

Terms

Acronyms

CI Continuous Integration. 13, 17

IEEE The Institute of Electrical and Electronics Engineers. 2

SDLC Software Development Life Cycle. 2

TCP Test Case Prioritisation. 17, 20, 21

TCS Test Case Selection. 17, 19, 20

TDD Test-driven development. 12

TSM Test Suite Minimisation. 17, 18, 20, 21

Chapter 1

Software Engineering

The Institute of Electrical and Electronics Engineers (IEEE) defines the practice of Software Engineering as the “Application of a systematic, disciplined, quantifiable approach to the development, operation and maintenance of software; that is, the application of engineering to software” [17, p. 421]. The word “systematic” in this definition emphasises the need for a structured process, depicting guidelines and models that describe how we should develop software in the most efficient way possible. Such a process does exist under the name of the Software Development Life Cycle (SDLC) [17, p. 420]. If a developer prefers not to abide by any model and act as they deem correct without following any guidelines, we employ the term *Cowboy coding* [19, p. 34].

1.1 Software Development Life Cycle

An implementation of the SDLC typically consists of several phases and a model that describes the transition from every phase to another. Depending on the nature of the software, we can either omit some or add more phases. The five phases below were compiled from multiple sources [12, 16] and describe a generic approach to which most software projects adhere.

1. **Requirements phase:** In the first phase of the development process, the developers acquaint themselves with the project and compile a list of the desired functionalities [16]. Subsequently, the developers can decide on the financial details, the required hardware specifications as well as which external software libraries 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 construct an architectural design of the application. This design consists of multiple documents, such as user stories and UML-diagrams. A user story describes which actions can be performed by which users, whereas a UML-diagram specifies the technical interaction between the individual components.
3. **Implementation phase:** In the third phase, the developers will write code according to the specifications defined in the architectural designs.

4. **Testing phase:** The fourth phase is the most critical. This phase will require the developers and quality assurance managers to test the implementation of the application thoroughly. The goal of this phase is to identify potential bugs before the application is made available to other users.
5. **Operational phase:** The final phase marks the completion of the project, after which the developers can integrate it into the existing business environment of their customer.

After we have identified the phases, we must define the transition from one phase into another phase using a model. Multiple models exist in the literature [12], with each model having its advantages and disadvantages. This thesis will consider the traditional model, which is still widely used as of today. The base of this model is the Waterfall model by Benington [5]. Similar to a real waterfall, this model executes every phase in cascading order. However, this imposes several restrictions. The most prevalent issue is the inability to revise a design decision when performing the actual implementation. To mitigate this problem, Royce has proposed an improved version of the Waterfall model [27], which does allow a phase to transition back to any preceding phase. Figure 1.1 illustrates this updated model.

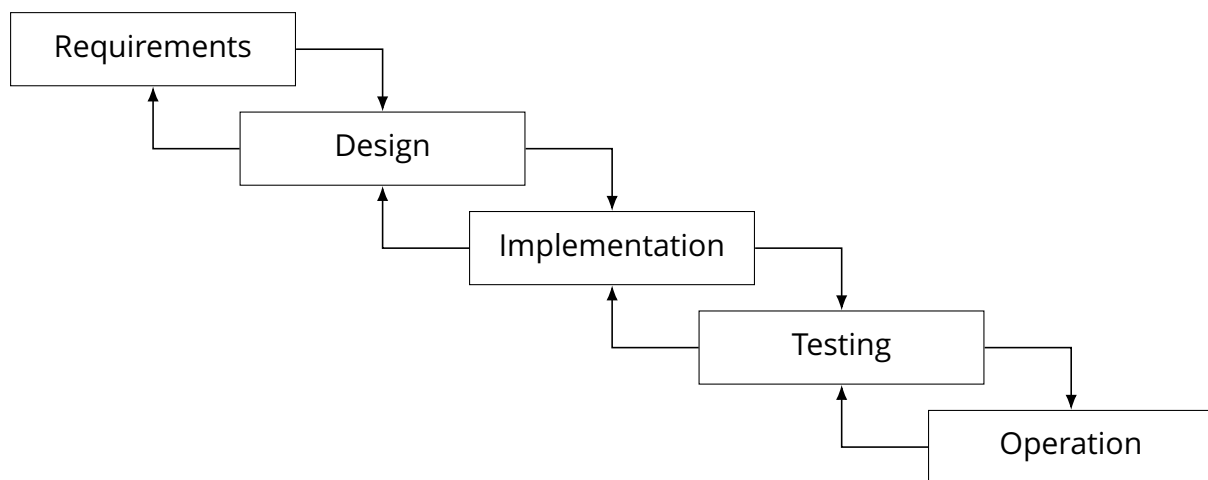


Figure 1.1: Improved Waterfall model by Royce

The focus of this thesis will be on the implementation and testing phases, as these are the most time-consuming phases of the entire process. The modification that Royce has applied to the Waterfall model is particularly useful when applied to these two phases in the context of *software regressions* [24]. We employ the term “regression” when a feature that was once working as intended is suddenly malfunctioning. The culprit of this problem can be a change in the code, but this behaviour can also have an external cause, such as a change in the system clock due to daylight saving time. Sometimes, a regression can even be the result of a change to another, seemingly unrelated part of the application code [15].

1.1.1 Taxonomy of test cases

Software regressions and other functional bugs can ultimately incur disastrous effects, such as severe financial loss or permanent 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 [20]. In order to reduce the risk of bugs, we should be able to detect malfunctioning components as soon as possible to warden the application against potential failures before they occur. Consequently, we must consider the testing phase as the most critical phase of the entire development process and therefore include sufficient test cases in the application. The collection of all the test cases in an application is referred to as the *test suite*. We can distinguish many different types of test cases. This thesis will consider three categories in particular.

Unit tests

This is the most basic test type. The purpose of a unit test is to verify the behaviour of an individual component [30]. As a result, the scope of a unit test is limited to a small and isolated piece of code, e.g. one function. Implementation-wise, a unit test is typically an example of a white-box test [15, p. 12]. The term white-box indicates that the creator of the test case can manually inspect the code before constructing the test. As such, they can identify necessary edge values or corner cases. Common examples of these edge values include zero, negative numbers, empty arrays or array boundaries that might cause an overflow. Once the developer has identified the edge cases, they can construct the unit test by repeatedly calling the function under test, each time with a different (edge) argument value, and afterwards verifying its behaviour and result. These verifications are referred to as *assertions*. Example 1.1 contains a unit test written in Java using the popular JUnit test framework.

```
1 public class ExampleUnitTest {
2     static int square(int base) { return base * base; }
3
4     @Test
5     public void testSquare() {
6         Assert.assertEquals(25, square(5));
7         Assert.assertEquals(4, square(-2));
8     }
9 }
```

Example 1.1: Java unit test in JUnit.

Integration tests

The second category involves a more advanced type of tests. An integration test validates the interaction between two or more individual components [30]. Ideally, accompanying unit tests should exist that test these components as well. As opposed to the previous unit tests, a developer will usually implement an integration test as a black-box test [15, p. 6]. A black-box test differs from the earlier white-box tests in the fact that the implementation details of the code under test are irrelevant for the construction of the test. Since a black-box test does not require any details about the code, we can, in fact, construct the integration tests before we implement the actual feature itself. A typical example of an integration test is the communication between the front-end and the back-end side of a web application. Another example is illustrated in Example 1.2.

```
1 public class ExampleIntegrationTest {
2     @Test
3     public void testOrderPizza() {
4         // Authenticate a test user.
5         Session session = UserSystem.login("JohnDoe", "password");
6         session.wallet.balance = 1000.0;
7         Assert.assertNotNull(session);
8
9         // Find an item to order.
10        Pizza pizza = new Pizza(Flavour.PEPPERONI);
11        Assert.assertNotNull(pizza);
12
13        // Create an order.
14        Order order = OrderSystem.createOrder(session, pizza);
15        Assert.assertNotNull(order);
16
17        // Checkout.
18        double oldBalance = session.wallet.balance;
19        order.checkout(session.wallet);
20        double newBalance = session.wallet.balance;
21        Assert.assertEquals(oldBalance - pizza.price, newBalance);
22    }
23 }
```

Example 1.2: Java integration test in JUnit.

Regression tests

After a developer has detected a regression in the application, they will add a regression test [17, p. 372] to the test suite. This regression test must replicate the exact conditions and sequence of actions that have triggered the failure. The goal of this test is to prevent similar failures to occur in the future if the same conditions would reapply. An example is provided in Example 1.3.

```
1 public class ExampleRegressionTest {
2     // Regression #439: A user cannot remove their comments after
3     they have changed their first name.
4     @Test
5     public void testRegression439() {
6         // Authenticate a test user.
7         Session session = UserSystem.login("johndoe", "password");
8         Assert.assertNotNull(session);
9
10        // Create a comment.
11        String content = "This is a comment by John Doe.";
12        Comment comment = CommentSystem.create(content, session);
13        Assert.assertNotNull(comment);
14
15        // Change the first name of the user.
16        Assert.assertEquals("Bert", session.user.firstName);
17        session.user.setFirstName("Matthew");
18        Assert.assertEquals("Matthew", session.user.firstName);
19        Assert.assertEquals("Matthew", comment.user.firstName);
20
21        // Try to remove the comment.
22        CommentSystem.remove(comment);
23        Assert.assertTrue(comment.removed);
24    }
25 }
```

Example 1.3: Java regression test in JUnit.

1.2 Test Suite Assessment

1.2.1 Coverage

The most frequently used metric to measure the quantity and thoroughness of a test suite is the *code coverage* or *test coverage* [17, p. 467]. The test coverage indicates which fraction of the application code is hit by at least one test case in the test suite. Internally, a coverage framework calculates the coverage ratio by augmenting every application statement using binary instrumentation. A hook is inserted before and after every statement to detect which statements are executed by the test cases. Many different criteria exist to interpret these results and thus to express the fraction of covered code [23]. The two most commonly used criteria are *statement coverage* and *branch coverage*.

Statement coverage Statement coverage expresses the fraction of statements that are executed by any test case in the test suite, over the total amount of statements in the code [15]. Similarly, we can calculate the *line coverage* as the fraction of covered code lines. Since one statement can span multiple lines and one line may also contain more than one statement, both of these criteria are intrinsically related. Statement coverage is heavily criticised in literature [23, p. 37] since it is possible to achieve a statement coverage percentage of 100 % on code of which we can prove it is malfunctioning. Consider example 1.4. If a test case would call the `example`-function twice with the arguments $\{a = 1, b = 2\}$ and $\{a = 5, b = 0\}$, then both test cases will pass and every statement will be covered, resulting in a statement coverage of 100 %. However, suppose we would call the function with arguments $\{a = 0, b = 0\}$. The first argument matches the first condition of the branch but will trigger a division-by-zero error, even though the previous combination of arguments reported a complete statement coverage. This simple example was already sufficient to illustrate that statement coverage is not trustworthy. However, statement coverage may still prove useful for other purposes, such as detecting unreachable code which we may safely remove.

```
1 int example(int a, int b) {  
2     if (a == 0 || b != 0) {  
3         return a / b;  
4     }  
5  
6     return 0;  
7 }
```




Example 1.4: Irrelevant statement coverage in C.

Branch coverage requires that the test cases traverse every branch of a conditional statement at least once [23, p. 37]. For an if-statement, we require two test cases, one for every possible outcome of the condition (`true` or `false`). For a loop-statement, we require at least two test cases as well. One test case should never execute the loop, and the other test case should execute every iteration. Optionally, we can add additional test cases for specific iterations. Observe that, while this criterion is stronger than statement coverage, it will still not detect the bug in Example 1.4. In order to mitigate this, we can use *multiple-condition coverage* [23, p. 40]. This criterion requires that for every conditional expression, every possible combination of subexpressions is evaluated at least once. If we apply this requirement to Example 1.4, the if-statement will only be covered if we test the following four cases.

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

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 every line of code, every statement or every branch must be explicitly covered [7]. Some parts of the code might be irrelevant or untestable, such as wrapper or delegation methods that only call a library function. All major programming languages have frameworks and libraries that enable the collection of coverage information during test execution, and each of these frameworks allows the exclusion parts of the code from the final coverage calculation. As of today, the most popular options are JaCoCo¹ for Java, coverage.py² for Python and simplecov³ for Ruby. These frameworks report in-depth statistics on the covered code and indicate which parts require more extensive testing, as illustrated in Figure 1.2.

io.github.thepieterdc.http.impl

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

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

¹<https://www.jacoco.org/jacoco/>

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

³<https://github.com/colszowka/simplecov>

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>				
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 1.2: Statistics from Code coverage frameworks.

1.2.2 Mutation testing

The previous section has explained how we can identify which parts of the code require additional test cases. However, we cannot yet measure the quality and resilience of the test suite nor its ability to detect future failures. To accomplish this, we can employ *mutation testing*. This technique creates several *mutants* of the application under test. A mutant is a syntactically different instance of the source code. We can create a mutant by applying one or more *mutation operators* to the original source code. These mutation operators attempt to simulate typical mistakes that developers tend to make, such as the introduction of off-by-one errors, removal of statements and the replacement of logical connectors [26]. 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 [18].

Creating and evaluating the mutant versions of the code is a computationally expensive process which typically requires human intervention. As a result, very few software developers have managed to employ this technique in practice. Figure 1.3 illustrates the process of applying mutation testing. The first step is to consider the original program P and a set of test cases TS , to which we apply mutation operators to construct a broad set of mutants P' . Next, we evaluate every test case $t \in TS$ on the original program P to determine the correct behaviour. Note that this step assumes that if the original source code passes the test cases, it is correct. This assumption will only be valid if the test suite contains sufficient thorough unit tests. If at least one of these test cases fails, we have found a bug which we must first resolve before continuing with the mutation analysis. When P successfully passes every test case, we evaluate every test case for each of the mutants. A mutant p' is said to be “killed” if its outcome is different from P for at least one test case. Otherwise, we refer to the mutant as “surviving”. After we have finished executing the test cases on every mutant, we analyse the set of surviving mutants. Every mutant that managed to survive implies a change in the source code that did not trigger any failure in the test cases. As a result, we need to introduce subsequent test cases until we have killed every mutant. However, it is also possible that the surviving mutants are functionally equivalent to P and are, therefore, correct. Since the detection of program equivalence is an impossible problem, we need to verify this manually [18, 26]. Finally, note that although we can use mutation testing to estimate the adequacy of the test suite, it is not flawless, as several mutation operators can cancel each other out [25].

After every mutant has either been killed or marked equivalent to the original problem, we can calculate the *mutation score* of the test suite using Equation (1.1). In an adequate test suite, this score is equal to 1, which indicates that the test suite was able to detect every mutant instantly.

$$\text{Mutant Score} = \frac{\text{killed mutants}}{\text{non-equivalent mutants}} \quad (1.1)$$

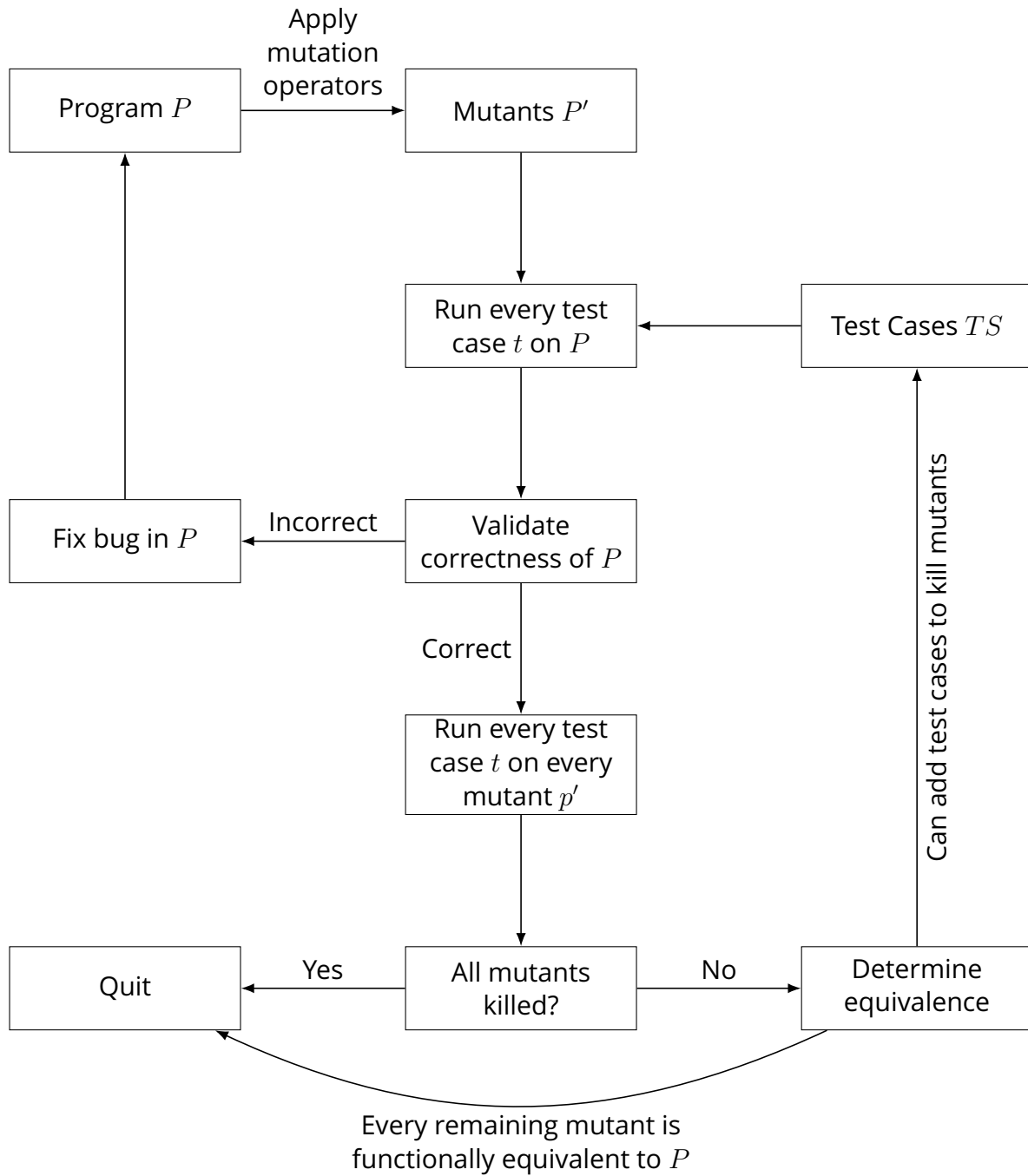


Figure 1.3: Process of Mutation Testing (based on [26]).

1.3 Agile Software Development

1.3.1 Agile Manifesto

Since the late 1990s, developers have tried to reduce the time occupied by the implementation and testing phases. As a result, several software pioneers have proposed new implementations of the SDLC, which were later collectively referred to as the *Agile development methodologies*. This term was coined during a meeting of seventeen prominent software developers, in which they have defined the following four fundamental values of Agile development in the *Agile Manifesto* [4].

1. *Individuals and interactions* over processes and tools.
2. *Working software* over comprehensive documentation.
3. *Customer collaboration* over contract negotiation.
4. *Responding to change* over following a plan.

According to the authors, we should interpret these values as follows: “While there is value in the items on the right, we value the items on the left more” [4]. When we examine these values more closely, we can observe that they all share a common philosophy, which is that software engineering should be a fast process in which communication and a short feedback loop is critical to avoid missteps. Since 2001, a variety of different programming models have arisen, each incorporating these agile principles in their own way. The most remarkable new practice is Test-driven development (TDD). Recall that an integration test is a black-box test and that as such, we can actually construct the test case in advance and write the implementation afterwards. This concept is also prevalent in TDD. This practice depicts that if we want to extend the functionality of the application, we should first modify the test cases (or add new test cases) and then modify the application code until every test case is passing [3].

1.3.2 The need for Agile

In the wake of the world economic crisis, software companies have been forced to devote efforts into researching how they can reduce their overall expenses. This research has concluded that in order to cut financial risks, developers should reduce the *time-to-market* of their applications [14]. As a result of this, the agile methodologies have received increased attention in scientific literature since this philosophy strives to deliver a minimal version as soon as possible. Afterwards, we can incrementally add additional features. This practice indeed results in a shorter time-to-market and lower costs, since one can decide to cancel the project much earlier in the process.

Additionally, 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 traditional methodologies (Figure 1.4).

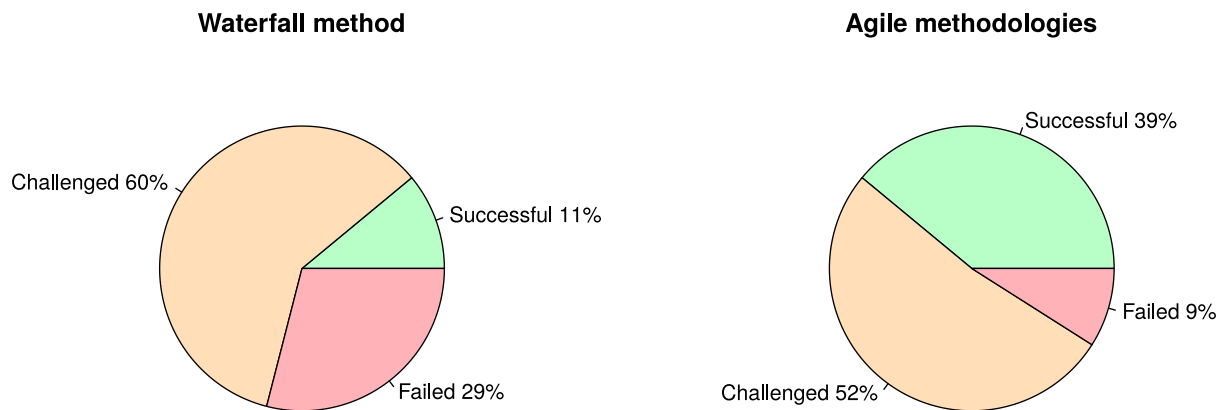


Figure 1.4: Success rate of Agile methodologies [13].

1.3.3 Continuous Integration

In traditional software development, the design phase usually leads to a representation of the required functionality in multiple, stand-alone modules. These modules are then implemented separately by the individual developers and are afterwards integrated into one monolithic application. This operation can prove to be very complicated since every developer can make their assumptions at the start of the project. Ultimately, these assumptions may render the components mutually incompatible. Furthermore, since it can take several weeks to months before this integration takes place, the developers often need to rewrite old code. Eventually, this will lead to unanticipated delays and costs [28].

Contrarily, agile development methodologies advocate the idea of frequent, yet small deliverables. In order to obtain frequent deliverables, we require to build the code often and integrate the modules multiple times, *continuously*, rather than just once at the end. Another advantage of frequent deliverables is the early identification of problems [11]. We refer to this practice as Continuous Integration (CI) [28]. Note that this idea has existed before the creation of the Agile manifesto. The first notorious software company that has adopted CI is Microsoft in 1989 [6]. Cusumano reports that Microsoft typically builds the entire application at least once per day and as such requires developers to integrate and test their changes multiple times per day.

The introduction of Continuous Integration in software development has significant consequences on the life cycle. Where the waterfall model used a cascading life cycle, CI employs a circular, repetitive structure consisting of three phases, as visualised in Figure 1.5.

1. **Implementation:** In the first phase, every developer writes code individually for their assigned module. At a regular interval, the code is committed to the remote repository.
2. **Integration:** When the developer commits their changes, they simultaneously fetch the changes other developers have made to their modules. Afterwards, the developer must integrate these remote changes with their local module to ensure these remain compatible. In case of a conflict, the developer is responsible for resolving this locally [22].
3. **Test:** After the developer has successfully integrated the remote modules to the locale module, the test suite must be executed to verify that no regressions have been introduced.

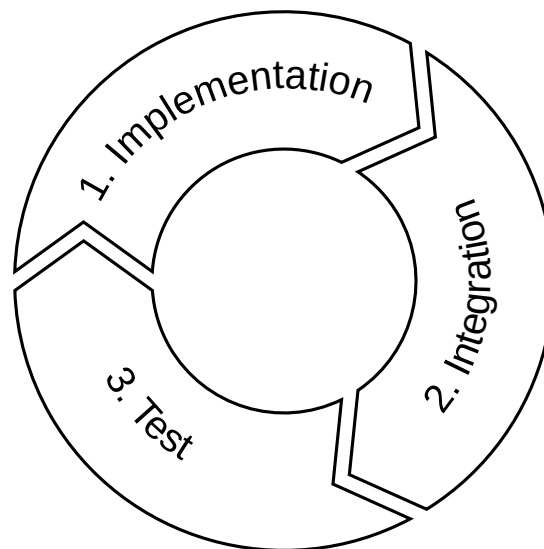


Figure 1.5: Development Life Cycle with Continuous Integration.

Adopting Continuous Integration can prove to be a lengthy and repetitive task. Luckily, a variety of tools and frameworks exist to automate this process. These tools are typically attached to a version control system (e.g. Git, Mercurial, ...) using a *post-receive* hook [28]. Every time the developers push a commit to the repository, the hook will notify the CI system. The CI system will respond accordingly by automatically building the code and executing the test cases. Optionally, we can configure the CI system to automatically publish successful builds to the end-users, a process referred to as *Continuous Delivery*. This paper will now proceed by discussing four prominent CI systems.

Jenkins

Jenkins CI⁴, “the leading open source automation server”, was started in 2004 by Kohsuke Kawaguchi as a hobby project. Initially launched as Hudson, the project was later renamed to Jenkins due to trademark issues [28]. Jenkins is programmed in Java and is currently maintained by volunteers. 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 in a private environment. Furthermore, Jenkins provides an open ecosystem that encourages developers to create new plugins and extend its functionality. Market research conducted by ZeroTurnaround in 2016 revealed that Jenkins is the preferred CI tool by 60 % of the developers [21].



Figure 1.6: Logo of Jenkins CI (<https://jenkins.io/>).

GitHub Actions

Following the successful beta of GitHub Actions which had started in August 2019, GitHub launched its Continuous Integration system later that year in November⁵. GitHub Actions executes builds in the cloud on servers owned by GitHub, restricting its use to GitHub repositories. Support for GitHub Enterprise repositories is not currently available. Developers can define builds using workflows, which can run both on Linux, Windows as well as macOS 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, we can extend GitHub Actions with custom plugins. These plugins can be composed either using a Docker container or using native JavaScript [1]. However, since this service is recent, it does not yet offer many plugins.



Figure 1.7: Logo of GitHub Actions (<https://github.com/features/actions>).

⁴<https://jenkins.io/>

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

GitLab CI

GitLab, the main competitor to GitHub, announced its own Continuous Integration service in late 2012 named GitLab CI⁶. The build configuration is specified in a *pipeline* and is executed by *GitLab Runners*. Developers may host these runners by themselves, or use shared runners hosted by GitLab [2]. Equivalent to the previously mentioned GitHub Actions, shared runners can be used for free by public repositories and are bound by quota for private repositories [10]. A downside of using GitLab CI is the absence of a community-driven plugin system, but support for plugins is planned⁷.



Figure 1.8: Logo of GitLab CI (<https://gitlab.com/>).

Travis CI

The final CI platform that will be discussed is Travis CI. This CI system was launched in 2011 and is only compatible with GitHub repositories. Travis CI build tasks are configured similarly to GitLab CI but lack support for self-hosted runners. In addition to commit-triggered builds, we can also schedule daily, weekly or monthly builds using *cronjobs*. Similar to GitHub Actions, open-source projects can use this service at zero cost, and a paid plan exists for private repositories [8]. It is not possible to create custom plugins, but Travis CI already features built-in support for a variety of programming languages. In 2020, almost one million projects are using Travis CI [29].



Figure 1.9: Logo of Travis CI (<https://travis-ci.com/>).

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

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

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](https://doi.org/10.26685/urncst.152).
- [3] Beck. *Test Driven Development: By Example*. USA: Addison-Wesley Longman Publishing Co., Inc., 2002. ISBN: 0321146530.
- [4] Kent Beck et al. *Manifesto for Agile Software Development*. 2001. URL: <https://www.agilemanifesto.org/>.
- [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] Michael 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](https://doi.org/10.1109/icsme.2019.00044). eprint: [arXiv:1904.09416](https://arxiv.org/abs/1904.09416).
- [9] *Features • GitHub Actions*. URL: <https://github.com/features/actions>.
- [10] *GitLab CI/CD*. URL: <https://docs.gitlab.com/ee/ci/>.
- [11] *GitLab Continuous Integration & Delivery*. URL: <https://about.gitlab.com/stages-devops-lifecycle/continuous-integration/>.
- [12] 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.
- [13] Standish Group et al. "CHAOS report 2015". In: *The Standish Group International* (2015). URL: https://www.standishgroup.com/sample_research_files/CHAOSReport2015-Final.pdf.
- [14] 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.

- [15] "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](https://doi.org/10.1109/IEEESTD.2013.6588537).
- [16] "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](https://doi.org/10.1109/IEEESTD.2015.7106435).
- [17] "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](https://doi.org/10.1109/IEEESTD.2017.8016712).
- [18] 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.
- [19] 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>.
- [20] 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](https://doi.org/10.1109/ECBS.1997.581900).
- [21] Simon Maple. *Development Tools in Java: 2016 Landscape*. July 2016. URL: <https://www.jrebel.com/blog/java-tools-and-technologies-2016>.
- [22] Robert C. Martin and Micah Martin. *Agile Principles, Patterns, and Practices in C# (Robert C. Martin)*. USA: Prentice Hall PTR, 2006. ISBN: 0131857258.
- [23] Glenford J. Myers, Corey Sandler, and Tom Badgett. *The Art of Software Testing*. 3rd. Wiley Publishing, 2011. ISBN: 1118031962.
- [24] 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.
- [25] 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](https://doi.org/10.1007/978-3-030-35510-4_4).
- [26] 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](https://doi.org/10.1007/978-1-4757-5939-6_7). URL: https://doi.org/10.1007/978-1-4757-5939-6_7.

- [27] 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>.
- [28] 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/>.
- [29] Travis. *Travis CI - Test and Deploy Your Code with Confidence*. Feb. 2020. URL: <https://travis-ci.org>.
- [30] 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](https://doi.org/10.1109/52.819971).