Johannes Anton Rieder, BSc

Applying Software Engineering Research with Focus on Testing in an Industry Project

MASTER'S THESIS

to achieve the university degree of

Diplom-Ingenieur

Master's degree programme: Software Development and Business Management

submitted to

Graz University of Technology

Supervisor

Univ.-Prof. Dipl.-Ing. Dr.techn. Wolfgang Slany

Institute of Software Technology
Head: Univ.-Prof. Dipl.-Ing. Dr.techn. Wolfgang Slany

Graz, September 2019

This document is set in Palatino, compiled with pdfLaTeX2e and `Biber`.

The LaTeX template from Karl Voit is based on KOMA script and can be found online: https://github.com/novoid/LaTeX-KOMA-template

Affidavit

I declare that I have authored this thesis independently, that I have not used other than the declared sources/resources, and that I have explicitly indicated all material which has been quoted either literally or by content from the sources used. The text document uploaded to TUGRAZonline is identical to the present master's thesis.

Date

Signature

Abstract

This thesis will begin by discussing a paper by C.A.R. Hoare in chapter 1, which will act as a starting point for all the topics discussed in this thesis with plenty of mentions of another paper by B. A. Kitchenham et al.

The concept of Evidence-Based Software Engineering (EBSE) which "*is concerned with determining what works, when and where, in terms of software engineering practice, tools and standards*"[1] will be introduced in chapter 3. While a lot of the theory behind computer science (which a lot of the current software infrastructure is based on) can mostly be proven as they often have deep roots in formal, mathematical definitions stemming from the early origins of computer science — software engineering or development (two terms in many cases used interchangeably) practices are frequently based on anecdotal evidence about what used to work for a certain team, organization, or company in their specific context, rather than research and scientific methods which could provide solid proof, or, at least, strong correlation between project success or failure and certain practices. EBSE tries to apply Systematic Literature Review (SLR) as one tool to find empirical evidence of methods, techniques and processes that can be proven to work in a defined context.

After this, the very broad subjects of testing in chapter 2, code coverage in chapter 4 and finally Test-Driven Development (TDD) in chapter 5 will be discussed. TDD will be covered intensively, taking a look at the benefits and downsides of a test-first approach and it will be attempted to answer the question if these methodologies and techniques used by practitioners can be supported not just by anecdotal, but by scientific evidence. A couple of industry studies will be discussed, as well as their results and what conclusions can be drawn for other projects.

[1] *EBSE Website* 2012.

Type systems will be briefly discussed as well in chapter 6, which, compared to the industry studies, have deep roots in formal computer science theory. In this context, two languages in particular, JavaScript and Ruby will be discussed with regards to their type system properties because of their relevance to OSKAR.

At the end in chapter 7 it will be discussed how the acquired knowledge was applied in an industry project in the context of a server backend of the software "OSKAR" at NR.Systems GmbH, and to which extend.

Contents

Abstract iv

List of Figures viii

List of Tables ix

Listings x

1 Background 1
- 1.1 The Human Factor . . . 5

2 Automated Self-Testing Code 7
- 2.1 Self-Testing . . . 7
- 2.2 Automated . . . 8
- 2.3 Types of Tests . . . 10
- 2.4 Conclusion . . . 12

3 Testing as a Suitable Goal for EBSE 13
- 3.1 Lack of tests are Technical Debt . . . 14

4 Code Coverage 17
- 4.1 Types of Coverage . . . 18
- 4.2 "Mythical Unit Test Coverage" . . . 19
  - 4.2.1 The Influence of Code Complexity on Defect Rate and Code Coverage . . . 22
- 4.3 What is Code Coverage measuring, if not Test Sufficiency? . . . 24
- 4.4 SQLite . . . 25

5 Test-Driven Development 26
- 5.1 Goals of TDD . . . 27

5.2 The TDD Development Cycle . . . 28
5.2.1 Regression Tests . . . 29
5.2.2 Refactoring . . . 30
5.3 Criticism . . . 32
5.3.1 Criticism on Unit Testing . . . 33
5.3.2 Criticism on Refactoring . . . 33
5.3.3 Criticism on Simple Design and Lack of Planning . . . 35
5.4 Efficacy of Test-Driven Development . . . 35
5.4.1 George and Williams, 2004 . . . 36
5.4.2 Bhat and Nagappan, 2006 . . . 38
5.4.3 Nagappan et al., 2008 . . . 40
5.5 Is TDD dead? . . . 43
5.5.1 3X by Kent Beck . . . 46
5.5.2 Example of Test-Induced Design Damage . . . 48
5.5.3 Conclusion . . . 52

6 Type Systems 53
6.1 About Type Systems . . . 53
6.2 Strict and Dynamic Typing . . . 54
6.3 Claimed Benefits of Strict Typing . . . 55
6.4 Efforts being made for JavaScript . . . 58
6.4.1 Reason . . . 59
6.5 Efforts being made for Ruby . . . 59
6.5.1 Sorbet . . . 60
6.5.2 Ruby 3.0 . . . 62

7 OSKAR 64
7.1 Background . . . 64
7.2 OSKAR-Server Overview . . . 65
7.3 Principles of the Ruby Community . . . 65
7.4 TDD in OSKAR . . . 66
7.5 Coverage in OSKAR . . . 68
7.5.1 Code Complexity in OSKAR . . . 69

8 Conclusion 70

Bibliography 72

List of Figures

2.1 Testing pyramid with the different types of tests 11

3.1 Study overlap of Rios et al.'s tertiary study 16

4.1 Focus of the "Mythical Unit Test Coverage" study 20
4.2 Contour plot showing relation between max. block depth, coverage and defects 23
4.3 Contour plot showing relation between no. of versions, coverage and defects 23

5.1 TDD lifecycle 28
5.2 Waterfall model 37

6.1 Error model of Gao et al.'s study 57
6.2 Ruby 3 static analysis 62

List of Tables

4.1 List of measures with strong correlations 21
5.1 12 key XP practices 34

Listings

5.1 Original Rails code . . . 49
5.2 Hexagon-inspired, test-induced, "damaged" controller . . . 49
5.3 Hexagon-inspired, test-induced, "damaged" runner . . . 50
5.4 Hexagon-inspired, test-induced, "damaged" repository . . . 51
5.5 Hexagon-inspired, test-induced, "damaged" model . . . 51

6.1 Ruby's strong typing results in a `TypeError` . . . 55
6.2 JavaScript's type coercion leads to concatenation . . . 55
6.3 A more curious example of JavaScript's type coercion . . . 55
6.4 Ruby code annotated with Sorbet . . . 61
6.5 The resulting type errors when using the Sorbet runtime . . . 61
6.6 Ruby code without type signatures . . . 63
6.7 Corresponding Ruby Interface file . . . 63

7.1 Writing a failing test . . . 67
7.2 Running the test to see it fail . . . 67
7.3 Adding the minimal implementation to make the test green . 68
7.4 Running the test again to see it turn green . . . 68

1 Background

As an introduction to give this thesis a little background, it will start by citing a few passages of a paper written by Sir Charles Antony Richard Hoare (C.A.R. Hoare), more commonly referred to as Tony Hoare, a British computer scientist known for inventing the Quicksort sorting algorithm[1], Hoare logic[2] and often quoted for his apology for inventing the null-reference, calling it his billion-dollar mistake in a talk given in 2009[3]. In his paper "How Did Software Get So Reliable Without Proof?" from 1996 he describes how — in his opinion — software could grow from programs consisting of mere tens of thousands LOC[4] to millions of LOC in recent times[5] without causing too many catastrophic failures that were being predicted at the time.

His paper will act as a sort of common thread for this thesis, as it touches topics such as Evidence-Based Software Engineering (EBSE, chapter 3), testing and Test-Driven Development (TDD, chapter 5), metrics for measuring the efficacy of testing methods such as code coverage (chapter 4), regression testing (subsection 5.2.1), Return on Investment (ROI, section 5.4) of testing software (or rather the cost of not doing so) and several examples of how research has benefited practitioners of software development over the years (for example type systems, see chapter 6). Although not all of this is directly mentioned, one can easily draw parallels of what Hoare wrote in 1996 and what other researchers have found and how languages, frameworks and software development practices developed in the years afterwards.

[1] *Quicksort* 2019.

[2] *Hoare logic* 2019.

[3] *Null References* 2009.

[4] The source code for the Apollo 11 contains about 130.000 LOC. (Garry, 2019)

[5] As mentioned by Hoare in 1996, when this paper was written. But many modern software projects easily exceed millions of LOC.

Catastrophic disasters caused by software defects

> *On 4 June 1996, the maiden flight of the Ariane 5 launcher ended in a failure. Only about 40 seconds after initiation of the flight sequence, at an altitude of about 3700m, the launcher veered off its flight path, broke up and exploded.*
>
> (Lions, 1996, p. 1)

The primary cause of the explosion in the chain of technical events that led up the the catastrophe was a software defect, namely an unprotected (of an "Operand Error" to occur) conversion from a 64-bit floating point to a 16-bit integer value. (Lions, 1996, p. 4)

> *In March 2019, aviation regulators and airlines around the world grounded all Boeing 737 MAX passenger airliners after two MAX 8 aircraft crashed, killing the 346 people aboard.*
>
> (*Boeing 737 MAX groundings* 2019)

The primary cause for the crashes of two Boeing 737 MAX passenger airplanes was a software flaw in the MCAS (Maneuvering Characteristics Augmentation System). This software was supposed to emulate behavior of previous generations of 737 airplanes so pilots would not have to be trained separately for the 737 MAX. (*Lion Air Flight 610* 2019; *Boeing 737 MAX groundings* 2019)

> *The death of Elaine Herzberg was the first recorded case of a pedestrian fatality involving a self-driving (autonomous) car, following a collision that occurred late in the evening of March 18, 2018.*
>
> (*Death of Elaine Herzberg* 2019)

In this case the autonomous driving software detected Herzberg 6 seconds before impact, but did not determine an emergency break was needed until 1.3 seconds before impact, at which point it did not initiate the emergency breaking maneuver since these maneuvers were not enabled during computer control of the vehicle. (*Death of Elaine Herzberg* 2019)

Compilers, operating systems, large telecommunication network software etc. working without grand failures every day. Hoare asks if this was all thanks to rigorous application of scientific research results and comes to the conclusion that while if one were to just take a quick glimpse at software engineering practices they might think that research has not influenced practitioners of software development and that a large gap exists between practice and theory. But taking a closer look, one finds that research has greatly improved common practice, although technology transfer between computer science theory and software development practice is very slow. This can also be seen in today's efforts to retroactively fit type systems into languages whose strong selling point was indeed their dynamic nature, such as Python, Ruby and JavaScript, a topic which will be discussed later on in sections 6.4 and 6.5.

Coming back to Hoare's paper from 1996 he states the following:

> *Above all, the strictest management is needed to prevent premature commitment to start programming as soon as possible. This can only lead to a volume of code of unknown and untestable utility, which will act forever after as a dead weight, blighting the subsequent progress of the project, if any.*
>
> (Hoare, 1996, p. 3)

It can be seen that Hoare advocates quite strongly that proper planning and testing of software is of the utmost importance when writing software projects, although he does not mention any particular technique for either. Some hints towards what Kent Beck will later call Test-Driven Development in his book from 2003 can also be found in his paper, albeit not limited to self-testing code, but rather broadly applied to engineering as a whole:

> *It is to ensure that a test made on a product is not a test of the product itself but rather of the methods that have been used to produce it — the processes the production lines, the machine tools, their parameter settings and operating disciplines.*
>
> (Hoare, 1996, p. 5)

Hoare follows up with a mention that software engineering should take inspiration from other branches of science and engineering:

> *A testing strategy for computer programs must be based on lessons learned from the successful treatment of failure in other branches of science and engineering.*
>
> (Hoare, 1996, p. 5)

This is in line with the paper by B. A. Kitchenham et al. from 2004 which compares EBSE with evidence-based medicine (EBM), or rather uses EBM as an analogy and proposes adoption of certain practices known from EBM but also mentions where this analogy breaks down.

B. A. Kitchenham et al. suggest the goal of EBSE should be "*to provide the means by which current best evidence from research can be integrated with practical experience and human values in the decision making process regarding the development and maintenance of software.*" (B. A. Kitchenham et al., 2004, p. 274) as to have the means to measure, verify and possibly certify that software has been developed with scientifically proven best practices rather than the rather vague definition of "enterprise best practices" which can sometimes be found in marketing language, escaping any actual proper definition.

Furthermore it is suggested that a standard should be established to make research more comparable because existing research at the time of the writing of the paper was fragmented and limited. While having empirical studies is important, they are often targeted towards individual publications, researching an area of interest, rather than trying to provide more material for a reproducible body of studies to make them comparable and providing the possibility to systematically review them (Systematic Literature Review, SLR).

> *Without agreed standards. There are no generally accepted guidelines or standard protocols for conducting individual experiments. . . . empirical software engineering is badly in need of guidelines and protocols.*
>
> (B. A. Kitchenham et al., 2004, p. 277)

It can be seen that both Hoare and B. A. Kitchenham et al. advocate that software engineering practices should be based on scientific evidence rather

than fashion and hype. B. A. Kitchenham et al. state that practitioners often perceive the issues discussed by researchers to be of little relevance to their work in the industry and papers are often not written in a language which directly dictate or suggest a way of applying the newly found results. This is in line with Hoare's observation that a direct correlation between theory and practice is often not easy to find, as mentioned above. Hoare gives a few examples were theory and research have greatly impacted software development practices, from type theory to data types to the widespread adoption of structured programming (the avoidance of jumps and `goto`s) which can be traced back to the Böhm-Jacopini theorem[6] which provided proof that any program can be expressed in code that uses purely structured code instead of jumps. This practice has since been widely adopted, although some usages of `goto` still remain and are valid[7], even though it continues to pop up in negative contexts, such as the "gotofail"[8] which occurred in security relevant code of Apple's SSL implementation[9].

1.1 The Human Factor

> *Design and programming are human activities; forget that and all is lost.*
>
> (Stroustrup, 1997, p. 693)

Stroustrup, the inventor of the C++ programming language, said that people too often forget that the software development process — often defined as a chain of steps one has to take, with specific inputs resulting in specific outputs, resulting in the desired results — has a lot of human aspects as well. And because of the language used, programming language as well as natural language used to describe the requirements, these aspects are often concealed.

[6] *Structured Program Theorem (Böhm–Jacopini theorem* 2019.

[7] For example, the `tmux` program uses it, amongst other use cases, for error and failure handling. (*tmux usage of "goto"* 2019)

[8] *NVD - CVE-2014-1266* 2014.

[9] *ImperialViolet - Apple's SSL/TLS bug* 2014.

Both Hoare and B. A. Kitchenham et al. describe the challenge of the human factor in their papers, however in quite a different tone. B. A. Kitchenham et al. describe it more formally.

> *Although there are opportunities for individual software engineers and managers to adopt* EBSE *principles, the decision to adopt a technology is often an organizational issue that is influenced by factors such as the organizational culture, the experience and skill of the individual software developers, the requirements of clients, project constraints, and the extent of training required.*
>
> (B. A. Kitchenham et al., 2004, p. 277)

Whereas Hoare is directly, and quite harshly addressing the individual.

> *The real value of tests is not that they detect bugs in the code, but that they detect inadequacy in the methods, concentration and skills of those who design and produce the code. Programmers who consistently fail to meet their testing schedules are quickly isolated, and assigned to less intellectually demanding tasks.*
>
> (Hoare, 1996, p. 6)

But at the end of the paper he also advises that practitioners should be taught the basics of computer science theory, such as finite state machines and the concepts of types and functional programming. Thus further closing the gap, if not the gap of time, at least the gap of a common terminology and a common conceptual framework.

Copyright notice
Reprinted by permission from Springer Nature Customer Service Centre GmbH: Springer Nature, FME'96: Industrial Benefit and Advances in Formal Methods ("How Did Software Get So Reliable Without Proof?" Hoare), © (1996)

Proceedings. 26th International Conference on Software Engineering, B. A. Kitchenham et al., "Evidence-based Software Engineering," pp. 273–281, © 2004 IEEE

2 Automated Self-Testing Code

Having established the importance of testing as underlined by Hoare and B. A. Kitchenham et al. in chapter 1, this chapter will focus on giving a short introduction about testing software, as done by practitioners. It is not meant to be a complete overview of testing, intentionally leaving out onboarding, maintenance, documentation, specification, communication that are important parts of testing. Instead, the necessary terminology will be given, including a little background, to serve as a guideline for the following chapters.

2.1 Self-Testing

Martin Fowler describes self-testing code as a concept that he first came across during an OOPSLA conference (Fowler, 1999) where it was described as a way to have your code self-test itself much like hardware does a few self-tests during startup, for example the POST (power-on self-test[1]), in personal computers. (Fowler, 2014b) Classes should have a test method that when invoked, tested the functionality of the class. Later Fowler, whilst working with Kent Beck, discovered that Beck and Eric Gamma[2] were working on and later released *JUnit*, a comprehensive tool of the "xUnit"[3] family, providing a better way to have self-testing code without mixing the production and test code.

[1] *Power-on self-test* 2019.

[2] One of the "Gang of Four" who wrote one of the most widely regarded books about design patterns in object-oriented programming and software design. (*Gang Of Four* 2013)

[3] *"xUnit is the collective name for several unit testing frameworks that derive their structure and functionality from Smalltalk's SUnit."* (*xUnit* 2019)

For Fowler, code is only self-testing when it can be tested by invoking a single command to run the whole suite of tests and be confident that, with great certainty, the code does not contain substantial defects that are covered by the tests and regards any code that does not have tests to be broken.

2.2 Automated

> *Continuous Integration is a software development practice where members of a team integrate their work frequently, usually each person integrates at least daily — leading to multiple integrations per day. Each integration is verified by an* ***automated build (including test)*** *to detect integration errors as quickly as possible. Many teams find that this approach leads to significantly reduced integration problems and allows a team to develop cohesive software more rapidly.*
>
> (Fowler, 2006)

Having self-testing code is only one part of a bigger picture. If the tests are not run regularly and often, defects might slip in and are then harder to track down and possibly to remove if they remain undetected for a long time. Even worse if they are then shipped to production, with customers facing the errors. One way to avoid this is having a compact and fast test suite so that testing does not impact the development cycle too much. There are no hard numbers on this, but Kent Beck often mentions that 10 minutes is the maximum time a test suite should run, dedicating a whole chapter in the second edition of his XP book to this “10 minute rule” (Beck and Andres, 2004, chapter 7). Of course many software products exist where the test suites run much longer, even with parallelization, but the point of the “10 minute rule” is that testing should be able to be done often to give continuous feedback, even if it is more than 10 minutes. For example, small parts of the software can usually be tested in under 10 minutes, while the whole test suite might still take longer. Especially nowadays, software is often tested on multiple platforms and platform versions, potentially with multiple versions of the language being used, creating a huge test matrix that is impossible to test within 10 minutes.

> *A build that takes longer than ten minutes will be used much less often, missing the opportunity for feedback.*
>
> (Beck and Andres, 2004)

Another way is automating the tests, for example with "Continuous Integration" (CI) where typically a CI server exists that checks out, for example, either every single commit developers made and pushed to a remote repository, and then continues to build and test the code, or, if the feedback loop would be too slow given a large enough test suite, run the whole test suite during the night, creating "nightly-builds", so that at most there is a window of less than 24 hours (during a workweek) of when defects can slip through undetected and can be fixed soon after the fact.

In the last paragraph there are already a few concepts and prerequisites hidden for employing such a CI strategy, namely having a single source of truth, usually a version control system (VCS) such as Apache Subversion[4], git[5] or Mercurial SCM[6] with git seemingly being the winner[7], with git not only being used by the Linux kernel but also within large companies such as Microsoft, which has one repository (a monorepo) exceeding the size of 300 GB[8]. The benefits are obvious. Even in distributed teams, defects can be captured soon as everyone has to integrate their changes back to the repository, which as a by-product should, in theory, result in the ability to continuously deliver a working product every day.

Another concept is having the possibility to automate the build process which implies that everything needed to build a software product is included in the aforementioned code repository. Within this automation process the execution of the test suite should already be included, resulting in the desired automated self-testing code.

[4] *Apache Subversion* 2019.

[5] *Git* 2019.

[6] *Mercurial SCM* 2019.

[7] Big companies such as Microsoft (recently having bought GitHub, see *Microsoft acquires GitHub* 2019) have switched to using git and other companies such as Atlassian sunsetting Mercurial support in their code management product, Bitbucket (Chan, 2019).

[8] *The largest Git repo on the planet* 2017.

2.3 Types of Tests

There exist different types of categories of tests, each of them serving a different purpose, but all of them complementing each other. In this section a brief description of the different types will be given. However this list does not aim to be exhaustive nor complete, but its aim is to act as a guide for the following chapters, especially chapter 4 in which code coverage will be discussed. A hierarchy, called the "testing pyramid", of these types can be found in Figure 2.1, with a lot of unit tests at the base and end-to-end testing at the very top.[9]

Unit test The definition of what a unit test is, and what it is not, varies depending on the source. However, there are some typical elements that they usually have in common, such as that unit tests test only a very small part of a software, for example a single method or single object[10]. They usually execute very fast and there are typically more of them than other types of tests. The amount can vary depending on what kind of software is being tested, for example a library might have a lot of unit tests for their API, while a modern web application might have more integration or end-to-end tests, extensively testing its UI. (Fowler, 2014c)

Integration test Compared to a unit test which only tests one component, for example one class in isolation, an integration test tests how two or more such components that interact with each other integrate, and if they do so properly. These are valuable because the individual units might work 100% correctly, but when the these components interact with each other they might not behave as expected, as different types of defects can arise than when tested individually. (Fowler, 2018)

System test System testing verifies that a software program is compliant with the requirements. It is a form of black-box testing and can be done automatically as well as manually if some parts are hard or impossible to automate. (Shinde, 2019)

[9] Some similar figures in literature add manual testing above E2E testing.

[10] Testing a whole object is already of a different scope than just testing a single method. Also questions can arise, for example, if only public or also private methods should be tested directly.

End-to-end test Closely related to system testing, end-to-end or E2E tests test the whole software stack, which includes testing certain workflows within the software. One example would be the testing of a multi-step setup process in a web application which includes multiple screens with possibly different branches of workflows in a workflow tree, depending on user input. A system test would test the UI by simulating user input (for example using headless Firefox[11] or Chrome[12], using a webdriver like Selenium[13]), which also exercises the network stack, the middleware, the backend and business logic, persistence layers such as a database and other related services, for example mobile push notifications, email confirmations and other side effects. End-to-end tests can, to a certain extent, replace manual testing.
Drawbacks include that end-to-end tests have the tendency to be much more "flaky" (unreliable) than the other types of tests because of things like animations, asynchronous execution of code etc. making automation of these type of tests harder. (Shinde, 2019; Ham Vocke, 2018)

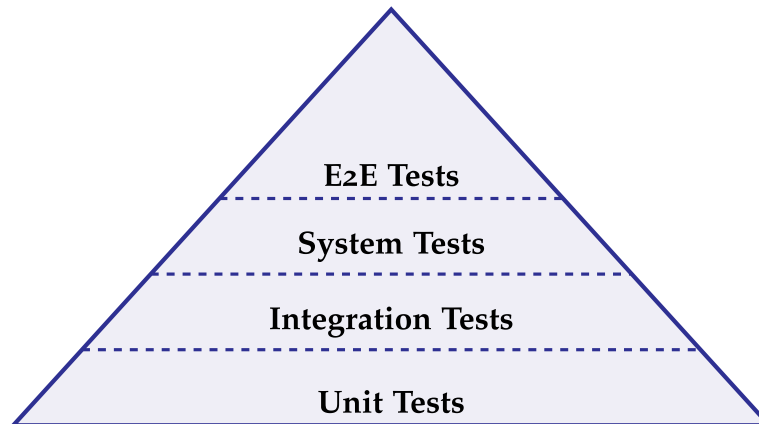

Figure 2.1: Testing pyramid with the different types of tests

[11] *Firefox Headless mode* 2019.
[12] *Getting Started with Headless Chrome — Web* 2019.
[13] *Selenium WebDriver* 2019.

2.4 Conclusion

Much more could be written about automated self-testing code, digging into the details of what a process could look like to introduce CI, "Continuous Delivery" (CD) into a project, which software could be useful to have and why, how to build a company culture around testing, avoiding testing-fatigue, branching strategies for VCS that work for agile development and many things more such as automated mutation testing[14]. However, this thesis will focus on some of the research being done around evidence based software development and how testing relates to it, adding some information as necessary.

[14]Mutation testing is a technique to test existing test suites and identify weak tests and weakly tested parts of the code as well as generating new tests. (*Mutation testing* 2019)

3 Testing as a Suitable Goal for EBSE

Thorough testing is the touchstone of reliability in quality assurance and control of modern production engineering.

(Hoare, 1996, p. 4)

Given the difficulties described in chapter 1 that researchers are facing when trying to provide evidence based suggestions, there is still one topic which is both covered by Hoare and B. A. Kitchenham et al., with the latter saying that while there are many challenges based on the human and more soft factors of the development lifecycle, that testing is a promising candidate for EBSE and should be studied further. This thesis will mostly cover Test-Driven Development and the studies that were made around it over the years, trying to provide a bigger picture of what it is and what it is not, and how a hybrid ("test-first" *and* "test-last") approach can in practice also be of value, namely when exploring ideas. One such approach, 3x by Kent Beck, will be described in subsection 5.5.1.

Microsoft Research's Empirical Software Engineering (ESE) group[1] tries to study different topics of the software development process, from human factors to the influence of organizational structures on software quality and also testing, as will be discussed in sections 5.4.2 and 5.4.3. They are in a unique position because unlike many academic researchers they have access to a plethora of data and software source code from a lot of different industry projects often spanning multiple years, sometimes decades. (Bird et al., 2011) Indeed, Hoare is a principal researcher at Microsoft Research in Cambridge.[2]

---

[1] *Empirical Software Engineering Group (ESE)* 2019.

[2] *And the Winners Are ...* 2006.

However, besides TDD there is one more topic where it is possible to draw numbers from, to further analyze the testing effort and try to derive metrics for code quality or defect rate from, which is the topic of code coverage. This topic would deserve its own thesis, but will be outlined in chapter 4 in context to EBSE.

According to Hoare it is the goal of an initial test suite to be a coverage test, meaning that it should be used to drive the program to "*execute each line of its code at least once*" (Hoare, 1996, p. 7). Much like what will be discussed later on when discussing TDD, he suggest "white box" tests, which are tests that are aware of the underlying implementation and its side effects. For example when a request is made to a server and creates a resource, a white box test might also check for other related resources being created and side effects such as email notifications being sent via an asynchronous background job and also check which priority the job has in a potential queue. They are the opposite to a "black box" test, where the implementation is not know, but known input-output pairs are, usually defined by a specification. Such a specification could just demand that one resource is created and that an email is being sent, but makes no mention of the implementation details (inside the "black box").

Hoare argues that full coverage is necessary for a test suite to be useful because errors are being discovered until even the last line of code is tested (Hoare, 1996, p. 8). And while practices like TDD should automatically lead to 100% code coverage (Beck, 2003, p. 105), some studies suggest that an increase in code coverage does not automatically lead to a proportional decrease in defect rate, as discussed in section 4.2.

3.1 Lack of tests are Technical Debt

> *The concept of technical debt (TD) contextualizes problems faced during software evolution considering the tasks that are not carried out adequately during its development. . . . Nonexecution of tests, pending code refactoring, and outdated documentation are examples of TD.*
>
> (Rios et al., 2018, p. 117)

Technical Debt (TD), as described in the previous quote, is often used knowingly for its short-term benefits such as increased development speed and thus possible shorter time to market. However if this debt is not repaid in a timely manner the amount of work required to pay back this debt increases until development might even come to a halt. Cunningham describes it like this in 1992:

> *Shipping first time code is like going into debt. A little debt speeds development so long as it is paid back promptly with a rewrite. . . The danger occurs when the debt is not repaid. Every minute spent on not-quite-right code counts as interest on that debt.*
>
> (Cunningham, 1992, p. 30)

There are different kinds of TD which Rios et al. tried to define in a taxonomy to help practitioners identify, and thus manage TD. If TD is known, it can be managed and planned for, but if it goes unmanaged for a prolonged period of time it can lead to crisis within a software project. The different kinds of TD can be managed in different ways and at different times within a project lifetime, and to help practitioners with their decision-making process, Rios et al. performed a tertiary study to help identify the current state of the art technical debt research to find out what researchers are currently studying and if any practical recommendations can be derived from it.

> *A tertiary study aims to synthesize data from secondary studies providing a comprehensive view of the state of research in a given knowledge area. It supports the organization of evidence-based body of knowledge that can be used by practitioners and researchers to support their activities.*
>
> (Rios et al., 2018)[3]

By reviewing existing research they were able to identify that cost estimation related reviews had the potential of being valuable for EBSE and also for making this knowledge available to practitioners to employ in their engineering practice (Rios et al., 2018, p. 119). The most recurring types of debt mentioned in primary studies were design (51 primary studies and 4 secondary studies), code (48 primary, 4 secondary) and architecture (46

[3] See also B. Kitchenham and Charters, 2007 and Verner et al., 2012

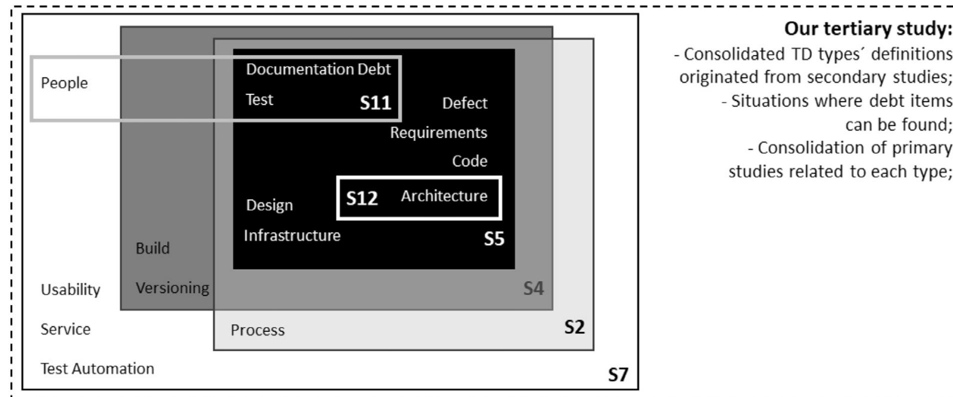

Figure 3.1: Study overlap of Rios et al.'s tertiary study (Rios et al., 2018, p. 134)

primary, 5 secondary) with tests coming in with a bit of a distance as the fourth-most discussed type of TD with 37 primary studies and 5 secondary studies, simply called "test debt". As can be seen in Figure 3.1 there is an overlap over test and documentation debt and "People", giving a slight hint at the human factor, previously discussed in section 1.1. The situations where this kind of test TD could be found were identified as follows (Rios et al., 2018, p. 128):

- Insufficient test coverage;
- Lack of tests (e.g., unit tests, integration tests, and acceptance tests);
- Deferred testing;
- Lack test case planning.

In the following chapters about test coverage (chapter 4) and TDD (chapter 5) certain practices will be discussed that can help reducing test debt and if they are indeed effective.

Copyright notice
Reprinted from Information and Software Technology, Volume 102, Rios et al., "A tertiary study on technical debt," pp. 117–145, Copyright (2018), with permission from Elsevier.

4 Code Coverage

Code coverage is a metric which describes how much of a code base was exercised during the run of a test suite and is usually given in percent. Using code coverage as a metric enables a project to quantify test success and the efficiency of their testing strategy.[1] Of course, a single metric is not an absolute measure and has to be used in conjunction with other metrics, for example defect rate, bugs found during development vs. found in production, bugs fixed within a certain time unit (for example when using agile software development methodology, this would be a "sprint") etc. Additionally it is important to note that there might not always be a strong correlation between an increase in code coverage and a decrease in defect rate as will be discussed later in this chapter.

There are different kinds of code coverage metrics, three of which (statement coverage, decision coverage and function coverage) will be described here as a help to further discuss a case study done at Ericsson and described in a paper released in 2018 by Antinyan et al. which researched the influence of code coverage on defect rate. The definition of each kind of code coverage will be taken directly from the paper for better understanding.

Copyright notice
IEEE Software, Volume 35, Antinyan et al., "Mythical Unit Test Coverage," pp. 73–79, © 2018 IEEE

[1] *Code Coverage Tutorial: Branch, Statement, Decision, FSM* 2019.

4.1 Types of Coverage

Below, three types of coverage will be described and how they are calculated. Later in this chapter they will be mentioned again when discussing the effectiveness of code coverage as a metric for test sufficiency.

Statement coverage *"Statement coverage is the percentage of statements in a file that have been exercised during a test run."* (Antinyan et al., 2018, p. 73). *"A statement is a syntactic unit of an imperative programming language that expresses some action to be carried out."*[2] See Equation 4.1[3] for the formula on how to calculate it.

$$Statement\ Coverage = \frac{Number\ of\ executed\ statements}{Total\ number\ of\ statements} \tag{4.1}$$

Decision coverage *"Decision coverage is the percentage of decision blocks in a file that have been exercised during a test run."* (Antinyan et al., 2018, p. 73). *"In this context the decision is a boolean expression composed of conditions and zero or more boolean operators."*[4] See Equation 4.2[5] for the formula on how to calculate it.

$$Decision\ Coverage = \frac{Number\ of\ Decision\ Outcomes\ exercised}{Total\ number\ of\ Decision\ Outcomes} \tag{4.2}$$

Function coverage *"Function coverage is the percentage of all functions in a file that have been exercised during a test run."* (Antinyan et al., 2018, p. 73). See Equation 4.3 for the formula on how to calculate it. (Antinyan et al., 2018, p. 73)

$$Functional\ Coverage = \frac{Number\ of\ Functions\ exercised}{Total\ number\ of\ Functions} \tag{4.3}$$

[2] *Statement (computer science)* 2019.
[3] *Code Coverage Tutorial: Branch, Statement, Decision, FSM* 2019.
[4] *Code coverage* 2019.
[5] *Code Coverage Tutorial: Branch, Statement, Decision, FSM* 2019.

4.2 "Mythical Unit Test Coverage"

Antinyan et al. conducted a case study at Ericsson to evaluate if the company's unit testing strategy is adequate with regard to their measurement of code coverage. They studied an existing software product with about two million lines of code (LOC) by means of collecting all software defects per file over the course of a year. If a defect occurred in a file this file was considered defective and it was counted how many bugs and bug fixes went into a certain file. For this study, only unit tests were taken into account with integration and system tests not being measured using code coverage, but were used as an indicator. That means that they took a look at code coverage and studied if an increase in unit test code coverage decreases the number of defects found in integration and system tests (which happen at a later stage in the development cycle of the product) and consequently if the opposite, a decrease in unit test code coverage leads to an increase in defect rate in integration and system tests, see also Figure 4.1 for a graphical representation of their study focus.

Furthermore, complexity (for example maximum nesting depth), size of files and the number of changes and their influence was collected to derive measures of adequacy of Ericsson's unit testing strategy. Like many other studies mentioned in this thesis, they intentionally left out other factors which might have an effect on the outcome of the study, such as "*developers' experience in coding and testing, the programming language in which the product was developed, and the integrated development environment that offers testing tools*" (Antinyan et al., 2018, p. 75), but these were assumed to be randomly distributed and thus have only an insignificant effect on the study.

The results found by Antinyan et al. were that they found that all three coverage metrics (statement, decision, function coverage) had a strong correlation between each other, meaning that they are very similar to each other, thus they only took a look at statement coverage.

In their further analysis they found that coverage measures are insufficient as to provide a measure for test sufficiency, however they noted that they did not control for size of files. It was assumed that a large file with, for example, 100 LOC and 50% coverage had half of the code untested had to have the same defect rate as a file with 1000 LOC and the same coverage

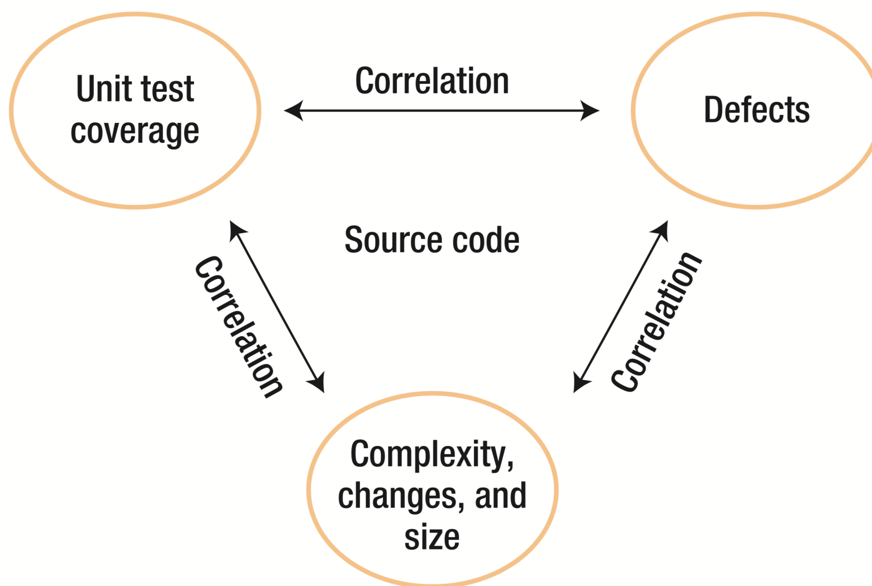

Figure 4.1: Focus of the "Mythical Unit Test Coverage" study (Antinyan et al., 2018, p. 75), © 2018 IEEE

rate of 50%, which results in 500 lines of untested code. However, quite intuitively, they found that large files with more LOC showed a higher defect rate than those with fewer lines of code. The same result was observed when looking at the number of changes a file has undergone, namely a file with more changes was more prone to more defects than a file with fewer changes.

Measure	Correlation with defects	Correlation with stmt. cov.
Statement coverage	−0.19/−0.13	1
Decision coverage	0.19/−0.13	0.91/0.87
Function coverage	−0.18/−0.14	0.87/0.86
LOC	0.67/0.53	−0.18/−0.12
Maximum block depth	0.42/0.42	−0.40/−0.33
Stmt. cov./LOC	−0.06/−0.25	—
Stmt. cov./no. of versions	−0.18/−0.30	—

Table 4.1: List of measures with strong correlations from Antinyan et al., 2018, p. 76, © 2018 IEEE

To account for this, Antinyan et al. introduced two more metrics, average coverage per LOC (Equation 4.4) and average coverage per version (Equation 4.5). Even after this the Pearson and Spearman correlation coefficients[6] showed only a weak correlation between an increase in unit test code coverage and a decrease in defects. Some files did show a downwards trend, but most of the files analyzed had no such association. Some selected measures and their correlation with defects and statement coverage can be found in Table 4.1.

$$\textit{Average Coverage per LOC} = \frac{\textit{Statement Coverage}}{\textit{LOC}} \tag{4.4}$$

[6]Spearman's rank correlation coefficient is a measure of statistical dependence between the rankings of two variables. When the coefficient is negative, it means that variable Y tends to decrease when X increases. It is positive if X increases and Y tends to increase as well. Compared to the Pearson correlation coefficient (r) which describes *linear* relationships, Spearman's correlation coefficient (ρ) describes *monotonic* relationships. It can be useful to calculate both to see if the two produce a similar or different result to see if one is handling a linear or monotonic relationship between variables. (*Spearman's rank correlation coefficient* 2019; *Pearson correlation coefficient* 2019; Minitab, LLC, 2019)

$$Average\ Coverage\ per\ Version = \frac{Statement\ Coverage}{Number\ of\ Versions} \tag{4.5}$$

4.2.1 The Influence of Code Complexity on Defect Rate and Code Coverage

One result the research yielded was that while some complexity measurements such as cyclomatic complexity, parameter count, percentage of comments had no tangible effect on coverage, maximum block depth had a strong negative correlation with code coverage, meaning that if a file contained code with very deep nesting this actively hindered the increase of code coverage, since nesting — depending on the language — often results from conditional statements or control structures, creating a lot of branches to cover, making it harder to test it. The same files often exhibited a higher defect rate, meaning that deeply nested code was harder to test and had a correlation to higher defect rate. See Figures 4.2 and 4.3 for contour plots showing the relation between maximum block depth, respectively the number of version of a file, and statement coverage.

The conclusion of the paper was that, at least at Ericsson, unit test coverage alone did not prove to be an adequate metric for the sufficiency of testing. This is in line with hat Juristo et al. found out in 2004 in their "Reviewing 25 Years of Testing Technique Experiments" paper. They also concluded that complexity is usually easier to manage than lines of code or number of versions made to a file, since active development often naturally results in both, more lines of code and changes being made and suggest that practitioners should use complexity as an additional metric for reducing the defect rate. Lastly, they recommend that testing standards (such as DO-178B, which is briefly discussed in section 4.4) adopt their suggestions with regards to test sufficiency using code coverage as a metric.

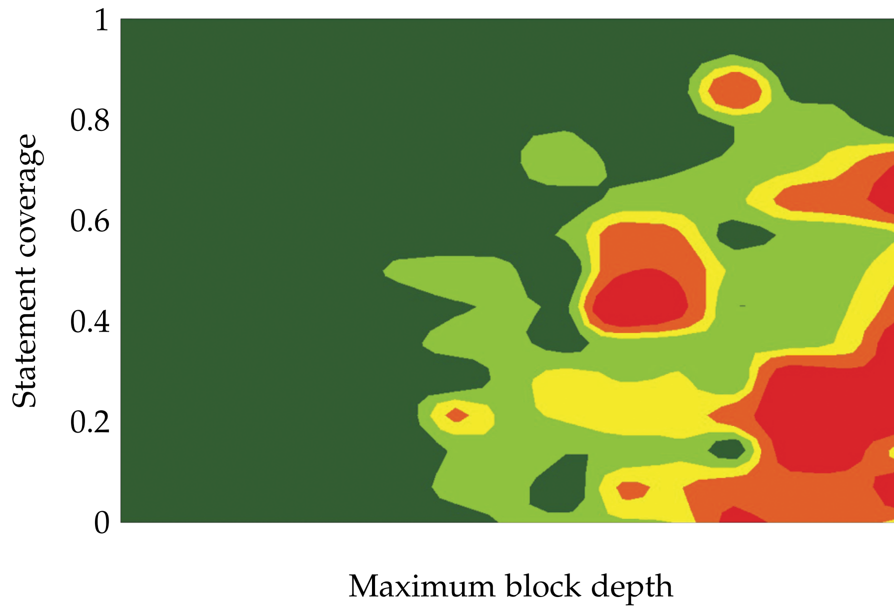

Figure 4.2: Contour plot showing the relation between defects related to maximum block depth and statement coverage (Antinyan et al., 2018, p. 78), © 2018 IEEE

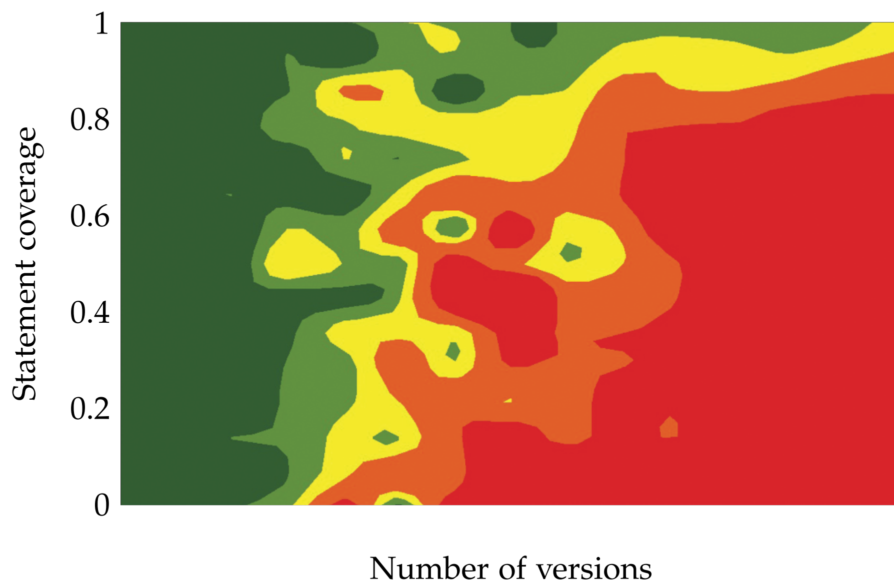

Figure 4.3: Contour plot showing the relation between defects related to the number of versions and statement coverage (Antinyan et al., 2018, p. 78), © 2018 IEEE

4.3 What is Code Coverage measuring, if not Test Sufficiency?

Martin Fowler noted in 2012 what Antinyan et al. later empirically found out at Ericsson — that code coverage is not a good metric to measure if the tests being written are good, but serves as a tool to find which parts of the code are not being tested. Setting a certain goal, for example a minimum of 90% code coverage will, according to Fowler, lead to developers trying to optimize their tests for that metric instead of writing tests that make sense and are of high quality. This can be seen as the analog of Goodhart's law (*Goodhart's law* 2019), named after British economist Charles Goodhart, which states:

> *Any observed statistical regularity will tend to collapse once pressure is placed upon it for control purposes.*

Or as Marilyn Strathern later summarized:

> *When a measure becomes a target, it ceases to be a good measure.*

Fowler also makes a reference to Test-Driven Development, which will be discussed at length in the next chapter, calling it a "*useful, but certainly not sufficient, tool*" (Fowler, 2012) to write valuable test, stating that when executed properly, he would expect a code base developed in such manner to have code coverage around 80% or 90% but would be very skeptical of a code base that has 100% code coverage (although attainable in very small programs and even huge code bases, such as SQLite, see section 4.4).
However, he does state that low numbers such as 50% and below are a sign that the risk of undiscovered bugs and that something could be missed during refactoring could be higher and thus a sign for trouble.

Instead of using code coverage as a measure for test sufficiency, he gives two statements that should hold true for the development process:

- You rarely get bugs that escape into production, and
- You are rarely hesitant to change some code for fear it will cause production bugs.

(Fowler, 2012)

4.4 SQLite

SQLite deserves its own section since it is a special piece of software. It is estimated to be used on over 14 billion devices[7]. It has 100% branch coverage (see section 4.1) as well as 100% MC/DC coverage.[8] MC/DC states that the following must be true[9]:

1. Each entry and exit point is invoked
2. Each decision takes every possible outcome
3. Each condition in a decision takes every possible outcome
4. Each condition in a decision is shown to independently affect the outcome of the decision.

It is used, among other things, in the (de-facto) technical standard for developing avionics software to ensure adequate testing of critical parts of the software, DO-178B.[10]

Additionally, SQLite contains 711 times more test code than production code, which possibly makes it the most well-tested open source software. However, while SQLite is open source, it is not open-contribution, meaning that no one can directly contribute to SQLite, rather than suggest new features. This is to ensure that SQLite can remain in the open domain. Another fact that makes SQLite special is the fact that while SQLite is fully open source, most of its tests are proprietary and can only be acquired by buying a license. This underlines the importance of tests, since in the case of SQLite, tests are what bring value to the product and not having the whole test suite makes it much harder to copy it.[11]

---

[7] Every Android device (around 2.5 bn), every Mac and iOS Device (around 1.5 bn), every Windows 10 machine (around 2 bn), every Chrome and Firefox browser (around 5 bn), Skype, iTunes, WhatsApp (around 3 bn). (Richard Hipp, 2019)

[8] *Richard Hipp about the history of testing SQLite — Hacker News* 2019; *How SQLite Is Tested* 2019.

[9] *Modified condition/decision coverage* 2019.

[10] *DO-178B* 2019.

[11] *How SQLite Is Tested* 2019; *SQLite Copyright* 2019; *SQLite TH3 (Test Harness 3)* 2019.

5 Test-Driven Development

Test-Driven Development (TDD) is mentioned indirectly[1] in the book *Extreme Programming Explained* from 1999, but has since then sparked interest of its own, in a less "extreme" way[2]. Kent Beck is often cited as the inventor of TDD, however he claims he did not invent, but *rediscovere* it (Beck, 2012). Beck wrote a book about TDD in 2003 called "Test-Driven Development: By Example" in which he explains the rationale behind TDD and what its intention is and how certain goals can be achieved using this development methodology.

TDD is a practice which, much like the name suggests, focuses on driving development and software design by splitting tasks into testable units (thus, using unit tests) which are written before the actual code is written, making sure they fail at least once. By doing so it tries to mandate thinking about what kind of minimal implementation is needed to make a test succeed, and, in the case of Object Oriented Programming (OOP) which objects would be useful to have and to which messages they should respond. Beck calls this "*writing a story*" (Beck, 2003, p. 30), in the sense that it should be thought about what is necessary to bring this story to its expected conclusion and argues that using a test-first strategy, code becomes less coupled and more cohesive than without employing this strategy (Beck, 2003, p. 142).

> *Even if I don't know how to implement something I can almost always figure out how to write a test for it, and if I can't figure out how to write a test for it, I have no business programming it in the first place.*
>
> Kent Beck in *Is TDD dead?* 2014, 07:40

[1]The practice of writing tests first and that work is not done until all tests pass is one of the 12 key XP practices, although not called "Test-Driven Development" yet.

[2]The practice of testing in XP and TDD are the same, however TDD was "extracted" from XP, with "less extreme" meaning that the other practices of XP are being left out (while not forbidden), such as pair programming.

Using TDD, defects are discovered early and quickly during development and, as a consequence, making it easier to determine the source of the defect. This is the result of constantly running the tests and very often the whole test suite. This benefit should compensate for the additional time spent on writing and executing the tests. (Nagappan et al., 2008, p. 292)

5.1 Goals of TDD

The bigger picture goals of TDD are not unique to TDD but to software development in general. It aims to provide a better way to plan and estimate software development, thus also helping project managers, avoiding nasty surprises by reducing the number of defects discovered only after a product has been shipped to costumers.

> *TDD is an awareness of the gap between decision and feedback during programming, and techniques to control that gap.*
>
> (Beck, 2003, p. 10)

Additionally, if TDD can succeed in these goals set by itself, Beck claims that the result would be software that has much fewer defects than it would without and could theoretically be shipped every day, because of an extensive test suite that would at the very least ensure that the defects that are being tested for do not exist.

> *Testing shows the presence, not the absence of bugs.*
>
> (Buxton and Randell, 1970, p. 16, citing Edsger W. Dijkstra)

Another goal stated by Beck is a bit more informal, which is to reduce programming stress. He claims that if TDD is followed, a programmer can rest easy at home after work, knowing that his code did not break anything else and there should be no unpleasant surprises when returning to work the next day. (*Is TDD dead?* 2014, 09:39)

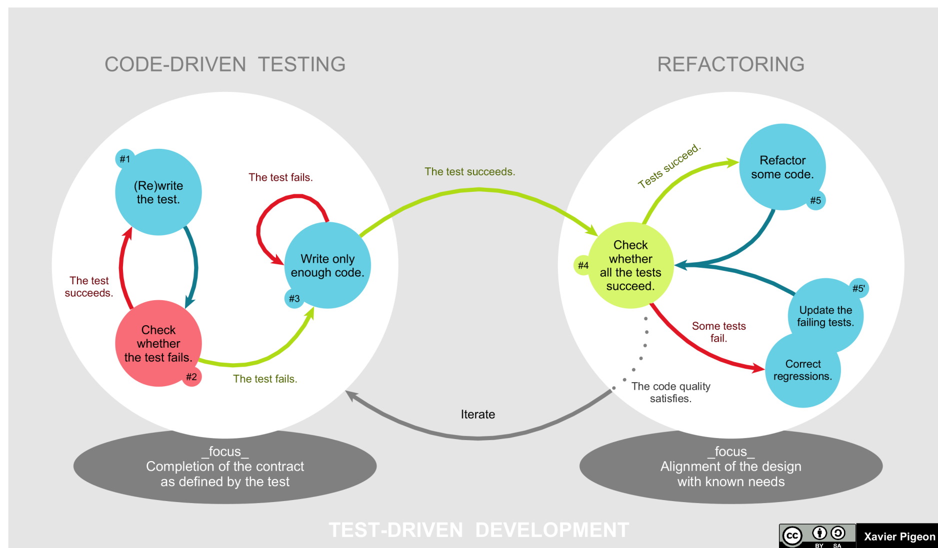

Figure 5.1: The TDD lifecycle as described by Beck (Xarawn, 2015)

5.2 The TDD Development Cycle

The TDD development cycle is often found in its short version "Red, Green, Refactor", but in the book written by Beck it is written as follows (Beck, 2003, p. 5), see also Figure 5.1[3].

1. Quickly add a test
2. Run all tests and see the new one fail
3. Make a little change
4. Run all tests and see them all succeed
5. Refactor to remove duplication

When starting off with TDD, Beck suggest going in the tiniest steps possible to get a feeling for what the benefits of TDD can be, for example changing just the name of a variable and seeing tests fail because of it, then renaming it everywhere else in a next step and only then using it in a different context or adding new functionality. Given enough experience these tiny steps can be squashed into bigger leaps but learning to go in these tiny steps enables

[3]Xarawn [CC BY-SA 4.0 (https://creativecommons.org/licenses/by-sa/4.0)]

a developer to go in tiny steps when needed, for example when debugging or refactoring a code base to get a better sense of action and consequence. (Beck, 2003, p. 112)

If these steps are followed religiously, code coverage, more precisely statement coverage, should theoretically be 100%, but as discussed in chapter 4, code coverage is not a sufficient measure of test quality, as Beck also states (only providing anecdotal evidence), but is seen as a good starting point. (Beck, 2003, p. 105) Using these simple techniques should also result in a simpler design that is easier to test, resulting in a suite of tests that should inspire confidence[4] in the programmer. (Beck, 2003, p. 18)

> *I conclude that there are two ways of constructing a software design: One way is to make it so simple that there are obviously no deficiencies and the other way is to make it so complicated that there are no obvious deficiencies.*
>
> (Hoare, 1981, p. 81)

5.2.1 Regression Tests

The same process can be followed when new defects are being reported. First, one writes a minimal failing test that reproduces the reported defect, then writes the minimal changes needed to fix it and refactors if necessary. Such tests are called regression tests and are a very useful way for users to explicitly state the behavior they see in an application and which one they expected, which makes these kinds of tests — however unfortunate as they were written because a defect occurred — very valuable because they act as living and maintained documentation of the software, a concept also found in XP[5].

Nagappan et al. describe the tests written using TDD as *"a high-granularity, low-level, regression test"* (Nagappan et al., 2008, p. 292) which help defect identification early on and avoiding late discovery.

[4] Indeed, variations of the word "confidence" can be found over 30 times in Beck's book.

[5] However, XP does not forbid writing documentation completely, but Beck argues that there should be less of it and it should be considered who it is for. (Beck and Andres, 2004, p. 26; Josuttis, 2000)

5.2.2 Refactoring

In this chapter a brief explanation of what refactoring is will be given as well as a small section about how tests can help while refactoring, using Martin Fowler's *Refactoring* book from 1999[6] as a basis, since it was written around the time XP and later TDD became more widely known.

Definition

> *Refactoring is a disciplined technique for restructuring an existing body of code, altering its internal structure without changing its external behavior.*
>
> (Fowler, 2019)

Fowler describes "refactoring" — used both as a noun and a verb — as a small *behavior preserving* transformation of a piece of code (Fowler, 2004a). Small, because while sometimes phrases like "refactoring the whole code base" are thrown around, what is actually meant is a sequence of these small "refactorings", making the code easier to understand, extend and maintain. Refactoring is also behavior preserving because *what* the software does should not change, only the way it does it, which should be an implementation detail. One simple example is exchanging one sorting algorithm with another that has better performance characteristics or properties. This way a certain input will still yield the same output, while overall improving the software. Another example would be to make certain parts of the code cheaper to modify or enhance in the future. In this sense, refactoring is different from rewriting/restructuring or changing the code where the public interfaces might change and callers of code have to be adapted during which time a test suite might fail (see also "Refactoring Malapropism" by Fowler, 2004b). However these two concepts are often conflated, but it is an important distinction to make, especially with regards to using testing to aid refactoring which will be discussed in in the next subsubsection.

In many software development methodologies, such as XP and TDD, refactoring is not a separate task that will be found on project plans, but should

[6] A 2nd edition was released in 2018.

be a constant, day-to-day task. If there is a chance to enhance the code base it should be refactored immediately. (Fowler, 2019)

Tests as a Way to aid Refactoring

In his book, Fowler describes the importance of tests during refactoring a code base. According to him, they are not just important, but an essential precondition before beginning to even attempt to refactor a code base. In his experience, having tests increases his speed rather than slowing him down, although this is a counterintuitive thought for many programmers. Fowler insists that if you want to refactor, you have to have tests, and if you do not have any, tests have to be written before going on to refactor. Having these tests in place, even a legacy code base can be transformed to be more maintainable, have better extensibility or be adapted to more modern technologies without the fear that every single step might lose the functionality and contained knowledge that went into writing the code and coverage of edge cases that were initially written.

Since this book was released in the same year as *Extreme Programming Explained*, but four years before *Test-Driven Development* by Beck was released, the phrase "TDD" is not explicitly mentioned, while testing as an important concept in XP is[7]. However some hints towards TDD exist, such as the mention that the best time to write a test is before writing the code. Additionally it is mentioned that one should have fast tests, a concept also found in TDD, with the note to run the complete test suite often enough so no unpleasant surprises will arise.

One difference to *some* modern software development practices can be found, which is the mention that function tests that test a software as a whole should not be written by the developers themselves but by a different team, putting a strong focus on unit tests. Nowadays, while not universally true, but also due to the advent of better and easier-to-use testing frameworks, developers can in fact write functional and system tests with relative ease.

---

[7]The quote "*Development is driven by tests*" can be found in the first edition of *Extreme Programming Explained* in Beck, 1999, Chapter 2 with TDD being mentioned explicitly beginning with the second edition of the book.

Fowler advocates to write tests in a risk-driven fashion. Which tests would one like to have to know that they will fail in case a new defect is introduced or a regression happens. For example, this means that simple getters and setters do not have to be explicitly tested since they are not expected to fail. (Fowler, 1999, Chapter 4)

5.3 Criticism

George and Williams who conducted a set of experiments in 2004 with 24 professional pair programmers concluded that TDD does yield better code in terms of code coverage (compared to "test-last" approaches) and that simple design is indeed achieved by following TDD practices (the study will be discussed in detail in subsection 5.4.1). They however also list several shortcomings to the TDD approach, which are mostly equivalent with sources such as Deursen's paper "Program Comprehension Risks and Opportunities in Extreme Programming" from 2001 in which he analyzes five XP key practices (pair programming, unit testing, refactoring, simple design, and planning as a team activity (Deursen, 2001, p. 4). See Table 5.1 for an overview of the twelve key XP practices) and shows benefits as well as risks associated with them in relation to program comprehension.

In the conclusion of his paper Deursen, like many others before and after him, ask for empirical support of these practices, some of which have already been discussed in this thesis and some that will be discussed later.

Copyright notice
Proceedings Eighth Working Conference on Reverse Engineering, Deursen, "Program Comprehension Risks and Opportunities in Extreme Programming," pp. 176–185, © 2001 IEEE

5.3.1 Criticism on Unit Testing

Leaving out the practices which are not relevant to testing, Deursen postulates that the practice of having unit tests as living documentation leads to a paradox situation where a practitioner, in order to comprehend the code they are reading, must read another piece of code. Code that must be maintained and is not necessarily correct, as there is no guarantee that test code is always correct. Furthermore some projects contain not less, but much more test code than actual production code, so one has to read even more code than there is production code, with SQLite being a notable open source software extreme with 711 times more test code than program code (see section 4.4).

Another concern was that much of the decision making (in the case of XP often done during pair programming) is made through oral communication and thus can be lost because of the lack of written documentation in prose form if the tests are lacking in expressiveness. Of course writing documentation is not forbidden by XP, although mostly reasons for *not* writing it are given, thus the inclination to write documentation is low and thus the likeliness that documentation gets written at all is low as well.

Deursen's last concern with regard to testing is that parts of the program that are particularly hard to test such as user interfaces, asynchronous code etc. are either left untested or "*require skill, experience, and determination*" (Deursen, 2001, p. 7) which are not always available within a project's life.

5.3.2 Criticism on Refactoring

One other key practice of both TDD and XP is refactoring, of which Deursen said that while it can have a positive influence on a code base, the risk is that with the constantly changing shape of the code base, practitioners need to re-learn parts of code they already understood previously.

Furthermore it was mentioned that the lack of comments within a codebase (Martin Fowler strongly discourages leaving comments and rather recommends refactoring the code so that it does not need comments anymore) might lead to no comments at all and leave practitioners at a loss as to what

Pair programming	Production code is developed by pairs of programmers
Testing	Unit tests and acceptance tests are run continuously
Refactoring	Continuously improve the design without changing the functionality
Simple Design	The guiding design principle is to do the simplest thing that could possibly work
Planning Game	Development estimates user stories, and business prioritizes them
Collective ownership	Developers can modify any piece of code
Continuous integration	Integrate changes immediately instead of developing them in separate branches
40-hour week	Programmers work 40 hours max., to keep them fresh and creative
On site customer	A customer is on th team to discuss feature requests and domain concepts
Frequent releases	Release code into production as often as possible
System metaphor	Simple shared story of how the system works
Coding standards	Ensure agreement on simple coding conventions

Table 5.1: 12 key XP practices, summarized by Deursen, 2001, p. 5, © 2001 IEEE

and why code has certain properties and does certain things (which — in XP and TDD — should then be explained by a corresponding test, see criticism in subsection 5.3.1).

5.3.3 Criticism on Simple Design and Lack of Planning

TDD, as described by Beck, should lead to a simple design with the fewest necessary classes, the fewest necessary methods and the least amount of code needed to fulfill the requirements (or "making the tests green"). Deursen finds that this practice might lead to a lack of design, because it only focuses on small portions of the code with the possibility to miss the "big picture" or macro view on the program overall. If not done "right", having many small different design decisions for different parts of the code might lead to inconsistent code with a mixture of different styles.

He concludes that adhering to these XP practices (the subset interesting to this thesis, unit testing, refactoring) play a game with high risks and possible high returns. If it works, it might work incredibly well, resulting in programs with superior code quality and better code coverage than programs developed with an alternative development style, but leave practitioners at risk that if XP fails them, there is little to no fallback, since no explicit design and technical documentation were produced.

5.4 Efficacy of Test-Driven Development

After the introduction of the term Test-Driven Development and the claims made by Beck in his book about the same, researchers and companies alike studied the practice in detail, trying to find a reproducible environment in which case-studies could be compared to each other, as well as separate studies performed with very small programs and multiple participants. The goal of the research was to find out if what is suggested in the TDD book (fewer defects, a testable and maintainable code base, simple design etc., see section 5.1 about TDD) and what practitioners reported (both positive

and negative, negatives for example such as additional time overhead and bending tests to make it testable) was true, and if so, to what extent.

> *Case studies can be viewed as "research in the typical",* (Barbara Kitchenham et al., 1995; Fenton and Pfleeger, 1998) *increasing the degree to which the results can be compared with those of other industrial teams. However, case studies cannot be performed with the rigor of experiments because they involve real people, real projects, and real customers over a relatively long period of time. In empirical research, replication of studies is a crucial factor to building an empirical body of knowledge about a practice or a process.*
>
> (Nagappan et al., 2008, p. 290)

Copyright notice
Reprinted by permission from Springer Nature Customer Service Centre GmbH: Springer Nature, Empirical Software Engineering, Volume 13, ("Realizing quality improvement through test driven development," Nagappan et al.), © (2008) Proceedings of the 2006 ACM/IEEE International Symposium on Empirical Software Engineering, ("Evaluating the Efficacy of Test-driven Development," Bhat and Nagappan, pp. 356–363), Copyright 2006 ACM

5.4.1 George and Williams, 2004

In 2004, George and Williams conducted a set of structured experiments to examine two hypotheses, one being that the TDD practice will result in code with a higher quality than if it were developed using the more traditional waterfall practice, where each phase of development is executed after another (requirements, design, implementation, verification, maintenance, see also Figure 5.2[8])[9]. This was verified by measuring the code quality, based on the number of test cases passed. The second hypothesis was that employing a TDD style development practice will result in faster code development.

[8] Kemp and Smith [CC BY 3.0 (https://creativecommons.org/licenses/by/3.0)]
[9] *Waterfall model* 2019.

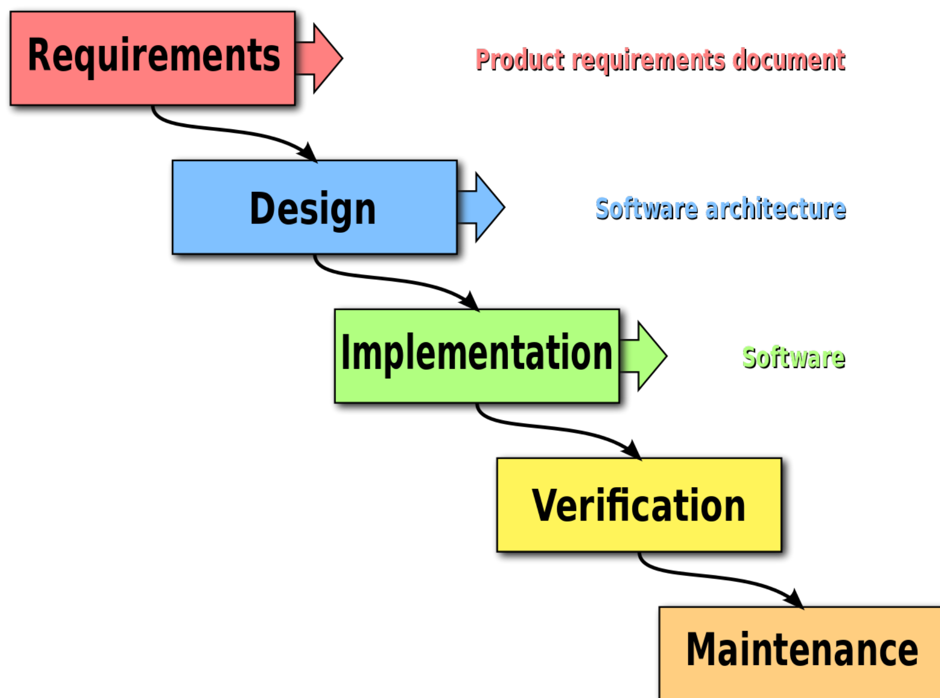

Figure 5.2: The (unmodified) waterfall model (Kemp and Smith, 2010). Compare with Figure 5.1.

The experiment was carried out with 24 professional pair programmers, developing a small Java program. George and Williams found that the TDD approach does yield higher quality code, when measured by the number of tests passed, but their second hypothesis was wrong as it was shown that the TDD group took on average 16% more time than the waterfall group.

Two other conclusions were drawn, first, 80% of the developers answered in a survey that TDD was an effective approach and that it improved their productivity. Second, it could also be observed that the group using the TDD approach did in fact come up with a simpler design and that the missing, separate design phase at the beginning was not an impediment.

Concerns regarding the validity of the research are of course that this experiment was a controlled study with the practitioners knowing that they were being observed and their code would be analyzed. George and Williams suggested that further studies in industrial environments should take place. Such a study can be found in sections 5.4.2 and 5.4.3.

5.4.2 Bhat and Nagappan, 2006

In 2006, Bhat and Nagappan from Microsoft discussed the TDD methodology in two different environments at Microsoft. The environments were the Windows Operating System and MSN division to study the methodology in two different contexts to increase the chance that if TDD works in both contexts, that the results would be more easily validated or at least plausible, than looking at just one type of software product. They found that the code produced in projects employing TDD had an increase in quality of up to more than two times compared to projects that did not use the TDD methodology. But they also found that the projects took about 15% longer when using TDD[10], but had the additional benefit of having documentation for the code in the form of unit tests in the case of libraries and public APIs (Application Programming Interface). Furthermore in a follow-up paper from 2008 Nagappan et al. imply that by using TDD and making small changes in code, design and adding new functionality in tiny steps

[10]This is in line with the 16% George and Williams observed in their set of structured experiments

a developer can retain "*intellectual control*" over what they are doing at a certain point in time at a relatively consistent rate. (Nagappan et al., 2008, p. 291)

They enhanced the explanation of the TDD process of Beck as follows (Bhat and Nagappan, 2006, p. 356):

1. Writing a (very) small number of automated unit test case(s);
2. Running the new unit test case(s) to ensure they fail (since there is no code to run yet);
3. Implementing code which should allow the new unit test cases to pass;
4. Re-running the new unit test cases to ensure they now pass with the new code;
5. Refactoring the implementation or test code, as necessary, and
6. Periodically (preferably once a day or more) re-running all the test cases in the code base to ensure the new code does not break any previously-running test cases.

As discussed in the introduction of this chapter, their goal was to set a kind of framework to scientifically research the TDD methodology with their contributions to it listed below (Bhat and Nagappan, 2006, p. 358):

- TDD evaluation using commercial software in two different divisions at Microsoft
- Accurate quality measurements with comparable projects in terms of defect density to measure quality
- Quantification of increase in development time due to adoption of TDD
- Contribute to strengthening the existing empirical body of knowledge regarding investigations of TDD

To measure quality they used the measure of defect density, which was introduced in this thesis as "defect rate" in chapter 4, meaning they measured the number of defects and normalized it per thousand lines of code (KLOC). The amount of defects was extracted from Microsoft's bug tracking system.

For the first project which was a networking library written in C++ they found that while block coverage was 79% (compare with the 100% in an ideal

scenario mentioned by Beck and keeping in mind the lack of test sufficiency, even given an increase in code coverage mentioned in the paper by Antinyan et al., see section 4.2), with test LOC being 66% of the actual program source LOC, the defects/KLOC were 2.6 times less than in a comparable project that did not use the TDD methodology with an estimated (by management) increase in development time of 25–35%.

In the other project in the MSN division that develops web services applications a similar effect was discovered by Bhat and Nagappan. This project had more than four times the source LOC than the first project in the Windows division and also had a higher ratio of test LOC with 89% test LOC and 88% test coverage. It was observed that in a similar project not using TDD the defect density was 4.2 times as high as in the project using TDD with an estimated increase of development time of 15% in the TDD team.

In summary, while there was additional time needed to write tests (estimated between 15% and up to 35%), code quality measured in defect density was much lower in the projects using a TDD approach. However, drawing general conclusions from an individual, empirical study is always prone to variation caused by effects not assessed, such as skill factor, human factors and the simple fact that while projects might be similar in context and scope, no two projects are directly comparable to each other.

Bhat and Nagappan suggest in 2006, that more studies should be done in a similar fashion as to build up an empirical body of knowledge on the efficacy of TDD and looking into possible cost-benefit tradeoff analyses to see if the return on investment (ROI) is worth taking the TDD approach, with the investment being the additional development time needed and the return being higher software quality.

5.4.3 Nagappan et al., 2008

In 2008, Nagappan et al. published another paper in which they reiterated most of the paper by Bhat and Nagappan from 2006, adding some more explanations and giving more detail to the goals stated in the 2006 paper. One big addition in the 2008 paper was the inclusion of a case-study conducted at IBM. The IBM case study compared a legacy product from 1998, a device

driver with seven releases and a newly developed product that targeted the same devices, but with a new architecture from 2002.

The new product that was developed using the TDD methodology had a ratio of 70% test LOC compared to source LOC and a block coverage from unit tests of 95%. Complete unit testing was enforced, meaning that all public classes and public methods had to be tested. IBM set a target goal of at least 80% coverage. The legacy project showed an over 60% higher defect density when compared to the newly developed product which had a 15–20% time overhead caused by writing the tests.

These results are in line with what has been observed at Microsoft and further strengthen the argument that using TDD does reduce the maintenance cost and increase software quality (when measured as defect density). However it was mentioned again that one threat to validity is that empirical studies cannot be generalized.

> *...a family of case studies is likely not to yield statistically significant results, though Harrison (2004) observes that statistical significance has not been shown to impact industrial practice.*
>
> (Nagappan et al., 2008, p. 298)

Another strong hint that TDD does decrease defect rate was that the IBM team reported that because some team members took shortcuts and did not run the tests they did temporarily increase defect rate compared to the time before shortcuts were made. (Nagappan et al., 2008, p. 299)

Nagappan et al. end their research paper with a few recommendations, listed below, as they extend on what Beck wrote in his book, derived from case studies of various software products at IBM and Microsoft. (Nagappan et al., 2008, pp. 299–300)

- *Start* TDD *from the beginning of projects.* Do not stop in the middle and claim it doesn't work. Do not start TDD late in the project cycle when the design has already been decided and majority of the code has been written. TDD is best done incrementally and continuously.
- *For a team new to* TDD*, introduce automated build test integration towards the second third of the development phase — not too early but not too late.* If this is a "Greenfield" project, adding the automated build test towards

the second third of the development schedule allows the team to adjust to and become familiar with TDD. Prior to the automated build test integration, each developer should run all the test cases on their own machine.

- *Convince the development team to add new tests every time a problem is found*, no matter when the problem is found. By doing so, the unit test suites improve during the development and test phases.
- *Get the test team involved and knowledgeable about the TDD approach.* The test team should not accept new development release if the unit tests are failing.
- *Hold a thorough review of an initial unit test plan*, setting an ambitious goal of having the highest possible (agreed upon) code coverage targets.
- *Constantly running the unit tests cases in a daily automatic build (or continuous integration)*; tests run should become the heartbeat of the system as well as a means to track progress of the development. This also gives a level of confidence to the team when new features are added.
- *Encourage fast unit test execution and efficient unit test design.* Test execution speed is very important since when all the tests are integrated, the complete execution can become quite long for a reasonably-sized project and when using constant test executions. Tests results are important early and often; they provide feedback on the current state of the system. Further, the faster the execution of the tests the more likely developers themselves will run the tests without waiting for the automated build tests results. Such constant execution of tests by developers may also result in faster unit tests additions and fixes.
- *Share unit tests.* Developers' sharing their unit tests, as an essential practice of TDD, helps identify integration issues early on.
- *Track the project using measurements.* Count the number of test cases, code coverage, bugs found and fixed, source code count, test code count, and trend across time, to identify problems and to determine if TDD is working for you.
- *Check morale of the team at the beginning and end of the project.* Conduct periodical and informal surveys to gauge developers' opinions on the TDD process and on their willingness to apply it in the future.

5.5 Is TDD dead?

> *Any headline that ends in a question mark can be answered by the word* ***no****.*
>
> Betteridge's law of headlines[11]

In 2014 David Heinemeier Hansson (often referred to simply as DHH) wrote a blog post[12] in which he described his discontent with the TDD methodology, saying that people took it to an extreme, adding abstraction layers over abstraction layers rather than writing clear, direct and concise code, in the name of testability and decoupling. It was meant as a general note, but examples were given in the context of Ruby on Rails, a MVC (Model View Controller) web-application framework written in Ruby. People argued that testing Rails — at that point in time — was not decoupled enough, making unit tests hard, because different components of the framework could not be tested in isolation or have dependencies, such as a database. Heinemeier Hansson argued that some coupling makes sense and that people's test-first approach was leading to a "test-induced design damage", which is also the name of the article he wrote.

This article sparked some controversy on the internet, with some people arguing that TDD is dead (among other sources, also an article by Heinemeier Hansson (Heinemeier Hansson, 2014a)) and led to a five-part video chat between David Heinemeier Hansson, Kent Beck and Martin Fowler, where points were further clarified and discussed (Fowler, 2014a).

In the video chat Heinemeier Hansson began with the statement that there is no overall consent over the definition of "unit test", whereas some people argue only if there are no dependencies and no coupling and if code can be tested in absolute isolation it is a unit test, others have broader and looser definitions over what a "unit" is. Further he argues that heavy mocking and intense use of abstraction leads to above mentioned "test-induced design damage", with TDD being used as an argument to justify such an architecture. His second argument is that the red-green-refactor loop is not comfortable for him in a lot of situations where he is exploring the problem, which for

[11] *Betteridge's law of headlines* 2019.
[12] Heinemeier Hansson, 2014b.

him goes against what the Ruby and Ruby on Rails (Heinemeier Hansson, 2016) community is about and what Ruby's creator, Yukihiro Matsumoto calls "developer happiness" (Venners, 2003). However he also states that he agrees that a feature is not done until automated tests are written for it. He differentiates between "test-first" and "test-last" as "going through the test" and "going to the test".

Kent Beck answered by saying that TDD is mostly about what has already been described at the beginning of this chapter, summarized as the question if a developer can sleep at night, knowing their code still works, giving them confidence going further. TDD is especially useful to produce a "*nice sequence of tests*" as he calls it, for data structures or any sort of code where given inputs and their corresponding outputs are known.

Martin Fowler's response is best described by quoting him directly:

> [It is] *really important to have self-testing code, the ability to be able to run a single command, have your whole system self-test itself in an acceptable amount of time. That is really powerful, because then if you can do that, you can refactor with confidence, you've got a good chance of keeping your code base healthy and that means you can be fast, deliver new features etc. etc. ...* TDD *is one way to approach that.* TDD *is a very particular technique, and if done well, one of the benefits is that it gives you self-testing code ... which for me is the primary benefit.*
>
> Martin Fowler in *Is TDD dead?* 2014, 24:10

It was argued by Heinemeier Hansson that additional code and levels of indirection result in more code, and in a gross oversimplification that more code means it is harder to comprehend, giving an example from Jim Weirich's talk "Decoupling from Rails" from 2013, comparing the two proposed solutions, one being the "Rails-way" and one being an approach of hexagonal architecture[13], abstracting away models from the persistence layer, data retrieval away from the database queries, resulting in a design

[13] Hexagonal architecture, is a software architecture which tries to loosely couple components of a software to make them easily testable and replaceable. It was invented by Alistair Cockburn in 2005 as an attempt to clearly separate business logic from other layers of the software, such as the user interface. (*Hexagonal architecture (software)* 2019)

that enables a practitioner to, for example, replace the database with an in-memory store or a web service. However, since in this example, the three possible persistence layers have very different performance characteristics and properties, Heinemeier Hansson argued that they would also be used in a very different way, but these implementation details were now hidden behind a repository pattern[14], for the sake of testability in isolation. Fowler and Beck countered that saying that TDD is causing these effects and the overuse of mocks is a conflation of cause and effect, with Fowler saying that in these cases, isolation, not TDD is the driver of this design.

While Heinemeier Hansson did admit that tests do sometimes reveal a good design, he believes that where people end up with TDD is rarely the best solution, calling it "*faith-based* TDD", as in having faith that using TDD will eventually lead you to the correct or best solution, to which Beck countered that TDD is not taking a developer anywhere, since TDD does not create the design, but the developer is making the design decisions, which are being made with any software development practice.

Another topic that was covered in the discussion and also mentioned in Beck, 2003, p. 9, was that many companies including Basecamp shifted from having dedicated Quality Assurance (QA) teams to relying more heavily on automated testing and building up confidence that no serious bugs would slip through from one release to the next. But Basecamp does still have QA, but the role of the QA team transformed from doing what automated tests are now doing — following a list of actions and checking if no new defect or regressions happened — to testing softer goals, such as User Experience (UX). QA in this sense is a very powerful and valuable thing to have as a company, since there are dedicated people testing the software who are not intimate with the code. This kind of QA can only be replaced with automated tests with much more effort than normal unit, functional and system testing.

At the time of the video chats Beck said that at Facebook[15] had no QA beyond the fact that they have a motto that no problem is a single person's responsibility, so everyone is being accountable for quality and defects that

[14]Maybe not really a software pattern in the classical sense, it is used to abstract away data access via a repository, so that the persistence layer for storing data can be replaced and remain an implementation detail.

[15]Beck worked at Facebook for 7 years between 2011 and 2018.

might occur. Fowler added, that at his workplace, ThoughtWorks, QA is still happening, but it has shifted from being a completely separate team to something much more collaborative.

At last, the topic of "overtesting" was covered, where Heinemeier Hansson argued that testing everything in multiple places, for example testing a validation of an attribute or class member in a unit test, again in an integration test and yet again in an end-to-end test made it hard to change the behavior of software. While this is not true for refactoring and might have huge benefits for it, he argued that developers are more likely to add new functionality or change existing behavior, rather than refactoring existing code in a behaviour preserving way.

Beck concurred, saying that overtesting is definitely something that exists and mentioned a metric for this, the so called "delta coverage". This metric is used to evaluate how much unique value a test adds to a test suite that no other test provides. If there are lots of tests that add 0% delta coverage, these tests can usually be safely deleted, unless they serve a communication purpose, for example to show how a feature is to be properly used. Covering the same functionality with more than one test leads to coupling within the tests and in an extreme scenario could lead to testing-fatigue if developers find themselves spending more time adapting or rewriting tests than writing actual production code. However, there should always be enough coverage and tests that code can be written or changed, knowing that at least one test will fail if a new defect is introduced.

5.5.1 3X by Kent Beck

After joining Facebook Beck had the chance to give a course at one of Facebook's internal hackathons. Of course, given his reputation and expertise in the field, he decided to give a talk about TDD. Much to his surprise nobody signed up[16], which made him question why. After some wondering he came to the conclusion that Facebook works in a different way and that

[16]According to Beck, the course right above his on the sheet where people could write their course names, "Advanced Excel" and the course below him, "Argentinian Tango" were full, whereas his course had zero sign-ups. (Beck, 2016, 02:00)

IBM Black Team

Compared to what has been found out about the importance of testing and the shift from exclusively dedicated software testers to developers writing their own tests and QA teams being reduced, there is an anecdote about an infamous team of testers at IBM, the *Black Team*. They were testers that achieved slightly better than the rest of testers at IBM and were put together in a team. The result was that they performed exceptionally well, but also build a culture and a certain reputation around them. The goal shifted from testing software, to trying to break it. They took pride in breaking and torturing the software written by others. Soon they were feared and little comradery was left towards the programmers of the software. (DeMarco and Lister, 2013; Romero, 2002)

This is a counter example of what TDD and XP offer, which is a much more collaborative approach and to find bugs before they occur at all.

engineers there often do things in a different fashion, often in a waterfall model approach, something he thought he killed over a decade ago. So he asked himself, maybe the others are not wrong, but maybe they are solving a different problem than what he was trying to solve with TDD.

This thought process resulted in the creation of 3X, where the three X stand for *Explore*, *Expand*, *Extract*. These describe different phases of a software product development lifecycle that will be briefly discussed here.

Explore This phase is used to experiment. A lot. Beck says that one should do a hundred little experiments, in the hopes that one might yield an extraordinary result. It can be ideas for work or for a new startup. In this phase, Beck says, that the rules of normal software development do not apply. For example instead of using a TDD approach here, one should do without tests, because if an experiment takes two days with testing and one without, one more experiment could have been done in the time spent on testing. And in this phase, the experiments create the value.

Expand The *Expand* phase is where projects hit obstacles that prevent them from growing (expand). During this phase, all bottlenecks and hurdles should be removed to aid in expansion and growth.

Extract During extraction, goals, problems and solutions to them are becoming clear. The project that started as an experiment and expanded beyond it can now be extracted. Extraction can mean many things, for example pulling out some services, features etc. and putting them into their own project. This is also where the more "classical" software development methodologies such as TDD can come to fruition.

Startups, as well as grown companies, can make use of 3x, as well as TDD in later stages or more mature products. Both approaches have their validity and place and create value in their own way, but do not contradict themselves in the sense that practitioners must choose between one or the other and use them exclusively. (Beck, 2016)

5.5.2 Example of Test-Induced Design Damage

Listing 5.1[17] is an example of idiomatic Rails code without any additional abstractions added to aid better unit testing. It is a typical controller class with a controller action named `create`. Testing this class mixes testing of the model and thus the persistence layer as well as the networking layer, if exercised through a system test.[18]

Listings 5.2 to 5.5[17] show a "damaged" version of the code in Listing 5.1. Concerns have been separated in different classes, sometimes referred to as "Service Objects"[19]. The controller does not know anything about a model anymore, but rather simply calls a `CreateRunner`.

[17]Usage rights granted by David Heinemeier Hansson.

[18]Controller tests do exist in Rails, but they are discouraged from being used as they in fact *do* mock some parts of the middleware stack of Rails and thus do not actually reflect what is happening in a real world scenario. (*RSpec 3.5 has been released!* 2016; *Deprecate assigns() and assert_template in controller testing · Issue #18950 · rails/rails* 2015)

[19]A "Service Object" is a PORO (Plain Old Ruby Object), which is just a normal object, but was given the "service" name in the Ruby on Rails community, because it is usually used to execute business logic, sometimes involving more than one model, talking to third party APIs etc. It can be thought of as a variation of the "command pattern" (*Command pattern* 2019).

```ruby
class EmployeesController < ApplicationController
  def create
    @employee = Employee.new(employee_params)

    if @employee.save
      redirect_to(
        @employee,
        notice: "Employee #{@employee.name} created"
      )
    else
      render :new
    end
  end
end
```

Listing 5.1: Original Rails code

```ruby
class EmployeesController < ApplicationController
  def create
    CreateRunner.new(self, EmployeesRepository.new)
                .run(params[:employee])
  end

  def create_succeeded(employee, message)
    redirect_to employee, notice: message
  end

  def create_failed(employee)
    @employee = employee
    render :new
  end
end
```

Listing 5.2: Hexagon-inspired, test-induced, “damaged” version of a Ruby on Rails controller action: Controller (Heinemeier Hansson, 2019)

The `CreateRunner` (see Listing 5.3) which uses dependency injection[20] (DI) to inject an `EmployeesRepository` (see Listing 5.4) which abstracts away the persistence layer using the repository pattern.

```
class CreateRunner
  attr_reader :context, :repo

  def initialize(context, repo)
    @context = context
    @repo    = repo
  end

  def run(employee_attrs)
    @employee = repo.new_employee(employee_attrs)

    if repo.save_employee
      context.create_succeeded(
        employee,
        "Employee #{employee.name} created"
      )
    else
      context.create_failed(employee)
    end
  end
end
```

Listing 5.3: Hexagon-inspired, test-induced, "damaged" version of a Ruby on Rails controller action: Service Object (Heinemeier Hansson, 2019)

The `CreateRunner` also does not know about the model directly, it merely orchestrates the creation via the `EmployeesRepository` and depending on the success of the saving of a `Biz::Employee` (see Listing 5.5) record it calls the appropriate methods of the controller which renders the results back to the user.

[20] "Dependency Injection" means that an object gets its dependencies from another object, for example objects that need to talk to a JSON (JavaScript Object Notation) API over HTTP get their HTTP-client "injected" instead of creating an instance of one themselves. (*Dependency injection* 2019)

```ruby
class EmployeesRepository
  def new_employee(*args)
    Biz::Employee.new(Employee.new(*args))
  end

  def save_employee(employee)
    employee.save
  end
end
```

Listing 5.4: Hexagon-inspired, test-induced, "damaged" version of a Ruby on Rails controller action: Repository (Heinemeier Hansson, 2019)

```ruby
require 'delegate'

module Biz
  class Employee < SimpleDelegator
    def self.wrap(employees)
      employees.wrap { |e| new(e) }
    end

    def class
      __getobj__.class
    end

    # Business logic
  end
end
```

Listing 5.5: Hexagon-inspired, test-induced, "damaged" version of a Ruby on Rails controller action: Model (Heinemeier Hansson, 2019)

This kind of design allows for unit testing each class in isolation, resulting in faster tests since individual components can be injected with mocks. For example the repository would typically be backed by some sort of database, while during testing it can just be a Ruby object or an in-memory database that acts like one.

However, while it is a contrived "Hello World" example (which was also mentioned in Jim Weirich's talk), it shows what Heinemeier Hansson meant

by test-induced design damage. He argues that the first version is much clearer to write but also to read, leading to a lower cognitive load since only *one* class has to be kept in mind while reading and writing instead of four or more. Nagappan et al. called this "*intellectual control*", see subsection 5.4.2.

5.5.3 Conclusion

In conclusion, like most discussions titled "*Is X dead?*" the answer in engineering is often "it depends". Valid concerns regarding "test-induced design damage" have been raised, such as people justifying certain software architectures that make for easily testing code in isolation, often introducing layers of indirection in the process which might make understanding the code harder. The counter argument was that TDD does not force any design on its own and that design decisions are still left to the developer, but using TDD might lead you to a design that is easily testable, also arguing that if some code is hard to test, it might not be the best design.

Other arguments such as TDD leading to overtesting and TDD not lending itself to all sorts of code being written were also discussed and resulted in agreement, even if not in all points. TDD is just one of many tools available to a developer, and teams have to carefully evaluate which process they want to adopt and carefully weigh the cost-benefit factor for the kind of work they do. For explorative work, the 3x model was mentioned as one alternative to TDD.

However as was shown in the previous sections of this chapter (for example section 5.4), there are measurable benefits for certain kinds of software development projects which adopted TDD.

6 Type Systems

Already mentioned in chapter 1, Hoare wrote in 1996 that one of the many advancements for software engineering were type systems. In this chapter a few kinds of categories of type systems will be briefly discussed and their claimed benefits and if they have been shown to improve the development of software in a measurable way. This includes static versus dynamic typing and within dynamic typing weak and strong typing will be discussed. Since this is not the main topic of this thesis, some parts are intentionally described in an informal way, since type systems can be discussed at length and warrant their own theses and books.

Most notably JavaScript — given its recent rise in popularity and the amount of work being done by software giants such as Google, Microsoft and Facebook, to alleviate some of the pain points that stem from its dynamic nature — and Ruby — since this is the language used in the server backend of OSKAR, which will be discussed in chapter 7 — will be shortly described.

6.1 About Type Systems

> *The fundamental purpose of a* ***type system*** *is to prevent the occurrence of execution errors during the running of a program.*
> *A program variable can assume a range of values during the execution of a program. An upper bound of such a range is called a* ***type*** *of the variable.*
>
> (Cardelli, 2004, pp. 2277–2278)

A type system is a collection of type rules, whereas a type rule is a rule that states under which conditions forbidden errors will not occur in a particular program. Forbidden errors are errors that occur when certain

invalid operations are applied to a value. One example of such a forbidden error is `not(3)`, where the boolean operator `not` was applied to an integer (3). (Cardelli, 2004, p. 2278) However, typed languages exist where such an operation is allowed such as Ruby and JavaScript.

Type systems are checked by a type checker, which, as the name suggests, checks if all types and operations that are being applied to them make sense. For compiled languages this is mostly done during the compile phase, with some type checking going on during the runtime as well, for example Java's `instanceof` can check a type during runtime.

But even if a type checker concludes that all operations are permitted, a program might still contain errors if the type system is not sound. Soundness means that a type checker might catch all type errors during the static analysis and none during runtime. For example, TypeScript does not have a sound type system and cannot guarantee a variable has a certain type at runtime.

This very brief and mostly informal, often — due to brevity — not completely accurate description shall suffice to go into the differences of certain type systems and how they correlate with research about their positive benefits and negative downsides.

6.2 Strict and Dynamic Typing

Strong and weak typing are concepts that describe how a programming language handles situations where incompatible types are combined with an operator they both share.

One such example can be given for Ruby, which is strongly typed, and JavaScript which is weakly typed and uses implicit type coercion. Type coercion means the usage of one type as if it were another. It is "*a controversial feature that enriches a language's expressivity at the cost of undermining type safety and code understandability*" (Gao et al., 2017, p. 3). In the below example it can be seen that adding a string to an integer is a `TypeError` in Ruby, while in JavaScript the integer is silently converted to a string and then concatenated to the string `"0"`, which can result in subtle bugs.

```
> 3 + "0"
Traceback (most recent call last):
    5: from irb:23:in `<main>'
    4: from irb:23:in `load'
    3: from irb:11:in `<top (required)>'
    2: from (irb):1
    1: from (irb):1:in `+'
    TypeError (String can't be coerced into Integer)
```

Listing 6.1: Ruby's strong typing results in a `TypeError`

```
> 3 + "0"
'30'
```

Listing 6.2: JavaScript's type coercion leads to concatenation

```
> ('b' + 'a' + + 'a' + 'a').toLowerCase()
'banana'
```

Listing 6.3: A more curious example of JavaScript's type coercion

6.3 Claimed Benefits of Strict Typing

JavaScript being a trending programming language[1], no longer restricted to the browser due to JavaScript runtime environments like Node.js[2], was in need for better tooling regarding its type system which is both dynamic and weak. Maintenance, while entirely possible, and refactoring with very clever tooling was no longer sufficient for big projects, which resulted in a lot of new tools and new languages being developed that either compile (or "transpile"[3]) to JavaScript or extend the JavaScript language, among

[1]Metrics used to measure the popularity of programming languages are always up to controversy, but nonetheless two sources using different metrics are the *GitHub Octoverse Report* 2018 and the *The RedMonk Programming Language Rankings* (O'Grady, 2016)

[2]Node.js Foundation, 2019.

[3]"Transpilation" is the process of taking the source code of one programming language and transforming ("transpiling") it into equivalent source code of either the same or a different programming language. (*Source-to-source compiler* 2019)

them Google's Closure compiler, Microsoft's TypeScript[4] and Facebook's Flow[5] that sought to fix some of the shortcomings of JavaScript or reduce unwanted behavior.

One of the downsides when refactoring huge codebases of dynamically typed code, in case of JavaScript and Ruby, is that one has to run the code, or a test suite if one exists, to see if everything was moved, renamed, refactored correctly, since due to the dynamic nature of these languages and the metaprogramming[6] possible in Ruby, certain passages of code *might* make sense during the runtime but are hard to examine statically. With added types, for example going through Flow or TypeScript's compiler `tsc`, at least some potential errors can be caught immediately.

Gao et al. conducted a study in 2017 called "To Type or Not to Type," researching if adding type declarations would help in reducing the defect rate for JavaScript codebases. More precisely, detecting defects before code is made public, either via a commit in a version control system such as git, or worse, being shipped to customers of a software product or consumers of a library.

Their study focused on public defects, meaning defects that were fixed after being published, but they assume that via typing many private defects would have been caught even earlier, reducing the number of public defects. Private in this context means that during the writing of code or refactoring, many little defects and errors can occur and are already caught during development, for example typographical errors, missing import statements, etc. Using types an additional class of errors can be potentially be caught, from nullability errors (variables that can potentially be `null` or `undefined` in JavaScript) to type errors (for example trying to assign a string to a number variable) etc. But to detect these sort of private errors, deep integration into the workflow of a practitioner would be needed for example via editor or IDE (Integrated Development Environment) integration. Since this is a very intrusive and often infeasible technique, Gao et al. concentrated on public

[4] *TypeScript - JavaScript that scales.* 2019.
[5] *Flow* 2019.
[6] In the case of Ruby this is, for example the ability to write programs that can modify themselves during runtime.

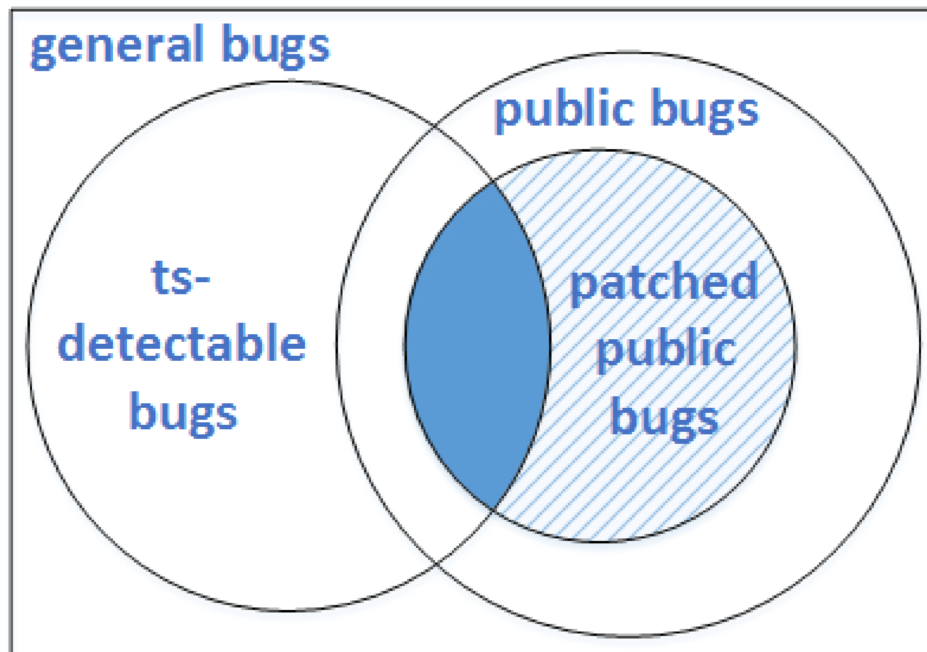

Figure 6.1: Error model of Gao et al.'s study (Gao et al., 2017, p. 3), © 2017 IEEE

defects on open source projects available on GitHub. See Figure 6.1 for their error model. (Gao et al., 2017, pp. 1, 9)

Four hundred projects were selected and sampled, collecting the hash value of the commit that potentially fixes the defect and a potential parent commit that still contains the defect. Using this database of defects each of the researchers was given a maximum of ten minutes to decide if adding types would have resulted in this defect not being made in the first place. Flow and TypeScript annotations were added and the samples were split into three categories, *ts-detectable*, *ts-undetectable* and *unknown*, where *ts* stands for *type system*. After the initial classification, the *unknown* defects were discussed among the group and some more defects were reclassified from *unknown* to *ts-detectable* or *ts-undetectable*.

As a result, Gao et al. found that 15% of defects could have potentially be prevented by using a type system, which suggests that there is some evidence that type systems could help reduce the defect rate of dynamically

typed code bases. As a manager at Microsoft noted — when being presented the result of the study — that a 15% decrease in defects is almost a "no-brainer", since few other technologies can have the same impact. However, what has not been studied in the paper was the added time needed to type a code base and if these errors could have been prevented if the code base had an adequate test suite. Type errors can be covered to some extent via unit tests, but as has been found out by Nagappan et al. in 2008, using a TDD approach can result in 15–35% longer development time. It has to be noted that the study done by Nagappan et al. was also done on statically typed languages, such as Java.

Copyright notice
2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE), Gao et al., "To Type or Not to Type," pp. 758–769, © 2017 IEEE

6.4 Efforts being made for JavaScript

Two such efforts have been indirectly discussed in section 6.3, Flow and TypeScript. In simple terms, Flow is a static type checker for JavaScript, using type inference and annotations otherwise, while TypeScript is its own language, and a superset of JavaScript[7].

Other solutions exist that do not try to augment the JavaScript language but rather provide their own syntax and type guarantees during development which are much stronger than those of the gradual typing solutions mentioned above. Notable examples are Elm[8], PureScript[9] and Reason[10].

[7] Although not all JavaScript code is valid *type-correct* TypeScript, `tsc` will still output valid JavaScript, given syntactically valid JavaScript as input. (Earwicker, 2019)

[8] *Elm - A delightful language for reliable webapps* 2019.

[9] *PureScript* 2019.

[10] *Reason* 2019.

6.4.1 Reason

Reason (previously ReasonML) provides a familiar syntax for JavaScript developers to make the onboarding easier and builds on top of OCaml, providing a different syntax for the OCaml language, which has a very strong type system and type inference. OCaml has a strong academic background and has been in development for almost 25 years, suggesting a very mature and robust language.

Like all solutions presented, it is used in production, but Reason is (with a high degree of certainty) used in one of the products that is used by the largest amount of users[11], which is the Facebook Messenger[12], resulting in little to no runtime errors for the parts written in Reason[13].

6.5 Efforts being made for Ruby

Ruby is dynamically typed and has strong metaprogramming capabilities that make it possible, among other things, to build methods and even whole classes at runtime that are very hard if not impossible to statically analyze before running the code. However, since it is used in some big and popular projects such as the macOS package manager Homebrew (McQuaid et al., 2019) and the web framework Ruby on Rails (used by companies such as GitHub, Shopify, Airbnb, Netflix) there exist large codebases generating billions of revenue every year. This fact and the advent of gradual typing introduced in other languages via third party solutions such as TypeScript and Flow for JavaScript, or officially in other languages such as Python since version 3.5 (Rossum et al., 2014) lead to some discussions in the past[14] and development of some third party solutions, most notably Sorbet by Stripe (Stripe, 2019). Types will be introduced in Ruby 3 and will be an optional feature (for more information, see subsection 6.5.2).

[11] According to Facebook, Messenger had over 1.3 billion monthly active users in 2019. (Facebook, 2019)

[12] *Messenger* 2019.

[13] *Messenger.com Now 50% Converted to Reason · Reason* 2017.

[14] *Feature #9999: Type Annotations (Static Type Checking) - Ruby Issue Tracking System* 2019.

6.5.1 Sorbet

Sorbet is a gradual type checker for Ruby with backwards compatible syntax. The advantage here is for large and old codebases where a complete rewrite in a different language is either infeasible or would come with a great risk, gradual typing can be a solution, because type checking can be introduced file by file and team by team.[15]

Sorbet has a static type checker[16] as well as a runtime, because not all type errors can be caught statically in highly dynamic languages like Ruby[17]. A runtime also enables constant checking of type signatures as they will be exercised constantly, during development and testing. Types can be inferred to a certain degree, but also manually written type signatures[18] can be used, in addition to RBI files[19], "Ruby Interface" files, that can contain type information. See Listing 6.4[20] for such annotated code and Listing 6.5 for the resulting errors. All this is done to have it work and be compatible with existing Ruby versions. RBI files will be part of Ruby 3 (Endoh, 2019b, p. 5) and the Sorbet and Ruby developer teams are in close contact to combine their efforts to introduce typing into Ruby 3.

[15] *Gradual Type Checking & Sorbet · Sorbet* 2019.
[16] *Enabling Static Checks · Sorbet* 2019.
[17] *Enabling Runtime Checks · Sorbet* 2019.
[18] `sigs` in Sorbet.
[19] *RBI Files · Sorbet* 2019.
[20] Copyright 2017–2019 Stripe Inc.

```
class A
  extend T::Sig

  sig {params(x: Integer).returns(String)}
  def bar(x)
    x.to_s
  end
end

def main
  A.new.barr(91)    # error: Typo!
  A.new.bar("91")   # error: Type mismatch!
end
```

Listing 6.4: Ruby code annotated with Sorbet

```
editor.rb:11: Method barr does not exist on A
    11 |   A.new.barr(91)    # error: Typo!
           ^^^^^^^^^^^^^^

  Autocorrect: Use `-a` to autocorrect
    editor.rb:11: Replace with bar
    11 |   A.new.barr(91)    # error: Typo!
                 ^^^^


editor.rb:12: Expected Integer but found String("91") for
   ↪  argument x
    12 |   A.new.bar("91")   # error: Type mismatch!
           ^^^^^^^^^^^^^^^

    editor.rb:4: Method A#bar has specified x as Integer
     4 |   sig {params(x: Integer).returns(String)}
                       ^

  Got String("91") originating from:
    editor.rb:12:
    12 |   A.new.bar("91")   # error: Type mismatch!
                     ^^^^

Errors: 2
```

Listing 6.5: The resulting type errors when using the Sorbet runtime

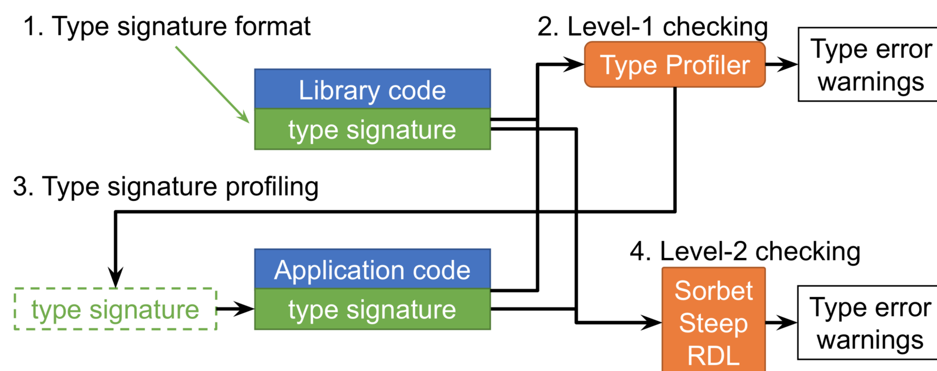

Figure 6.2: Ruby 3 static analysis (Endoh, 2019b, p. 4)

6.5.2 Ruby 3.0

Ruby 3 will have types, as announced at RubyKaigi 2019 in Japan. It will consists of several layers. First of all, a standard type signature format is being developed (the aforementioned RBI files[21]), which is planned to then cover the whole Ruby standard library. On top of this there will be two kinds of type checkers. First a "Level-1" type checker that tries to suggest types using type inference for code that is not annotated. It can find potential `NoMethodErrors` and `TypeErrors`. One such potential solution is the `ruby-type-profiler` being developed by Endoh, but it is still in an experimental stage as of writing.[22] Then, there will be a "Level-2" type checker that checks code according to given type signatures. One such possible solution, Sorbet, was discussed in subsection 6.5.1. An example of Ruby code without type annotations can be found in Listing 6.6[23] with the corresponding RBI file in Listing 6.7[23] that contains the type annotations. (Endoh, 2019b, pp. 5–10). See also Figure 6.2[23] for an overview of the type checking efforts being made for Ruby 3.

[21]Progress can be followed on GitHub. (*Type signature for Ruby classes.* 2019)
[22]Progress can be followed on GitHub. (Endoh, 2019a)
[23]Usage rights granted by Yusuke Endoh.

```ruby
class Foo
  def inc(num)
    num + 1
  end
end
```

Listing 6.6: Ruby code without type signatures (Endoh, 2019b, p. 5)

```ruby
class Foo
  def inc : Integer->Integer
end
```

Listing 6.7: Corresponding Ruby Interface file (Endoh, 2019b, p. 5)

7 OSKAR

7.1 Background

In 2013 Norbert Rabl Ziviltechniker GmbH started developing the app OSKAR in collaboration with the University of Technology Graz, which resulted in two master's theses being written: "Implementing reliable Android applications" by Markart, 2014 and "Agile Entwicklung einer Anwendung zur Kontrolle und Dokumentation der Umsetzung von sicherheitstechnischen Maßnahmen aus dem Bereich des Arbeitnehmerschutzes für Bauarbeiten" by Taferner, 2015. These investigated, among other topics, how the digitalization of requirements faced by health and safety requirements on construction sites could look like.

The result was OSKAR, which simplifies the implementation of safety-related requirements such as identification of dangerous situations, setting measures to avoid dangerous situations and allows documentation of the results to achieve the legally required "effective control system"[1].

Between 2016 and 2017 OSKAR received research funding from the FFG (Österreichische Forschungsförderungsgesellschaft) in connection to the program AT:net.

[1] The Supreme Administrative Court (VwGH) has — over the years — asked for an "effective control system" ("*Wirksames Kontrollsystem*"), without specifying how such a system could look like. OSKAR is, or at the very least gets very close to, being such a system. (*Österreichischer Verwaltungsgerichtshof - Arbeitnehmerschutz: Feststellungen zum wirksamen Kontrollsystem erforderlich* 2018)

7.2 OSKAR-Server Overview

The OSKAR[2] project which is implemented using the Ruby on Rails framework, consists of more than 2.400 automated tests. In this project tests are written for the majority of features although not always first, as a test-driven development style would mandate, depending on context.

Out of the over 2.400 tests in the OSKAR test suite, about 900 are unit tests, 270 are E2E tests, and the rest are integration and system tests. The ratio of production code to test code is 1:2.8.

In the following sections, some examples of the development of the project will be given and how they relate to the lessons learned from the research discussed in the previous chapters applies to it.

7.3 Principles of the Ruby Community

Ruby and its community have a strong focus on developer happiness (Venners, 2003) — as does Ruby on Rails (Heinemeier Hansson, 2016) — and testing, which manifests not only in the community itself (Elliott, 2018) but also in the tools that were written to make testing easy and tests a joy to write, such as RSpec[3], a BDD (Behavior-Driven Development)[4] framework, that is used in OSKAR for a total of over 2.400 tests using an easily readable DSL (Domain Specific Language)[5].

[2]When talking about OSKAR in the context of this thesis, only the backend server is meant, not the Android or iOS app.

[3]*RSpec* 2019.

[4]BDD, in very broad terms, evolved out of TDD and focuses on getting all parties (developers, QA, business participants etc.) involved in the formalization of requirements and specifications, typically using natural language constructs (stories) that almost read like English. (North, 2006)

[5]A DSL is a computer language tailored towards a specific domain of an application. For example RSpec is tailored towards BDD. (*Domain-specific language* 2019)

7.4 TDD in OSKAR

TDD is used not exclusively for development of OSKAR, depending on the task, but is still heavily employed during the development of new features and the fixing of defects. One such example will be given, where the TDD methodology was used to add a new field to the payload of the JSON API. First, the test that already exists is adapted, see Listing 7.1. Then, the tests are run, with the result that the test indeed fails, as expected, see Listing 7.2. After this step, the minimal implementation that will make the test pass is added in Listing 7.3. The tests are run again, and indeed, they are green and the feature has been implemented successfully, see Listing 7.4. After pushing this to the companies git repository, all tests are being run by a CI system.[6]

Just as the research mentioned in chapter 5, this practice has proven very useful during the development of OSKAR. So far, over 2.400 tests have been written, many using a TDD approach, especially regression tests for reported defects. In extreme cases, it took hours to get a failing test while the fix for the defect took less than 5 minutes. But over the course of over 5 years[7], these tests have proven invaluable, time and time again.

Also during the time when the code responsible for sending push notifications to the mobile clients of OSKAR for Android and iOS got more complex to create different payloads depending on platform, app version etc. the class was first extended, making it a very hard to reason about because so many conditionals and control statements were being used. But during that time, over 70 tests were written, documenting the intended behavior. When it was then refactored during the same development cycle and the code rewritten to use a OOP style to get rid of all the conditionals and control statements, the tests found many edge cases, not covered by the refactored code during development. Using small iterations and behavior preserving transformations, as recommended by Beck and Fowler (see sections 5.2 and 5.2.2) the refactoring was completed without introducing any new defects.

[6]Code has been simplified for better results in print.
[7]The first commit was on June 10th 2014.

```
--- a/spec/api/v3/controllers/projects_api_spec.rb
+++ b/spec/api/v3/controllers/projects_api_spec.rb
@@ -100,7 +100,7 @@ RSpec.describe Api::V3::
    ↪ ProjectsController, type: :controller do
it 'returns the correct data' do
  get :show, params: { id: project.encoded_id }

- expect(payload.size).to eq 10
+ expect(payload.size).to eq 11
  expect(payload['id']).to eq project.encoded_id
  expect(payload['name']).to eq project.name
+ expect(payload['has_map']).to eq project.has_map
  expect(payload).to have_key 'address'
  expect(payload).to have_key 'users'
```

Listing 7.1: Writing a failing test

```
1) Api::V3::ProjectsController returns the correct data
    Failure/Error: expect(payload.size).to eq 11

      expected: 11
           got: 10

    # ./spec/api/v3/controllers/projects_api_spec.rb:103

2) Api::V3::ProjectsController returns the correct data
    Failure/Error: expect(payload['has_map']).to eq
       ↪ project.has_map

      expected: false
           got: nil

    # ./spec/api/v3/controllers/projects_api_spec.rb:112

Finished in 0.59078 seconds
1 example, 1 failures
```

Listing 7.2: Running the test to see it fail

```
--- a/app/views/api/v3/projects/_project.json.jbuilder
+++ b/app/views/api/v3/projects/_project.json.jbuilder
@@ -9,6 +9,7 @@
 json.id project.encoded_id
 json.name project.name
+json.has_map project.has_map

 if project.archived?
   json.archived_at project.archived_at.utc.iso8601(3)
```

Listing 7.3: Adding the minimal implementation to make the test green

```
Finished in 0.49149 seconds
1 example, 0 failures
```

Listing 7.4: Running the test again to see it turn green

7.5 Coverage in OSKAR

OSKAR uses SimpleCov[8] to measure test coverage. The reported coverage is 95.92%, which is not 100% as the diligent use of TDD would result in, but the reason for this is easy to give. For one, there are features which only exist for internal users and developers, such as internal dashboards with statistics or data export features. Some of them are one-off features which remain dormant or are at an incubation phase, where too many things change too fast to justify a TDD approach. This is similar to what Kent Beck describes as the exploration phase in his 3x concept (see subsection 5.5.1 for details).

It is important to note which parts of the code are untested. Typically a codebase is not tested in an evenly distributed way. For example, not every class is 95.92% covered but the majority of user-facing features has a coverage of 100%, while other internal features are dark spots on the map, having as little as 33% code coverage. So when a codebase reports less than 100% coverage it is important to check how coverage is distributed.

[8] *SimpleCov* 2019.

7.5.1 Code Complexity in OSKAR

One of the parts that is theoretically well tested and has high coverage is the PDF report that OSKAR can generate for its users. The test code generates almost 1.000 (not part of the 2.400 mentioned at the beginning) test cases, which take almost two hours to run. Deep nesting in the production as well as the test code make it hard to impossible to trace back errors to their origin and fixes are hard to write. The maximum block depth in the whole OSKAR code base is in the PDF report code, as well as most other code complexity measures such as cyclomatic complexity[9], ABC size[10] and method length[11] report their maximum in this part of the codebase. To measure these, the Ruby linter RuboCop[12] is used. Anecdotal evidence suggests that what Antinyan et al. found out about the correlation of defects and code complexity also holds true for this project (see also section 4.2), with many of the metrics used my Antinyan et al. also being measured by RuboCop in OSKAR.

Because of the long time the PDF report tests run, they are never run, supporting the argument Beck makes with his "10 minute rule" (see also section 2.2).

It has been an ongoing effort to remedy the situation by splitting methods into smaller ones, reducing block depth and writing fewer, more valuable tests that run in an appropriate amount of time, be it local or on a CI system[13].

[9]Cyclomatic complexity is a quantitative metric, using the control flow graph of a program as its base. (*Cyclomatic complexity* 2019)

[10]ABC size counts the numbers of Assignments, Branches and Conditionals, hence ABC. (*ABC Software Metric* 2018)

[11]Number of lines a method has.

[12]*RuboCop* 2019.

[13]Some CI providers charge per minute, so faster tests can also save money.

8 Conclusion

This thesis began with a lot of quotes by Hoare and B. A. Kitchenham et al. which served as background and motivation for the other chapters.

In chapter 2 it was explained what automated self-testing code is with a short description of the different types of tests that exist. Later in chapter 3 it was argued that testing is a suitable goal for Evidence Based Software Engineering (EBSE), with the definition and goals of EBSE being given. In chapter 4 one metric that lends itself to research, code coverage, was discussed and the question answered if a high enough unit test coverage implies adequacy of test suites. However Antinyan et al. were able to show in a case study done at Ericsson that code coverage alone is not a suitable metric.

Microsoft Research's own Empirical Software Engineering (ESE) group's research was at the center of chapter 5, which talked about Test-Driven Development (TDD) at length. The efficacy of TDD was discussed as well as its potential risks and downsides. However, at least for Microsoft the research showed the clear benefit of using TDD to develop software. Code quality was much higher and defect rate much lower compared to projects using a test-last approach or not testing at all. Although this development style takes 15–35% longer the benefits outweigh the costs.
At the end of this chapter some controversy that occurred in 2014 was discussed, where the question was raised if TDD was dead. Different standpoints were being described as well as an alternative development methodology, 3x by Kent Beck, that can be used in situations where TDD might not be suitable, was described.

A short introduction to type systems was given in chapter 6 with a paper by Gao et al. from 2017 serving as a discussion point to see if strict typing can be useful for development and if its benefits can be measured. It was

shown that in the case of JavaScript, adding types can eliminate up to 15% of bugs that occurred due to its dynamically and weakly typed nature.

Lastly, in chapter 7 everything that has been described so far was put into context to a software product being written at NR.Systems GmbH named OSKAR and how practices and results from research have been employed and how they affected quality and the development process of the project.

Bibliography

ABC Software Metric (Jan. 27, 2018). In: *Wikipedia*. Page Version ID: 822653124. URL: https://en.wikipedia.org/w/index.php?title=ABC_Software_Metric&oldid=822653124 (visited on 09/10/2019) (cit. on p. 69).

And the Winners Are ... (Sept. 26, 2006). Microsoft Research. URL: https://www.microsoft.com/en-us/research/blog/and-the-winners-are/ (visited on 09/06/2019) (cit. on p. 13).

Antinyan, Vard et al. (May 2018). "Mythical Unit Test Coverage." In: *IEEE Software* 35.3, pp. 73–79. ISSN: 0740-7459. DOI: 10.1109/MS.2017.3281318. URL: https://ieeexplore.ieee.org/document/8354427/ (visited on 07/14/2019) (cit. on pp. 17–21, 23, 24, 40, 69, 70).

Apache Subversion (2019). URL: https://subversion.apache.org/ (visited on 08/28/2019) (cit. on p. 9).

Beck, Kent (1999). *Extreme Programming Explained: Embrace Change*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc. ISBN: 978-0-201-61641-5 (cit. on pp. 26, 31).

Beck, Kent (2003). *Test-Driven Development: By Example*. The Addison-Wesley signature series. Boston: Addison-Wesley. 220 pp. ISBN: 978-0-321-14653-3 (cit. on pp. 14, 26–29, 31, 45).

Beck, Kent (May 11, 2012). *Kent Beck answering why he redsicovered, not invented Test-Driven Development*. URL: https://www.quora.com/Why-does-Kent-Beck-refer-to-the-rediscovery-of-test-driven-development-Whats-the-history-of-test-driven-development-before-Kent-Becks-rediscovery/answer/Kent-Beck (visited on 08/26/2019) (cit. on p. 26).

Beck, Kent (Nov. 14, 2016). *3X with Kent Beck*. URL: https://www.youtube.com/watch?v=YX2XR73LnRY (visited on 09/10/2019) (cit. on pp. 46, 48).

Beck, Kent and Cynthia Andres (Nov. 16, 2004). *Extreme Programming Explained: Embrace Change, Second Edition*. Second. Addison-Wesley Professional. 224 pp. ISBN: 978-0-321-27865-4. URL: https://proquest.techbus.safaribooksonline.de/0321278658 (visited on 08/28/2019) (cit. on pp. 8, 9, 29).

Betteridge's law of headlines (Aug. 4, 2019). In: *Wikipedia*. Page Version ID: 909347429. URL: https://en.wikipedia.org/w/index.php?title=Betteridge%27s_law_of_headlines&oldid=909347429 (visited on 08/29/2019) (cit. on p. 43).

Bhat, Thirumalesh and Nachiappan Nagappan (2006). "Evaluating the Efficacy of Test-driven Development: Industrial Case Studies." In: *Proceedings of the 2006 ACM/IEEE International Symposium on Empirical Software Engineering*. ISESE '06. event-place: Rio de Janeiro, Brazil. New York, NY, USA: ACM, pp. 356–363. ISBN: 978-1-59593-218-1. DOI: 10.1145/1159733.1159787. URL: http://doi.acm.org/10.1145/1159733.1159787 (visited on 07/15/2019) (cit. on pp. 36, 38–40).

Bird, Christian et al. (Mar. 19, 2011). "Empirical Software Engineering at Microsoft Research." In: URL: https://www.microsoft.com/en-us/research/publication/empirical-software-engineering-at-microsoft-research/ (visited on 09/04/2019) (cit. on p. 13).

Boeing 737 MAX groundings (Sept. 2, 2019). In: *Wikipedia*. Page Version ID: 913611919. URL: https://en.wikipedia.org/w/index.php?title=Boeing_737_MAX_groundings&oldid=913611919 (visited on 09/02/2019) (cit. on p. 2).

Buxton, J. N. and B. Randell, eds. (1970). *Software Engineering Techniques: Report of a Conference Sponsored by the NATO Science Committee, Rome, Italy, 27-31 Oct. 1969, Brussels, Scientific Affairs Division, NATO* (cit. on p. 27).

Cardelli, Luca (2004). "Type Systems." In: *Computer Science Handbook, Second Edition*. Second. Chapman & Hall/CRC. Chap. 97, pp. 2277–2308. ISBN: 978-1-58488-360-9 (cit. on pp. 53, 54).

Chan, Denise (Aug. 20, 2019). *Sunsetting Mercurial support in Bitbucket*. Bitbucket. URL: https://bitbucket.org/blog/sunsetting-mercurial-support-in-bitbucket (visited on 08/28/2019) (cit. on p. 9).

Code coverage (Aug. 7, 2019). In: *Wikipedia*. Page Version ID: 909756417. URL: https://en.wikipedia.org/w/index.php?title=Code_coverage&oldid=909756417 (visited on 08/26/2019) (cit. on p. 18).

Code Coverage Tutorial: Branch, Statement, Decision, FSM (2019). URL: https://www.guru99.com/code-coverage.html (visited on 08/26/2019) (cit. on pp. 17, 18).

Command pattern (Aug. 21, 2019). In: *Wikipedia*. Page Version ID: 911818765. URL: https://en.wikipedia.org/w/index.php?title=Command_pattern&oldid=911818765 (visited on 09/02/2019) (cit. on p. 48).

Cunningham, Ward (1992). "The WyCash Portfolio Management System." In: *Addendum to the Proceedings on Object-oriented Programming Systems, Languages, and Applications (Addendum)*. OOPSLA '92. event-place: Vancouver, British Columbia, Canada. New York, NY, USA: ACM, pp. 29–30. ISBN: 978-0-89791-610-3. DOI: 10.1145/157709.157715. URL: http://doi.acm.org/10.1145/157709.157715 (visited on 08/27/2019) (cit. on p. 15).

Cyclomatic complexity (Sept. 1, 2019). In: *Wikipedia*. Page Version ID: 913449633. URL: https://en.wikipedia.org/w/index.php?title=Cyclomatic_complexity&oldid=913449633 (visited on 09/10/2019) (cit. on p. 69).

Death of Elaine Herzberg (Aug. 5, 2019). In: *Wikipedia*. Page Version ID: 909453995. URL: https://en.wikipedia.org/w/index.php?title=Death_of_Elaine_Herzberg&oldid=909453995 (visited on 09/02/2019) (cit. on p. 2).

DeMarco, Tom and Tim Lister (2013). *Peopleware: Productive Projects and Teams (3rd Edition)*. 3rd. Addison-Wesley Professional. ISBN: 978-0-321-93411-6 (cit. on p. 47).

Dependency injection (Aug. 9, 2019). In: *Wikipedia*. Page Version ID: 910131552. URL: https://en.wikipedia.org/w/index.php?title=Dependency_injection&oldid=910131552 (visited on 09/02/2019) (cit. on p. 50).

Deprecate assigns() and assert_template in controller testing · Issue #18950 · rails/rails (Feb. 15, 2015). GitHub. URL: https://github.com/rails/rails/issues/18950 (visited on 08/29/2019) (cit. on p. 48).

Deursen, Arie van (2001). "Program Comprehension Risks and Opportunities in Extreme Programming." In: *Proceedings Eighth Working Conference on Reverse Engineering*, pp. 176–185. DOI: 10.1109/WCRE.2001.957822 (cit. on pp. 32–35).

DO-178B (July 21, 2019). In: *Wikipedia*. Page Version ID: 907233189. URL: https://en.wikipedia.org/w/index.php?title=DO-178B&oldid=907233189 (visited on 09/03/2019) (cit. on p. 25).

Domain-specific language (July 27, 2019). In: *Wikipedia*. Page Version ID: 908097306. URL: https://en.wikipedia.org/w/index.php?title=Domain-specific_language&oldid=908097306 (visited on 09/06/2019) (cit. on p. 65).

Earwicker, Daniel (2019). *ES6 should be valid TypeScript · Issue #2606*. GitHub. URL: https://github.com/Microsoft/TypeScript/issues/2606#issuecomment-89266470 (visited on 09/04/2019) (cit. on p. 58).

EBSE Website (Mar. 15, 2012). URL: http://community.dur.ac.uk/ebse/ (visited on 08/26/2019) (cit. on p. iv).

Elliott, Thomas (Dec. 7, 2018). *The State of the Octoverse: communicating with emoji on GitHub*. The GitHub Blog. URL: https://github.blog/2018-12-07-octoverse-emoji-on-github/#percent-of-reactions-by-emoji-type-and-programming-language (visited on 09/04/2019) (cit. on p. 65).

Elm - A delightful language for reliable webapps (2019). URL: https://elm-lang.org/ (visited on 09/04/2019) (cit. on p. 58).

Empirical Software Engineering Group (ESE) (2019). Microsoft Research. URL: https://www.microsoft.com/en-us/research/group/empirical-software-engineering-group-ese/ (visited on 09/06/2019) (cit. on p. 13).

Enabling Runtime Checks · Sorbet (2019). URL: https://sorbet.org/ (visited on 09/05/2019) (cit. on p. 60).

Enabling Static Checks · Sorbet (2019). URL: https://sorbet.org/ (visited on 09/05/2019) (cit. on p. 60).

Endoh, Yusuke (Sept. 9, 2019a). *An experimental type-level Ruby interpreter for testing and understanding Ruby code*. original-date: 2019-02-22T07:26:19Z. URL: https://github.com/mame/ruby-type-profiler (visited on 09/09/2019) (cit. on p. 62).

Endoh, Yusuke (Mar. 26, 2019b). *Ruby 3 Progress Report*. URL: https://docs.google.com/presentation/d/1z_5JT0-MJySGn6UGrtdafK1oj9kGSO5sGlTtEQJz0JU/view#slide=id.g57cf166414_14_5 (visited on 09/05/2019) (cit. on pp. 60, 62, 63).

Facebook, Inc (Apr. 27, 2019). *F8 Messenger Fact Sheet 2019*. URL: https://messengernews.fb.com/wp-content/uploads/2019/04/3-2019-Messsenger-F8-Fact-Sheet-1.pdf (visited on 08/28/2019) (cit. on p. 59).

Feature #9999: Type Annotations (Static Type Checking) - Ruby Issue Tracking System (2019). URL: https://bugs.ruby-lang.org/issues/9999 (visited on 09/05/2019) (cit. on p. 59).

Fenton, Norman E. and Shari Lawrence Pfleeger (1998). *Software Metrics: A Rigorous and Practical Approach*. 2nd. Boston, MA, USA: PWS Publishing Co. ISBN: 978-0-534-95425-3 (cit. on p. 36).

Firefox Headless mode (2019). MDN Web Docs. URL: https://developer.mozilla.org/en-US/docs/Mozilla/Firefox/Headless_mode (visited on 08/28/2019) (cit. on p. 11).

Flow (2019). *Flow: A Static Type Checker for JavaScript*. Flow. URL: https://flow.org/en/ (visited on 09/04/2019) (cit. on p. 56).

Fowler, Martin (1999). *Refactoring: Improving the Design of Existing Code*. In collab. with Kent Beck. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc. 431 pp. ISBN: 978-0-201-48567-7 (cit. on pp. 7, 30, 32).

Fowler, Martin (Sept. 1, 2004a). *Definition of Refactoring*. martinfowler.com. URL: https://martinfowler.com/bliki/DefinitionOfRefactoring.html (visited on 09/02/2019) (cit. on p. 30).

Fowler, Martin (Jan. 3, 2004b). *Refactoring Malapropism*. martinfowler.com. URL: https://martinfowler.com/bliki/RefactoringMalapropism.html (visited on 09/02/2019) (cit. on p. 30).

Fowler, Martin (May 1, 2006). *Continuous Integration*. martinfowler.com. URL: https://martinfowler.com/articles/continuousIntegration.html (visited on 08/28/2019) (cit. on p. 8).

Fowler, Martin (Apr. 17, 2012). *TestCoverage, Martin Fowler*. martinfowler.com. URL: https://martinfowler.com/bliki/TestCoverage.html (visited on 08/26/2019) (cit. on p. 24).

Fowler, Martin (May 9, 2014a). *Is TDD Dead*. martinfowler.com. URL: https://martinfowler.com/articles/is-tdd-dead/ (visited on 08/29/2019) (cit. on p. 43).

Fowler, Martin (Jan. 5, 2014b). *Self Testing Code*. martinfowler.com. URL: https://martinfowler.com/bliki/SelfTestingCode.html (visited on 09/06/2019) (cit. on p. 7).

Fowler, Martin (May 5, 2014c). *UnitTest*. martinfowler.com. URL: https://martinfowler.com/bliki/UnitTest.html (visited on 08/28/2019) (cit. on p. 10).

Fowler, Martin (Jan. 16, 2018). *IntegrationTest*. martinfowler.com. URL: https://martinfowler.com/bliki/IntegrationTest.html (visited on 08/28/2019) (cit. on p. 10).

Fowler, Martin (2019). *Refactoring Home Page*. URL: http://refactoring.com (visited on 08/28/2019) (cit. on pp. 30, 31).

Gang Of Four (Oct. 16, 2013). URL: http://wiki.c2.com/?GangOfFour (visited on 08/28/2019) (cit. on p. 7).

Gao, Zheng et al. (May 2017). "To Type or Not to Type: Quantifying Detectable Bugs in JavaScript." In: *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. 2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE). Buenos Aires: IEEE, pp. 758–769. ISBN: 978-1-5386-3868-2. DOI: 10.1109/ICSE.2017.75. URL: http://ieeexplore.ieee.org/document/7985711/ (visited on 07/14/2019) (cit. on pp. 54, 56–58, 70).

Garry, Chris (Sept. 2, 2019). *Original Apollo 11 Guidance Computer (AGC) source code for the command and lunar modules.: chrislgarry/Apollo-11*. original-date: 2014-04-03T15:45:02Z. URL: https://github.com/chrislgarry/Apollo-11 (visited on 09/02/2019) (cit. on p. 1).

George, Boby and Laurie Williams (Apr. 15, 2004). "A structured experiment of test-driven development." In: *Information and Software Technology*. Special Issue on Software Engineering, Applications, Practices and Tools from the ACM Symposium on Applied Computing 2003 46.5, pp. 337–342. ISSN: 0950-5849. DOI: 10.1016/j.infsof.2003.09.011. URL: http://www.sciencedirect.com/science/article/pii/S0950584903002040 (visited on 07/15/2019) (cit. on pp. 32, 36, 38).

Getting Started with Headless Chrome — Web (2019). Google Developers. URL: https://developers.google.com/web/updates/2017/04/headless-chrome (visited on 08/28/2019) (cit. on p. 11).

Git (2019). URL: https://git-scm.com/ (visited on 08/28/2019) (cit. on p. 9).

GitHub Octoverse Report (Oct. 16, 2018). The State of the Octoverse. URL: https://octoverse.github.com/projects#languages (visited on 08/28/2019) (cit. on p. 55).

Goodhart's law (July 26, 2019). In: *Wikipedia*. Page Version ID: 907945035. URL: https://en.wikipedia.org/w/index.php?title=Goodhart%27s_law&oldid=907945035 (visited on 08/26/2019) (cit. on p. 24).

Gradual Type Checking & Sorbet · Sorbet (2019). URL: https://sorbet.org/ (visited on 09/05/2019) (cit. on p. 60).

Ham Vocke (Feb. 26, 2018). *The Practical Test Pyramid*. martinfowler.com. URL: https://martinfowler.com/articles/practical-test-pyramid.html (visited on 08/28/2019) (cit. on p. 11).

Heinemeier Hansson, David (Apr. 29, 2014a). *TDD is dead. Long live testing*. URL: https://dhh.dk/2014/tdd-is-dead-long-live-testing.html (visited on 08/29/2019) (cit. on p. 43).

Heinemeier Hansson, David (Apr. 29, 2014b). *Test-induced design damage*. URL: https://dhh.dk/2014/test-induced-design-damage.html (visited on 08/29/2019) (cit. on p. 43).

Heinemeier Hansson, David (Jan. 2016). *The Rails Doctrine*. Ruby on Rails. URL: https://rubyonrails.org/doctrine/ (visited on 08/29/2019) (cit. on pp. 44, 65).

Heinemeier Hansson, David (2019). *Test Induced Design Damage example*. URL: https://gist.github.com/dhh/4849a20d2ba89b34b201 (visited on 09/02/2019) (cit. on pp. 49–51).

Hexagonal architecture (software) (Aug. 14, 2019). In: *Wikipedia*. Page Version ID: 910783815. URL: https://en.wikipedia.org/w/index.php?title=Hexagonal_architecture_(software)&oldid=910783815 (visited on 09/06/2019) (cit. on p. 44).

Hoare logic (June 18, 2019). In: *Wikipedia*. Page Version ID: 902408484. URL: https://en.wikipedia.org/w/index.php?title=Hoare_logic&oldid=902408484 (visited on 08/29/2019) (cit. on p. 1).

Hoare, C. A. R. (Feb. 1981). "The Emperor's Old Clothes." In: *Commun. ACM* 24.2, pp. 75–83. ISSN: 0001-0782. DOI: 10.1145/358549.358561. URL: http://doi.acm.org/10.1145/358549.358561 (visited on 09/03/2019) (cit. on p. 29).

Hoare, C. A. R. (1996). "How Did Software Get So Reliable Without Proof?" In: *FME'96: Industrial Benefit and Advances in Formal Methods*. Ed. by Marie-Claude Gaudel and James Woodcock. Red. by Gerhard Goos et al. Vol. 1051. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 1–17. ISBN: 978-3-540-60973-5. DOI: 10.1007/3-540-60973-3_77. URL: http://link.springer.com/10.1007/3-540-60973-3_77 (visited on 07/14/2019) (cit. on pp. 1, 3–7, 13, 14, 53, 70).

How SQLite Is Tested (2019). URL: https://www.sqlite.org/testing.html (visited on 08/26/2019) (cit. on p. 25).

ImperialViolet - Apple's SSL/TLS bug (Feb. 22, 2014). URL: https://www.imperialviolet.org/2014/02/22/applebug.html (visited on 08/27/2019) (cit. on p. 5).

Is TDD dead? (May 9, 2014). URL: https://www.youtube.com/watch?v=z9quxZsLcfo (visited on 09/03/2019) (cit. on pp. 26, 27, 44).

Josuttis, Nicolai (Jan. 2000). *eXtreme Programming An interview with Kent Beck*. URL: https://accu.org/index.php/journals/509 (visited on 09/02/2019) (cit. on p. 29).

Juristo, Natalia et al. (Mar. 2004). "Reviewing 25 Years of Testing Technique Experiments." In: *Empirical Software Engineering* 9.1, pp. 7–44. ISSN: 1382-3256. DOI: 10.1023/B:EMSE.0000013513.48963.1b. URL: http://link.springer.com/10.1023/B:EMSE.0000013513.48963.1b (visited on 07/14/2019) (cit. on p. 22).

Kemp, Peter and Paul Smith (June 14, 2010). *Waterfall Model of System Development*. URL: https://commons.wikimedia.org/wiki/File:Waterfall_model.svg (visited on 09/02/2019) (cit. on pp. 36, 37).

Kitchenham, B. A. et al. (2004). "Evidence-based Software Engineering." In: *Proceedings. 26th International Conference on Software Engineering*, pp. 273–281. DOI: 10.1109/ICSE.2004.1317449 (cit. on pp. iv, 4–7, 13, 70).

Kitchenham, B. and S. Charters (2007). *Guidelines for performing Systematic Literature Reviews in Software Engineering* (cit. on p. 15).

Kitchenham, Barbara et al. (July 1995). "Case Studies for Method and Tool Evaluation." In: *IEEE Softw.* 12.4, pp. 52–62. ISSN: 0740-7459. DOI: 10.1109/52.391832. URL: https://doi.org/10.1109/52.391832 (visited on 09/02/2019) (cit. on p. 36).

Lion Air Flight 610 (Aug. 26, 2019). In: *Wikipedia*. Page Version ID: 912578683. URL: https://en.wikipedia.org/w/index.php?title=Lion_Air_Flight_610&oldid=912578683 (visited on 09/02/2019) (cit. on p. 2).

Lions, Jacques Louis (1996). "ARIANE 5 Flight 501 Failure: Report by the Enquiry Board." In: (cit. on p. 2).

Markart, Daniel (2014). "Implementing reliable Android applications." Dissertation/Thesis. PhD thesis (cit. on p. 64).

McQuaid, Mike et al. (Sept. 3, 2019). *Homebrew: The missing package manager for macOS (or Linux)*. original-date: 2016-03-06T05:08:38Z. URL: https://github.com/Homebrew/brew (visited on 09/03/2019) (cit. on p. 59).

Mercurial SCM (2019). URL: https://www.mercurial-scm.org/ (visited on 08/28/2019) (cit. on p. 9).

Messenger (2019). Facebook. URL: https://www.messenger.com/ (visited on 08/28/2019) (cit. on p. 59).

Messenger.com Now 50% Converted to Reason · Reason (Sept. 8, 2017). URL: https://reasonml.github.io/blog/2017/09/08/messenger-50-reason (visited on 08/28/2019) (cit. on p. 59).

Microsoft acquires GitHub (2019). Stories. URL: https://news.microsoft.com/announcement/microsoft-acquires-github/ (visited on 08/28/2019) (cit. on p. 9).

Minitab, LLC (2019). *A comparison of the Pearson and Spearman correlation methods*. URL: https://support.minitab.com/en-us/minitab-express/1/help-and-how-to/modeling-statistics/regression/supporting-topics/basics/a-comparison-of-the-pearson-and-spearman-correlation-methods/ (visited on 09/03/2019) (cit. on p. 21).

Modified condition/decision coverage (Mar. 16, 2019). In: *Wikipedia*. Page Version ID: 888092011. URL: https://en.wikipedia.org/w/index.php?title=Modified_condition/decision_coverage&oldid=888092011 (visited on 09/03/2019) (cit. on p. 25).

Mutation testing (Aug. 23, 2019). In: *Wikipedia*. Page Version ID: 912170781. URL: https://en.wikipedia.org/w/index.php?title=Mutation_testing&oldid=912170781 (visited on 09/03/2019) (cit. on p. 12).

Nagappan, Nachiappan et al. (June 1, 2008). "Realizing quality improvement through test driven development: results and experiences of four industrial teams." In: *Empirical Software Engineering* 13, pp. 289–302. DOI: 10.1007/s10664-008-9062-z. URL: https://doi.org/10.1007/s10664-008-9062-z (cit. on pp. 27, 29, 36, 38–41, 52, 58).

Node.js Foundation (2019). *Node.js*. Node.js. URL: https://nodejs.org/en/ (visited on 08/28/2019) (cit. on p. 55).

North, Dan (Sept. 20, 2006). *Introducing BDD*. Dan North & Associates. URL: https://dannorth.net/introducing-bdd/ (visited on 09/05/2019) (cit. on p. 65).

Null References (Aug. 25, 2009). *Null References: The Billion Dollar Mistake*. InfoQ. URL: https://www.infoq.com/presentations/Null-References-The-Billion-Dollar-Mistake-Tony-Hoare/ (visited on 08/27/2019) (cit. on p. 1).

NVD - CVE-2014-1266 (Feb. 22, 2014). URL: https://nvd.nist.gov/vuln/detail/CVE-2014-1266 (visited on 08/27/2019) (cit. on p. 5).

O'Grady, Stephen (Feb. 19, 2016). *The RedMonk Programming Language Rankings*. tecosystems. URL: https://redmonk.com/sogrady/2016/02/19/language-rankings-1-16/ (visited on 08/28/2019) (cit. on p. 55).

Österreichischer Verwaltungsgerichtshof - Arbeitnehmerschutz: Feststellungen zum wirksamen Kontrollsystem erforderlich (July 4, 2018). URL: https://www.vwgh.gv.at/rechtsprechung/aktuelle_entscheidungen/2018/ra_2017020240.html?0 (visited on 09/11/2019) (cit. on p. 64).

Pearson correlation coefficient (Aug. 25, 2019). In: *Wikipedia*. Page Version ID: 912485501. URL: https://en.wikipedia.org/w/index.php?title=Pearson_correlation_coefficient&oldid=912485501 (visited on 09/03/2019) (cit. on p. 21).

Power-on self-test (Apr. 7, 2019). In: *Wikipedia*. Page Version ID: 891389351. URL: https://en.wikipedia.org/w/index.php?title=Power-on_self-test&oldid=891389351 (visited on 08/28/2019) (cit. on p. 7).

PureScript (2019). URL: http://www.purescript.org/ (visited on 09/04/2019) (cit. on p. 58).

Quicksort (Aug. 27, 2019). In: *Wikipedia*. Page Version ID: 912786103. URL: https://en.wikipedia.org/w/index.php?title=Quicksort&oldid=912786103 (visited on 08/29/2019) (cit. on p. 1).

RBI Files · Sorbet (2019). URL: https://sorbet.org/ (visited on 09/05/2019) (cit. on p. 60).

Reason (2019). URL: https://reasonml.github.io/ (visited on 09/04/2019) (cit. on p. 58).

Richard Hipp (June 25, 2019). *Richard Hipp SQLite ViennaDB Talk 2019.06.25* (cit. on p. 25).

Richard Hipp about the history of testing SQLite — Hacker News (2019). URL: https://news.ycombinator.com/item?id=18686695 (visited on 08/26/2019) (cit. on p. 25).

Rios, Nicolli et al. (Oct. 2018). "A tertiary study on technical debt: Types, management strategies, research trends, and base information for practitioners." In: *Information and Software Technology* 102, pp. 117–145. ISSN: 09505849. DOI: 10.1016/j.infsof.2018.05.010. URL: https://linkinghub.elsevier.com/retrieve/pii/S0950584918300946 (visited on 07/14/2019) (cit. on pp. 14–16).

Romero, Tim (Dec. 17, 2002). *Tangled Webs 7.6 - The Black Team*. URL: http://www.t3.org/tangledwebs/07/tw0706.html (visited on 09/03/2019) (cit. on p. 47).

Rossum, Guido van et al. (Sept. 29, 2014). *PEP 484 – Type Hints*. Python.org. URL: https://www.python.org/dev/peps/pep-0484/ (visited on 09/03/2019) (cit. on p. 59).

RSpec 3.5 has been released! (July 1, 2016). URL: http://rspec.info/blog/2016/07/rspec-3-5-has-been-released/ (visited on 08/29/2019) (cit. on p. 48).

RSpec (2019). *RSpec: Behaviour Driven Development for Ruby*. URL: http://rspec.info/ (visited on 09/05/2019) (cit. on p. 65).

RuboCop (Sept. 10, 2019). original-date: 2012-04-21T10:09:58Z. URL: https://github.com/rubocop-hq/rubocop (visited on 09/10/2019) (cit. on p. 69).

Selenium WebDriver (2019). URL: https://www.seleniumhq.org/projects/webdriver/ (visited on 08/28/2019) (cit. on p. 11).

Shinde, Vijay (2019). *What Is END-TO-END Testing*. URL: https://www.softwaretestinghelp.com/what-is-end-to-end-testing/ (visited on 08/28/2019) (cit. on pp. 10, 11).

SimpleCov (Sept. 8, 2019). original-date: 2010-08-15T15:28:56Z. URL: https://github.com/colszowka/simplecov (visited on 09/10/2019) (cit. on p. 68).

Source-to-source compiler (July 16, 2019). In: *Wikipedia*. Page Version ID: 906556977. URL: https://en.wikipedia.org/w/index.php?title=Source-to-source_compiler&oldid=906556977 (visited on 08/28/2019) (cit. on p. 55).

Spearman's rank correlation coefficient (Aug. 1, 2019). In: *Wikipedia*. Page Version ID: 908824374. URL: https://en.wikipedia.org/w/index.php?title=Spearman%27s_rank_correlation_coefficient&oldid=908824374 (visited on 09/03/2019) (cit. on p. 21).

SQLite Copyright (2019). URL: https://www.sqlite.org/copyright.html (visited on 09/03/2019) (cit. on p. 25).

SQLite TH3 (Test Harness 3) (2019). URL: https://www.sqlite.org/th3.html (visited on 09/03/2019) (cit. on p. 25).

Statement (computer science) (June 16, 2019). In: *Wikipedia*. Page Version ID: 902154255. URL: https://en.wikipedia.org/w/index.php?title=Statement_(computer_science)&oldid=902154255 (visited on 08/26/2019) (cit. on p. 18).

Stripe (Sept. 3, 2019). *A fast, powerful type checker designed for Ruby.* original-date: 2018-06-26T18:13:06Z. URL: https://github.com/sorbet/sorbet (visited on 09/03/2019) (cit. on p. 59).

Stroustrup, Bjarne (1997). *The C++ Programming Language, Third Edition*. 3rd. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc. ISBN: 978-0-201-88954-3 (cit. on p. 5).

Structured Program Theorem (Böhm–Jacopini theorem (Aug. 17, 2019). In: *Wikipedia*. Page Version ID: 911286009. URL: https://en.wikipedia.org/w/index.php?title=Structured_program_theorem&oldid=911286009 (visited on 08/27/2019) (cit. on p. 5).

Taferner, Philipp (2015). "Agile Entwicklung einer Anwendung zur Kontrolle und Dokumentation der Umsetzung von sicherheitstechnischen Maßnahmen aus dem Bereich des Arbeitnehmerschutzes für Bauarbeiten." Dissertation/Thesis. PhD thesis (cit. on p. 64).

The largest Git repo on the planet (May 24, 2017). Brian Harry's Blog. URL: https://devblogs.microsoft.com/bharry/the-largest-git-repo-on-the-planet/ (visited on 08/28/2019) (cit. on p. 9).

tmux usage of "goto" (2019). GitHub. URL: https://github.com/tmux/tmux (visited on 09/09/2019) (cit. on p. 5).

Type signature for Ruby classes. (Sept. 9, 2019). original-date: 2019-03-10T08:22:27Z. URL: https://github.com/ruby/ruby-signature (visited on 09/09/2019) (cit. on p. 62).

TypeScript - JavaScript that scales. (2019). URL: https://www.typescriptlang.org/ (visited on 09/04/2019) (cit. on p. 56).

Venners, Bill (Sept. 29, 2003). *The Philosophy of Ruby*. URL: https://www.artima.com/intv/rubyP.html (visited on 08/29/2019) (cit. on pp. 44, 65).

Verner, J. M. et al. (May 2012). "Systematic literature reviews in global software development: A tertiary study." In: *16th International Conference on Evaluation Assessment in Software Engineering (EASE 2012)*. 16th International Conference on Evaluation Assessment in Software Engineering (EASE 2012), pp. 2–11. DOI: 10.1049/ic.2012.0001 (cit. on p. 15).

Waterfall model (Aug. 30, 2019). In: *Wikipedia*. Page Version ID: 913171517. URL: https://en.wikipedia.org/w/index.php?title=Waterfall_model&oldid=913171517 (visited on 09/02/2019) (cit. on p. 36).

Weirich, Jim (Oct. 15, 2013). *October CincyRb - Jim Weirich on Decoupling from Rails*. URL: https://www.youtube.com/watch?v=tg5RFeSfBM4 (visited on 09/03/2019) (cit. on p. 44).

Xarawn (Nov. 4, 2015). *Lifecycle of the Test-Driven Development method*. URL: https://commons.wikimedia.org/wiki/File:TDD_Global_Lifecycle.png (visited on 09/02/2019) (cit. on p. 28).

xUnit (Aug. 7, 2019). In: *Wikipedia*. Page Version ID: 909756510. URL: https://en.wikipedia.org/w/index.php?title=XUnit&oldid=909756510 (visited on 08/28/2019) (cit. on p. 7).