Usage of Test-Driven Development in Open Source Projects
Abstract

The aim of this thesis is to provide an understanding of the use of test-driven development (TDD) in unit testing level in open source software (OSS) projects. TDD is a software development approach relying in short development cycles. Test-first approach, in which software tests are written first and then just enough code to pass them, is a central part of TDD. This way tests and production code are developed synchronously together supporting quality assurance processes in OSS projects which are sometimes reported to suffer from insufficient testing.

To get an understanding of the use of TDD in OSS projects, 2500 OSS projects were downloaded from GitHub and an automatic script was developed to analyse the projects and particularly test files which have one precise target file. These tests were divided into three categories depending on if the tests were committed before, with or after their target files.

It turned out that 74% (1822 out of 2500) projects contained at least one test file. 52% (1292) of all projects contained at least one test file with one precise target file and which was committed to the project’s repository either before or with its target file. 561 projects contained at least 10 tests with a single target and of which at least 50% were committed before or with their target files. These 561 projects are 22% from all projects and nearly one third (31%) of projects containing tests. Furthermore, it turned out that only half of all tests committed before or after their target files were committed by the same author who also committed the production code. Naturally tests committed with the production code were added by the same committer.

Based on the results it seems that a distinct minority of the downloaded projects are certainly following the test-first aspect of TDD in the sense that tests are added to the project’s repository before the production code, although it is possible that tests are made first and only committed later with the target file. Since majority of the tests were committed with the target file, it can be assumed that most of them were created at least closely together. Thus, the most important finding in this study is that there is evidence of the open source community's commitment to creating tests for their code.

Keywords
Open source software, software testing, test-driven development

Supervisor
Henrik Hedberg
Foreword

I want to take the opportunity to thank certain people who have helped me and made it possible to complete this thesis.

First and foremost I want to thank my supervisor Henrik Hedberg whose knowledge and patient guidance has helped me a lot through the entire time I have spent writing this thesis. I also want to thank my girlfriend who has also helped me by proofreading and giving new ideas and viewpoints.

In Oulu, spring 2015

Jari Hanhela

“On matka pitkä, voi käydä mitä vaan.”
## Contents

Abstract ............................................................................................................................ 2  
Foreword ........................................................................................................................... 3  
Contents ............................................................................................................................ 4  

1. Introduction .................................................................................................................. 6  
   1.1 Research question and method .................................................................................. 7  

2. Software testing and test-driven development ........................................................... 8  
   2.1 The concept of test-driven development ................................................................. 8  
   2.2 Advantages and disadvantages of test-driven development ...................................... 9  
   2.3 Previous research on the effectiveness of test-driven development ......................... 9  
   2.4 Conclusion .............................................................................................................. 10  

3. Open source software development ........................................................................... 11  
   3.1 Social structure ....................................................................................................... 11  
   3.2 Motivation .............................................................................................................. 12  
   3.3 Development tools .................................................................................................. 13  

4. Testing in open source software development ............................................................ 14  
   4.1 Popularity of testing in open source software projects ........................................... 14  
   4.2 Evolution of tests and production code ................................................................... 15  
   4.3 Software testing challenges in open source development ....................................... 15  
      4.3.1 Software requirements ....................................................................................... 16  
      4.3.2 Globally distributed development ................................................................... 16  
      4.3.3 Gaining user contribution ............................................................................... 17  
      4.3.4 Creating sufficient documentation .................................................................. 17  
      4.3.5 Insufficient resources ...................................................................................... 18  
   4.4 Advantages and disadvantages of Test-driven development in open source software development ................................................................................................................. 18  
      4.4.1 Popularity ........................................................................................................... 18  
      4.4.2 Evolution of tests and production code ............................................................ 19  
   4.5 Conclusion .............................................................................................................. 19  

5. Research method ........................................................................................................ 20  
   5.1 Mining software repositories ................................................................................... 20  
   5.2 Selecting open source projects ................................................................................ 21  
   5.3 Download process ................................................................................................... 22  
   5.4 Analysing the projects ............................................................................................ 23  
      5.4.1 Finding test files ................................................................................................ 23  
      5.4.2 Finding test targets and comparing commit dates .............................................. 24  
      5.4.3 Validity of the script .......................................................................................... 25  

6. Results ....................................................................................................................... 26  
   6.1 Background information about the downloaded projects ........................................ 26  
      6.1.1 Project age ......................................................................................................... 26  
      6.1.2 Software size ...................................................................................................... 26  
      6.1.3 Contributors ...................................................................................................... 27  
      6.1.4 Popularity .......................................................................................................... 28  
   6.2 Testing in the analysed projects .............................................................................. 28  
   6.3 Test-driven development in the analysed projects ................................................... 31  
      6.3.1 Analysed files ..................................................................................................... 31  
      6.3.2 Test-driven development in projects containing single-target tests .................. 33  
      6.3.3 Project size and the number of contributors affecting the usage of test-driven development ................................................................................................................. 34  
      6.3.4 Test-target pairs and their committers ............................................................... 36  

7. Discussion .................................................................................................................. 40  
   7.1 Number of test cases ............................................................................................... 40
7.2 Testing in popular projects ................................................................. 40
7.3 The use of test-driven development .................................................... 40
7.4 Test committers .................................................................................. 41
7.5 Limitations ......................................................................................... 41
7.6 Suggestions for future research .......................................................... 42
8. Conclusion ............................................................................................ 44
9. References ............................................................................................. 46
1. Introduction

Open source software (OSS) has gained significant attention in the previous decade and has been a viable alternative for closed software production for some time now. Many successful software products have been created using this development model, among the best known of them are operating system Linux1 and web server Apache2. These and many other OSS products are comparable to commercially made products regarding quality (Michlmayr, Hunt, & Probert, 2005b, p. 24; Wahyudin, Schatten, Winkler, & Biffl, 2007, p. 230). OSS has also gained increasing adoption rate in governments and many industries consider it as an alternative to proprietarily made software products (Wahyudin et al., 2007, p. 229).

Despite the popularity of OSS, the quality and development methods of OSS projects have been a matter of debate for a long time. One reason for this might be the fact that OSS projects tend to vary significantly in terms of used development practices, which also makes it a difficult subject to study (Michlmayr, 2005a, p. 27; Deshpande & Riehle, 2008, p. 273). Also, even though the OSS phenomenon is not new anymore, there still exists a lack of sufficient scientific evidence that would entirely justify the faith in high quality in OSS projects. This is one important reason to produce scientific research on OSS.

Even though company participation in OSS has increased in the previous decade, many OSS projects are still developed by volunteers. Developers are often located in different geographical sites and they communicate over the internet. Therefore, many OSS projects are examples of globally distributed software development.

The main obstacles in globally distributed software development are often seen to be difficulties in communication and software integration (Sangwan & Laplante, 2006, p. 1; Noll, Beecham, & Richardson, 2010, p. 1). Integrating components developed in different sites may also be challenging and may require additional focus in communication and coordination. Failing to integrate new components properly with the previous parts of the project can result in bugs and this is obviously an outcome that is desirable to avoid. Software testing plays an important role in achieving this: it is a good practice to create tests for each new component and run them locally before sharing the modifications with other developers. Test-driven development (TDD), which encourages developers to create tests before the production code, might help achieving this. Having tests created together with production code makes it easy to see if additions to the existing code broke the already tested functionality, and in fact “break the build”.

Despite focusing on testing, TDD as a software development practice can help creating a design for the software before the actual implementation is made. Thus, TDD can help software developers to spot possible defects in the code that test-last testing might not uncover (Sangwan & Laplante, 2006, p. 25). Since the open nature of OSS development usually gives everyone the possibility to contribute also in the software design phase, possible problems can be noticed well in advance.

---

1 https://www.linux.com

2 https://www.apache.org
Even though the advantages of TDD are widely acknowledged it seems to be somewhat unclear how much TDD is used in OSS projects. The results of a quick search of references to TDD in GitHub, a popular website for sharing OSS projects, consist mainly of personal learning projects. The search does not find many popular and well-established projects in which every contributor is asked to write tests for their code and strictly follow the rules of TDD. It is also unclear how its usage affects the project’s success. Thus, there seems to be a lack of scientific research in this area.

1.1 Research question and method

The purpose of this study is to get an understanding of the use of TDD in OSS projects. The scope of this thesis is to focus on the test-first aspect of TDD in unit testing level. The following research question has been set for this purpose:

RQ: How widely is test-driven development used in open source software projects in unit testing level?

As already discussed, there has been debate about whether the quality assurance practices in OSS projects are sufficient or not. The adoption rate of TDD can give hints of the developers’ commitment to the quality of each component before introducing it to a larger project. The findings may also provide guidance in answering other software quality related questions such as how much testing is used in OSS and what kind of correlations can be found between testing and other project characteristics.

Also, many OSS projects are developed across different sites and they face many problems related to globally distributed software development, such as integrating different parts of the software. There seems to be many advantages in using TDD in projects like this, for example helping the developers not to share broken code with each other. On the other hand, keeping test and production code in sync might be difficult especially if integration cycles are long. The results of this study should give an indication of whether such projects using TDD have been successful or not.

To answer the research question an empirical study was conducted, in which 2500 OSS projects were downloaded from GitHub and their test files were analysed using an automatic script. The purpose of this study is to quantitatively find signs on the use of TDD in OSS projects.

The structure of this thesis is as follows. Sections 2 and 3 give an introduction to TDD and OSS to get an understanding of both. Section 4 will look into earlier studies and challenges related to testing in OSS projects. Section 5 introduces the research method. Section 6 consists of the results of the empirical study. These results are discussed and effects of limitations are considered in Section 7. The conclusions that can be drawn from the results are reviewed in the final section of the study.
2. Software testing and test-driven development

This section begins with a brief introduction of the concept of software testing. After that test-driven development and previous research related to it is presented to get an understanding of how the effects of TDD have been studied and what kind of results the researchers have got.

Software quality is one of the most important factors when differences between similar software products are considered. Even though many developers and end-users can determine the differences between low and high quality software, there is no one universally accepted method to determine software quality and views may vary (Michlmayr et al., 2005b, p. 27). One of the most obvious ways, however, is to validate the software against its specified requirements. Software testing can obviously help in this process by providing evidence of the degree to which the requirements have been met.

IEEE Standard for Software Test Documentation (1983, p. 11) defines software testing as a process in which the purpose is to analyse a software item to detect the difference between existing system and its stated conditions. A later IEEE Standard for Software and System Test Documentation (2008, p. 3) defines it as a process in which the purpose is to validate that the software satisfied its requirements, solves the right problem and satisfies its intended use and user needs.

Even though many software projects allocate lots of time and resources in testing, large software programs are so complex that it is impossible to detect and eliminate all bugs. Even a large amount of test cases does not guarantee that software is free of defects (Kochhar, Bissyande, Lo, & Jiang, 2013). Furthermore, regardless of the amount of modern testing tools available today, many software programs still suffer from insufficient testing. Software testing cannot prove that the product behaves correctly in all different situations but it can remarkably reduce the issues the software might suffer from.

Software testing can be done on different levels, ranging from testing the output of a single function to testing the completely integrated system (Tosi & Tahir, 2013, p. 47). Numerous different testing methods exist in software engineering, but this study focuses mainly on unit testing, since it is a fundamental part of TDD.

2.1 The concept of test-driven development

A traditional approach in software development has been creating the working code first and writing unit tests afterwards (Gupta & Jalote, 2007, p. 285). This method is sometimes referred to Test-last development. In many traditional software development models, like the Waterfall model, software testing is one of the last actions to be performed before the software enters the maintenance phase. On the contrary in modern agile software development methods testing is often adopted as an integrated part of the whole development process. These methods help developers to find and fix bugs starting from the early phases of the development. In test-driven development, however, software tests are written before the actual production code (Beck, 2002, p. 1).

As the name suggests, TDD is a software development approach in which software code and its tests are written together so that tests are always written first and then just enough code to pass them (Gupta & Jalote, 2007, p. 285; Beck, 2002, p. 1). This way tests and
production code are developed synchronously together and all code gets tested at least in some form. The basic process of TDD is the following as introduced by the creator of TDD Kent Beck (2002, p. 1):

1. Add a test
2. Run all the tests and see the new one fail
3. Write the minimum amount of code to pass the test
4. Run all tests and see them all pass
5. Refactor code to remove possible duplication

The development process is then continued by repeating the steps mentioned above.

2.2 Advantages and disadvantages of test-driven development

Perhaps the most obvious advantage of TDD is that it makes it more likely that there are tests for each part of the software, meaning that more defects are caught than with tests written using test-last approach. Other advantages are more comprehensive code coverage and better software design, since with TDD the software is “designed for testability”. If the code seems to be too difficult to test with TDD, it indicates that there is a room for improvement in terms of simplicity and testability. Simpler and more testable code often tends to be faster and more maintainable because it tends to be easier to understand for developers who have not written it. Other often mentioned advantages are better developer productivity, which often occurs as higher amount of written tests, and higher production code quality (George & Williams, 2003, p. 1136; Ciglarič & Matjaž, 2011, p. 558.)

There are also some disadvantages in the TDD method. One of these is a possible difficulty of applying TDD in certain domains, such as highly distributed software and event-based user interfaces. It is suggested that tests written using TDD are not always sufficient enough to catch all possible defects especially when the bugs are related to the external dependencies used in real processing environment (Hunter, 2002). This is further compounded if the tests are written by the same developer who writes the testable code in which case the developer may not know all defects in the code and therefore does not write tests for them.

Even though the process of using TDD introduced above does not mention it, there often also exists the need of keeping the tests up to date with production code when the software evolves further. Maintaining tests continuously with the production code requires time and if the quality standards set for production code are not used in test code, maintenance of the tests can become difficult. As Martin (2008, p. 154) has said, having dirty tests is equivalent to, if not worse than, having no tests at all.

2.3 Previous research on the effectiveness of test-driven development

There have been some experiments evaluating the efficiency of TDD in limited scope but it seems that comprehensive studies in large scale projects have not been investigated yet (Gupta & Jalote, 2007, p. 285). One typical empirical research approach of the effectiveness of TDD has often been letting two groups develop the same or similar small or midsize software program and comparing the results. The experimental group uses TDD while the control group uses traditional approach where tests are written last.
Erdogmus, Morisio and Torchiano (2005, p. 236) found that the group that wrote the tests first created more tests and were more productive. Gupta and Jalote’s (2007) study showed that the code quality achieved by the TDD group was considerably better and the development effort was smaller than achieved by the group using a traditional development method. Similar positive results were also found in other experiments (for example Bhat & Nagappan, 2006; Janzen & Saiedian, 2006; Williams, Maximilien, & Vouk, 2003; George & Williams, 2003; Hagner & Muller, 2002).

On the other hand, the investigations of Muller and Hangner (2002, p. 135) showed that Test-driven approach neither made the implementation of the program faster nor more reliable but it seemed to support better program understanding. Other examples of studies that also do not show increased productivity in projects using TDD are for example Ciglarič & Matjaž, 2011; Janzen & Saiedian, 2006; Erdogmus, Morisio, & Torchiano, 2005.

A review by Shull, Melnik, Turhan, Layman, Diep and Erdogmus (2010) shows that majority (13 out of 21) of the analysed studies found TDD to be beneficial in terms of external code quality and only in two studies using TDD resulted in lower quality. The effects of TDD on productivity were mixed between different types of studies and TDD did not show consistent effect on internal code quality. Similarly, a review by Rafique and Mišić (2013), consisting of 27 studies on TDD, shows that TDD had a small positive effect on the quality but little effect on the productivity.

It is notable that the results of different experiments cannot be directly compared to each other; for example some studies measure code quality as the number of passed tests while others focused on code readability and complexity. Nevertheless, all of these can be said to be ways to measure code quality.

2.4 Conclusion

As a summary it can be said that the effectiveness of TDD is not fully proven. The results in many studies have been mixed; mostly the use of TDD has given benefits in terms of code quality or productivity, sometimes it has had no effect or has it introduced negative effects, which are usually related to increased development time. Also, there seems to be a lack of empirical research on TDD in large software projects.
3. Open source software development

Since this study focuses on the usage of TDD in open source software (OSS) projects, this section aims to get an understanding of the special characteristics of OSS development and communities. OSS development is a complex socio-technical phenomenon which is not easy to study. It is also a relatively new field in research and thus not all results presented here are based on comprehensively covered scientific studies, but assumptions based on empirical observations are also used.

The term open source is not quite unambiguous, i.e. there are many definitions for the concept of open source software (Khanjani & Sulaiman, 2011, p. 548). Open Source Initiative (OSI), which is one of the best known global non-profit organisations promoting OSS, defined open source at its simplest as software that “can be freely used, changed, and shared (in modified or unmodified form) by anyone” (The Open Source Initiative, 2014). In practice, this simple definition may be problematic, because even if a program is published freely and it is stated to be open source, questions of what the code may be used for and how does copyright limit its use. One solution to this is to require OSS productions to be published under OSI certified software license (OSS Watch, 2005; The Open Source Initiative, 2014). To gain this license the software must comply with OSI's definition of OSS and must meet their criteria for code distribution and derived work. OSI has certified many popular open source licenses, such as GNU General Public License (GPL) and BSD 2-Clause "FreeBSD" license.

The terms closed source software and proprietary software, on the contrary, refer to a software for which the source code availability is limited. The software and its source code is owned by an individual person or a company which has set major restrictions on how the software can be used.

In this thesis, OSS is defined simply as software for which the source code has been released to the general public and is free for anyone to study and modify.

3.1 Social structure

OSS is often developed by voluntarily participating programmers who write source code often in geographically distributed environment and who communicate over the Internet (Otte, Moreton, & Knoell, 2008, p. 1247; Hars & Ou, 2001, p. 1). In addition to these volunteers many companies are also promoting and developing a number of important OSS projects these days (Wahyudin et al., 2007, p. 230).

The social structure of typical OSS projects is composed of two major groups: the developer community and the user community. The difference is not clear, however, since developers of OSS are often also users of it (Wahyudin et al., 2007, p. 230; Khanjani & Sulaiman, 2011, p. 550). Compared to proprietary software development it is notable that even though a hierarchical structure can be seen, it is not often very strict and the project leader cannot forcefully assign tasks to individual developers.
Figure 1. Onion model of OSS teams (Crowston & Howison, 2005)

The social structure of OSS development teams is often described using so-called Onion model (Figure 1). Many variations of this model are used, but the one presented in Figure 1 seems to be one of the best known. At the center of the onion are core developers who make the most important decisions regarding the project. Usually the core group is small compared to the whole development team. Active users are users who have a special interest in the project and usually contribute by giving feedback and reporting bugs. Passive users, on the other hand, just use the program and are not a very visible part of the development community. (Crowston, Wei, Li, & Howison, 2006; Crowston, Annabi, Howison, & Masango, 2005.)

The onion-model is not reported to be scientifically proven but empirical observations seem to support this overview of OSS development teams. However, as mentioned previously, OSS participants can be seen to have multiple roles and they can adopt different roles at different times in the development. Thus it might be hard to place some participants strictly in one layer of the onion.

Aberdour (2007, p. 59) states that building a large and active community around an OSS project is the key factor to success. The members of the community should participate in the project in many different ways, such as developing new code, testing it and giving feedback. This should not be very difficult, since it is well known that user participation is very high in OSS projects (Zhao & Elbaum, 2002, p. 72; Wahyudin et al., 2007, p. 230). In fact, user participation is seen to be the backbone and one reason for high quality in many OSS projects (Zhao & Elbaum, 2002, p. 70; Otte et al, 2008, p. 1250; Aberdour, 2007, p. 59).

3.2 Motivation

One point of interest is why developers create code. In proprietary software development, pay, benefits, recognition and opportunities are often seen the main motivators for developing software (Baddoo, Hall, & Jagielska, 2006, p. 219). However, OSS developers sometimes have different motivations to work in OSS projects. One of the most common questions about OSS development is why software developers want to participate in such projects, sometimes even without pay. A number of surveys has been conducted to answer this question.

OSS developers have different motivations to work on OSS projects, as shown by Otte, Moreton and Knoell (2008, p. 1249). In their paper the most common reasons were
personal needs, community needs and company needs. The latter seems to increase with project growth. A survey-based study by Zhao and Elbaum (2002, p. 72) also indicates that majority (60%) of OSS projects included in their study are started to meet developers’ personal needs. Khanjani and Suleiman (2011, p. 550) suggest that programmers are mainly motivated by an opportunity to be recognized and the possibility to improve OSS. Aberdour (2007, p. 60) also got similar findings. He stated that programmers do not contribute in OSS projects selflessly but they want to achieve certain status and recognition in the community. Learning new things seems to be also an important reason participating in OSS projects (Ye & Kishida, 2003, p. 419). Finally, many OSS developers believe in freedom and free software, which is one reason for voluntary cooperation (Elliott, 2003).

3.3 Development tools

OSS development is largely based on the usage of different software tools that support the OSS development methodology. For example, version management and issue management tools are very common in OSS projects (Otte et al., 2008, p. 1250; Robbins, 2007, p. 246). Tools like these are not used only to support the different activities of software engineering but can also be seen to support the culture of OSS by keeping the whole development process and communication available for everyone to see.

It is notable that many OSS developers prefer to use tools that are OSS products themselves (Elliott & Scacchi, 2003, p. 27). The reasons for this can be described with personal preferences, but it also seems to be a part of building the community. Otte, Moreton and Knoell (2008, p. 1251) suggest that tools that are OSS themselves should lower the barriers for other community members to use them.

Regarding software testing, the usage of OSS only testing tools could have a positive effect on testing since such tools are free to use for everyone. It could be assumed that OSS community pays attention to software testing since there is a wide range of software testing tools and frameworks available that are OSS products themselves, such as JUnit3. On the other hand some studies suggest that the general usage of testing tools is not very high in OSS projects (Otte et al., 2008, p. 1250; Zhao & Elbaum, 2000, p. 56).

There are also possible downsides in sticking only to OSS tools. It restricts the freedom to choose the preferred tools in individual level. Furthermore there are important characteristics of software engineering in which OSS tools seem to be lacking. Examples of these are requirements management, project management and test suite design tools (Robbins, 2007, p. 258)

3 http://junit.org
4. Testing in open source software development

To get an idea of how much TDD is used in OSS projects it is first important to understand how widely testing is done in OSS projects in general level and what things might have an effect on the amount of testing. These things are discussed in this section.

4.1 Popularity of testing in open source software projects

These days there are a number of popular web sites focusing on spreading OSS, such as GitHub\(^4\) and SourceForge\(^5\). These sites have been the research subject to a few quantitative studies, in which data is mined from these repositories to get an understanding of the popularity of testing activities in OSS projects. In this section the main findings of these studies are introduced.

Tosi and Tahir (2013) studied the popularity of testing activities in OSS projects. They selected 33 well-established OSS projects and wanted to find out how they address software testing. They found that slightly more than half of the selected projects (58%) contained testing activities. According to this study it also seems that majority of the projects did not contain a well-planned testing strategy and approach. Tosi and Tahir’s conclusion is that OSS projects are not tested and validated enough. Their results do not reveal a clear reason for this, but they assume that a wider adoption of testing tools can support the whole testing process, starting from test planning to testing the complete system and reporting the results. (Tosi & Tahir, 2013)

Similar research method was also used by Kochhar, Bissyande, Lo and Jiang (2013) in their empirical study on adoption of software testing in OSS projects. Their study included 20,000 OSS projects from GitHub. The purpose was to find out how many projects contained test cases and what things affected the amount of testing. Since it would be practically impossible to find and check test cases of each project manually, the authors analysed all files which contained the word “test” in their name, indicating that the file contains tests. According to the results of the study, relatively high number of projects (39%) did not contain a single test case. It should be noted that this means the results were similar to Tosi and Tahir’s (2013) as 61% of projects did contain test cases. Large projects, contained more test cases than smaller projects. (Kochhar et al., 2013)

Kochhar et al. (2013a) were also interested in finding if there is a connection between the number of developers and amount of test cases. They found a weak positive correlation, meaning that the number of test cases increases slightly when there are more developers in the project. Interestingly, while the amount of developers increases, the number of test cases per developer decreases. Similar results were also found in a survey study by Otte, Moreton and Knoell (2008), in which when coding effort increases, it happens with expense of testing effort. Kochhar et al. did not give an explanations for this but they suggested that not all developers wrote test cases.

Kochhar et al. (2013a) also examined the relationship between popular programming languages and the number of test cases in the project written in that language. They found

\(^4\) https://github.com

\(^5\) http://sourceforge.net
that popular programming languages had higher numbers of test cases than projects written in other languages. Zhao and Elbaum (2000) had similar results from their earlier study.

In another similar study, performed also by Kochhar et al. (2013b), additional 50,000 GitHub projects were analysed in much the same way as in the previously mentioned study. The results in these studies were in line with each other, but this study added the finding that while the number of tests increases in large projects, the number of tests per code line decreases. Furthermore Kochhar et al. (2013b) found that over 90% of the selected projects had less than 100 test cases in the project repository. Less than 3% contained more than 500 tests.

Some limitations can be found in the study by Kochhar et al. The test file investigation might have given false results, since it is not certain that all test files contain the word test, nor that all files containing the word are actually test files. Furthermore the test cases used in the project may not have been shared to the public. Kochhar et al. (2013a) did not filter out projects in any way in their first repository examination study. They believed the dataset did not contain many “toy projects” because it contained famous projects, such as jQuery and Ruby on Rails. However, this does not necessarily mean that no irrelevant projects, such as those created only for personal learning purposes, were included in the data set. In their later study, though, small projects were excluded (Kochhar et al., 2013b).

The quantitative studies presented above give some idea of the amount of testing effort in OSS projects. All studies had roughly the same conclusion that many OSS projects do not contain a large amount of test cases.

4.2 Evolution of tests and production code

Since publicly available OSS repositories typically store most of the development activities and code changes made by the authors of the project, code modifications have been a popular research target in mining data from these repositories. One of these studies, made by Zaidman, van Rompaey, Demeyer and van Deursen (2008), focused on studying the co-evolution of production and test code in two open source projects: CheckStyle and ArgoUML. Their aim was to create awareness with developers and managers about the testing process that was followed in the project. The authors also mention the importance of maintaining production code and test code synchronously to make sure that all new functionality gets tested and that tests are kept up to date with the production code.

The results of the study indicate that a large part of the production code and related test code was changed together in same commits. Both of them also seemed to be growing together, indicating synchronous co-evolution. According to the authors, the finding can be seen as an indication of the usage of TDD as production and tests are development synchronously together. (Zaidman, van Rompaey, Demeyer, & van Deursen, 2008.)

4.3 Software testing challenges in open source development

Numerous challenges exist in adopting testing methods in OSS projects. Most of these are the same as in any software development project, such as what needs to be tested and how much testing needs to be done to ensure high quality software. However, there are also many challenges that are explicitly related to OSS development, like how the requirements are defined and who is responsible for testing certain parts of code in voluntary based projects. These things also have an effect on the decision of using TDD
in the project or not. Getting an understanding of the typical testing challenges is thus important.

4.3.1 Software requirements

As mentioned previously, one goal of software testing is to ensure that the software meets the requirements set for it. Usually these requirements are stated in the requirements analysis and management phase of software development according to the wishes of the project’s stakeholders. However, in OSS development the requirements elicitation phase can be far from traditional textbook example (Scacchi, 2002, p. 24).

Based on his empirical findings, Scacchi (2002, p. 24) suggests that OSS projects do not elicit software requirements in traditional way. Instead the requirements are typically asserted based on discussion inside the community. These findings are supported by Wahyudin et al. (2007, p. 231). They stated that every individual in OSS project’s community can be considered a stakeholder. A case study on OSS project GNOME showed that its stakeholders were the developers who also acted as users, coders, testers and documenters (German, 2003).

Based on these findings, it seems that the systematic requirements elicitation mentioned in text-books is often missing in OSS projects and stakeholders cannot always be easily defined. Every stakeholder can also have different and undocumented assumptions of the software which makes it difficult to decide what things need to be verified through software testing.

4.3.2 Globally distributed development

OSS projects are typically distributed in a way that developers are located in different places around the world and may never meet each other (Michlmayr et al., 2005b, p. 24). This is often seen as a challenge in software development due to the difficulties related to communication and physical meetings. Software testing and continuous integration are also seen as challenges in globally distributed software development (Sangwan & Laplante, 2006, p. 3).

In a single-site software projects developers usually rely on face-to-face communication regarding the development process. When the development environment becomes distributed, such communication is more difficult. Thus, communication must rely mainly on emails, instant messaging and video conferences. There might be only a few, if not any, face-to-face meetings on a frequent basis. The communication is further complicated by the differences in time-zones, cultures and the way how people communicate.

Continuous integration is often seen an as good practice in software development to make the integration process easier, especially in case when the code is distributed in different sites. OSS projects are often mentioned following this practice but on the other hand there does not seem to be enough scientific research to justify it. For example a study conducted by Deshpande and Dirk (2008, p. 5) did not find evidence to validate the belief that OSS developers are practicing continuous integration.

It looks like global distribution does not directly make software testing more challenging. The difficulties arise mainly from other problems, such as difficulties in communication and code integration. Still it is interesting to note that many OSS projects seem to successfully overcome the challenges related to distributed software development, such as GNOME (German, 2003).
4.3.3 Gaining user contribution

Large user base and the amount of possible contributors is often seen as an advantage in OSS development model (Khanjani & Sulaiman, 2011, p. 548). For those willing to participate in OSS projects there are many ways to contribute. One can act simply as an end user and give the developers feedback from that perspective, or participate in the actual development process by programming, peer reviewing and testing other people’s code.

Many difficulties can still be seen in user contribution to testing. It seems that many possible contributors are only interested in developing new code and only few of them are interested in helping testing the already written code (Michlmayr et al., 2005b, p. 27). Thus, it seems that even though OSS projects are supposed to take advantage of the large user base to test and validate the software, it is not guaranteed that this will happen. There is also a risk that trusting the community to test and validate the software reduces the developer’s interest in utilizing testing techniques and tools (Zhao & Elbaum, 2002, p. 73).

Trusting the user community to test software is a risk if majority of users have few technical skills. It is not guaranteed that these users will actually give useful feedback or complete bug reports. Sometimes they might also send duplicated bug reports, which can be time-consuming. (Farooq & Quadri, 2012, p. 129.)

4.3.4 Creating sufficient documentation

Comprehensive documentation is often seen as a mark of good software quality. It indicates that the developers are interested in producing consistent code and have set defined guidelines for contributing in the project. For example Mockus’s (2002, p. 334) case study on Mozilla project showed that it had problems attracting contributors. The number of participants increased only after the core team had redefined development tools and processes and also improved the documentation.

According to Michlmayr, Hunt, and Probert (2005b, p. 26), contributing in OSS projects often requires following specific procedures and guidelines in order to join the project. From the testing point of view the documentation should help the possible participants to understand what guidelines have been set for testing in a specific OSS project. According to Tosi and Tahir (2013, p. 47) the documentation itself should contain testing-related things like test specifications, test plans and test results reports. One of the specifications could be a requirement of writing tests for all new code commits.

The problem regarding documentation appears to be that comprehensive documentation is often not available in OSS projects, which has been showed in many studies (Zhao & Elbaum, 2002, p. 73; Tosi & Tahir, 2013, p. 47; Michlmayr et al., 2005a, p. 26). Nevertheless, many large projects seem to contain comprehensive documentation (Michlmayr et al., 2005, p. 26).

Michlmayr, Hunt and Probert (2005b, p. 27) listed the lack of documentation as one significant quality problems in OSS development. This problem is seen to decrease the motivation of new participants who would like to contribute in OSS projects for example by testing its functionality. It may be difficult for them to contribute in a project if they have difficulties to understand it. The lack of documentation may also indicate that there is no assurance that every contributor follows the same guidelines in testing.
4.3.5 Insufficient resources

In proprietary software development the amount of available resources, such as available funding and budget, are often seen to directly be related to the amount and quality of software testing that is possible to achieve. Funding is one of the most significant resources that is often seen as an advantage in proprietary software development and which often seems to be lacking in OSS projects.

OSS projects are funded in different ways and some are not funded at all. According to Kamp (2014), the lack of available funding is a significant problem in many OSS projects. The consequences can be extensive especially in security-related applications. One well-known example of this is OpenSLL Heartbleed security hole, which Kamp (2014) says to be originated from insufficient funding of the project.

The availability of source code often encourages people to create modifications and variants of the software. However, as Gokhale et al. (2006, p. 95) mention, this makes software testing more complicated due to the increase in the amount of configuration options that needs to be tested. The large user base can help testing the different configurations, but since OSS projects often run on a tight budget, a comprehensive testing may be a challenge for the project.

Michlmayr, Hunt & Probert (2005b, p. 25) conducted an interview with seven OSS developers. According to them, even though OSS has an advantage in the number of possible contributors, there is still a lack of resources and infrastructure compared to proprietary source software development. This might have an effect on testing in a negative way. Aberdour (2007, p. 62) states that formal testing techniques and test automation are expensive and not all OSS project can achieve them. Still, as mentioned before, many OSS developers tend to prefer testing tools that are OSS themselves, which could indicate that expensive testing tools are not mainly favoured by OSS developers.

Based on the findings, lack of money and resources can be seen as a factor of reducing the amount of quality assurance in OSS projects and thus having a negative effect on the amount of testing. However, the intensity of the effect does not seem to be completely proven.

4.4 Advantages and disadvantages of Test-driven development in open source software development

As mentioned in previous sections, there are many challenges in adopting software testing in OSS projects. This section describes the ways how TDD could possibly help OSS projects to overcome some of these challenges and help the development team to achieve high quality of testing activities in the project. The possible disadvantages of TDD are also discussed.

Inefficient software design can easily lead to inefficient code and bad quality (Khanjani & Sulaiman, 2011, p. 550). In proprietary software development the software design and code are usually kept private. OSS development has an advantage here because it can permit anyone to view and review the software design. Therefore the project team can get feedback from the design before they even start implementing it.

One of the mentioned challenges in testing OSS products was defining software requirements. TDD could help OSS project in software design process. Sengupta, Sinha and Chandra (2004, p. 2) believe that early availability of test suites can act as a
documentation of requirements, perhaps the most precise form of them, and increases the understanding of what a specific part of the program is supposed to do and how it behaves in unexpected situation. This helps the developers on different sites to keep track of the changing software requirements. This can also be seen as an indirect way helping developers communicate with each other since tests help them to understand what a software is supposed to do.

There seem to be dissenting opinions about the usefulness of TDD along with continuous integration, which is often seen one of the most important software development practices in globally distributed software development. According to Clerc, Lago and Vliet (2007, p. 232), TDD serves as a good basis in continuous integration helping the team to run the tests automatically and get immediate response. On the other hand Sangwan and Laplante (2006, p. 1) mention that TDD focuses more on low-level unit testing rather than integration and system-level testing and thus is not very useful in continuous integration process.

TDD has some clear advantages in globally distributed OSS projects and should fit the culture of OSS development. However, TDD is definitely not an all-in-one solution for all the mentioned testing-related challenges mentioned previously and it is not clear whether these benefits are bigger than possible disadvantages. As mentioned earlier, TDD has often been studied in relatively small experiments and there seems to be a lack of scientific research in large globally distributed software projects.

4.5 Conclusion

Many previous studies related to testing and quality assurance processes in OSS products conclude that OSS projects often suffer from insufficient testing. Many software testing difficulties could indeed be found in OSS development, such as defining the requirements for which the software is tested against and creating sufficient documentation to support testing. Insufficient resources can also be seen as a factor which makes testing more difficult. The use of TDD is unlikely to solve all testing related challenges but it could make the software testing easier in OSS projects. However, the lack of empirical research in this area leaves the possible benefits of TDD invalidated.
5. Research method

Case studies and surveys have traditionally been popular research methods in software engineering for a long time. Today, the existence of the internet, OSS and modern version management tools have made it possible to mine data from publicly available software repositories (Hassan, 2008, p. 49). This method is called data mining, and in the case of mining data from software project repositories, the process is better known as Mining Software Repositories (MSR).

Using MSR allowed using a rich source material from projects downloaded from GitHub. These projects were analysed using an automated script to get an understanding of the use of TDD in OSS projects. The following sections will introduce MSR and the research method used in this study in higher detail.

5.1 Mining software repositories

Software project repositories are seen as a valuable source of data (Hassan, 2008, p. 48; Anbalagan & Vouk, 2009, p. 171; Gerlec, Krajnc, Heričko, & Božnik, 2011, p. 1). The primary goal of software repositories is to store the software’s source code and its modification history (Gerlec et al., 2011, p. 1). The data can used to conduct research for example on software evolution through code modifications history and studying different development methods. However, the method is not used only by software engineering researchers but it can be a valuable tool for software engineers themselves too. For example, software engineers can use data mining algorithms for bug hunting and finding defects from the code (Xie, Thummalapenta, Lo, & Liu, 2009, p. 56).

As a data collection method MSR has increased in popularity in recent years and it is considered as an important area of research (Chaturvedi, Singh, & Singh, 2013, p. 89; Gerlec et al., 2011, p. 1; Hassan, 2008, p. 49). There is also an international workshop on MSR in International Conference on Software Engineering (ICSE) to bring practitioners interested in MSR together (Hassan, 2008, p. 49).

According to Xie et al. (2009, p. 56) a typical MSR method consists of the following five steps:

1. Collect / investigate software engineering data
2. Determine software engineering task
3. Preprocess data
4. Adapt / develop data mining algorithm
5. Post-process / apply mining results

The research typically starts with the mixture of the first two steps: knowing which software engineering data to collect and which task to assist. In step three the data is then preprocessed, i.e. relevant data is extracted from raw software engineering data for further analysis. Then in step four the necessary data mining tools is adapted or developed. The final step consists of post-processing the results.

In this thesis the steps presented above were followed with one exception: once the data was downloaded, it was not preprocessed in any way but instead the data mining algorithm was programmed to automatically extract and analyse the relevant data for analysis.
Since most of the data related to the development process of OSS projects is available for researchers in public software repositories, it makes it possible to study these projects using data mining algorithms and applying them to a very large number of projects with a small amount of effort. Nevertheless, the large amount of available data does not mean that the relevant data is easy to extract and analyse. This is further complicated by the fact that many software repository systems were not designed data mining in mind (Hassan, 2008, p. 52).

According to Jensen and Scacchi (2006, p. 457), extracting individual development practices from observed OSS data can be difficult and is not likely to give completely accurate results. Another often mentioned challenge in MSR seems to be either finding a proper tool or developing an algorithm for a specific data mining task (Xie et al., 2009, p. 57; Hassan, 2008, p. 53).

5.2 Selecting open source projects

GitHub is one of the most popular platforms for hosting and sharing OSS code. In 2013 the site reported having over 10 million hosted repositories (Doll, 2014). As of 2014, the site stated having over 3.4 million users (Whitaker, 2014), making it the largest OSS community in the world. Many popular OSS projects have been hosted on GitHub, such as Bootstrap\(^6\), jQuery\(^7\) and Ruby on Rails\(^8\). GitHub also provides an API for searching and downloading large number of projects relatively easily and also useful tools exist for analysing Git projects in data mining. These were the main reasons for choosing GitHub as the target site for downloading OSS projects.

Even though numerous popular OSS projects have been hosted on GitHub, it also contains many small and personal projects created mainly for learning purposes. A study by Blincoe, Gousios and Kalliamvakou (2014) shows that majority of projects hosted in GitHub are created mainly for personal use and are inactive. Even though the authors believe that there is a need to automatically analyse GitHub projects, they also warn that this is one of the most important questions to consider when doing data-mining research on GitHub.

In practice it would be difficult to exclude all irrelevant projects from the dataset but at least the amount of them can be decreased by setting certain requirements for the downloadable project. To decrease the number of possible irrelevant projects, in this study the following requirements regarding age, popularity, number of contributors, programming language and size were set for the projects to be downloaded.

*Age*

This study aims to focus on well-established projects with moderately long history and so only projects that were created before 1st of December 2013 were included in this study. Since the projects were downloaded between December of 2014 and January of 2015, only projects that were at least one year old were included. The age limitation was chosen so that the downloaded projects were less likely to be in their early phases and

\(^6\) [http://getbootstrap.com](http://getbootstrap.com)

\(^7\) [https://jquery.com](https://jquery.com)

\(^8\) [http://rubyonrails.org](http://rubyonrails.org)
that the contributors had already spent some time working in the project, possibly facing some of the problems in globally distributed software development mentioned in Section 4.3.

**Popularity**

There are many ways to measure the popularity of OSS projects. In GitHub users can give a “star” to repositories and the number of stars can be seen on the repository page. According to GitHub, the purpose of stars is to help users bookmark repositories and it gives other users an approximation of the level of interest shown to the project. Thus, the number of stars was chosen as the measure of popularity. Projects included in this study had at least one star in GitHub, indicating it has been noticed in the OSS community. It should be noted, however, that developers can also give a star to their own project in GitHub.

**Number of contributors**

In GitHub, contributors of a repository are all users who have contributed to project. Only projects that had at least two contributors were included in this study. The aim of this limitation is to exclude majority of learning projects which Kochhar et al. (2013a) called “toy projects” in their study.

**Programming language**

Only projects using Java as their main programming language were included. The reason for choosing to study only Java projects was that it is one of the world’s most popular programming language. Also in GitHub Java was reported to be the second most popular programming language in 2014 (Bard, 2014). The huge popularity means that there are plenty of testing tools available for Java and many of them are OSS. According to Robbins (2007, p. 257), JUnit, a popular unit testing framework for Java, has been widely adopted in OSS projects. An empirical analysis made by Weiss (2013) indicated that testing plays an important role in Java projects in GitHub, which gives a good dataset to analyse the forms of testing in these projects. Also Java projects often have one-to-one relationship between files and classes which makes Java projects relatively easy to study and analyse resulting in less errors in automatical analysis.

**Size**

In order to exclude extremely small projects the size of the project was taken into account. Projects size was measured in lines of code (LOC) and projects containing less than 500 were excluded. Very small projects do not possibly benefit of the use of TDD very much and thus were not taken into account. Only files with the file extension .java were taken into account in counting LOC. Code comments and blank lines were included in LOC.

### 5.3 Download process

The goal of this research is to get a good representation of OSS projects, thus it is desirable to get randomly chosen projects because analysing all existing projects is practically impossible in the scope of this study. However, GitHub does not provide a way to list projects in a truly random order so the results were sorted by last update date. This made the dataset reasonably random considering the characteristics of the projects and assured that only active projects were listed. Since GitHub limits the results to 1000 per search, the list of results was saved and a new list created after minimum wait of 10 minutes. The
saved lists of repositories were then used to download the projects. It was ensured that a single project was downloaded only once and failed downloads were not included in the dataset.

GitHub API provides a way to get a list of repositories matching a certain inclusion criteria, but it does not provide search parameters for all inclusion criteria used in this research. The missing parameters were the number of contributors and project size in LOC. Thus, projects not having at least two contributors or 500 LOC were downloaded but they were later deleted manually.

Many software projects take advantage of using different development branches. According to GitHub, downloaded projects’ state is by default the latest version of the “default branch”. Project developers are able to change the default branch, but by default it is called “master”, which usually contains the latest stable version of the project. This was fine for the analysis and thus the projects were analysed in the state they were after downloading.

### 5.4 Analysing the projects

To analyse the projects for the usage of TDD an automated script was developed in Java using JGit software library. The script was used to find test files and compare their commit date with the commit date of the test target file. All downloaded projects were analysed separately. Figure 2 shows the activity diagram of the script. In the following sections the execution process of the script is introduced in more detail.

5.4.1 Finding test files

As shown in Figure 2 the script takes a GitHub project as an input and analyses it. First, the script searches through all code files to find files that contain test cases. In Java it is considered a good practice to name test files in a way that the file name starts with the
name of the class being tested and ends with the word “Test”. Similar connection between production and test code files that is based on file naming conventions was also used by Demeyer et al. (2008). However, it is possible that some test files were named differently. To include as many test files as possible, a random sample of 50 downloaded projects were analysed manually to find typical test file names. Based on these findings the script was configured to include files with names ending with Test.java, Tests.java, TestCase.java, TestCases.java or Tester.java. All names were case-sensitive. Once all test files were found, each one of them was analysed separately.

5.4.2 Finding test targets and comparing commit dates

In Java a widely accepted development practice is that every class is written in its own file (excluding inner-classes), which means that there should be a one-to-one relationship between files and classes. Regarding the development practice of TDD, it is assumed that for every class and its file there exists a test file, which contains methods to test the behaviour of the written class. If the development practices of TDD are followed, the test file should have been written first. One way, which is also used in this study, is to resolve this is by comparing the commit dates of the test file and its target file.

As mentioned earlier, it is a good principle to name test files in a way that the file name starts with the target class name. The script uses this assumption to find the test target file, which contains the target class. The script ensures that the testable class is also used in the test file; the class is either initialized or a static method is called. The script does not, however, check that the usage of the class is not a comment but it should have hardly any effect on the results.

If the script finds a single target file for the test, it compares the dates when the test file and its target file were committed to the project for the first time. The test file will be called a single-target test file and analysed test file and its target are called a test-target pair. If a single target file for the test is not found, however, the test is classed as other test. Tests like this are often smoke tests or regression tests, which are out of the scope of this thesis regarding test-driven development.

The script divides the tests-target pairs into following categories:

- Tests committed before target file (test-first)
- Tests committed with target file (test-with)
- Tests committed after target file (test-after)
- Other tests

It is notable that if the test file was committed with the target file, it is not possible to be certain which one was actually created first. In this case, it might be that the target file was written before the test. Nevertheless if the test and its target file were committed at the same time, it possibly indicates that testing is considered to be important part of software development, even though TDD might not have been strictly complied with.

The script also compared the author or committer between the test file and its target file. The script used the name and email address attached to the committer information when comparing the committers. For each analysed test-target pair the script included information on how many of them were committed by the same author.
5.4.3 Validity of the script

Before the large sample of projects was analysed using the script, it was tested multiple times with a smaller number of projects. To decrease the amount of possible defects and wrong analyses, random samples of the analysis done by the script were repeated manually to see if the script produces intended results. The script was improved multiple times based on these findings and the final form of the script was then used to analyse all the projects selected for the study. Still, the script contains some limitations, which are discussed in Section 7.5.
6. Results

This section presents the results of the research. Section 6.1 first presents some background information about the downloaded projects, after which the results of the automatical analysis of the project files is examined. The discussion and comparison to previous studies is done in Section 7.

6.1 Background information about the downloaded projects

All in all 2500 projects were downloaded and analysed. Since GitHub search was unable to list all the projects meeting the inclusion criteria used in this research, it is unclear what percentage of them was downloaded. A quick search of Java projects rated with at least one star, created before 1st of December 2013 and modified after 1st of September 2014 returned approximately 12,000 projects. Considering that not all of these projects were more than 500 LOC and had at least 2 contributors, it could be assumed that the sample taken for this study was sufficient.

6.1.1 Project age

Figure 3 shows that majority of the projects (73%) were created in 2012 and 2013. Thus it can be said that most of the projects were relatively new, but there were also many projects that have been in development for multiple years. One reason for the majority of projects being created 2012 or later might be that GitHub has evenly increased in popularity as a hosting platform of choice. Another reason could be the fact that the download process was focusing on projects that were updated recently, possibly taking more unfinished projects. It must be noted that reliable analysis on whether the use of TDD has increased or decreased in time cannot be made for this reason.

![Figure 3. Projects by creation year.](image)

6.1.2 Software size

The software size of the projects varied a lot. The smallest downloaded project contained 501 LOC and the largest 4 245 552 LOC. On the average a project was 78 757 LOC in
size. Table 1 gives an overview of the sizes of the projects. The projects are divided into five different categories according to their size. These categories are also used later in the result analysis. The table shows that majority of the projects (80%) were small or medium-sized, between 1000 and 99,999 LOC in size. Only a very small amount of projects (1%) were more than one million LOC in size.

Table 1. Distribution of projects software sizes.

<table>
<thead>
<tr>
<th>LOC</th>
<th>Size description</th>
<th># of projects</th>
<th>% of projects</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt; 1000</td>
<td>Very small</td>
<td>101</td>
<td>4%</td>
</tr>
<tr>
<td>1000 - 9999</td>
<td>Small</td>
<td>801</td>
<td>32%</td>
</tr>
<tr>
<td>10,000 - 99,999</td>
<td>Medium</td>
<td>1188</td>
<td>48%</td>
</tr>
<tr>
<td>100,000 - 1,000,000</td>
<td>Large</td>
<td>384</td>
<td>15%</td>
</tr>
<tr>
<td>&gt; 1,000,000</td>
<td>Very large</td>
<td>26</td>
<td>1%</td>
</tr>
</tbody>
</table>

6.1.3 Contributors

On the average a project had 13 contributors. Table 2 gives an overview of distribution of contributors in projects. Majority of the project teams were relatively small, 64% had less than 10 contributors. The biggest downloaded project contained 310 contributors.

Table 2. Distribution of contributors.

<table>
<thead>
<tr>
<th># of contributors</th>
<th>Size description</th>
<th># of projects</th>
<th>% of projects</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt; 5</td>
<td>Very small</td>
<td>944</td>
<td>38%</td>
</tr>
<tr>
<td>5 - 9</td>
<td>Small</td>
<td>640</td>
<td>26%</td>
</tr>
<tr>
<td>10 - 49</td>
<td>Medium</td>
<td>809</td>
<td>32%</td>
</tr>
<tr>
<td>50 - 100</td>
<td>Large</td>
<td>76</td>
<td>3%</td>
</tr>
<tr>
<td>&gt; 100</td>
<td>Very large</td>
<td>31</td>
<td>1%</td>
</tr>
</tbody>
</table>

The correlation between contributors and project size in LOC was examined to get an understanding if large projects have more contributors, as it could be easily assumed. Number of contributors in projects of different sizes varied a lot; there were some small projects that had many contributors and there were also some large projects with only a few contributors. However, as Figure 4 suggests, the correlation between the number of contributors and LOC is positive ($r = 0.368$) and the correlation is significant ($p = 0.000$). This indicates that when project size grows it is likely to have more contributors and vice versa.

---

9 Percentages are rounded to the nearest integer. Due to this the sum is not always 100%.
6.1.4 Popularity

As mentioned in Section 5.2, popularity of downloaded projects was measured in the number of stars GitHub users had given to projects. On the average a project was starred 137 times. The most popular downloaded project was starred 6913 times. Table 3 shows the distribution of stars. It tells that majority of the projects (60%) were starred at least 10 times, indicating that they had been noticed in the community. Only a very small amount of downloaded projects (3%) were starred over 1000 times, implying huge popularity.

<table>
<thead>
<tr>
<th># of stars</th>
<th># of projects</th>
<th>% of projects</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt; 10</td>
<td>996</td>
<td>40%</td>
</tr>
<tr>
<td>10 - 99</td>
<td>945</td>
<td>38%</td>
</tr>
<tr>
<td>100 - 1000</td>
<td>489</td>
<td>20%</td>
</tr>
<tr>
<td>&gt; 1000</td>
<td>70</td>
<td>3%</td>
</tr>
</tbody>
</table>

6.2 Testing in the analysed projects

In this section the amount of test files found from the downloaded projects is examined. All test files are taken into account in this analysis. It is also investigated what project characteristics seem to increase or decrease the amount of test files.

Table 4 shows how many projects contained test files. Majority of the projects (73%) contained at least one test file. 1125 (45%) contained 10 test files or more and on the average a project contained 57 test files.

<table>
<thead>
<tr>
<th>Project</th>
<th># of projects</th>
<th>% of projects</th>
</tr>
</thead>
<tbody>
<tr>
<td>Contained at least one test file</td>
<td>1822</td>
<td>73%</td>
</tr>
<tr>
<td>Did not contain test files</td>
<td>678</td>
<td>27%</td>
</tr>
</tbody>
</table>
It could be assumed that larger projects contain more test files. Figure 5 shows that there seems to be a positive correlation between these two variables: projects with higher LOC are likely to have more test files and vice versa \( r = 0.679, p = 0.000 \).

![Figure 5. Correlation between LOC and found test files.](image)

Although the correlation between the number of test files and LOC is positive, it is also interesting to look at the correlation between the lines of production code and lines of test code per production code. This would help to get and understanding how much of the production code is actually tested. In this analysis, all code except tests are counted as production code. The correlation is shown in Figure 6. It shows that there is a weak negative correlation between these two variables: when the amount of production code grows, lines of test code per production code decreases \( r = −0.044, p = 0.028 \). Based on this, it could be assumed that bigger projects are not as comprehensively tested, but it is also possible that for bigger projects code base is just growing at a much faster rate than the test-code base.

![Figure 6. Correlation between production code and test code per production code.](image)
Interestingly Figure 6 also shows that in some projects the percentage of test code per production code is greater than one, meaning that there are projects which contain more test code than production code. A quick examination of these projects revealed that there were projects which contained only test files and there were only some production code files. Some of these repositories also had the word “tests” in their name. This suggests that for some OSS projects the test code is distributed in a separate repository.

It is also interesting to see if projects with higher number of contributors contain more test code per production code, i.e. is higher number of contributors helping the project to gain better testing coverage. The relationship between these two variables was investigated and the results can be seen in Figure 7.

![Figure 7. Correlation between contributors and test code per production code.](image)

There is a weak positive correlation between the number of contributors and the lines of test code per the lines of production code ($r = 0.063$). The calculated p value suggests that the results is also statistically significant ($p = 0.002$).

Finally, it could be assumed that projects with higher percentage of test code per production code are higher quality and thus possibly more popular in GitHub community. To find this out, the correlation between stars and test code per production code was examined and the results can be seen in Figure 8. There is a weak positive correlation between these two variables and that this correlation is statistically significant ($r = 0.043, p = 0.033$).
6.3 Test-driven development in the analysed projects

This section describes how many of the found test files were single-target tests and which of them were committed before, with or after their target files. After that the usage of TDD in projects containing single-target tests in examined in higher detail.

6.3.1 Analysed files

As it can be seen from Table 5, a total of 141 785 tests were found from all projects and 54 712 (39%) of them were categorised as single-target test file. 1476 projects contained at least one single-target test file. The remaining 87 073 (61%) test files were categorised as other tests. From now on, only the found single-target test files are further analysed to get an understanding of the usage of TDD in the projects.

Table 5. The number of tests with a single target.

<table>
<thead>
<tr>
<th>Category</th>
<th># of all tests</th>
<th>% of all tests</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single target was found</td>
<td>54 712</td>
<td>39%</td>
</tr>
<tr>
<td>Other tests</td>
<td>87 073</td>
<td>61%</td>
</tr>
<tr>
<td>Total</td>
<td>141 785</td>
<td>100%</td>
</tr>
</tbody>
</table>

All found single-target test files and their target files were divided to three categories according to their commit dates. Figure 9 contains an overview of the results. It seems that only a small fraction (5%) of the analysed tests were test-first files. However, majority (61%) of the analysed test files were committed with their target file. The remaining (34%) test files were committed after their target. Taking all projects into account, on the average a project contained 1 test-first test, 13 test-with tests, 8 test-after tests and 35 other tests.
If all test-first and test-with files were grouped together, they would cover approximately two thirds (66%) of all single-target tests. This means that two thirds of all single-target tests files were committed either before or with their target file and only one third (33%) were committed after the target was created. This could indicate that many programmers consider it a good practice to commit their test files before or with the target file.

The distribution of test-first files between projects is shown in Table 6. Majority of downloaded projects, 2024 (81%), did not contain any test-first tests and only 74 (3%) projects contained at least 10 test-first tests. One project contained by far the highest amount of test-first tests, 226, while the second highest project contained only 64 test-first tests. The top 5 projects containing more than 50 test-first tests contained a total of 453 test-first tests, which is 16% of all test-first tests.

<table>
<thead>
<tr>
<th># of test-first tests in project</th>
<th># of projects</th>
<th>% of all projects</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2024</td>
<td>81%</td>
</tr>
<tr>
<td>1 - 9</td>
<td>403</td>
<td>16%</td>
</tr>
<tr>
<td>10 - 49</td>
<td>68</td>
<td>3%</td>
</tr>
<tr>
<td>50 - 100</td>
<td>4</td>
<td>0%</td>
</tr>
<tr>
<td>&gt; 100</td>
<td>1</td>
<td>0%</td>
</tr>
</tbody>
</table>

The distribution of test-with and test-after files between projects is shown in Table 7 and Table 8. The number of projects containing test-with and test-after tests was much greater compared to projects containing test-first files. Test-with and test-after tests were also distributed more evenly and there were higher amount of projects containing more than 100 test-with or test-after tests.

<table>
<thead>
<tr>
<th># of test-with tests in project</th>
<th># of projects</th>
<th>% of all projects</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1255</td>
<td>50%</td>
</tr>
<tr>
<td>1 - 9</td>
<td>692</td>
<td>28%</td>
</tr>
<tr>
<td>10 - 49</td>
<td>376</td>
<td>15%</td>
</tr>
<tr>
<td>50 - 100</td>
<td>104</td>
<td>4%</td>
</tr>
<tr>
<td>&gt; 100</td>
<td>73</td>
<td>3%</td>
</tr>
</tbody>
</table>
6.3.2 Test-driven development in projects containing single-target tests

As mentioned previously, 1822 (73%) of the downloaded 2500 projects contained at least one test file. From these 1476 contained at least one single-target test file. This is 81% of projects containing tests and 59% of all downloaded projects.

From the projects containing single-target test files, 1292 (88%) contained at least one single-target test file that was committed either before or with its target file. A more detailed view of test-first and test-with files in projects containing test files is introduced in Table 9. It suggests that 42% of all projects containing single-target tests had over 80% of their single-target test files committed before or with their target file. This is 30% of all projects containing tests and 22% of all downloaded projects. On the average a project containing single-target tests had 61% of all its single-target test files committed before or with their target files.

Table 9. Percentage of test-first and test-with files of all single-target test files in projects containing single-target tests.

<table>
<thead>
<tr>
<th>% of test-first and test-with tests of single-target tests</th>
<th># of projects</th>
<th>% of projects containing single-target tests</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt; 20%</td>
<td>249</td>
<td>19%</td>
</tr>
<tr>
<td>20 - 49%</td>
<td>222</td>
<td>18%</td>
</tr>
<tr>
<td>50 - 80%</td>
<td>465</td>
<td>36%</td>
</tr>
<tr>
<td>&gt; 80%</td>
<td>540</td>
<td>42%</td>
</tr>
</tbody>
</table>

Since there were many projects containing only a couple of test files they might have a great effect on the relative amount of test-first and test-with files. Because of this the previous analysis was conducted again for projects containing at least 10 single-target test files. There were 773 projects like this, 41% of all projects containing tests and 31% from all projects. The results can be seen in Table 10.

Table 10. Percentage of test-first and test-with files of all single-target test files in projects containing at least 10 single-target tests.

<table>
<thead>
<tr>
<th>% of test-first and test-with tests of single-target tests</th>
<th># of projects</th>
<th>% of projects containing single-target tests</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt; 20%</td>
<td>67</td>
<td>9%</td>
</tr>
<tr>
<td>20 - 49%</td>
<td>145</td>
<td>19%</td>
</tr>
<tr>
<td>50 - 80%</td>
<td>291</td>
<td>38%</td>
</tr>
<tr>
<td>&gt; 80%</td>
<td>270</td>
<td>35%</td>
</tr>
</tbody>
</table>

As it can be seen, the percentages did not change much, which makes it more clear that majority of projects containing many single-target test files have preferred to commit tests before or with its target.
It was also investigated whether there is a difference between the average percentages of test-first, test-with and test-after files in projects of different sizes containing single-target tests. For this analysis, only projects containing at least 10 single-target tests were taken into account as is visible in Table 11.

**Table 11.** The average percentage of test-first and test-after tests in projects containing at least 10 single-target tests.

<table>
<thead>
<tr>
<th>Size description</th>
<th># of projects</th>
<th>% of test-first &amp; test-with</th>
<th>% of test-after</th>
</tr>
</thead>
<tbody>
<tr>
<td>Very small</td>
<td>0</td>
<td>n/a</td>
<td>n/a</td>
</tr>
<tr>
<td>Small</td>
<td>72</td>
<td>59%</td>
<td>41%</td>
</tr>
<tr>
<td>Medium</td>
<td>438</td>
<td>67%</td>
<td>33%</td>
</tr>
<tr>
<td>Large</td>
<td>243</td>
<td>64%</td>
<td>36%</td>
</tr>
<tr>
<td>Very large</td>
<td>20</td>
<td>47%</td>
<td>53%</td>
</tr>
</tbody>
</table>

The average percentage of test-first and test-with tests per all single-target tests is highest in medium sized and large projects. Writing tests after production code seems to be more popular in small and very large projects, although the differences between project sizes are not radically different.

### 6.3.3 Project size and the number of contributors affecting the usage of test-driven development

It is interesting to know whether certain project characteristics increase or decrease the amount of test-after tests. To examine this, the correlation between different project characteristics and the percentage of test-after tests of all single-target tests was analysed. Projects containing at least one single-target test were included in the analysis.

Figure 10 shows that there is a weak positive correlation between LOC and the percentage of test-after test files: when the project size increases, the percentage of tests committed after target seems to increase ($r = 0.029$), but the finding is not statistically significant ($p = 0.266$).

![Figure 10. Correlation between LOC and the relative amount of test-after tests per single-target tests.](image)
The same kind of relationship was found between contributors and tests-after tests per single-target tests. Figure 11 indicates that there is a weak positive correlation between the percentage of test-after tests and the number of project contributors ($r = 0.039$). However, like in the case of correlation between LOC and test-after tests per single-target tests, the result is not statistically significant ($p = 0.266$).

![Figure 11](image)

**Figure 11.** Correlation between contributors and the relative amount of test-after tests per single-target tests.

It was previously found that the popularity, measured with stars given to the project, has a weak correlation with test code per production code. It is interesting to find out if the correlation between the number of stars and the percentage of test-after files have a relationship, i.e. whether projects applying TDD practices have been more popular than those in which the tests are mainly committed afterwards.

As it can be seen from Figure 12 the situation is pretty much the same as it was when comparing stars with tests code per production code: there correlation between the variables is very weak ($r = 0.008, p = 0.761$). Thus, it can be said that the amount of stars the project has does not have any relationship with the amount of tests or the usage of TDD in OSS projects.
Finally, it was investigated how many of the test-target pairs were committed by the same author. TDD does not directly take a stand on whether the same developer should write both the tests and production code, but the concept is often seen as a guideline for developers who write them both. However, as discussed in Section 2.2, tests written by the same developer who also writes production code may not be as effective, i.e. catch as many defects, as tests written by other developers.

Regarding projects on GitHub it can be assumed that if the test and its target is committed by different author, the files are also written by the different authors. This was used as an assumption when comparing test-target pairs’ authors.

Taking all single-target tests into account, 80% of them were committed by the same author and 20% by different author. In the following figures it is shown how the percentages are divided between test-first test and test-after tests. For test-first tests the results can be seen in Figure 13.
The same data about test-after tests are shown in Figure 14.

![Figure 14](image)

**Figure 14.** Amount of test-after files committed by same and different authors.

As it can be seen, 55% of all test-first and 51% of all test-after files were committed by the same person who also committed their target. This means that in both cases approximately half of the tests were committed by a different author. When analysing the results, it should be taken into account that the amount of test-first files was smaller than test-after files. The large amount of tests committed by same committer when all single-target tests are considered is due to that test-with tests are always committed by the same author. This is why the test-with files are not included in figures 13 and 14.

As it was previously mentioned, taking all projects into account, on the average there were 1 test-first test, 13 test-with tests, 8 test-after tests and 35 other tests per project. Also, on the average 0.5 of test-first files were committed by the same person and 4 of test-after files was committed by the same person.

Of all 2500 projects, 852 of them contained at least one single-target test file that was committed by a different person than who committed the target file. This is approximately one third (34%) of all projects and two thirds (65%) of projects containing single-target tests.

A more detailed view of the amount of projects in which test-first and test-after files were committed by the same person is shown in Table 12 and Table 13. These tables show data from projects that contained at least 10 test-first or test-after files. Even though the percentage of test-first and test-after tests committed by the same author as the target file is high, the tables show that when looking at individual projects only in a few of them majority of tests are committed by different authors.

**Table 12.** Test-first tests committed by the same author as the target file in projects containing at least 10 test-first files.

<table>
<thead>
<tr>
<th>% of same committer in test-first tests</th>
<th># of projects</th>
<th>% of projects containing test-first tests</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt; 20%</td>
<td>29</td>
<td>8%</td>
</tr>
<tr>
<td>20 - 49%</td>
<td>51</td>
<td>14%</td>
</tr>
<tr>
<td>50 - 80%</td>
<td>77</td>
<td>21%</td>
</tr>
<tr>
<td>&gt; 80%</td>
<td>208</td>
<td>57%</td>
</tr>
</tbody>
</table>
Table 13. Test-after tests committed by the same author as the target file in projects containing at least 10 test-after files.

<table>
<thead>
<tr>
<th>% of same committer in test-after tests</th>
<th># of projects</th>
<th>% of projects containing test-after tests</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt; 20%</td>
<td>62</td>
<td>16%</td>
</tr>
<tr>
<td>20 - 49%</td>
<td>106</td>
<td>28%</td>
</tr>
<tr>
<td>50 - 80%</td>
<td>112</td>
<td>29%</td>
</tr>
<tr>
<td>&gt; 80%</td>
<td>103</td>
<td>27%</td>
</tr>
</tbody>
</table>

The relationship between the percentage of differently categorised test-target pairs committed by the same person and other project characteristics was investigated. All projects containing single-target tests were taken into account in this analysis.

Figure 15 suggests that when project size grows, the relative amount of test-first files committed by the same author seems to decrease ($r = -0.16, p = 0.000$).

Figure 16 suggests the same kind of relationship between LOC and test-after files that was seen in Figure 15; the relative amount of same authors committing both the test file and its target seem to be smaller in big projects ($r = -0.116, p = 0.00$). Although the findings are interesting, far-reaching conclusions cannot be made from the results since the amount of test-first tests and big projects was relatively small.
Figure 16. Percentage of same committer in test-after tests.
7. Discussion

In this section the findings are discussed and compared to the previous studies. However, as Hassan (2008, p. 54) has mentioned, repository data can only be used to find correlation between variables and is not very good alone at describing causations and reasons behind them. Thus, only assumptions can be made about the reasons for the phenomena observed in this study, and further research would be needed to validate them.

7.1 Number of test cases

The percentage of projects containing test files seems to be somewhat higher than in previous studies. For example in a study by Kochhar, Bissyande, Lo and Jiang (2013, p. 106) nearly 62% of the downloaded projects contained test cases while in this study the amount was 73%. Previous studies on testing in OSS have generally focused on all projects regardless of the used programming language while this study focused only in Java projects. The previous studies have shown that there seems to be a great differences between the number of test cases in different programming languages. Java has been available for two decades now and also has a pretty good amount of open source testing tools available. Thus, it can be assumed that the chosen programming language is one of the most significant reasons for the difference in high number of test cases between this and previous studies.

According to Kochhar et al. (2013), they selected files that contained the word test in their file name. It can be assumed that this way their study has contained files that contained the word test but are not likely tests. One example of such file could be SiteStats.java. The script used in this study picked only files that ended with the word test or its variants. Thus, the requirement for catching test files was stricter in this study and the difference is even more significant that the pure percentages indicate.

7.2 Testing in popular projects

It could be assumed that well-tested software products are more popular, meaning that they have more stars, in GitHub community. It was somewhat unexpected that the correlation between test code per production code and the number of stars the project had was almost non-existent. Based on that, it can be concluded that testing correlates with the popularity of the project, but the correlation is very weak.

One explanation for the very weak correlation could be the fact that unit testing, which was examined in this study, is not the only testing method available in software development. Popular projects could have also used other quality assurance processes than pure software testing which cannot be seen from the results of this study. Peer reviewing, for example, seems to be a popular quality assurance method in OSS projects (Otte et al., 2008, p. 1251).

7.3 The use of test-driven development

It was not surprising that distinct minority of all single-target tests were test-first files (5%), but the amount of test-with files was unexpected. Combining them with test-first files it turned out that two thirds (66%) of all tests where either test-first or test-with tests. The finding seems to be in line with Demeyer et al. (2008) whose study also indicated strong relationship between the evolution of production and test code in two open source projects.
Even if it is not possible to resolve the time when the test file and its target file were actually created in test-with cases, the high number of these cases indicates that adding tests for newly committed code files seems to be important in many OSS projects. If we grouped test-first files and test-with files together, two thirds (66%) of all analysed target files would have had test file when they were committed. Even though this does not tell anything about the quality of the tests, the result can be considered good since majority of the newly committed code gets tested in some form.

One explanation for the high percentage of test-with tests could be seen that some developers might be using software development tools that automatically create test files for each new production code file they create. These files were included in this study even if they were commented. This way the production code and its tests are easily committed together and it does not tell whether the developer was paying attention to writing real tests in the test file or not.

When discovering the relationship between different project characteristics and test-after tests the relationships found were mostly weak and thus any firm conclusions about the relationship between the usage of TDD and different project characteristics cannot be made.

7.4 Test committers

As the results indicate, large part of the test files and their target were committed by different persons. This was unexpected, especially in test-first cases, and is probably the most interesting finding in this thesis. The finding indicates that developers in open source projects are interested in testing code written by other developers.

Another interesting finding was the fact that the percentage of production code files and their test files committed by the same person seems to strongly decrease when project size grows. This seems to be true for both test-after and test-first files and especially the latter was highly unexpected. One obvious reason for this finding is the fact that small projects contain generally smaller amount of tests and developers and thus the relative amount of same committer in test-after tests varies more, as it can be easily seen from the scatter diagrams (Figure 15 and Figure 16).

Another explanatory factor is probably the fact that large projects, measured in LOC, also generally contain more contributors than smaller projects. The more contributors the project has the more likely it is that a single test was written by someone else than the person who wrote the production code.

7.5 Limitations

The empirical section in this thesis contains some limitations and weaknesses, which are discussed in this section along with how they have been addressed and taken into account when analysing the results. The most obvious limitations are related to the technical nature of the script.

It is possible that files not containing any tests might have been counted as test files. On the other hand, some test files might have been missed completely, especially if the tests were added into a different repository. To maximise the number of collected test files, the test file names were analysed manually to resolve the most typical name types and thus it can be assumed that great majority of the test files were collected correctly.
The script focused only on unit tests that were used to test only one single target file. If the script did not find a single specific target class for the test, it was classified as other test. It is possible that the target class was not always resolved correctly, which might have an effect on the results. However, manually checking some parts of the execution log gave an indication that at least most of the analysis was done correctly.

Test files committed at the same time with their target files does not guarantee that the files were created together. This applies only if the project followed good practices which say that commits should be made often and they should not be too large: if a large amount of files are committed together, it is likely they have been created over a longer period of time and commit dates say little or nothing about the order of their creation. Also, it is possible that some automated tools have created nearly empty test files for new production code files, which can have an effect on the results.

Only first commit dates of test files and their targets were taken into account and further changes of the files were not analysed. It is possible that even though the test and its target were committed at the same time, the test file was not maintained further with its target file.

It is notable that only the public data that was available in OSS repositories was analysed in study. It can be assumed that the downloaded projects contained at least the majority of the important code files and their tests that were made for the project. Still, it is possible that not all code files were shared to the public and thus not analysed in this thesis. It was also noticed that sometimes GitHub developers store their test files in separate repositories and this may have been the case also with the projects downloaded for this study that had no tests at all. This should not affect the results very much, since the dataset also contained projects with mainly test files.

Only Java projects were analysed in this thesis. This was a conscious choice, reasons for which were explained in Section 5.2. Still, it would be interesting to find out what kind of effect the used programming language has on the results. This would require a much larger dataset as well as different scripts for different programming languages, and thus it would be in the scope of another study. As mentioned in Section 5.2, there are a number of OSS testing tools available for Java and many testing tools are bundled with Java development tools. This helps OSS developers to start writing unit tests easily but the amount of tests and their simultaneous committing with target files might be highly different in less popular languages.

Finally, the fact that only single-target test files were analysed in this study leaves majority of all test files out. However, since TDD highly encourages to produce new tests for production code on frequent basis, it could be assumed that for every class file there is one precise test file, which should be enough to get accurate results. This addition to the difficulties related to analysing multi-target tests were the reasons why they were categorised as other tests and left without further analysis.

### 7.6 Suggestions for future research

As mentioned in the previous section this thesis contains some weaknesses. To get support for the findings, it would be possible to take the projects that are likely using development practices of TDD according to the results of this thesis, contact the developer of these projects and interview them to get support for the findings. The interview would also give results on the developers’ experiences of TDD whether its working in OSS projects in global software development context.
This thesis, like many previous studies, used the number of test files as an approximation of the amount of testing in OSS projects in addition to comparing the lines of test code to the lines of production code. However, the number of test files or code lines does not tell anything about the number and quality of single unit tests. To fill this gap, it would be possible to download OSS projects and analyse the quality of those tests using different characteristics, for example the code coverage level of the test files. Projects on GitHub also often have all the data of the development history available so it would also be possible to investigate how the tests are maintained during the whole development phase.
8. Conclusion

The purpose of this study was to examine the test-first aspect of test-driven development (TDD) in open source software projects (OSS). The research question set for this study was: “How widely is test-driven development used in open source software projects in unit testing level?”

A literature review of this study introduced the concept and previous studies of TDD as well as the general concept of OSS development. For TDD it turned out that its advantages and disadvantages are well known but results of many small experiments have been mixed. It looks like the concept of TDD fits in the world of OSS development culture but the usefulness of TDD in large multi-site projects, which many OSS projects also are, seems to be under debate.

A literature review revealed many testing-related challenges that exist in all software development, but are common in open source projects. The found challenges were difficulties in eliciting software requirements, communication and continuous integration in globally distributed software development, gaining user contribution, creating sufficient documentation and often insufficient resources. While TDD cannot alone solve these problems, it can probably help with them to some degree. However, the literature contained barely any studies that were directly related to examining TDD in OSS context.

To examine the current situation of the usage of TDD in OSS projects a quantitative research process was used to download OSS projects from GitHub and analyse them using an automated script. The script was used to find test files with one precise target file (single-target test file). Other tests were taken into account when counting the number of tests in projects, but they were not used in the analysis of the usage of TDD. When all single-target test files were found, they and their target file’s commit dates were compared to get an indication of if the test was added to the project’s repository before or with its target file. All in all 2500 active and at least one year old projects that contained at least two contributors were downloaded for the analysis.

It turned out that majority of the projects were created in 2012 and 2013. On the average a project was nearly 80 000 LOC in size, had 13 contributors and was starred 137 times. Majority of the projects (73% or 1822) contained at least one test file and on the average a project contained 57 test files. Also majority of all projects (59% or 1476), contained at least one single-target test file that was used in the analysis examining the usage of TDD. From these projects 1292 (87%) contained at least one single-target test file that was committed either before or with its target file.

Perhaps the clearest characteristic affecting the amount of test files is project size, measured in LOC, since a positive correlation between LOC and number of test files was found. However, somewhat unexpectedly, there was hardly any correlation between project popularity, measured with the number of stars, and lines of test code.

When counting all single-target test files together, only 5% of them were committed before the target file. An unexpected result was that 61% of them were committed with the target file. Combining these together it showed that two thirds of these tests were committed either before or with the target file, leaving only one third of tests committed after the target.
Correlations between test-after and different project characteristics were examined to get an understanding if certain things correlate with the usage of TDD. However, only weak correlations were found and none of them were statistically significant. Thus, it seems that the analysed project characteristics do not have much to do with the usage of TDD.

The authors of the test-target pairs were also examined in this study. It turned out that 55% of tests committed before target were committed by a different person than who committed the target. The situation was almost the same for tests written after target, 49% were committed by different persons. Percentages this high can be considered unexpected and no clear reasons for them could be determined in the scope of this study.

Most of the limitations of this study are related to the technical nature of the script, the most obvious ones being that only tests with one precise target file and their first commit dates were taken into account. Also, even though the script was tested and its execution log was analysed multiple times before the actual research, it is possible that some faulty analysis has been made. Some discussion is done based on empirical findings and the found literature review to analyse the reasons or purposes regarding the usage of TDD. However, since this type of study cannot give a clear answer to that, a survey or an interview with the developers known to be using TDD could be a topic of a future research.

To give a short answer to the research question, a distinct minority (5%) of all found single-target tests were test-first tests, although majority of the projects (73%) contained at least one test file. Also, there were 561 projects which contained at least 10 single-target tests and in which at least half of the single-target tests were either test-first or test-with tests. These 561 projects form 22% of all projects and nearly one third (31%) of projects containing tests. Thus, from the sample of OSS projects downloaded for this study, it seems that a distinct minority are certainly following the test-first aspect of TDD in the sense that tests are added to the project’s repository before the production code, although it is possible that tests are made first and only committed later with the target file. Since majority of the tests were committed with the target file, it can be assumed that most of them were created at least closely together. Thus, the most important finding in this study is that there is evidence of the open source community’s commitment to creating tests for their code.
9. References


Source Development, Communities and Quality (Vol. 275, pp. 273-280). Springer US.


Gerlec, Č., Krajnc, A., Heričko, M., & Božnik, J. (2011). Mining source code changes from software repositories. 7th Central and Eastern European Software Engineering Conference in Russia (pp. 1-5). Moscow: IEEE.


