| *Original* | 0 | 68 | 2 | 278 |
| Alpha | 0 | 10 | 2 | 57 |
| GreedyCoverAll | 0 | 11 | 3 | 57 |
| HGSAll | 0 | 9 | 4 | 50 |
| ROCKET | 0 | 42 | 27 | 216 |

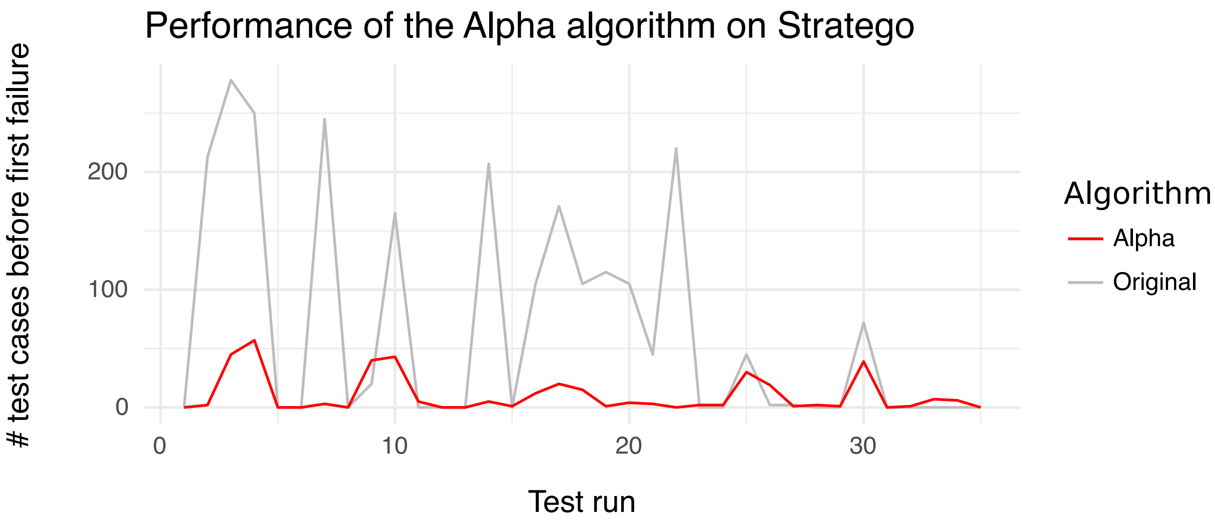
Table 5.4: Amount of executed test cases until the first failure.

Even though the performance of the ROCKET algorithm is suboptimal in the previous table, Table 5.5 does indicate that it outperforms every other algorithm time-wise. Notice that the predicted sequence of the HGS algorithm takes the longest to execute, while the previous table suggested a good prediction accuracy. The Alpha and Greedy algorithms seem very similar on both the amount of executed test cases, as well as the execution time.

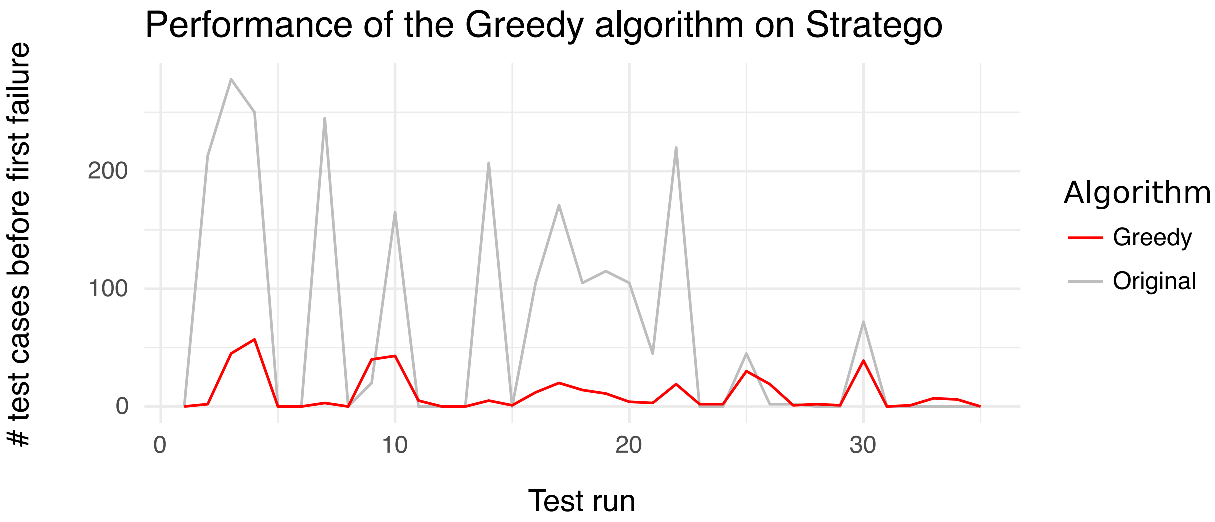
| **Algorithm** | **Minimum** | **Mean** | **Median** | **Maximum** |
|---|---|---|---|---|
| *Original* | 0 s | 62 s | 8 s | 233 s |
| Alpha | 0 s | 11 s | 2 s | 103 s |
| GreedyCoverAll | 0 s | 12 s | 2 s | 103 s |
| HGSAll | 0 s | 19 s | 1 s | 130 s |
| ROCKET | 0 s | 6 s | 0 s | 85 s |

Table 5.5: Amount of executed test cases until the first failure.

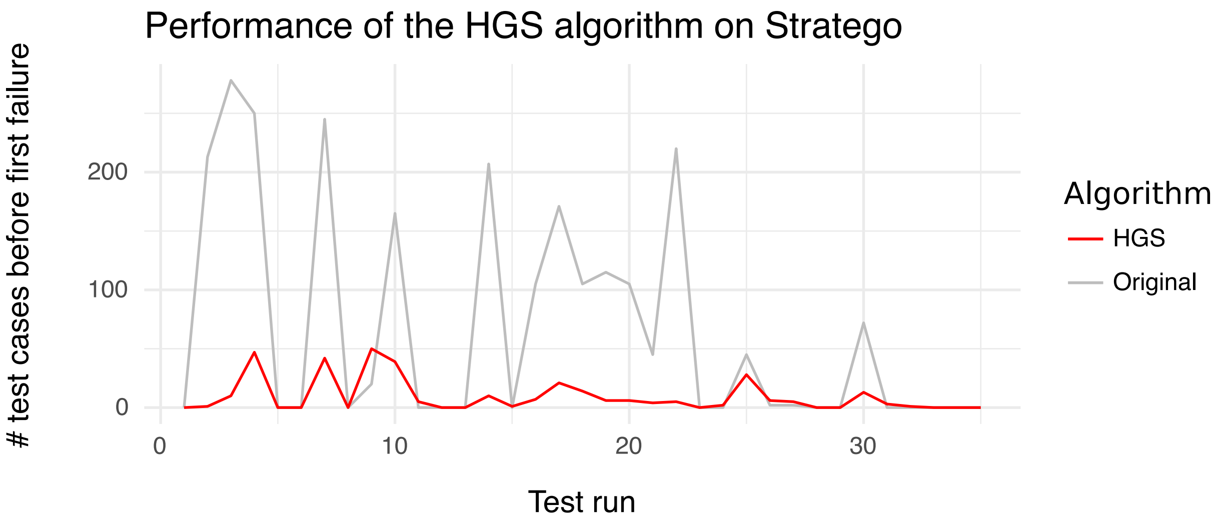
Figure 5.0 further confirms the above statements. Notice the close resemblance of the charts of the Greedy algorithm and the Alpha algorithm, which indicates that a different failing test case is the cause of every test run failure. The ROCKET algorithm performs better on this project than on Dodona, yet not accurate.



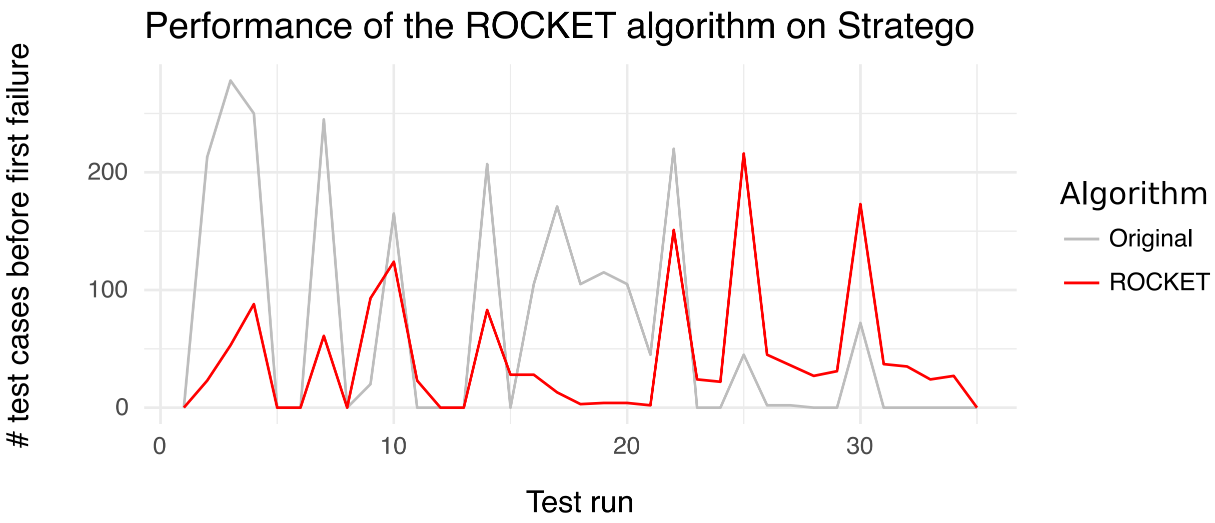
(a) Alpha algorithm



(b) Greedy algorithm



(c) HGS algorithm



(d) ROCKET algorithm

Figure 5.0: Prediction performance on the Stratego project.

# Chapter 6

# Conclusion

The main objective of this thesis was to study different approaches to optimise the test suite of a typical software project. Three approaches have been introduced to this extend: Test Suite Minimisation, Test Case Selection and Test Case Prioritisation. We have successfully implemented the latter approach using the VeloCIty framework. Furthermore, this framework features the Alpha algorithm as a novel prioritisation algorithm. The performance of the Alpha algorithm has been evaluated, mainly on the UGent Dodona project. The results are promising, resulting in 95 % fewer executed test cases and 97 % less time spent waiting for the first test case to fail.

The second purpose of this thesis was to gain useful insights into the characteristics of a regular test suite, formalised into three research questions. The first question was to estimate the expected failure probability of a test run. To answer this question, we have analysed more than 28 million test runs on Travis-CI. This analysis has indicated that 18 % of those test runs have failed. Additionally, we have used this dataset to answer another question, which was to determine the typical duration of a test run. Statistical analysis has revealed that developers mainly use Travis-CI for small projects, with an average test suite duration of seven minutes. 0.20 % of the test suites take longer than one hour to execute, and some projects use mutation testing. The final question was to examine the probability of consecutive failing test runs. This probability was estimated at 52 % using a second Travis-CI dataset from the TravisTorrent project[4].

## 6.1 Future work

The proposed architecture currently features a Java agent, which supports the prediction of Gradle projects using ten available predictors. However, there is still room for improvements. The paragraphs below will suggest some ideas for possible enhancements.

### 6.1.1 Java agent

We can extend the functionality of the Java agent in multiple ways. Its current biggest weakness is the lack of support for parallel test case execution. To allow parallel testing, we must first solve a problem related to the scheduling process. Since the execution time of a test case can vary significantly, a coordination mechanism is required to schedule which test case should be executed on which thread. One possibility would be to consider the average execution time per test case, which we can obtain by examining prior runs. Alternatively, the scheduling can be performed at runtime using an existing inter-thread communication paradigm, such as message passing. Specific to the Java agent, implementing parallel execution requires us to modify the current `TestProcessor` to extend the `MaxNParallelTestClassProcessor` instead. A thread pool should ideally be used to diminish the overhead of restarting a new thread for every test case.

### 6.1.2 Predictions

We can make four different enhancements to the predictors.

For the first enhancement, the predictor should be able to discriminate between a unit test or an integration test. Recall that the scope of a unit test is limited to a small fraction of the application code and that its execution time is usually low. Contrarily, an integration test usually takes much longer to execute and tests multiple components of the application at once. The predictor should ideally make use of this distinction and assume that a failing unit test will almost certainly result in a failed integration test as well, and as such, prioritise unit tests over integration tests.

Secondly, the prediction algorithms currently take into account which source code lines have either been modified or removed to identify which test cases have been affected. Likewise, a change in the code of the test case itself should also consider that test case affected, as the change might have introduced a bug as well.

A third possible improvement would be to examine the performance of combining multiple prediction algorithms. Currently, the algorithms operate independently from each other, but there might be hidden potential in combining the individual strengths of these algorithms dynamically at runtime. A simple implementation is possible by modifying the existing meta predictor. Instead of assigning a score to the entire prediction, we could combine several predictions using predefined weights from earlier predictions.

Finally, the predictors do not currently consider branch coverage in addition to statement coverage. Not every coverage tool is capable of accurately reporting which branches have been covered, therefore this has not been implemented. Branch coverage can alternatively be supported by instrumenting the source code and rewriting every conditional expression as separate `if`-statements.

### 6.1.3 Meta predictor

The current implementation of the meta predictor increments the score of the predictor if the prediction was above-average, and decreases the score otherwise. However, a possible problem with this approach is that the nature of the source code might evolve and change as time progresses. As a result, it might take several test suite invocations for the meta predictor to prefer an alternative predictor. We can mitigate this effect if we would use a saturating counter (Figure 6.1) instead. This idea is also used in branch predictors of microprocessors and allows a more versatile meta predictor.

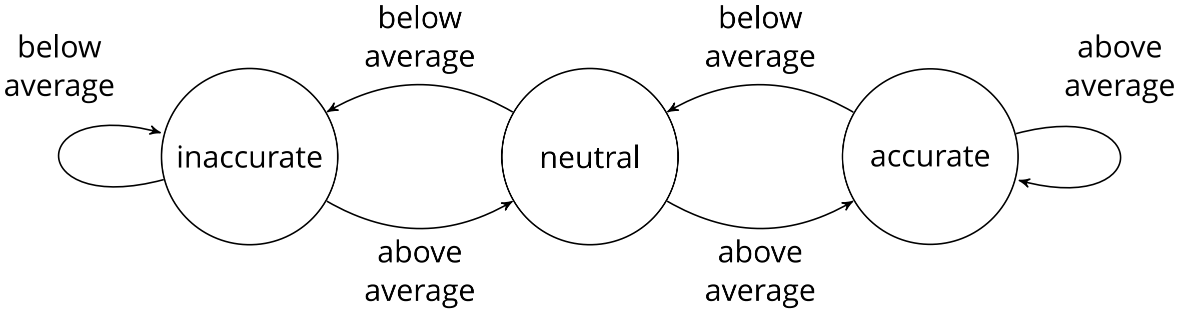


Figure 6.1: Saturating counter with three states

In addition to implementing a different update strategy, it might be worth to investigate the use of machine learning or linear programming models as a meta predictor, or even as a prediction algorithm.

### 6.1.4 Final enhancements

Finally, since we can apply every implemented algorithm to Test Suite Minimisation as well, we might extend the architecture to support this technique explicitly. Executing fewer test cases will result in even lower execution times.

Support for new programming languages and frameworks is possible by implementing a new agent. A naive implementation would be to restart the test suite after every executed test case, should test case reordering not be supported natively by the test framework.

# Bibliography

[1] *About GitHub Actions*. URL: https://help.github.com/en/actions/getting-started-with-github-actions/about-github-actions.

[2] Mohammed Arefeen and Michael Schiller. "Continuous Integration Using Gitlab". In: *Undergraduate Research in Natural and Clinical Science and Technology (URNCST) Journal* 3 (Sept. 2019), pp. 1–6. DOI: 10.26685/urncst.152.

[3] Kent Beck et al. *Manifesto for Agile Software Development*. 2001. URL: https://www.agilemanifesto.org/.

[4] Moritz Beller, Georgios Gousios, and Andy Zaidman. "TravisTorrent: Synthesizing Travis CI and GitHub for Full-Stack Research on Continuous Integration". In: *Proceedings of the 14th working conference on mining software repositories*. 2017.

[5] H.D. Benington. *Production of large computer programs*. ONR symposium report. Office of Naval Research, Department of the Navy, 1956, pp. 15–27. URL: https://books.google.com/books?id=tLo6AQAAMAAJ.

[6] Thomas H. Cormen et al. *Introduction to Algorithms, Third Edition*. 3rd. The MIT Press, 2009. ISBN: 0262033844.

[7] Michael Cusumano, Akindutire Michael, and Stanley Smith. "Beyond the waterfall : software development at Microsoft". In: (Feb. 1995).

[8] Charles-Axel Dein. *dein.fr*. Sept. 2019. URL: https://www.dein.fr/2019-09-06-test-coverage-only-matters-if-at-100-percent.html.

[9] Thomas Durieux et al. "An Analysis of 35+ Million Jobs of Travis CI". In: (2019). DOI: 10.1109/icsme.2019.00044. eprint: arXiv:1904.09416.

[10] *Features • GitHub Actions*. URL: https://github.com/features/actions.

[11] Martin Fowler and Jim Highsmith. "The Agile Manifesto". In: 9 (Nov. 2000).

[12] Michael R. Garey and David S. Johnson. *Computers and Intractability; A Guide to the Theory of NP-Completeness*. USA: W. H. Freeman & Co., 1990. ISBN: 0716710455.

[13] *GitLab CI/CD*. URL: https://docs.gitlab.com/ee/ci/.

[14] *GitLab Continuous Integration & Delivery*. URL: https://about.gitlab.com/stages-devops-lifecycle/continuous-integration/.

[15] Robert L Glass. "Agile versus traditional: Make love, not war!" In: *Cutter IT Journal* 14.12 (2001), pp. 12–18.

[16] A. Govardhan. “A Comparison Between Five Models Of Software Engineering”. In: *IJCSI International Journal of Computer Science Issues 1694-0814* 7 (Sept. 2010), pp. 94–101.

[17] Standish Group et al. “CHAOS report 2015”. In: *The Standish Group International* (2015). URL: `https://www.standishgroup.com/sample_research_files/CHAOSReport2015-Final.pdf`.

[18] M. Jean Harrold, Rajiv Gupta, and Mary Lou Soffa. “A Methodology for Controlling the Size of a Test Suite”. In: *ACM Trans. Softw. Eng. Methodol.* 2.3 (July 1993), pp. 270–285. ISSN: 1049-331X. DOI: `10.1145/152388.152391`. URL: `https://doi.org/10.1145/152388.152391`.

[19] Orit Hazzan and Yael Dubinsky. “The Agile Manifesto”. In: *Agile Anywhere: Essays on Agile Projects and Beyond*. Cham: Springer International Publishing, 2014, pp. 9–14. ISBN: 978-3-319-10157-6. DOI: `10.1007/978-3-319-10157-6_3`. URL: `https://doi.org/10.1007/978-3-319-10157-6_3`.

[20] Jim Highsmith. *History: The Agile Manifesto*. 2001. URL: `https://agilemanifesto.org/history.html`.

[21] Naftanaila Ionel. “AGILE SOFTWARE DEVELOPMENT METHODOLOGIES: AN OVERVIEW OF THE CURRENT STATE OF RESEARCH”. In: *Annals of Faculty of Economics* 4 (May 2009), pp. 381–385.

[22] “ISO/IEC/IEEE International Standard - Software and systems engineering –Software testing –Part 1:Concepts and definitions”. In: *ISO/IEC/IEEE 29119-1:2013(E)* (Sept. 2013), pp. 1–64. DOI: `10.1109/IEEESTD.2013.6588537`.

[23] “ISO/IEC/IEEE International Standard - Systems and software engineering – System life cycle processes”. In: *ISO/IEC/IEEE 15288 First edition 2015-05-15* (May 2015), pp. 1–118. DOI: `10.1109/IEEESTD.2015.7106435`.

[24] “ISO/IEC/IEEE International Standard - Systems and software engineering–Vocabulary”. In: *ISO/IEC/IEEE 24765:2017(E)* (Aug. 2017), pp. 1–541. DOI: `10.1109/IEEESTD.2017.8016712`.

[25] Y. Jia and M. Harman. “An Analysis and Survey of the Development of Mutation Testing”. In: *IEEE Transactions on Software Engineering* 37.5 (2011), pp. 649–678.

[26] Soreangsey Kiv et al. “Agile Manifesto and Practices Selection for Tailoring Software Development: A Systematic Literature Review”. In: *Product-Focused Software Process Improvement*. Ed. by Marco Kuhrmann et al. Cham: Springer International Publishing, 2018, pp. 12–30. ISBN: 978-3-030-03673-7.

[27] N. Landry. *Iterative and Agile Implementation Methodologies in Business Intelligence Software Development*. Lulu.com, 2011. ISBN: 9780557247585. URL: `https://books.google.be/books?id=bUHJAQAAQBAJ`.

[28] G. Le Lann. "An analysis of the Ariane 5 flight 501 failure-a system engineering perspective". In: *Proceedings International Conference and Workshop on Engineering of Computer-Based Systems*. Mar. 1997, pp. 339–346. DOI: `10.1109/ECBS.1997.581900`.

[29] Simon Maple. *Development Tools in Java: 2016 Landscape*. July 2016. URL: `https://www.jrebel.com/blog/java-tools-and-technologies-2016`.

[30] D. Marijan, A. Gotlieb, and S. Sen. "Test Case Prioritization for Continuous Regression Testing: An Industrial Case Study". In: *2013 IEEE International Conference on Software Maintenance*. 2013, pp. 540–543.

[31] Robert C. Martin and Micah Martin. *Agile Principles, Patterns, and Practices in C# (Robert C. Martin)*. USA: Prentice Hall PTR, 2006. ISBN: 0131857258.

[32] Bertrand Meyer. "Overview". In: *Agile!: The Good, the Hype and the Ugly*. Cham: Springer International Publishing, 2014, pp. 1–15. ISBN: 978-3-319-05155-0. DOI: `10.1007/978-3-319-05155-0_1`. URL: `https://doi.org/10.1007/978-3-319-05155-0_1`.

[33] Glenford J. Myers, Corey Sandler, and Tom Badgett. *The Art of Software Testing*. 3rd. Wiley Publishing, 2011. ISBN: 1118031962.

[34] Dor Nir, Shmuel Tyszberowicz, and Amiram Yehudai. "Locating Regression Bugs". In: *Hardware and Software: Verification and Testing*. Ed. by Karen Yorav. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 218–234. ISBN: 978-3-540-77966-7.

[35] Raphael Noemmer and Roman Haas. "An Evaluation of Test Suite Minimization Techniques". In: Dec. 2019, pp. 51–66. ISBN: 978-3-030-35509-8. DOI: `10.1007/978-3-030-35510-4_4`.

[36] A. Jefferson Offutt and Roland H. Untch. "Mutation 2000: Uniting the Orthogonal". In: *Mutation Testing for the New Century*. Ed. by W. Eric Wong. Boston, MA: Springer US, 2001, pp. 34–44. ISBN: 978-1-4757-5939-6. DOI: `10.1007/978-1-4757-5939-6_7`. URL: `https://doi.org/10.1007/978-1-4757-5939-6_7`.

[37] W. W. Royce. "Managing the Development of Large Software Systems: Concepts and Techniques". In: *Proceedings of the 9th International Conference on Software Engineering*. ICSE '87. Monterey, California, USA: IEEE Computer Society Press, 1987, pp. 328–338. ISBN: 0-89791-216-0. URL: `http://dl.acm.org/citation.cfm?id=41765.41801`.

[38] John Ferguson Smart. *Jenkins: The Definitive Guide*. Beijing: O'Reilly, 2011. ISBN: 978-1-4493-0535-2. URL: https://www.safaribooksonline.com/library/view/jenkins-the-definitive/9781449311155/.

[39] Travis. *Travis CI - Test and DeployYour Code with Confidence*. Feb. 2020. URL: https://travis-ci.org.

[40] Kristen R. Walcott et al. "TimeAware Test Suite Prioritization". In: *Proceedings of the 2006 International Symposium on Software Testing and Analysis*. ISSTA '06. Portland, Maine, USA: Association for Computing Machinery, 2006, pp. 1–12. ISBN: 1595932631. DOI: 10.1145/1146238.1146240. URL: https://doi.org/10.1145/1146238.1146240.

[41] James Whittaker. "What is software testing? And why is it so hard?" In: *Software, IEEE* 17 (Feb. 2000), pp. 70–79. DOI: 10.1109/52.819971.

[42] S. Yoo and M. Harman. "Regression Testing Minimization, Selection and Prioritization: A Survey". In: *Softw. Test. Verif. Reliab.* 22.2 (Mar. 2012), pp. 67–120. ISSN: 0960-0833. DOI: 10.1002/stv.430. URL: https://doi.org/10.1002/stv.430.

# List of Figures

# List of Listings

# List of Tables

# Appendices

# Appendix A

# TravisTorrent queries

```
SELECT
  COUNTIF(tr_log_bool_tests_failed) as failed,
  COUNTIF(tr_log_bool_tests_ran) as ran,
  COUNT(1) as total
FROM ‘travistorrent‘
```

Listing A.1: TravisTorrent query: Find the amount of failed runs

```
(
  SELECT gh_build_started_at, true as failed
  FROM ‘travistorrent‘
  WHERE
    tr_build_id IN (
      SELECT DISTINCT(tr_prev_build)
      FROM ‘travistorrent‘
      WHERE tr_log_bool_tests_ran=true AND tr_log_bool_tests_failed=true
    )
    AND gh_build_started_at IS NOT null AND tr_log_bool_tests_failed=true
) UNION ALL (
  SELECT gh_build_started_at, false as failed
  FROM ‘travistorrent‘
  WHERE tr_build_id IN (
    SELECT DISTINCT(tr_prev_build)
    FROM ‘travistorrent‘
    WHERE tr_log_bool_tests_ran=true AND tr_log_bool_tests_failed=false
  )
  AND gh_build_started_at IS NOT null AND tr_log_bool_tests_failed=true
)
```

Listing A.2: TravisTorrent query: Find the probability of consecutive failures