

# Thesis (working draft)

Paper: Working draft

Pieter De Clercq

April 4, 2020

# Acknowledgements

(TODO)

# Contents

<b>1</b>	<b>Introduction</b>	<b>3</b>
<b>2</b>	<b>Software Engineering</b>	<b>4</b>
2.1	Software Development Life Cycle . . . . .	4
2.1.1	Test Suite Assessment . . . . .	6
2.2	Agile Software Development . . . . .	11
2.2.1	Agile Manifesto . . . . .	11
2.2.2	The need for Agile . . . . .	15
2.2.3	Continuous Integration . . . . .	15
<b>3</b>	<b>Related work</b>	<b>20</b>
3.1	Classification of approaches . . . . .	21
3.1.1	Test Suite Minimisation . . . . .	21
3.1.2	Test Case Selection . . . . .	22
3.1.3	Test Case Prioritisation . . . . .	22
3.2	Algorithms . . . . .	23
3.2.1	Greedy algorithm . . . . .	23
3.3	Existing implementations . . . . .	23
<b>4</b>	<b>Proposed framework: VeloCity</b>	<b>24</b>
4.0.1	Architectuur . . . . .	24
4.0.2	Junit-reorder . . . . .	24
<b>5</b>	<b>Results and evaluation</b>	<b>26</b>
<b>6</b>	<b>Other cost-reducing factors</b>	<b>27</b>
<b>7</b>	<b>Conclusion and future work</b>	<b>28</b>

# Chapter 1

## Introduction

(TODO)

- economische impact (minder tijdverlies) - ecologische impact (minder elektriciteit)

# Chapter 2

## Software Engineering

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" [22, 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) [22, 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 [25, 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 [15, 21], 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 [21]. 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 [15], each having advantages and disadvantages, but I will consider the basic yet most widely used model, which is the Waterfall model by Benington [4]. 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 [34]. This version allows a phase to transition back to a previous phase (Figure 2.1).

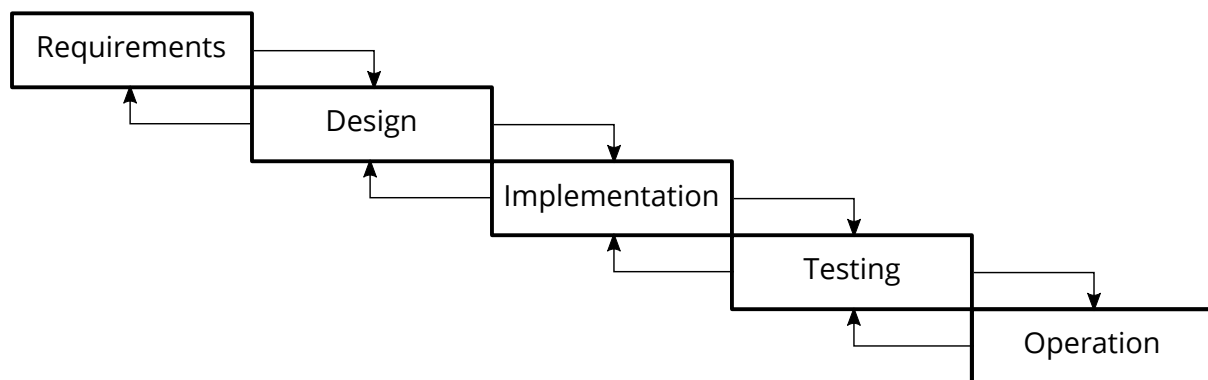


Figure 2.1: Improved Waterfall model by Royce

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 [31] 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 [20].

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 [26]. 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 [38]. 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* [20, 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 [38]. 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* [20, 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 [22, 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 warn the implementation against subsequent failures if the same conditions would reapply in the future.

## 2.1.1 Test Suite Assessment

### 2.1.1.1 Coverage

The most frequently used metric to measure the quantity and thoroughness of a test suite is the *code coverage* or *test coverage* [22, 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 [30], 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 [20], 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 [30, 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.

```
1 int example(int a, int b) {  
2     if (a == 0 || b != 0) {  
3         return a / b;  
4     }  
5 }
```

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 [30, 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* [30, 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 [7]. 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<sup>1</sup> for Java, coverage.py<sup>2</sup> for Python and simplecov<sup>3</sup> 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.

### 2.1.1.2 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 [33]. 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 [23].

**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

---

<sup>1</sup><https://www.jacoco.org/jacoco/>

<sup>2</sup><https://github.com/nedbat/coveragepy>

<sup>3</sup><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 [23, 33].

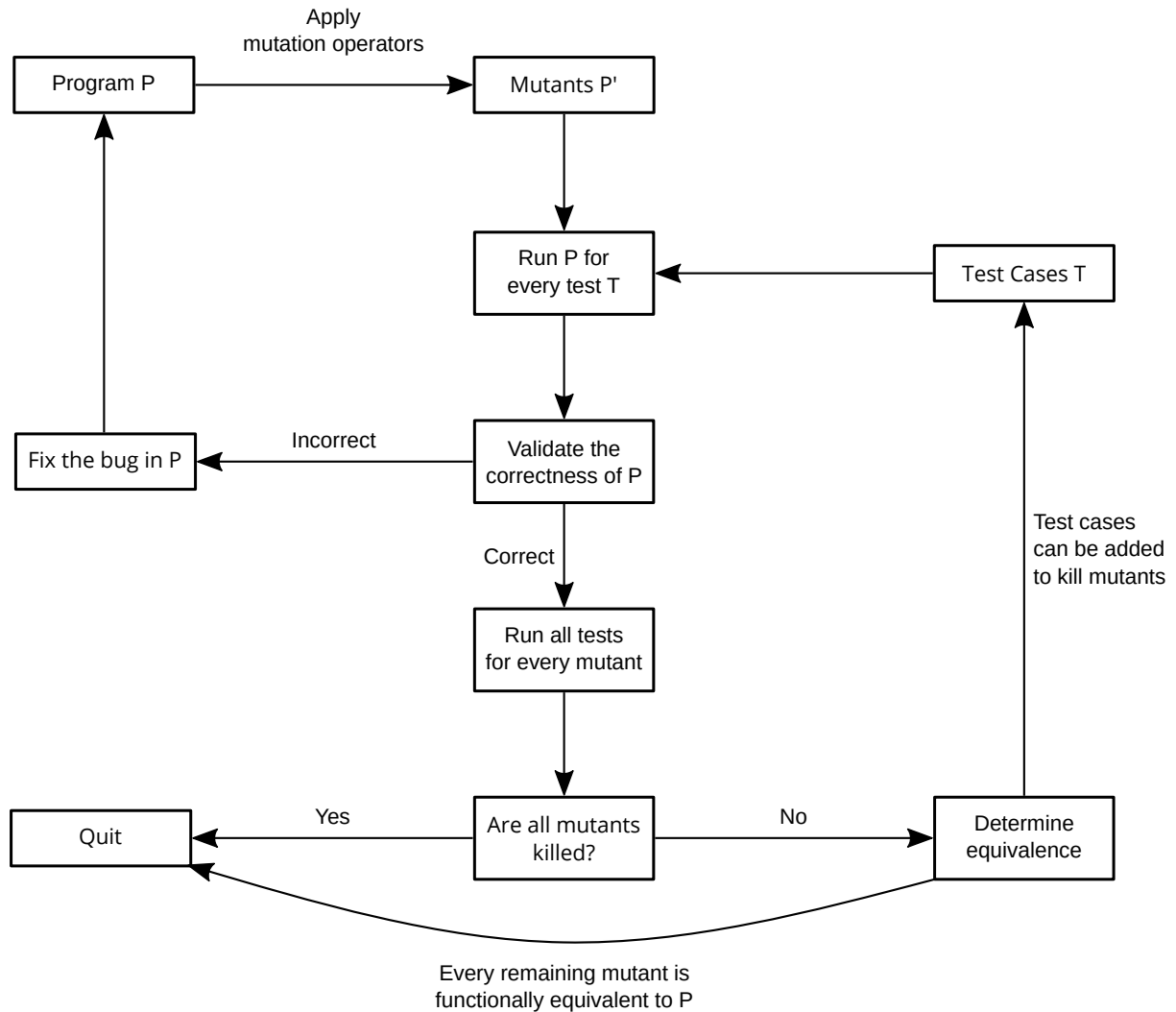





Figure 2.2: Process of Mutation Testing (based on [33])

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

$$\text{Mutant Score} = \frac{\text{killed mutants}}{\text{non-equivalent mutants}} \quad (2.1)$$

## io.github.thepieterdc.http.impl

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
<a href="#">HttpClientImpl</a>		59%		14%	7	14	18	40	2	9	0	1
<a href="#">HttpResponseImpl</a>		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%
<pre> 1   def smile(): 2       return ":" 3   4   def frown(): 5       return ":("</pre>				
<b>Total</b>	<b>4</b>	<b>1</b>	<b>0</b>	<b>75%</b>

(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 [18]. 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" [29, p. 2]. Martin identifies the principles as "the characteristics that differentiate a set of agile practices from a heavyweight process" [28, 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 [24, p. 12].

#### 2.2.1.1 *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 [14].

**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 [14]. Any factor that negatively impacts a healthy environment or decreases motivation should be changed [28, 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 [10, 14].

**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 [28].

**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 [28]. 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 [10].

**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 [28]. 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 [14].

#### **2.2.1.2 *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 [17].

**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 [28], 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 [10]. Glass criticises this statement and sees little point in delivering development versions to the customer [14].

**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 [14]. In agile software development, the progress is measured directly by the fraction of completed functionality [28].

**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 [14]. Secondly, as section 2.2.1.4 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 [28].

**2.2.1.3 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 [17].

**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.” [28]. This has already been emphasised before by the principles discussed in section 2.2.1.1. 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 [14].

**2.2.1.4 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 [14]. 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 [17].

**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 [28].

**Principle 9: Continuous attention to technical excellence and good design enhances agility.**

“High quality is the key to high speed”, according to Fowler [28]. 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 [10].

### 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 [19]. As was previously described in section 2.2.1.2, 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.

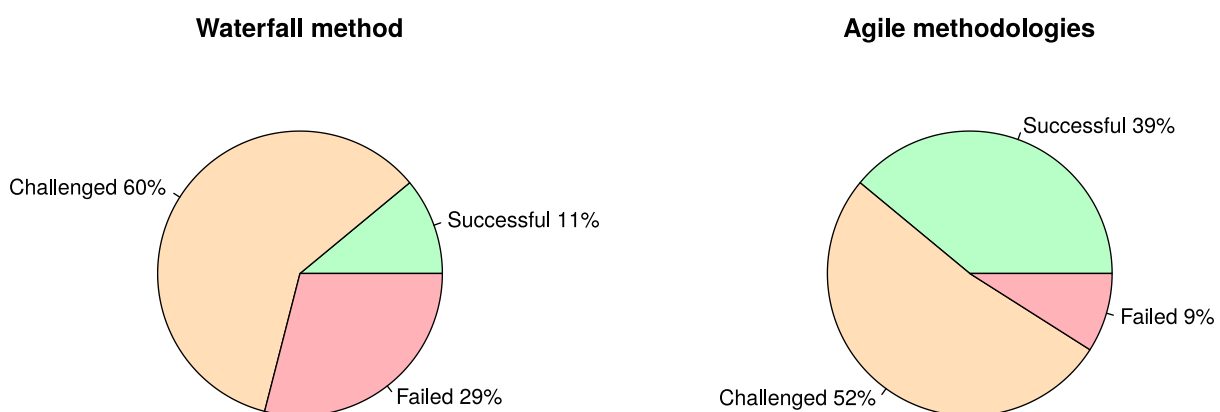


Figure 2.4: Success rate of Agile methodologies [16].

### 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” [29, 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 [35].

Contrarily, agile development methodologies advocate the idea of frequent, yet small deliveries (section 2.2.1.2). 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 [13]. This practice of frequent builds is referred to as *Continuous Integration* [28, 29]. 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 [6, p.11]. Cusumano reports that Microsoft typically builds the entire application at least once per day [6, 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 [28].
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.

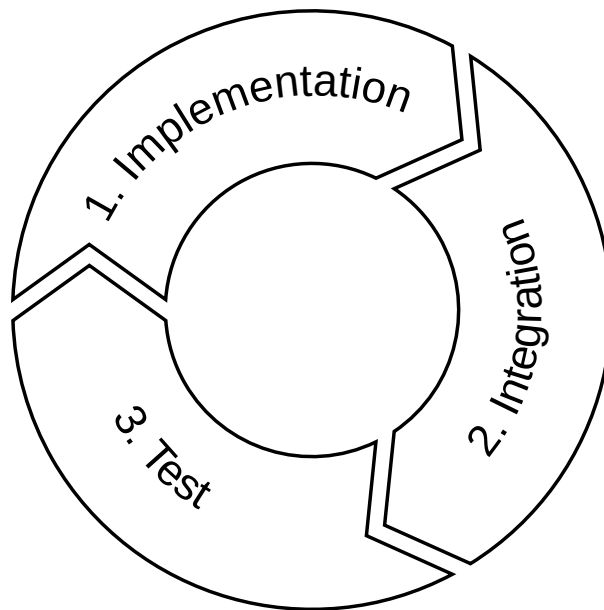


Figure 2.5: Development Life Cycle with Continuous Integration

### 2.2.3.1 Jenkins

Jenkins CI<sup>4</sup> 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 [35]. 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 [27].



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

---

<sup>4</sup><https://jenkins.io/>

### 2.2.3.2 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<sup>5</sup>. 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 [9]. 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>)

### 2.2.3.3 GitLab CI

GitLab, the main competitor of GitHub, announced its own Continuous Integration service in late 2012 named GitLab CI<sup>6</sup>. 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 [12]. A downside of using GitLab CI is the lack of a community-driven plugin system, however this is a planned feature<sup>7</sup>.



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

---

<sup>5</sup><https://github.blog/2019-08-08-github-actions-now-supports-ci-cd/>

<sup>6</sup><https://about.gitlab.com/blog/2012/11/13/continuous-integration-server-from-gitlab/>

<sup>7</sup><https://gitlab.com/gitlab-org/gitlab/issues/15067>

#### 2.2.3.4 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 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 [8]. 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 [36].



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

# Chapter 3

## Related work

In the previous chapter I have stressed the paramount importance of periodically integrating one's changes into the upstream repository. Additionally, Continuous Integration was introduced as both a practice and a tool to facilitate this often complex and time-consuming process. However, Continuous Integration is not the golden bullet for software engineering. In this chapter I will investigate the flip side of applying this practice. After every integration, all of the unit and regression tests in the test suite must be executed to ensure that the integration was successful and that no new bugs have been introduced. As the project evolves, the size of the codebase increases and consequently the amount of tests will increase as well in order to maintain a sufficiently high coverage level. An increase in the size of the test suite will inevitably lead to an increase in test duration [32], which imposes an issue of scaling. Walcott, Soffa and Kapfhammer illustrate the magnitude of this problem by providing an example of a codebase consisting of 20.000 lines, for which the tests require up to seven weeks to complete [37].

Fortunately, multiple developers and researchers have found some techniques that can be used to address the scalability issues of growing test suites. The techniques currently known to literature can be classified in three categories. Developers can either apply *Test Suite Minimisation*, *Test Case Selection* or *Test Case Prioritisation* [32]. All three techniques are applicable to any test suite, however there is a trade-off to be made. Depending on which technique is chosen, it will either have a major impact on the duration of the test suite execution in exchange for a reduced test coverage level, or it will result in a higher test adequacy.

In the following sections I will first elaborate on the details of these three approaches, then I will provide accompanying algorithms that can be used. Since the approaches share common ideas, the algorithms can (albeit with minor modifications) be reused across all approaches. In the final section I will illustrate some examples of the discussed techniques and algorithms that are currently integrated in existing software testing frameworks.

## 3.1 Classification of approaches

### 3.1.1 Test Suite Minimisation

Test Suite Minimisation, also referred to as *Test Suite Reduction*, aims to reduce the size of the test suite by permanently removing redundant tests. This problem is formally defined by Rothermel in definition 1 [39].

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

*Given:*

- $T = \{t_1, \dots, t_n\}$  a test suite consisting of tests  $t_j$ .
- $R = \{r_1, \dots, r_n\}$  a set of requirements that must be satisfied in order to provide the desired “adequate” testing of the program.
- $\{T_1, \dots, T_n\} \subseteq T$  subsets of test cases, 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  $r_i$ .

*Test Suite Minimisation is then defined as the task of finding a set  $T'$  of test cases  $t_j \in T$  that satisfies all requirements  $r_i$ .*

If we apply this definition to the concepts introduced in chapter 2, the requirements  $R$  can be interpreted as lines in the codebase that must be covered. With respect to the definition, a requirement can be satisfied by any test  $t_j$  that belongs to subset  $T_i$  of  $T$ . Observe that the problem of finding  $T'$  is closely related to the *hitting set problem* (definition 2) [39].

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

*Given:*

- $S = \{s_1, \dots, s_n\}$  a finite set of elements.
- $C = \{c_1, \dots, c_n\}$  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,  $T'$  is precisely 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 [39], which is an NP-complete problem as it can be reduced to the *Vertex Cover*-problem [11].

### 3.1.2 Test Case Selection

The second algorithm closely resembles the previous one. Instead of determining the minimal hitting set of the test suite in order to permanently remove tests, this algorithm has a notion of context. Prior to the execution of the tests, the algorithm performs a *white-box static analysis* of the codebase to identify which parts have been changed. Subsequently, only the tests regarding modified parts are executed, making the selection temporary and modification-aware [39]. 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  $P'$ .*

### 3.1.3 Test Case Prioritisation

Where the previous algorithms both attempted to execute as few tests as possible, it might sometimes be desired or even required that all tests pass. In this case, the previous ideas can be used as well. In Test Case Prioritisation, we want to find a permutation of the sequence of all tests instead of eliminating certain tests. The order of the permutation is chosen specifically to achieve a given goal as soon as possible, allowing for early termination of the test suite upon failure [39]. Some examples of goals include covering as many lines of code as fast as possible, or early execution of tests with a high probability of failure. A formal definition of this algorithm is provided in definition 4.

**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 tests to find the optimal permutation.

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

## 3.2 Algorithms

In subsection 3.1.1 I have already mentioned that applying Test Suite Minimisation is essentially the same as finding the minimal hitting set of the test suite and the set of requirements, which is an NP-complete problem. Therefore, we require the use of *heuristics* that are already known to literature to solve other, related problems. A heuristic is an experience-based method that can be used to solve a hard to compute problem by finding a fast approximation [20]. However, the found solution will mostly be suboptimal or might sometimes even fail to find any solution at all.

### 3.2.1 Greedy algorithm

The first algorithm is a *greedy* heuristic, which was originally designed to find an approximation for the set-covering problem [32]. A greedy algorithm always makes a locally optimal choice, assuming that this will eventually lead to a globally optimal solution [5]. Algorithm 1 presents the Greedy algorithm for Test Suite Minimisation. The greedy aspect in this algorithm is to always select the test case that contributes the most additional coverage.

---

**Algorithm 1** Greedy algorithm for TSM

---

1: *true*

---

## 3.3 Existing implementations

- OpenClover (enkel Java) heeft hier misschien support voor



# Chapter 4

## Proposed framework: VeloCity

(TODO)

- Implementatiedetails van algoritmes
- Uitwerking: nog onder voorbehoud (2e semester)
- Metapredictor: Voer alle algoritmes eens uit en rangschik ze volgens hoe goed ze het voorspeld hebben
- Scoringsmechanisme: Nog bepalen
- Junit: <https://www.baeldung.com/junit-5-test-order>

### 4.0.1 Architectuur

- Commit - POST /commit data: parent commit hash, huidige commit hash result: volgorde van tests - voer tests uit in gegeven volgorde - POST /result data: huidige commit hash, gefaalde tests

### 4.0.2 Junit-reorder

- Orde vastleggen in yaml files (plaats voorbeeld) - Out of the box support om tests binnen eenzelfde klasse in volgorde uit te voeren, maar niet class-wide support - Na elke uitvoering van een test wordt een nieuwe processor aangemaakt met een filter op de methodenaam, dit heeft als nadeel dat hergebruiken van initialisaties niet gaat. Misschien hier nog iets op vinden - Parallelliseren momenteel niet gesupport, geen idee hoe dat zou werken met een volgorde (kan niet gewoon verdelen in volgorde want stel dat 1 test heel lang duurt dan is die volgorde niet meer juist) - Shortcuts genomen wegens problemen met gradle (shaded jars) -> bepaalde JUnit functionaliteit weggegooid (vooral @Ignored annotatie)

Na elke uitvoering van test wordt coverage data bijgehouden om later te analyseren door test processor hack (maak hier zo'n mooi ding van).

Coverage data formaat optimaliseren zodat xmlfile niet gigantisch groot wordt -> done

Algoritmes aanpassen zodat branches ook rekening mee gehouden wordt (stel dat test A lijn 5 covert en test B ook, maar lijn 5 is een if-statement met 2 condities en test A triggert enkel de eerste conditie, dan zal test B met lagere prioriteit worden uitgevoerd of zelfs niet omdat de lijn al gecoverd is) -> eventueel via mutator die branches naar lijnen uitsplitst in zowel tests als source

Andere programmeertalen kunnen op zelfde manier door tests sequentieel uit te voeren

# Chapter 5

## Results and evaluation

(TODO)

1904.09416.pdf heeft een hele benchmark van 35.000.000 runs

- Experiment setup
- Data verzameling
- Bespreek de geselecteerde projecten
- Resultaat van toepassing van alle algoritmes op alle projecten, met wat grafieken
- travistorrent.testroots.org - zie die site voor cite source; gehost op gcp
- dodona
- onderzoek op dat travistorrent ding of het klopt dat tests de tendency te hebben om te falen in volgende commits

# Chapter 6

## Other cost-reducing factors

(TODO; provisional: chapter might be omitted completely)

- kost van server die staat te idle'n

# Chapter 7

## Conclusion and future work

(TODO) - TCP combieren met ideeën uit TSM om zo bepaalde tests te skippen -> Test Suite Optimisation

# 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 (URN CST) 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] 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>.
- [5] Thomas H. Cormen et al. *Introduction to Algorithms, Third Edition*. 3rd. The MIT Press, 2009. ISBN: 0262033844.
- [6] Cusumano, Akindutire Michael, and Stanley Smith. "Beyond the waterfall : software development at Microsoft". In: (Feb. 1995).
- [7] Charles-Axel Dein. *dein.fr*. Sept. 2019. URL: <https://www.dein.fr/2019-09-06-test-coverage-only-matters-if-at-100-percent.html>.
- [8] 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.
- [9] *Features • GitHub Actions*. URL: <https://github.com/features/actions>.
- [10] Martin Fowler and Jim Highsmith. "The Agile Manifesto". In: 9 (Nov. 2000).
- [11] 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.
- [12] *GitLab CI/CD*. URL: <https://docs.gitlab.com/ee/ci/>.
- [13] *GitLab Continuous Integration & Delivery*. URL: <https://about.gitlab.com/stages-devops-lifecycle/continuous-integration/>.
- [14] Robert L Glass. "Agile versus traditional: Make love, not war!" In: *Cutter IT Journal* 14.12 (2001), pp. 12–18.
- [15] 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.

- [16] Standish Group et al. "CHAOS report 2015". In: *The Standish Group International* (2015). URL: [https://www.standishgroup.com/sample\\_research\\_files/CHAOSReport2015-Final.pdf](https://www.standishgroup.com/sample_research_files/CHAOSReport2015-Final.pdf).
- [17] 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](https://doi.org/10.1007/978-3-319-10157-6_3).
- [18] Jim Highsmith. *History: The Agile Manifesto*. 2001. URL: <https://agilemanifesto.org/history.html>.
- [19] 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.
- [20] "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.
- [21] "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.
- [22] "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.
- [23] 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.
- [24] 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.
- [25] 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>.
- [26] 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.
- [27] Simon Maple. *Development Tools in Java: 2016 Landscape*. July 2016. URL: <https://www.jrebel.com/blog/java-tools-and-technologies-2016>.

- [28] Robert C. Martin and Micah Martin. *Agile Principles, Patterns, and Practices in C# (Robert C. Martin)*. USA: Prentice Hall PTR, 2006. ISBN: 0131857258.
- [29] 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](https://doi.org/10.1007/978-3-319-05155-0_1).
- [30] Glenford J. Myers, Corey Sandler, and Tom Badgett. *The Art of Software Testing*. 3rd. Wiley Publishing, 2011. ISBN: 1118031962.
- [31] 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.
- [32] 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.
- [33] 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](https://doi.org/10.1007/978-1-4757-5939-6_7).
- [34] 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>.
- [35] 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/>.
- [36] Travis. *Travis CI - Test and Deploy Your Code with Confidence*. Feb. 2020. URL: <https://travis-ci.org>.
- [37] 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>.
- [38] 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.



- [39] 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>.