Thesis (working draft)

Paper: Working draft

Pieter De Clercq

January 13, 2020

GHENT UNIVERSITY

FACULTY OF SCIENCES

Acknowledgements

(TODO)

Contents

1 Introduction **3**

2 Software Engineering **4**

2.1 Software Development Life Cycle 4

2.2 Continuous Integration 9

2.2.1 Agile Manifesto 9

2.2.2 The need for Agile 11

3 Related work **12**

4 Proposed framework: VeloCIty **13**

5 Results and evaluation **14**

6 Other cost-reducing factors **15**

7 Conclusion and future work **16**

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" [11, 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) [11, 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 [13, 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 [5] [10], 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 [10]. 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 [5], each having advantages and disadvantages, but I will consider the basic yet most widely used model, which is the Waterfall model by Benington [2]. 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 [18]. This version allows a phase to transition to a previous phase, as illustrated in Figure 2.1.

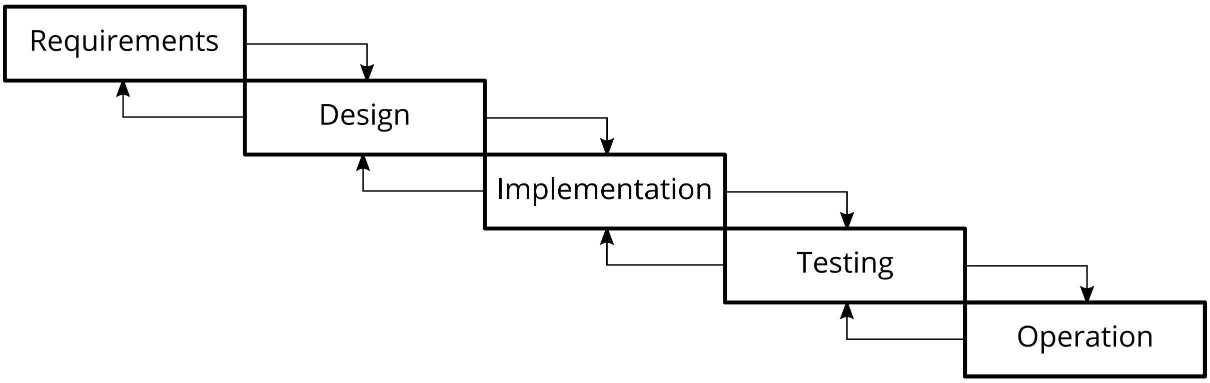

Figure 2.1: Improved Waterfall model by Royce

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 [17] 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 [9].

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 [14]. 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 [19]. 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* [9, 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 [19]. 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 [9, 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 [11, p. 372] is added to the test suite. This regression test should replicate the exact conditions and sequence of actions that have caused the regression, to warden the implementation against subsequent failures if the same conditions would reapply in the future.

A frequently used metric to measure the quantity and effectiveness and thoroughness of a test suite is the *code coverage* or *test coverage* [11, 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 [16], 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 [9], 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 [16, p. 37], since it is possible to achieve a statement coverage percentage of 100% on a code fragment which can be proven to be incorrect. Consider the code fragment in Listing 2.1. If a test would call the `example`-function with arguments $\{a = 1, b = 2\}$, the test will pass and every statement will be covered, resulting in a statement coverage of 100%. However, it is clear to see that if the function would be called with arguments $\{a = 0, b = 0\}$, a *division-by-zero* error would be raised, resulting in a crash. This very short example already indicates that statement coverage is not trustworthy, yet it may still be useful for other purposes, such as detecting unreachable code which may safely be removed.

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

Listing 2.1: Example of irrelevant statement coverage in C.

Branch coverage on the other hand, requires that every branch of a conditional statement is traversed at least once [16, 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* [16, 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 [3]. Some parts of the code might simply be irrelevant or untestable. Examples include wrapper or delegation methods that simply call a library function. All major programming languages have frameworks and libraries available to collect coverage information during test execution, and each of these frameworks allows the developer to exclude parts of the code from the final coverage calculation. As of today, the most popular options are JaCoCo[1] for Java, coverage.py[2] for Python and simplecov[3] for Ruby. These frameworks are able to generate in-depth statistics on which parts of the code are covered and which parts require more tests, as illustrated in Figure 2.2.

io.github.thepieterdc.http.impl

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

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

Coverage report: 75%

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

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

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

(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 2.2: Statistics from Code coverage tools

[1] `https://www.jacoco.org/jacoco/`
[2] `https://github.com/nedbat/coveragepy`
[3] `https://github.com/colszowka/simplecov`

2.2 Continuous Integration

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 [7]. 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" [1]. 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" [15, p. 2]. 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 briefly explain these values and their corresponding principles, using the mapping proposed by Kiv [12, 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 [4].

Principle 5: Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done.

Principle 6: The most efficient and effective method of conveying information to and within a development team is face-to-face conversation. TODO EXPLAIN

Principle 8: Agile processes promote sustainable development. The sponsors, developers, and users should be able to maintain a constant pace indefinitely. TODO EXPLAIN

Principle 11: The best architectures, requirements, and designs emerge from self-organizing teams. TODO EXPLAIN

Principle 12: At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behavior accordingly. TODO EXPLAIN

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 [6].

Principle 1: Our highest priority is to satisfy the customer through early and continuous delivery of valuable software. TODO EXPLAIN

Principle 3: Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale. TODO EXPLAIN

Principle 7: Working software is the primary measure of progress. TODO EXPLAIN

Principle 10: Simplicity–the art of maximizing the amount of work not done–is essential. TODO EXPLAIN

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 [6].

Principle 4: Business people and developers must work together daily throughout the project. TODO EXPLAIN

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 [4]. 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 [6]. This is accomplished by working in short iterations (sprints), instead of programming the entire application at once, which is the case with the traditional programming models.

Principle 2: Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage. TODO EXPLAIN

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

2.2.2 The need for Agile

Over the past decade, the agile methodologies have received increasing attention amongst software developers, following the world economic crisis of 2009 [8]. A consequence of this crisis was that software companies were forced to cut on their expenses and find ways to reduce the *time-to-market* of their applications.

- probleem met waterfall model -¿ een bron hiervoor?
- feedback loop
- 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

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; provisionary: chapter might be omitted completely)

- kost van server die staat te idle'n

Chapter 7

Conclusion and future work

(TODO)

Bibliography

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

[2] 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.

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

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

[5] 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.

[6] 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.

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

[8] 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.

[9] "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.

[10] "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.

[11] "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.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

[16] Glenford J. Myers, Corey Sandler, and Tom Badgett. *The Art of Software Testing*. 3rd. Wiley Publishing, 2011. ISBN: 1118031962, 9781118031964.

[17] 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.

[18] 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.

[19] 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.