

Thesis (working draft)

Paper: Working draft

Pieter De Clercq

November 25, 2019

Acknowledgements

(TODO)

Contents

1	Introduction	3
2	Software Engineering	4
2.1	Software Development Life Cycle	4
2.2	Continuous Integration	8
3	Related work	9
4	Proposed framework: VeloCity	10
5	Results and evaluation	11
6	Other cost-reducing factors	12
7	Conclusion and future work	13

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" [8, 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) [8, p. 420].

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 [4] [7], 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 [7]. 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 [4], each having advantages and disadvantages, but I will consider the basic yet most widely used model, which is the Waterfall model by Benington [1]. 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 [12]. This version allows a phase to transition to a previous phase, as illustrated in Figure 2.1.

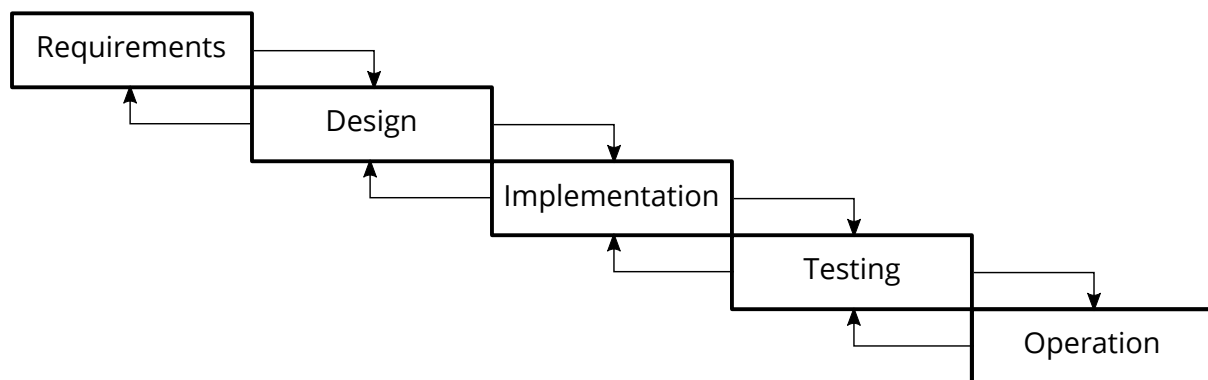


Figure 2.1: Improved Waterfall model by Royce

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 [11] 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 in the code of another, seemingly unrelated part of the application code [6].

Software regressions and other functional bugs can ultimately lead to 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 [9]. 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 include a plethora of tests. The set of all tests, or a smaller

chosen subset of certain tests of an application is referred to as the *test suite*. This thesis considers 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 [13]. 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* [6, 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 [13]. 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* [6, 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 [8, 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.

A frequently used metric to measure the quantity and effectiveness and thoroughness of a test suite is the *code coverage* or *test coverage* [8, 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 [10], 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 [6], 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 [10, 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. 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 [10, 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* [10, 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 [2]. 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 all of these frameworks include options to exclude 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 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 2.2.

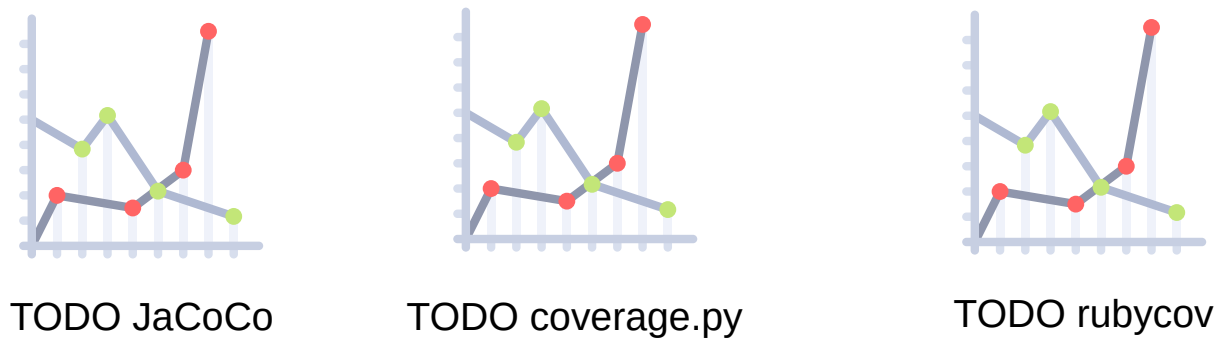


Figure 2.2: Statistics from Code coverage tools

2.2 Continuous Integration

(TODO)

Over the past decade, following the financial crisis of 2009 [5], extensive research has been put into reducing the time occupied by the implementation and testing phases. This resulted in a new development approach, known as *Agile programming* [3]. The main philosophy of Agile programming is that the time-to-market should be as short as possible, allowing end-users to provide feedback much earlier in the process. (TODO)

- buildup naar waarom tooling nodig is
- waarom
- wat
- voorbeelden: Jenkins, CircleCI, Travis-CI, recent GitHub Actions + screenshots
- Probleem en oplossingen met regression tests
- Test Case Prioritization -¿ Focus want geen tests weggooien
- Test Suite Minimization
- Test Suite Selection
- Test Suite Reduction

¹<https://www.jacoco.org/jacoco/>

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

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

Chapter 3

Related work

(TODO)

- OpenClover (enkel Java) heeft hier misschien support voor
- Machine Learning approaches
- Heuristieken

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

Chapter 5

Results and evaluation

(TODO)

- Experiment setup
- Data verzameling
- Bespreek de geselecteerde projecten
- Resultaat van toepassing van alle algoritmes op alle projecten, met wat grafieken

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)

Bibliography

- [1] 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>.
- [2] Charles-Axel Dein. *dein.fr*. Sept. 2019. URL: <https://www.dein.fr/2019-09-06-test-coverage-only-matters-if-at-100-percent.html>.
- [3] Torgeir Dingsøy, Tore Dybå, and Nils Moe. “Agile Software Development: An Introduction and Overview”. In: *Agile Software Development, by Dingsøy, Torgeir; Dybå, Tore; Moe, Nils Brede, ISBN 978-3-642-12574-4. Springer-Verlag Berlin Heidelberg, 2010, p. 1 -1* (Apr. 2010), p. 1. DOI: 10.1007/978-3-642-12575-1_1.
- [4] 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.
- [5] 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.
- [6] “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. ISSN: null. DOI: 10.1109/IEEESTD.2013.6588537.
- [7] “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. ISSN: null. DOI: 10.1109/IEEESTD.2015.7106435.
- [8] “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.
- [9] 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.
- [10] Glenford J. Myers, Corey Sandler, and Tom Badgett. *The Art of Software Testing*. 3rd. Wiley Publishing, 2011. ISBN: 1118031962, 9781118031964.
- [11] 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.

- [12] 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>.
- [13] 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.