Exploration of the code review process:
Can a code review checklist improve the process?

Maiko Goudriaan

Technische Universiteit Delft

TU Delft Delft University of Technology

Challenge the future

Exploration of the code review process:

Can a code review checklist improve the process?

by

Maiko Goudriaan

in partial fulfillment of the requirements for the degree of

Master of Science
in Computer Science

at the Delft University of Technology,
to be defended publicly on Tuesday May 18, 2021 at 9:00 AM.

Supervisor:	Prof. dr. A. E. Zaidman	
Thesis committee:	Prof. dr. A. E. Zaidman,	TU Delft
	Dr. U. K. Gadiraju,	TU Delft
	M. Fens MSc,	Greenchoice

An electronic version of this thesis is available at http://repository.tudelft.nl/.

TU Delft Delft University of Technology

Abstract

In the software engineering field the code review process has been widely adopted as a quality control, in the current day code reviews have also multiple additional advantages. In this research we explored how the code review process is used at Greenchoice, a Dutch energy supplier. Subsequently, we created a code review checklist which was used by the developers in order to improve the process. In the first step of the exploration we used a questionnaire to retrieve Greenchoice's motivation for the code review process. In the results we show that ensuring *quality* is the main motivation followed by *knowledge sharing* and *functionality* check. Secondly, we held 13 semi-structured interviews, performed a data analysis on the metadata of 8458 pull requests and performed a data analysis of 5400 code review comments to explore Greenchoice's code review process further. In the results of the interviews we show that the reviewers tend to focus on understanding the change and finding obvious mistakes. They also mentioned code reviews could be performed stricter improving the maintainability and evolvability aspects. However, it can be hard to know where to check on for these aspects. In the results of the metadata analysis we show that the code reviews are performed quick, frequent and small changes are made, similar to the modern code reviews processes described in the literature. In the results of the code review comments analysis we show that most comments are related to explaining or understanding the change. Next, we created together with a scrum team a code review checklist which was was used by three scrum teams for two periods of two weeks. In order to measure the effectiveness of the code review checklist we used a pre questionnaire to gauge the current situation and the expectations of the usage. A post questionnaire was used to gauge the new situation and whether the expectations were met. After comparing the results we show in the results that the checklist is less appropriate to guide the reviewer step by step in a code review. Furthermore, we show that the items in the checklists should be applicable to the different changes in the pull requests. The strength of the checklist lies in addressing the items which are often forgotten in the development process and the code reviews.

Preface

The completion of this thesis was not an easy task and sacrifices have been made in the creation of this master thesis, though no animals were hurt in this process. Lets get back to the start of my master thesis. It was December 2018 I found my first subject for my master thesis, unfortunately due to circumstances the subject did not go through. I decided to first complete the required courses and in the mean time look for a new subject. In a chat with a teammate from sport I told that I was looking for a new subject and he proposed to talk to his team leader at work to see whether they had a subject. No sooner said than done I started in July 2019 my master thesis at Greenchoice. The period at Greenchoice went well, at least in my opinion, I enjoyed the stay and the lessons that I learned working in an industrial environment compared to the university projects. It was the end of January 2020 when my time at Greenchoice was over and *just* had to finish up the report. Early March the coronavirus has also reached the Netherlands for some days and the country went into a smart lockdown. At that time I spent most of my time for a music venue which had to be closed due to the smart lockdown. Next, was the preparation and opening of the music venue in the new 1.5 meters society.

Hold up, I am forgetting something... I should also finish my master thesis.

Motivation was hard to be found and since June 2020 several attempts have been made attempting to finish the report. Small steps were made towards finishing the report, however, it was till January 2021 that I got the first impression that I was actually going to finish the report. The next months I spent finishing up the report and processing the (abundant) received feedback. After a long bumpy ride I present to you my final master thesis report.

First off, I would like to thank Roelant for introducing me to Greenchoice and the support throughout my stay at Greenchoice. Furthermore, I would like to thank Greenchoice as a company and the employees, especially the ones who participated in my research, for the fantastic time I had and helping me in my research. In special I would like to thank Mick and Jeroen for their supervision, guidance and support from the side of Greenchoice and most importantly their patience with me to complete my report after a slight delay. Furthermore, I would like to thank Andy for the supervision, guidance and support from the side of the TU Delft and also his patience with me. Subsequently, I would like to thank Ujwal for taking place in the thesis committee. In addition, I would like to thank everyone else for the support through the sometimes hard corona times with the lockdowns. Lastly, I would like to thank my parents for keeping me fed throughout the past period.

Maiko Goudriaan
Delft, May 2021

CONTENTS

1 Introduction **1**
1.1 Research design . . . 2
1.2 Introduction of Greenchoice . . . 3
1.2.1 Greenchoice's development process . . . 4
1.2.2 Greenchoice's coding guidelines . . . 4
1.2.3 Greenchoice's old code review checklist . . . 4
1.3 Research questions . . . 5
1.4 Structure of thesis . . . 6

2 Related work **7**
2.1 Formal software inspection . . . 7
2.2 Informal software inspection . . . 8
2.3 Modern code reviews . . . 9
2.3.1 Challenges in open source software development . . . 9
2.3.2 Challenges in industrial setting . . . 10
2.3.3 Resolving the size challenge . . . 10
2.3.4 Motivation for code reviews . . . 10
2.3.5 Actual defects discovered . . . 10
2.3.6 Usefulness and understanding code reviews . . . 10
2.3.7 Identify defect prone areas of code . . . 11
2.3.8 Checklists . . . 11

3 Methodology **13**
3.1 Initial code review questionnaire . . . 13
3.1.1 Questionnaire introduction . . . 14
3.1.2 Discovering the motivation for code reviews . . . 14
3.1.3 Comparing the different categories . . . 16
3.1.4 Discovering the initial thoughts with statements . . . 17
3.1.5 Error prevention in practice . . . 17
3.2 Exploring Greenchoice's current code review process . . . 18
3.2.1 Interviewing the developers . . . 18
3.2.2 Meta data analysis . . . 18
3.2.3 Comment data analysis . . . 20
3.3 Creating a code review checklist . . . 23
3.3.1 Expectations and intentions regarding the checklist . . . 23
3.3.2 Joining a scrum team . . . 23
3.4 Using and refining the checklist . . . 24
3.4.1 Explaining the general set-up . . . 24
3.4.2 Explaining the pre questionnaire . . . 24
3.4.3 Explaining the post questionnaire . . . 26

4 Motivation for code reviews **27**
4.1 Demographic data . . . 27
4.1.1 Function . . . 27
4.1.2 Experience and familiarization . . . 28
4.1.3 Frequency and roles . . . 28
4.2 Motivation for code reviews . . . 29
4.2.1 Comparing the motivations . . . 29
4.2.2 Global categories . . . 31

4.3 Code review statements . . . 33
4.4 Practice example . . . 34
4.5 Discussion . . . 34
4.5.1 Demographic data . . . 34
4.5.2 Motivations . . . 35
4.5.3 Code review statements . . . 36

5 Current code review process 39
5.1 Interviews . . . 39
5.1.1 Preparation for a pull request . . . 39
5.1.2 Performing a code review . . . 40
5.1.3 Strengths and weaknesses . . . 41
5.1.4 Potential improvements . . . 45
5.2 Metadata analysis . . . 46
5.2.1 General information . . . 46
5.2.2 Exploration of the time features . . . 49
5.2.3 Comparing the time features . . . 52
5.2.4 Exploration of the size features . . . 55
5.2.5 Comparing the size features . . . 56
5.2.6 Comparing the size feature to time features . . . 58
5.2.7 Explaining the reviewer interaction features . . . 59
5.2.8 Comparing the user interaction features to the time and size features . . . 61
5.3 Code review comment data analysis . . . 62
5.3.1 Results of the global categories . . . 62
5.3.2 Results of the remaining categories . . . 64
5.4 Discussion . . . 65
5.4.1 Interviews . . . 65
5.4.2 Metadata analysis . . . 67
5.4.3 Comment data analyse . . . 68

6 Creating a code review checklist 70
6.1 The exploration session . . . 70
6.2 Determining the checklist items . . . 72
6.3 Explaining the checklist . . . 73
6.4 Final checklist . . . 74
6.5 Discussion . . . 75
6.5.1 Finding the strengths and weaknesses of the scrum team . . . 75
6.5.2 Creating the checklist . . . 76

7 Using the code review checklist 78
7.1 Pre questionnaire . . . 78
7.1.1 Personal interest . . . 79
7.1.2 Thoughts towards code quality . . . 79
7.1.3 Thoughts towards checklists . . . 81
7.1.4 Thoughts towards code review comments usage and tooling . . . 81
7.1.5 Expectations usage checklist during code reviews . . . 84
7.1.6 Final remarks . . . 86
7.2 Checklist evaluation . . . 86
7.3 Refining the checklist . . . 87
7.3.1 Refined checklist . . . 89
7.4 Post questionnaire . . . 90
7.4.1 General experience of the experiment . . . 90
7.4.2 Review experience . . . 91
7.4.3 Final remarks . . . 94

7.5 Discussion . . . 95
7.5.1 Usage of the checklists in general . . . 95
7.5.2 Usage frequency . . . 96
7.5.3 Code review comment usage . . . 96
7.5.4 Increase in types of comments used . . . 96
7.5.5 Structure and applicability of the checklist . . . 97

8 Conclusion 99
8.1 Answers to the research questions . . . 99
8.1.1 Motivation for code reviews . . . 99
8.1.2 Current code review process . . . 100
8.1.3 Creation of the code review checklist . . . 101
8.1.4 Usage of the code review checklist . . . 101
8.2 Recommendations to Greenchoice . . . 101
8.3 Future work . . . 102
8.3.1 Dynamic checklist generator . . . 102
8.4 Threats to validity . . . 103

A Motivation questionnaire 105
A.1 Demographic questions . . . 105
A.2 Motivation for code reviews . . . 106
A.3 Motivation for code reviews (additional answers) . . . 106
A.4 Code review statements . . . 106
A.5 Code reviews in practice . . . 107
A.6 Final remarks . . . 107

B Motivations for code reviews 108
B.1 4 eyes principle . . . 108
B.2 According to architecture . . . 108
B.3 Consistency between teams . . . 108
B.4 Discussing solution . . . 108
B.5 Following code guidelines . . . 108
B.6 Functionality . . . 109
B.7 Knowledge sharing . . . 109
B.8 Preventing defects . . . 109
B.9 Quality . . . 109
B.10 Team Awareness and ownership . . . 110
B.11 Understandable and maintainable code . . . 110

C pre questionnaire 111
C.1 Introduction . . . 111
C.2 Personal interest . . . 111
C.3 Attitude against code quality . . . 111
C.4 Thoughts towards checklists . . . 112
C.5 Thoughts towards code reviews . . . 112
C.6 Tooling (optional section) . . . 113
C.7 Expectations about the use of a checklist during code reviews . . . 113
C.8 Remarks and feedback . . . 114

D Post questionnaire 115
D.1 Introduction . . . 115
D.2 Experience usage of checklist . . . 115
D.3 Review experience . . . 115
D.4 Remarks and feedback . . . 117

E Long list of checklist items **118**

F Shortlist of checklist items **121**

G Final checklist **123**

H Refined checklist **124**

Bibliography **126**

1
INTRODUCTION

In the earlier days of software development the waterfall model was commonly used in which the process went down into the next phase without the possibility to turn back [1, 2]. This means that the current phase should be completely finished and refined before the process could go the next phase. The short version of the waterfall model starts with a requirement phase in which the requirements of the desired software are extracted from the client and thoroughly documented. Next, the designs of the software architecture is created and thoroughly documented. Subsequently, the software is implemented according to the predefined requirements and design rules. Next, the created software is tested and defects are discovered and resolved. Finally, the software is deployed and maintained to ensure its functionality.

This process focuses on documentation and leaves little space for mistakes, once a mistake is made in one of the earlier phases it can have a tremendous effect in the later phases, without the possibility to go back [2]. However, in later versions of the waterfall model feedback loops were introduced to go back to earlier phases before advancing to the next phase [3]. This process is currently unimaginable in the fast changing world of software development in which we are overwhelmed with daily updates and we are wondering what has been changed.

In the current fast paced development process the waterfall model is no longer used and different types of the agile processes have been widely adopted [4]. In general, the phases of the agile process are still similar to the phases of the waterfall process, however, agile process focuses on small, short and flexible sprints. This means that in a sprint only a smaller part of the software is developed in periods of 1 to 4 weeks. The agile process stands for working on the software over extensive documentation, customer collaboration over contract negotiation, and responding to changes over following a predefined plan [5].

Part of the software development process is the quality control of the software in which code reviews are an important factor [6, 7]. Code reviews were first introduced in 1976 by Fagan [8] to detect as many functional defects as possible in the design and development process of the software. Throughout time the code review process also changed according to the faster paced development practice and turned into frequent, fast and lightweight code reviews [9]. Today's code reviews no longer address solely functional defects, but also maintainability and evolvability defects in the code [10]. In this thesis we will look into the code review process of Greenchoice and look into the usage of a code review checklist. Before we explain the research design further in Section 1.1, we first want to introduce what a review is, a brief introduction into the usage of code reviews and the usage of checklists.

WHAT IS A REVIEW?

The term *review* is defined by the Cambridge dictionary as following: *"to think or talk about something again, in order to make changes to it or to make a decision about it"* [11]. One of the most common reviews might be a book or film review, or even a random product in which the reviewer expresses their personal experience regarding the book, film or product. Based on these reviews recommendations could be made for a follow-up book or movie. Currently user reviews play an important role in collaborative filtering techniques to create large recommendations systems [12]. Netflix even held a competition between 2006 and 2009, the Netflix prize [13], to create the best recommendation system [14]. Furthermore, recommendation systems play an important role in the e-commerce and e-marketing for large companies such as Amazon [15].

In addition, reviews are also commonly used in the literature in the form of a literature review. According to Webster and Watson [16] the use of a literature review is essential to create a firm foundation to gain advancing knowledge in the area of research. A literature review exposes areas of less interest in which abundant research is performed and exposes interesting areas of research with limited previous research [16]. A common form of a literature review firstly introduced in the medical field is known as a systematic review [17–19] which is based on evidence and follows up the expert opinion which was often biased [17]. Cronin *et al.* [19] describe the systematic review as following: *"A systematic review is a comprehensive search, critical evaluation, and synthesis of all the relevant studies on a specific (clinical) topic."*. Kitchenham [20], Kitchenham *et al.* [21] introduced the systematic literature reviews into the software engineering field, however, at that time still limited systematic literature reviews have been performed in the software engineering field [21].

Thus a systematic review aims to gather as much information as possible in an area of research based on hard evidence and exposes the weaknesses, such as contradicting or inconclusive findings[18], of the area of research which requires further research. With the gathered information the authors gain insight into the area of research and gain insight in how to proceed in their research. Translating a literature review to a code review, the reviewer of a code review should also gather as much information as possible in order to assess the quality of the pull request and exposing the weaknesses and defects in the changed code.

What is a code review?

Coherent to performing a code review is the creation of a pull request, these terms will be used rather frequently throughout this report. We therefore want to explain and clarify the meaning of these two terms in our own words. A code review is simply the process of reviewing or evaluating a piece of software, either newly created piece of software or a modification in an existing part of the software, written by another developer. During a code review the intention of the change, the functionality of the change and the quality of the change will be assessed as we show further throughout this thesis. Nowadays software is often developed in parallel on multiple branches. This process is often managed through a version control system of which Git is widely adopted. However, managing a Git repository is a tedious task and therefore tools have been created to host and manage Git repositories whereby GitHub is probably the best known tool [22]. A pull request is the request to merge a change made on a separate branch into another branch, the target branch is often the master branch in which all changes are accumulated once the change is complete. A pull request itself is no feature of Git, however, a feature of the hosting and manage tool, thus for example GitHub. Merging the code frequently into the master branch is also called continuous integration (CI) [23]. The CI could be extended with additional automated steps to ensure the quality of the software development process. These additional steps could be for example: building the software on a server, integrating automated tools (e.g. assessing the code quality) and running the software tests. In pull requests an often integrated policy is that another developer should approve the pull request whereby we come back to performing a code review.

The usage of checklists

In complex and stressful situations with an increased chance of human errors exposing high safety risks, checklists are often used to decrease the chance of human errors and catastrophic consequences [24, 25]. According to Hales and Pronovost [24] a checklist is a list of items allowing the user to go systematically over the items to ensure all essential items are considered or completed in a particular area. In other words all potential known risks are minimized once the user has checked off all items in the checklist. Checklists also exist in a more informal setting such as todo notes on a post-it note. The usage of checklists has proven its effectiveness in the medical [26, 27] and aviation [28, 29] fields, however, the practice of a checklist is not yet widespread in other fields and is often a forgotten business tool [24, 25]. In the current outbreak of a new coronavirus (COVID-19) checklists are used to minimize the safety risks in the treatment of the patients [30, 31].

1.1. Research design

In this thesis we will explore the code review practices at Greenchoice, a Dutch energy supplier and will be further introduced in Section 1.2, and how to improve their code review process through the usage of a code review checklist. In our research we will perform an exploratory study [32] into Greenchoice's motivation for code reviews and the strengths and weaknesses of Greenchoice's code review process. The latter is done in three steps: 1) interviews with Greenchoice's developers, 2) a data analysis of the metadata regarding pull requests and code reviews and 3) a data analysis on the subjects mentioned in the code review comments.

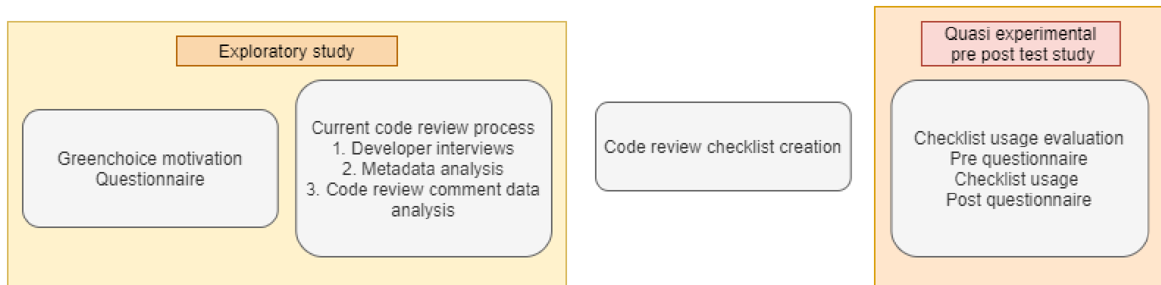


Figure 1.1: Overview of the research design

Furthermore, we will create and introduce a checklist which can be used during code reviews in order improve the code review process and minimize the risks exposed in a pull request. The usage of the checklist will be analysed through a quasi experimental pre post test study [32–34] using questionnaires. In other words, first a pre questionnaire is send out to retrieve a baseline of the current situation. Then the code review checklist is used creating a new situation. Lastly, a post questionnaire is send out retrieving the results of the new situation. An overview of the research design is shown in Figure 1.1.

In the literature there have been several research projects performed regarding the motivation, best practices and effects of code reviews. These research projects were mostly performed at large tech companies such as Google [35], Microsoft [9, 36], and Mozilla [37], at universities with student groups [38, 39], or in open source projects [22, 40, 41]. With our research obtain an insight into the code review process of a middle sized Dutch company which is by default not a software development company. However, in the current day Greenchoice is a tech company in which the data of the customer, product and grid operator come together. This requires modern day IT solutions, which are mainly developed in house, to be able to handle the increased data flow and provide the best service.

In addition, our research fills the gap in an area with limited research [42, 43] which was performed on the usage of checklists during code reviews. However, more recent literature which was released during the creation of this thesis is also looking into the usage of a checklist during code reviews [44, 45]. Whereas Chong *et al.* [46] looked into the quality of code review checklist created by students. Furthermore, the creation of code review checklist could be used as a teaching method for the code review process.

The rest of this chapter is structured as following, next in Section 1.2 we will introduce Greenchoice and its current software development process in which we also give a brief introduction of Greenchoice's code review process which we will further explore on throughout this thesis. Subsequently, in Section 1.3 we present the research questions used on which this thesis was build. Finally, in Section 1.4 we give an overview on the structure of the remaining chapters in this thesis.

1.2. INTRODUCTION OF GREENCHOICE

Greenchoice is an energy supplier which strives to provide 100% renewable energy. It is a medium sized Dutch company with roughly 350 employees and is in principle an administrative company that buys energy from the companies who generate energy and sells it to Greenchoice's customers. However, nowadays more aspects should be considered in running the company: marketing, customer service, analysing the energy market, transition into renewable energy and part of the company is to create and manage software created and used within the company. Furthermore, Greenchoice owns multiple windmills and solar parks to generate its own renewable energy. However, a difficult factor in generating and using renewable energy is that most methods are dependent on the weather especially sun and wind energy. The energy supply of wind and sun energy is not consistent and sometimes an abundant amount of energy is generated and sometimes a shortage occurs. This raises the challenge of distributing the energy according to the current energy need, requiring the storage of excess energy during the peaks in generation which can be later used during the peaks in energy demand. The storage of energy is a difficult task and several methods are used to store the energy [47–49]. Greenchoice is currently investing in the storage of energy in large batteries. Additional, Greenchoice's vision is to let the customers generate their own renewable energy locally, this can be through solar panels or smaller windmills which could be placed on top of buildings in the area.

1.2.1. GREENCHOICE'S DEVELOPMENT PROCESS

The IT department of Greenchoice consists of roughly 80 employees and is divided over an IT development and an IT operation team. The operation team is responsible for maintaining the network, (e-mail) servers, workplace management and also facilitates the hardware, for example personal work laptops and mobiles. The IT development team, in which we held our research, is responsible for developing and maintaining software which is used by the internal business departments. The development team consisted at the start of our research of 8 scrum teams and throughout our period a ninth team was started up. Most scrum teams consist of 3 developers, a tester, a business consultant and a product owner. However, a single team has, instead of a dedicated tester, an additional developer. Furthermore, Greenchoice had at the time we started our research a single architect, however, quickly after we started our research a second architect was promoted. Finally, the IT development team is lead by two team leaders and the IT department as a whole is managed by a single manager.

Greenchoice uses the scrum process for their planning process, though a single team uses kanban which is an even more flexible process compared to scrum [50]. A sprint within Greenchoice consists of 2 weeks after which the new code is deployed. However, Greenchoice is currently switching to an architecture of microservices instead of multiple larger applications, this provides the ability to the teams to continuously deploy their own microservices. In the transition into microservices Lenarduzzi *et al.* [51] performed an extensive study on the migration from a monolithic system, a single large application, into microservices in which they mainly focused on the technical debt aspect.

Greenchoice uses Azure DevOps [52] as their code hosting service and their planning tool. The tasks in a sprint are denoted in a product backlog item (PBI) by the product owner and business consultant. The business consultants and product owners are the link between the scrum teams and the business departments. They themselves are not actively involved in writing code. A PBI is developed in its own branch and once finished, merged into the protected master branch via a pull request. Before a merge is allowed another developer should perform a code review and approve the pull request. Additional, the build should pass on the CI and the PBI should be linked in the pull request. During a code review the reviewer checks whether the change is correct. Throughout this thesis we will get more familiar with the details of Greenchoice's code review process.

Furthermore, in the test process of Greenchoice the developers write their own unit tests and the dedicated tester performs additional tests. Most testers do not have a dedicated programming education and have learned the necessary skills in practice. This reflects in that most tests are performed manually, however, there is an ongoing process in test automation in which major steps are made past period and for which the testers are trained the concepts of programming.

1.2.2. GREENCHOICE'S CODING GUIDELINES

Looking in more detail into Greenchoice's development process, Greenchoice provides a Wiki page with architectural agreements and uses the following commonly known coding principles as guidance during programming: *Keep it simple, stupid* KISS, *You aren't gonna need it* YAGNI, *Don't repeat yourself* DRY and *SOLID* [53]. The latter is an acronym of a collection of principles, the principles are: *Single responsibility*, *Open/closed*, *Liskov substitution*, *Interface segregation* and *Dependency inversion* [53]. Most importantly in the development process is to use common sense.

Furthermore, Greenchoice uses SonarQube [54], which was later replaced by SonarCloud [55], to automatically check the code quality and detect code vulnerabilities. The biggest difference between SonarQube and SonarCloud is that SonarQube was self hosted and as the name suggest SonarCloud is hosted in the cloud, in terms of quality check and vulnerabilities detected they perform the same. Furthermore, the ReSharper [56] tool is used to check the code style and automatically provide quick fixes and provides quality of life features for refactoring and navigation of the code to the developers. Greenchoice does not provide additional coding guidelines, however, there exist an deprecated Wiki page with detailed (older) coding guidelines which is replaced by the coding principles and most importantly the common sense of the developers.

1.2.3. GREENCHOICE'S OLD CODE REVIEW CHECKLIST

During the introduction of code reviews in 2016, roughly 5 years ago, Greenchoice also introduced a code review checklist besides a definition of ready (DoR) and a definition of done (DoD). A DoR describes the requirements when a PBI is ready to be included in a sprint and DoD described the requirement when a PBI is finished and ready to be merged into the master branch. The introduced checklist was however not used in practice and is currently also not available on any Wiki page. The following items were included in the

checklist.

- Does the code work and does it do what it is supposed to do?
- Is the code easy to understand?
- Is there no redundant or repetitive code?
- Has the code been set up as modularly as possible?
- Is there no commented out code?
- Have no functions been written that are also available in another library?
- Is the code sufficiently testable?
- Have enough unit tests been written?
- Are the unit tests covering the load?
- Are there no build warnings?
- Are unmanaged resources disposed?

1.3. RESEARCH QUESTIONS

Our research is focused on two main goals namely analysing and improving Greenchoice's code review process. To achieve this we have divided this thesis into 4 research questions which were refined and answered throughout the study.

RQ 1 **What is Greenchoice's motivation behind code reviews?**
Although code reviews are commonly used in most software development companies, each company has their own desired goals with their code reviews. In the first research question we will look into the motivations for code reviews at Greenchoice. The motivations will be gathered via a questionnaire and we expect a list of motivations sorted by their priority as result. The list of motivations can be used to assist us in determining where and how to improve on in the code review process. Additionally, through the questionnaire we gain insight into the initial thoughts and experiences from the respondents regarding code reviews.

RQ 2 **What does the current code review process of Greenchoice look like?**
In order to improve the code review process at Greenchoice it is important to know how the code review process is currently performed. What are the positive and negative aspects in the current process? In order to explore the current code review process we split this research question in three sub questions, the perspective of the developers through interviews, a data analysis of the available metadata regarding code reviews in DevOps and the types of comments used also available in DevOps. The sub questions are formulated as following:

RQ 2.1 **What does the current code review process of Greenchoice look like in the perspective of the developers?**

RQ 2.2 **What does the current code review process of Greenchoice look like according to the metadata related to the code reviews?**

RQ 2.3 **What types of comments are found in the current code review process?**

Based on the findings of the sub questions we can relate the different aspects. The data can support or contradict the observations by the authors. Finally, the current process can be compared to state of the art code review processes. In this research question we gain an insight in the potential factors of the code review process to improve on.

RQ 3 **Which factors should be included in a well designed checklist for code reviews?**
Each developer has her own preferred method of performing code reviews. In this research question we look, together with a dedicated scrum team, into how to design a well suited code review checklist. Is it possible to create a single checklist that suits the needs of all developers? Furthermore, which items are needed in the checklist and at what level of detail? The expected outcome is a code review checklist that guides the developers in the code review process to find the most defects as possible. In addition, we gain insight into the factors that are deemed important, interesting and not necessary in the code review process.

RQ 4 **How do developers experience the usage of a checklist during code reviews?**
In this last research question we look, together with three scrum teams, into the experience of the actual usage of the checklist during code reviews. The usage of the checklist requires discipline to be used consistently and to go over all the items. Additional, the usage of the checklist may require more time. Are the developers willingly to spend additional time and bring up the discipline working with a checklist? In this research question we gain an insight in the positive and negative aspects of the usage of the checklist. The aspects are measured via a pre- and post questionnaire, and interviews with the scrum teams.

1.4. STRUCTURE OF THESIS

The remainder of the thesis is structured as following. In Chapter 2 we look into related work of code reviews. We discover how the process of code reviews was introduced and how the code review process evolved over time to the current widely adopted code reviews defined as *modern code reviews* [9]. Additionally, we describe the literature which previously looked into the known motivations for code reviews, the challenges and best practices and the usage of checklists during code reviews.

In Chapter 3 we describe in the methodology in detail the steps used to answer the previously stated research questions. In the methodology the used questionnaires, the interviews we performed and how we gathered and analysed the data are described. In addition, how we did create the code review checklist and analysed the usage of the checklist.

Chapter 4 describes the result of the motivations gathered through a questionnaire. In additional, we gain an insight with the questionnaire in the composition of the IT development department, their degree of experience with code reviews and reveals their initial thoughts regarding code reviews.

Next, in Chapter 5 we describe in the first Section 5.1 the findings of the developer interviews. We show how the developers go through code reviews and discover their perceived best practices and limitations in the code review process. In Section 5.2 the results of the metadata analysis are described. We show and compare the different characteristics of code reviews such as the size and merge times. In Section 5.3 the results of the types of comments used in code reviews are described. Finally, in Section 5.4 we compare the different results to each other and the known literature.

Next, in Chapter 6 we show the process of creating a code review checklist in cooperation with a dedicated scrum which will be used during code reviews. We held two session with the scrum in which we explored in the first session their code reviews process and in the second session their preferred items to be included in the checklist. After the second session we created a checklist which was adjusted to the latest feedback of the developers of the scrum team and the architects and present the final checklist.

Subsequently, we describe in Chapter 7 our findings of using the checklist in practice. First, we show the expectations of the developers which we gathered through a pre questionnaire. After the first period of usage we evaluate the used checklist and refine the checklist for the second usage period. Next, we show the real experiences of the usage of the checklists according to the developers and compare these to the earlier gathered expectations.

Finally, in Chapter 8 we summarize the answers to the research questions. In addition, we present our recommendations to Greenchoice for the code review process and the usage of a code review checklist. Lastly, we give an insight in the future work and the threats to validity of the study.

2

Related work

In this chapter we present the previous literature regarding the code review practice. In Section 2.1 we explain the start of the code review practice in the form of a formal inspection. Subsequently, we present in Section 2.2 the transition into a more informal setting. Finally, in Section 2.3 we present the contemporary code reviews described in the state of the art. We explore the challenges faced in the different code review practices used in open source software projects and industrial settings. In which we also look into the different known motivations for the code review process and the usage of checklists.

2.1. Formal software inspection

The term code review originated first in the literature as the term formal software inspection in 1976 by Fagan [8]. In a later paper by Fagan [57] he stated that the formal software inspection process was first introduced in 1972 at IBM Kingston. In the paper of 1976 Fagan [8] the formal inspection was experienced useful as it improved the productivity and quality of the software development process. The goal of the formal software inspection is to detect defects early on. The cost of resolving a defect becomes more expensive the later it is found in the process.

With the introduction of the formal code inspections the need of the measurement aspect is fulfilled. Measurement together with planning and control are three aspects to consider for any successful process [8]. Hereby discipline lays an important role in each aspect, clear rules with understanding of flexibility and preventing non-flexibility. The latter can be also described as using clear guidelines over strict rules, thus exceptions can be made if necessary.

The formal code inspection is performed by 4 participants: moderator, designer, programmer and tester [8]. Whereby the moderator is a neutral person who did not work on the software and is preferably specifically trained to be a moderator. It can occur that the programmer is also the designer or tester, hereby that person fulfills the role as programmer, the other roles are filled by others who were closely involved or working on similar software. The inspection is a meeting scheduled by the moderator and should not take longer than 2 hours as then the productivity decreases. During the inspection the moderator denotes the defects found. The defects considered are functional errors, meaning unexpected and unwanted behaviour of the software or missing functionality. To find the defects in the program a checklist can be used specifically designed for that project.

During the inspection the moderator reports all defects discovered and should sent within a day a report to the other participants. The team members can then start to resolve the defects they are responsible for. Once all defects are resolved a new inspection is held, depending on the amount of defects only the solution of the defects are checked or a second full inspection is held. The formal software inspection is the first process that actually provides real-time feedback on the product on predefined checkpoints. Without the inspections the programmers were dependent on the users to report bugs found after release. The user experience of the users could be unpleasant as they had to work with the bugs till a new version was released. In that time the machine costs were also more expensive than human hours and therefore the formal software inspection provides the best option for real-time feedback.

Fagan [57] describes in 1986 the development of the formal code inspection is adopted by many industrial companies. Throughout the time the process is refined by the many companies who have adopted the

process and the paper provides an overview of the results these companies achieved. Furthermore, it was found that 125 non commentary statements is the optimal amount per hour that can be checked during the inspection. In addition, Shull and Seaman [58] describes that it is no longer the question whether code inspections are effective or whether code inspections are effective in a particular environment. Sufficient research has been performed which proved the effectiveness of code inspections in different environments. Shull and Seaman [58] further explains the introduction of the code inspection process at NASA in the 80's. In the introduction process it was necessary to provide hard, quantitative and anecdotal data to convince the engineers. Once one or two prominent engineers were convinced by not only the amount of defects detected but also how quickly defects were detected and fixed other also adopted the code inspection process.

2.2. INFORMAL SOFTWARE INSPECTION

In 1993 Votta Jr [59] questioned whether every inspection needs a meeting. In which it are not the code inspections itself, however, the requirement of a meeting that is questioned. Votta showed that precious time is wasted scheduling the meetings and waiting for late comers at the meetings. In the inspections itself the moderator keeps the reviewers on track providing less space for distractions. Votta Jr [59] proposes three methods to make the inspections more efficient. The first method is structured meetings, although this is already well used in most inspection, the foremost change this requires is a culture change for the reviewers to be on time. The second option recommends an inspection with only 3 attendees in which the moderator also performs the role as recorder. This saves the time of a single person without increasing the time required for the remaining roles. Ideally the inspection is only held by 2 persons, the author and a reviewer, also omitting the moderator. This could be justified as most discussion in the meetings are held 1 on 1. The third and last recommendation is to not use a meeting at all and the reviewer can go solely over the changed code. The communication of the detected defects could be given verbal, handwritten or electronic.

The strict rules related to the collocated formal software inspection slowly faded away to a more lenient process, in which fewer participants are involved [42]. Furthermore, the open source software (OSS) projects changed the code review process in the way that the changes are broadcast via a mailing list to hundreds of potential stakeholders [60]. Rigby and Storey [60] explain the process of the broadcast based review process and the challenges that are faced. With the fast amount of patches it can be hard to determine what to review. This is the responsibility of the core developers and e-mail filters are used to find interesting patches. Hereby, it is possible that patches are ignored as too many patches are provided and limited time is available, however, this does not decrease the quality of the project. Where in the industry limited time and money is involved workarounds are encouraged, however in OSS projects there is no rush and every patch gets the needed attention to maintain the quality [60]. The patches that are ignored contain mostly feedback of no interest and no further time is spent on these patches. The type of errors addressed are mostly technical defects having clear guidelines. However, diplomatic skills can be used for project objectives or politics to direct the project. There is a risk exposed that too many stakeholders are involved in the the discussion, however, the individuals with a lower rank are given less attention. Lastly, the scalability of the e-mail based broadcast is guaranteed by multiple mailing lists, from general to very specific lists, bridging individuals in multiple lists and explicit requests to review a patch.

Rigby *et al.* [40] show the lessons learned from the broadcast based reviews in OSS projects compared to the traditional formal code reviews. They explain 5 lessons learned, 1 asynchronous reviews; the reviewers are no longer limited to the collocated reviews and the reviewers are more flexible to review. Passive readers could also benefit from the discussions held. 2 Frequent reviews; under the guise of the earlier the defect is detected the better [8] code reviews can be held more often and therefore defects are detected earlier. Additional, the feedback loop changed from several weeks to several days. 3 incremental reviews; the changes should be small and independent. The small changes allow a full review and less preparation time. Additional, the incremental changes allow the reviewers to keep an overview how the project evolved over time. 4 experienced reviewers; having reviewers who are already familiar with the code helps to understand the change better. Furthermore, developers who are also related to the source code share the common responsibility to keep the quality high. 5 empower expert reviewers; the reviewers should be able to determine by themselves what to review, this also leads to a natural load-balancing. The changes that remain unreviewed could be assigned to dedicated reviewers, however, this might also indicate issue related to the specific areas.

Sutherland and Venolia [61] analysed the review practices at Microsoft. Microsoft used the informal synchronous face to face meetings in 49% of the total reviews and the remaining 51% of the total reviews are asynchronous e-mail based reviews. They were interested in retention and the future usage of the review

data. They found that only 18% of the review data is made available of the synchronous reviews and 32% of the asynchronous reviews. There is a large log of mails, however finding and using the information is hard due to the poorly search system in the mail system. Looking further into the composition of an e-mail the patch is included as diff file. The initial mail contains a change list explaining the rationale behind each change. The follow up mails contain the discussion between the reviewers and the author. Finally, a new solution is proposed that is agreed on, rejected or postponed. Developers value mostly the design rationale when designing and implementing software. However, at Microsoft there is little retention of the design rationale discussed in the reviews and valuable information is lost.

2.3. MODERN CODE REVIEWS

Bacchelli and Bird [9] have conducted another research regarding code inspections at Microsoft. They thereby introduced the term *modern code reviews* for the lightweight code reviews. The process is earlier described by Rigby and Storey [60], a modern code review is an informal, tool based review and performed frequently. Although, tool based reviews were already used by several companies as Facebook [62] and Google [63]. Bacchelli and Bird [9] were the first to investigate the expectations, outcomes and challenges of these modern code reviews. Additional, upcoming coding hosting services such as GitHub and BitBucket provide a commonly used tool for pull based development for both closed software and open source projects [22]. Bacchelli and Bird [9] have found that discovering defects is still the main motivation for code reviews, however, additional benefits such as: knowledge transfer, team awareness and alternative solutions were also acknowledged. Furthermore, they discovered that understanding the code in the pull request is the key aspect for code reviews. Besides improving the understandability of a pull request they also recommend to automate parts of the process.

Rigby and Bird [64] compared three types of code reviews in different projects. The formal inspections, e-mail based review and contemporary (modern) peer reviews following a lightweight and tool based approach. Most project use RTC (review then commit) strategy, however CTR (commit then review) is used sometimes by trusted experienced developers in OSS. They discovered that the contemporary reviews have a lightweight and flexible process and is performed early and small changes are reviewed. Two reviewers is the optimal number of reviewers and a median of 3 to 4 comments are placed during a review. Furthermore, the contemporary review practice of defect finding changed to a problem solving activity.

2.3.1. CHALLENGES IN OPEN SOURCE SOFTWARE DEVELOPMENT

Gousios *et al.* [65, 66] looks into the role as contributor [66] and integrator [65] in modern code reviews on GitHub in OSS projects. They found that the pull based system is mainly used to perform code reviews and in lesser extend to discuss new features [65]. Whether a pull request is accepted in these OSS projects depends on the quality of the pull request. The quality is determined by whether the project guidelines are followed, the code quality itself and throughout the presence of tests. The challenges the integrators face are to maintain quality, feature isolation and the total volume [65]. The contributors face the challenges of writing the code, tools used and the testing environment [66]. Both perspectives face the challenge of the social aspect, reacting and receiving feedback in a timely manner without the other becoming inactive and reaching an consensus with each other without offending the other and demotivating them. In addition, Rigby and Bird [64] discovered that OSS projects have drive-by developers who commit to a few files and then leave. Furthermore, Gousios *et al.* [66] discovered the contributors expect transparency on what is going on, however, they themselves lack to provide the information for transparency or to look up and use the available information. This is reflected in the amount of duplicate pull requests which are encountered and rejected by the integrators [65].

Kononenko *et al.* [37] looks into how the code review quality is perceived in the open source project of Mozilla. They found that thoroughness of the feedback, familiarization with the code and the code quality itself are related to the review quality. The personal challenges the contributors face are keeping their technical skills up-to-date, manage personal priorities and dealing with context switches. Hereby the technical skills is mainly determined by getting familiar with the code base, complexity of the code and tool support.

The social impression of peers plays an important role in OSS peer reviews. Bosu and Carver [41] analysed the peer impression formation in code reviews via a survey. They discovered that code reviews have the most effect on the social impression formation in perception of expertise, reliability, friendship and trust. The four constructs are correlated with each other, so basically code reviews improve the social attitude in general towards the review partner.

2.3.2. CHALLENGES IN INDUSTRIAL SETTING

In the industrial setting MacLeod *et al.* [67] performed an internal analysis at Microsoft focused on the challenges in modern code reviews. They did that by observing four teams and conducting interviews and finally validated their findings through a survey. Developers recognized the value of reviews, appreciate the feedback and are more thorough once they know their work will be reviewed. There are no strict rules or polices used in the review process and can vary over the teams. They discovered that the developers face the following challenges: receiving and providing feedback in a timely manner, pull request size, time constraints, who to review, understanding the purpose and the motivation of the change.

In the research at Google, Sadowski *et al.* [35] found that the code review practice used is a lightweight and flexible process compared to the other modern code reviews. The reviews are done by a single reviewer instead of two in other processes, have quick iterations and smaller changes. In the review process ownership and readability are two aspects which are deemed important by Google. Google still faces challenges in the communication and unclear intention of the pull request. The distance between the author and the reviewer can delay communication both geographically and in teams. Additional, the tone of the comments and abuse of power can lead to friction between the developers in the code review process. The intention of the change is not always clear, for example several teams also perform design reviews while other teams have done the designing at forehand. Subsequently, a mismatch in the expectation of change can cause delay or frustration, there is a difference between a bug fix or a less important quality of life improvement. Furthermore, the developers have a highly desired wish for automation in the review process.

2.3.3. RESOLVING THE SIZE CHALLENGE

One of the main challenges in the modern code reviews is to deal with larger pull requests which are harder to assess and preferably the changes should be small and independent [9, 35, 37, 67]. In order to be able to deal better with larger pull requests Barnett *et al.* [68] introduced a tool called *"ClusterChanges"* separating the changes in the pull request into smaller independent changes. They were able to distinguish the changes based on the abstract-syntax tree provided by Roslyn. They marked each partition, a separate change, with a separate color in the code review tool. This allows the reviewer to understand each change separately without the distraction of the other changes. In the qualitative evaluation with 20 developers they discovered that the tool was rather useful.

2.3.4. MOTIVATION FOR CODE REVIEWS

Earlier we showed that the initial motivation for code reviews is to detect defects early on [8, 69]. In a later study Bacchelli and Bird [9] discovered besides detecting defects code reviews provide additional benefits: knowledge transfer, team awareness and alternative solutions. However, code reviews were introduced at Google to encourage the developers to write understandable code for the other developers [35]. During the practice of code reviews they discovered that detecting bugs is an additional benefit together with the educational value, consistency in the code base, adequate tests and security. The current motivation behind code reviews focuses on education, maintaining norms, gatekeeping and accident prevention. Furthermore, MacLeod *et al.* [67] discovered that the top 3 motivations for code reviews at Microsoft are: code improvements, detect defects and knowledge transfer.

2.3.5. ACTUAL DEFECTS DISCOVERED

Mäntylä and Lassenius [39] are one of the first researchers who looked into the actual types of defects discovered in code reviews instead of the amount of defects discovered. They discovered that 71.1%, industrial reviews, and 77.4%, student reviews, of the comments are evolvability defects. Additional, 21.4% and 13.2% are functional defects respectively and the left over comments were false positives. Hereby they state that the usage of testing as a quality control is biased as it does not look into the evolvability of the code. Similar Beller *et al.* [10] retrieved a ratio of 75:25 for maintainability defects against functional defects in OSS. Furthermore, they discovered that 7 to 35% of the comments are discarded and 10 to 22% of the changes are not triggered by a comment. However, it is possible a change was a result of face to face communication.

2.3.6. USEFULNESS AND UNDERSTANDING CODE REVIEWS

Bosu *et al.* [36] looked into what makes a comment useful and created based on their findings a classification model. They discovered based on interviews that comments addressing functional issues are most useful. Additional, code reviews are useful for new developers to get familiar with the project, APIs and used tool-

ing. Somewhat useful comments are nitpicking comments, they are not considered essential, however, it increase the quality over time. Furthermore, proposing alternative solution increasing the understandability of the code is also considered somewhat useful. False positives and comments out of scope are considered not useful. In their classification model Bosu *et al.* [36] included the following characteristics to determine the usefulness of a comment: a change happened after a comment, status of the comment, contribution of the author to the thread and specific keywords. Based on the result of their classifier they discovered that reviewers with prior experience give more useful feedback. This reflects in that in the first year of a developer the usefulness of her comments increases, however, this evens out after a year of experience.

Tao *et al.* [70] explored which information is needed in order to understand a code change. They discovered that the information needed is to understand the rationale behind the change to assess the quality of the change in terms of: completeness, correctness and design issues, and the risks the change exposes. The hardest part regarding the information gathering is consistency in the code base, does the change need to be applied elsewhere for a homologous code base. In order to provide this information developers should better support the risks of a change and decomposing a composite change, in other words smaller stand alone changes which are easier to understand. Understanding the rationale behind a change is rather easy through the commit logs and descriptions given. However, the developers face the challenge which risks the change exposes, these risks could either be minimized by testing and code reviewing. Whereby both methods have their own limitations in which testing is labour intensive and limited to existing test cases and code reviewing is error prone and do not have IDE features. In addition, static analyse tools could be used to assist the developers in the review process.

2.3.7. IDENTIFY DEFECT PRONE AREAS OF CODE

Staron *et al.* [71] analysed the development rate of code components to the amount of defects reported in the code base. They visualized the code change churn of different components in a heatmap over a period of 6 months. Based on the heatmaps code components were labeled as stable or unstable in terms of development. In addition, they linked the amount of reported defects in the whole code base to the changes in the heatmap. In some occasions they were even able to distinguish the defects reported to the specific components itself. Combining the information of the code change rate and the reported defects they were able to identify patterns in the development rate which leads to an increase in the amount of reported defects in the components. Based on these findings they created a warning system which assists the developers to identify oncoming batches of defects in the development of the components.

Although the previous study did not involve code reviews, information related to code reviews could also be used to detect defect prone areas of code. So did Thongtanunam *et al.* [72] investigate the code review activity in files that have had previous defects and possible future defects in the open source system of Qt. They showed that defective files undergo lax code reviews, this is expressed in less iterations, fever reviewers, faster review rate, shorter discussions and more revisions that are not inspired by reviewer feedback and the files tend to change more often. Risky files, files that included defects in history, still tend to undergo less intense code reviews similar to future defective files. In addition, they recommend to use multiple reviewers as this decreases the chance of future defects. This is based on Linus' law *"Given enough eyeballs, all bugs are shallow"* [73], in other words enough reviewers every bug is discovered and solved early on. Furthermore, they showed that the defects addressed in 82% of the defects in future defective files and 70% in clean files are mostly evolvability defects.

Similar McIntosh *et al.* [74] studied the relationship between code review coverage and review participation against the amount of post release defects found. Although components having a higher review coverage tend to have fever post release defects, there are also components with a high coverage rate having a higher amount of defects. This indicates that other factors besides the review coverage are as well at play in the amount of defects found. Furthermore, the lack of participation in a code review indicates a negative impact to the code quality. The participation in a code review is measured by self approved requests, enough time spend on the review and lack of comments and thereby discussion in the review, in which the latter is especially important for integration pull requests.

2.3.8. CHECKLISTS

In the earliest paper [8] on code reviews the usage of a checklist was already mentioned. Checklists were used as reference material consisting of clues hinting to detecting defects. A checklist should be specially created based on the needs of the project and based on the defects which are often detected in earlier inspections. Fagan [8] also indicates that the checklist should be updated regularly throughout time based on the insights of

the newer inspections.

In addition, Gilb and Graham [69] also indicates that the usage of a checklist is a fundamental part in the formal inspections. Furthermore, Gilb and Graham [69] explains the following characteristics to the checklist. The checklist should be build based on the process evaluated, meaning each project requires its own checklist. The items in the checklist should be formulated as question and should reflect the frequent defects detected in inspections. The length should not exceed a single A4 paper, this is equal to roughly 25 items. To detect the most important defects in the software the checklist should focus on the major defects. A checklist should not be copied from other projects as they might not address the major defects in the current project. The checklist should be kept updated based on the newest findings in the inspections. The checklist is owned by a single person which is the only one who is allowed to make direct changes to the checklist. Others can provide their updates in terms of improvements to the checklist owner. Furthermore, the goal of the checklist is to train the reviewers in what is expected in a formal inspection, provoke the reviewers to look into issues that were otherwise not found, increase the number of defects detected and can provide explanation of rules used, however, the items should not introduce (new) formal definition of these rules.

Halling *et al.* [42] analysed checklist based reading against scenario based reading focusing on the individual defect finding reviews. Checklist based reading does not define how to perform an inspection, however, it provides aspects to consider during inspections. A single checklist is often used within a team by all inspectors which is not always suitable for each role in the project. Scenario based reading provides scenarios as guidance to find defects, it is a process which can be repeated. Checklist reading was found more effective in all situations in detecting the highest amount of defects, either during individual and team inspections.

Hatton [43] analysed the effectiveness of checklists based on 308 code inspections on a 62 lines of code C program in workshops for industrial engineers. Their result is that no evidence was found that the usage of the checklist had any influence on the amount of defects found in their specific environment. In addition, further analyse by Hatton [43] revealed that two reviewers found more defects in code reviews, around a quarter, than a single reviewer.

Other literature provides additional insight in the usage of checklist. For example, Mäntylä and Lassenius [39] analysed the defects found in code reviews and based on the types of defects found a checklist could be made. In which the different types of defects found in different programming languages should be considered, for example the Java language has less memory issues than C(++) languages. Furthermore, Rigby *et al.* [40] discovered that code reviews in open source projects advised checklists and readings techniques which can help reviewers to focus during inspections.

3
METHODOLOGY

In the introduction we have listed 4 research questions to analyze and improve Greenchoice's code review process. In this chapter we will explain the steps and methods used to answers the 4 research questions. To answer the first research question we explain in Section 3.1 how we discovered the motivations for code reviews at Greenchoice through the usage of a questionnaire. In Section 3.1.1 we first explain the introduction of the questionnaire and 5 demographics question to get to know the respondents. Next, in Sections 3.1.2 and 3.1.3 we describe how got to the motivations of the respondents and how we compared the different motivation categories. Finally, in Sections 3.1.4 and 3.1.5 we describe how we gauged the initial thoughts on the code reviews process with 10 statements and asked for an in practice example of the usefulness of code reviews.

Next, in Section 3.2 we describe how we explored the current code review process in detail at Greenchoice to answer the second research question. The second research question consists of 3 sub questions which uses each a separate method to discover a different view on the current code review process. The first method is a qualitative study on the process through interviewing Greenchoice's developers, second and third methods are both quantitative data analyses either looking into the metadata or the code review comment data of the pull requests and code reviews. These methods are further explained in the Sections 3.2.1, 3.2.2 and 3.2.3 respectively.

Subsequently, in Section 3.3 we explain how we discovered the factors needed to create a checklist in order to answer the third research question. In Section 3.3.1 we first present our expectations and intentions of the checklist. Next, in Section 3.3.2 we describe how we explored in more detail the code review process of a dedicated scrum team and found its strengths and weaknesses. Subsequently, how we explored in collaboration with the dedicated scrum team the factors which are deemed important and which factors are deemed unnecessary in the checklist and eventually how we created the checklist.

Finally, in Section 3.4 we explain the pre and post questionnaire evaluation design used to analyse the usage of the checklist during code reviews in order to answer the fourth research question. In Section 3.4.1 we first explain the general set-up of the experiment. The pre and post questionnaires used to evaluate the experiment are respectively explained in Sections 3.4.2 and 3.4.3.

3.1. INITIAL CODE REVIEW QUESTIONNAIRE

In order to discover the motivations for the code review process at Greenchoice we used a questionnaire. With the motivation known we know the intention of the process and we get an thought on where to improve on. We choose for a questionnaire as it provides a fast method in gathering data from a group of respondents. We are interested in the opinion of all employees in the software development department as they are all directly or indirectly involved with code reviews. Therefore, the questionnaire was sent to the whole IT development department. The IT development department had, at the moment of the questionnaire, a total of 58 employees consisting of developers, testers, business consultants (BC), product owners (PO), specialists (e.g. architects and database managers) and managers. The complete questionnaire consisting of 6 sections can be found in Appendix A.

Besides discovering the motivations we also included several demographics questions explained in Section 3.1.1. Next, in Section 3.1.2 we explain how we actually discovered with the questionnaire the motiva-

tions for the code review process and the categories used to group the answers. Subsequently, we explain in Section 3.1.3 in detail the method used to compare the different motivation categories. Furthermore, in Section 3.1.4 we describe how we used 10 statements to discover the first thoughts towards code reviews. Lastly, in Section 3.1.5 we explain the last questions of the questionnaire in which we asked for an in practice example of a defect found during code reviews.

3.1.1. QUESTIONNAIRE INTRODUCTION

The first section of the questionnaire consists of a brief introduction on the questionnaire followed by five demographic questions to get to know the respondents. In the introduction we ask the respondents to answer the questionnaire as completely as possible. Aimed in particular to the roles who are not actively involved in the code reviews process. Some questions are hard or cannot be answered without participating in code reviews. However, we still want to include their answers as much as possible to get better understanding of these roles and get a complete view of their involvement in the code review process. Based on the description in the introduction we expect the developers to be the most active in pull requests followed by the testers and specialist, whereby the business consultants, product owners and managers are less to not active in code reviews.

The demographics questions comprises the function, familiarization, frequency and roles with code reviews. The demographics questions give an insight into the background of the respondents and a better understanding of the composition of the department. Additional, we will be able to differentiate the answers by function or experience.

3.1.2. DISCOVERING THE MOTIVATION FOR CODE REVIEWS

Continuing the questionnaire we asked the respondent to give three motivations for code reviews ranked by priority. This is followed by an optional section to give at most 5 additional motivations. The respondents had to fill in the motivations in open answer fields, in other words no predefined motivations were given to the respondents.

The given answers were initially classified in categories based on a subset of 7 categories derived from earlier research by Bacchelli and Bird [9] and Sadowski *et al.* [35]. When a given motivation did not fit into one of these 7 categories, a new category was introduced or assigned to an already newly created category. Once all motivations were classified into a category we reevaluated each motivation. During the reevaluation we looked whether a motivation fitted better into another category either a newly created or existing category.

When a category lacked answers, 5 answers or less, the category was merged, if possible, with another closely related category. This could happen due to the possibility that a new category was introduced for a specific answer and eventually no other answers fitted into that category. Likewise, the category could be part of the initial subset of categories from the literature and only a few given answers were found in our results. In the explanation of the categories we will mention which categories have been merged.

Furthermore, when the given motivation could fit into two or multiple categories the first mentioned category was chosen. In practice this means that *"Compliance with architecture and coding guidelines"* is categorized as *according to architecture.*

MOTIVATION CATEGORIES

The given motivations were in the end classified over 11 different categories. Of these categories 7 categories were based on the literature, these categories are explained in the first part. After that, the 4 other categories which were introduced by us based on the given answers are explained. The categories based on the literature have been adapted to better reflect the needs of the given answers.

The categories based on the literature are either described in our own words based on the definition used in the literature. Others are partly based on one of the categories described in the literature and we had redefine the categories ourselves to better suit the given answers.

LITERATURE CATEGORIES

1. **4 Eyes principle:** A category partly based on the security motivation at Google [35], one of three additional benefits from the initial motivations at Google: *"Improving security by making sure no single developer can commit arbitrary code without oversight"*. We define this category in our words as following; everyone can make a mistake and therefore deserves a review from a colleague. With the second pair of eyes performing a sanity check the reviewer prevents mistakes that can harm the system, such as a missing or additional file, or a potential functional issue that the author looked over. Furthermore,

working in a finance company and with personal data, the sanity check should prevent that a single developer can singly commit fraud within the code base. In other words, there is an acknowledgement from a colleague that the work done in a pull request is proper.

2. **Discussing solution:** This category is based on *"Alternative solution"* described by Bacchelli and Bird [9]. During a code review it is the last moment to consider if the proposed solution is the best solution. This offers the possibility for a discussion whether or not the proposed solution is good enough or a better, possibly a more simplistic solution is available.

3. **Following code guidelines:** This category is equal to *"maintaining norms"*, one of the current motivations at Google [35]. Greenchoice itself does not specify many coding guidelines; it relies mainly on general coding principles, the language conventions, skills of the developers and the agreements within the scrum teams. During a code review the reviewer checks if the delivered code follows these desired coding guidelines.

4. **Knowledge sharing:** In the code review process both the reviewer as the author learns from the code review. The author gets feedback on his work and learns from this feedback. The reviewer learns from the code and sees how the author solved the problem, the solution can be later applied by the reviewer facing a similar problem when programming. At Google [35] education was the main reasons to start code reviews and still is. At Microsoft [9] the learning aspect is also an important motivation for code reviews.

5. **Preventing defects:** This category represent the main motivation for the first systematic code reviews introduced by Fagan [8]. According to Fagan [8] finding and solving functional defects early on minimizes the cost and increases the productivity. In addition, the user encounters less defects in the software increasing the productivity of the user. Later Beller *et al.* [10] showed that only 25% of the comments are functional defects, another study showed similar results [39]. The category is similar to *"Finding defects"* at Microsoft [9]. At Google [35] finding bugs is considered as an additional benefit rather than a motivation.

6. **Team awareness and ownership:** This is a combination of *"team awareness and transparency"* and *"share code ownership"* at Microsoft [9]. Using pull requests provide other developers who are not actively involved in the pull request to read the pull requests and get aware of the change. Although the developers is not actively involved in the code change, she is aware of the change and can adopt her current or future work based on the change. With ownership not only the author is responsible for the code, also the reviewer gains an understanding of the change and should be able to adapt it in the future. This spreads the knowledge within the team and decreases knowledge loss when a developer leaves the company, reducing the bus factor [75].

7. **Understandable and maintainable code:** This category is based on one of the foremost reasons for code reviews at Google [35] *"improve code understandability and maintainability."*. We define understandable as; can the reviewer go quickly through the code, is it easy to understand, does it have clear names and no strange indentations or white lines. Furthermore we define maintainable as; whether the code in the change is easy to maintain in the future, does the change not contain too complex code or other code smells.

NEWLY INTRODUCED CATEGORIES

8. **According to architecture:** Within Greenchoice there are specific architectural requirements which should be taken into account when writing the code, for example the usage of the different services and the communication rules between the different services. With code reviews these architectural requirements are checked before releasing the code to production. Preventing that additional code is written on top, making it harder to refactor the code in the future.

9. **Consistency between teams:** Each scrum team is responsible for their own method of coding, consisting of: problem solving, use of design patterns and coding style. Code reviews can be part in learning and getting familiar with the standards of the team. The learning part is closely related to *knowledge sharing* category 4 of the earlier described literature categories. The coding style part is related to *following code guidelines* category 3, these are mostly small improvements based on a few lines of code.

Functional	Quality	Social
4 Eyes principle	According to architecture	Consistency between teams
Verifying behaviour	Following code guidelines	Discussing solutions
Preventing defects	Quality	Knowledge sharing
	Understandable and maintainable code	Team awareness and ownership

Table 3.1: Division of the specific motivation categories over the global categories.

We doubted whether we had to merge this category with the related categories, however the given motivations in *consistency between teams* do not specifically mention code guidelines or learning from each other.

10. **Quality:** A new category with a broad definition covering the quality of the request, in most cases this covers the code quality. The answers related to the code quality could not be placed into one of the more specific code quality categories *following code guidelines* and *understandable and maintainable code* due to the lack of specification in the answer. The category can be compared to *"Code improvement"* found by Bacchelli and Bird [9], however the goal of the *quality* category is in our interpretation to guarantee or control the code quality over actually improving the code quality.

11. **Verifying behaviour:** This definition is similar to *preventing defects* category 5 of the earlier described literature categories. Instead of preventing defects the delivered code is verified to see whether or not the code does what it is supposed to do. There is checked whether the code passes the acceptance criteria in the PBI and fulfills the additional requirements.

GLOBAL CATEGORIES

In the classification of the given answers into the 11 categories described above, further referred as specific categories, we noted that some categories are closely related to each other or have a common aspects. For example, multiple specific categories affect the quality of the code: *quality*, *understandable and maintainable code* and *following code guidelines*. To get a generalised understanding of the different specific categories we introduced three global categories; *functional*, *code quality* and *social*.

1. **Functional:** This are the specific categories having the main goal to improve or check the functionality of the code. This global category consists of *4 eyes principle*, *preventing defects* and *verifying behaviour*.

2. **Code quality:** This are the specific categories aiming to improve or control the code quality. This global category includes *following code guidelines*, *understandable and maintainable code*, *according to architecture* and *quality*.

3. **Social:** Covers the social aspects of the code reviews consisting of learning from code reviews *knowledge transfer*, being responsible as a team for the code base *team awareness and ownership*, finding the best solution through discussion *discussing solution* and developing in a similar method *consistency between teams*

The distribution of how the 11 categories are divided over the global categories is also show in Table 3.1. Most specific categories were naturally grouped into the global categories. However, *consistency between teams* could be part of both global categories *quality* and *social*, we ended up grouping it in the global category *social*. Achieving consistency between the teams on how to code is part of learning from each other which fits to the global category *social*. The fact that learning from each other also leads to improvement in code quality we considered as an additional benefit.

3.1.3. COMPARING THE DIFFERENT CATEGORIES

We have introduced 3 different methods, listed and explained further below, in order to compare the 11 categories of motivations. The introduction of the priority system allows us to compare the categories not only on the quantitative factor but also on the priority factor. To be able to differentiate the results based on these factors we introduced for both factors its own comparison method. Furthermore, we introduced a method combining both factors. Thus each method focuses on either the quantitative or priority factor, or the combination of both factors. By analyzing the results of these methods we can determine the most important category of motivations for code reviews at Greenchoice.

1.	I find code reviews an important factor in software development
2.	I think the effect of code reviews is visible
3.	I think enough attention is paid to code reviews
4.	I think I have enough time to perform code reviews
5.	I enjoy performing code reviews
6.	I find it difficult to perform code reviews
7.	I think I give useful feedback during code reviews
8.	I enjoy receiving feedback through code reviews
9.	I find it difficult to receive feedback through code reviews
10.	I think I get useful feedback through code reviews

Table 3.2: Ten statements used in the questionnaire to get to know the thoughts of the respondents regarding the code review process.

1. **Count of categories:** Simply counting the total number of occurrences of that category. This gives an overview of the quantitative factor of a category, however it does not take the priority factor into account.

2. **Weighted count of categories:** Assigning a different weight for each priority allows to distinguish in level of importance, the higher the priority the larger the weight. Together with counting the number of occurrences this method uses both the quantitative as the priority factor. The used equation is shown in Equation 3.1. Where *#priorityx* is the amount of occurrences of that category in the specific priority level x, and *#optional* is the amount of occurrences in the optional motivation fields. Priority 1 has the weight of 1, priority 2 weight of 0.75, priority 3 weight of 0.5 and the optional fields 0.25.

$$\text{weighted count} = \#\text{priority1} + 0.75 * \#\text{priority2} + 0.5 * \#\text{priority3} + 0.25 * \#\text{optional} \tag{3.1}$$

3. **Average weight:** This method reuses the values of the previous comparison methods, it divides the weighted count by the count, shown in Equation 3.2. This rules out the quantitative factor and uses only the priority factor, being the opposite of the first comparison. This comparison shows the difference between a category that occurs frequently with a lower priority and vice versa a category that occurs rarely with a higher priority.

$$\text{Average weight} = \frac{\text{weighted count}}{\text{count}} \tag{3.2}$$

3.1.4. DISCOVERING THE INITIAL THOUGHTS WITH STATEMENTS

The fourth section of the questionnaire contains ten statements with respect to the code review process, aiming to find the effects and satisfaction of the process. In addition, the results of these statements exposes the initial thoughts of the respondents which can be used as discussion material in the interviews, which are further explained in Section 3.2.1. For the statement we used a 4 point Likert scale [76] to force the respondent into a positive or negative direction. Though, we included an opt out option as the statements are not applicable by all the roles. Additionally, we added for each statement a text field in which the respondent could explain his or her reasoning. The used statements can be found in Table 3.2.

The first four questions address the importance, visibility of the effect, attention paid and time availability of the code review process. In questions 5 to 7 the enjoyment performing code reviews, difficulty of performing code reviews and usefulness of given feedback is gauged. The last three questions are similar, though in the position of the author of the code review. Thus enjoyment receiving feedback through code reviews, difficulty of receiving feedback through code reviews and usefulness of retrieved feedback.

3.1.5. ERROR PREVENTION IN PRACTICE

The fifth and penultimate section of the questionnaire challenges the respondent to think about the past effects of code reviews. We have asked for a bug or code smell which was caught during a code review.

Finally, in the last section we have asked the respondents for their e-mail address in order to contact the employee when needed. This was optional as it would break the respondent's anonymity. Additional, we gave the respondents the possibility to give their final remark.

3.2. EXPLORING GREENCHOICE'S CURRENT CODE REVIEW PROCESS

To get a better understanding where to improve on in the code review process we aimed to get a deeper insight of the current code review process. For this, we have used various methods to explore the current process at Greenchoice. First off, several initial meetings with the software architects were held learning the basics of Greenchoice's development process in general. Secondly, the results of the questionnaire gave insight into Greenchoice's motivations for performing code reviews. Additionally, we have interviewed one or two developers from each scrum team, in total we held 13 interviews, to get a further and deeper understanding of the code review process in practice. The set-up of the interviews is further described in Section 3.2.1. Furthermore, we made a crawler to gather the data regarding the pull requests and code reviews on DevOps. This data was split over a meta data analysis, further described in Section 3.2.2, and code review comment data analysis, further described in Section 3.2.3. The results of these analyses are compared to the findings of the interviews and state of the art. Finally, we kept our eyes and ears open at the workplace for discussions regarding the code review process.

3.2.1. INTERVIEWING THE DEVELOPERS

To get a better understanding of how code reviews are performed in practice we interviewed one or two developers of each scrum team consisting of 3 to 4 developers. We have chosen to only interview developers because they are the ones the most involved with code reviews. They have the most experience with code reviews and can provide us with the most information regarding the process. We used semi-structured interviews [77] based on three general questions we wanted to specifically address in the interviews. We chose the semi-structured approach as we wanted to discover the free thoughts of the developers concerning code reviews. Furthermore, each developer may have his own reasoning requiring different questions to get into the details. We aimed for a duration of 45 minutes to an hour for each interview. The interviews were based on the following three question:

1. How does the current code review process look like, both in reviewer and coder perspective?
2. What are the strengths and weaknesses of the code review process?
3. What can be improved in the code review process and how?

At the beginning of the interview the idea behind the interview and the three questions were explained to the interviewee. After the explanation, the first question was asked once more and it was up to the interviewee from which perspective he or she wanted to start. Based on the explanation of the interviewee additional questions were asked to reveal the strengths and weaknesses of the process. With the strengths and weaknesses exposed, the improvements could be deduced in a natural flow. However, in case there was no natural continuation, we have asked the second and third questions literally to keep the interview going.

3.2.2. META DATA ANALYSIS

The analysis of the data is divided over two analyses in here we explain the first analysis covering the meta data related to the pull requests and code reviews. The metadata of the pull requests and code reviews consists of 35 features, shown in Table 3.3, representing the characteristics of the code reviews. With the meta data analysis we try to expose hidden strengths and weaknesses in the data, in particular related to the time and size features of the pull requests. Additional, the findings of the meta data analysis can support the earlier findings in the interviews. First we will briefly explain how we gathered the data through the crawler we created.

Via the DevOps REST API [78] the data regarding the Git repositories is available via GET calls. Part of the data exposed is the data regarding pull request and code reviews. To specifically information related to the changed code in a pull request a specific POST call [79] should be used, this POST call is not officially released in the DevOps API. We created a C# MVC application making the necessary GET and POST calls to gather the data and processed the data and eventually create two files. The first file contains the metadata regarding the pull requests and code reviews and the second file contains the comment data used in the comment data analysis, explained in Section 3.2.3.

The retrieved meta data file was further analysed in RStudio [80]. In the analysis we first explored the general features as shown in Table 3.3. The analysis of the general data starts with a look into the evolution of Greenchoice's code review process to get a general idea how Greenchoice adopted their current code review process. The evolution over time of the code reviews is expressed through the amount of pull requests

per month and the corresponding amount of unique authors and projects for the corresponding month. To visualise this data we used a time series graph of the *ggplot2* library [81]. In addition, we looked into the average pull requests per author and project how these values evolved over time, once more, in a time series graph. This gives us a clearer insight whether the growth or decrease in amount of pull requests is due to an increase or decrease in the amount of authors, projects or other unknown factors. An example of calculating the monthly average pull requests per author for the month January is shown in Equation 3.3, for the other months or the projects a similar calculation is performed.

$$\text{average pull requests per author}_{January} = \frac{\text{amount of pull request}_{January}}{\text{unique authors}_{January}} \tag{3.3}$$

Next, we looked into the guidance given in the pull requests for the code reviewers. We did this by analysing the title and description of each pull request. In DevOps the default title and description are generated based on the latest commit and are by default similar. In order to get an idea of the guidance given in the pull requests we looked into the amount of empty descriptions and descriptions similar to the title, indicating no additional guidance was given. Furthermore, we looked into the average amount of committers, reviewers and approvers. The literature found 1 to 2 reviewers are optimal in modern code reviews [35, 43].

The remaining part of the analysis consisted of analysing the features of the three main aspects of code reviews: time, size and reviewer interaction. The time aspect consists of 6 features of which we mainly used the 4 features: TimeToOpen, TimeToReview, TimeToClosed and CycleTime. TimeToOpen starts at the last commit before creating the pull request and ends once the pull request is created, TimeToReview starts from that point till the first vote is given and TimeToClosed from that point till the actual merge of the pull request. CycleTime is the total time from the start, the last commit, till the end, the actual merge, and is therefore marked bold in Table 3.3.

Similar for the size aspect the features NumTotalChangedFiles and NumTotalChangedLines are marked bold as they are a total of their sub features. The feature NumTotalChangedFiles can be broken up over the following broken features: NumAddedFiles, NumDeletedFiles, NumEditedFiles, NumRenamedFiles, NumRenamedAndEditedFiles. The feature NumTotalChangedLines can be broken up over the following features: NumAddedLines, NumDeletedLines, NumModifiedLinesOld and NumModifiedLinesNew. Most feature names are self explanatory the latter two however require more explanation. Changing a line requires the old line to be deleted (NumModifiedLinesOld) and a new line to be added (NumModifiedLinesNew). This means a modification in lines are counted twice compared to addition and deletion. It also possible a single line is removed and replaced by two lines or vice versa giving a total of 3 changes lines. We did decide to not change the possibility that a modification is counted twice as the reviewer needs to check whether the removal of the old lines is correct and that new lines are correctly added.

In the review interaction analyses we used the following features: NumUserThreads, NumUserComments, NumIterations, HasCommitBeforeReview and HasCommitAfterReview. The first two features are threads and comments created by a reviewer. A thread consists of a single or multiple comments reacting on each other, within DevOps it is not possible to have multiple threads within a thread. The features, NumUserThreads and NumUserComments were filtered from the total amount of threads and comments in the pull request as the system and a bot also posted comments and thereby creating threads. The third feature, NumIterations, is the amount of pushes to the branch once the pull request is created, starting at 1. The latter two features are whether or not the author has edited the pull request before or after review interaction, either a vote or comment.

The analysis of the time and size aspects are especially interesting as modern code reviews are fast, small and frequent [9]. In the time analysis we are curious how the CycleTime can be explained by its sub features similar to the research performed by Davis [82]. Additional, we are interested how the total size can be explained by their sub features. Subsequently, we look into whether we can find evidence that larger requests or other specific types of pull request require more time. Furthermore, we interested whether larger or other specific requests affect the reviewer interaction. In other words, whether larger pull requests have a higher amount of code review comments given and whether they require more iterations. Finally, we looked also into the relations of the specific features to each other to find hidden relationships.

In general we first explored the separate features of each aspect. To get a better understanding of the distribution of the features we used 4 plots: histogram, density plot, Q-Q plot and box plot. Based on the findings in the exploration plots we zoomed into the denser parts of the data. This, to get a better understanding how these parts of the data are distributed themselves. To compare the features to each other and to detect a possible relationship we used scatter plot matrices together with the Spearman correlation. We used Spearman

General	Time	Size	User Interaction
Project	OpenDate	NumAddedFiles	NumThreads
Repository	CloseDate	NumDeletedFiles	NumUserThreads
RequestId	TimeToOpen	NumEditedFiles	NumComments
Title	TimeToReview	NumRenamedFiles	NumUserComments
Description	TimeToClosed	NumRenamedAndEditedFiles	NumIterations
AuthorId	**CycleTime**	NumBinaryFiles	HasCommitAfterReview
NumCommitters		**NumTotalChangedFiles**	HasCommitBeforeReview
NumReviewers		NumAddedLines	
NumApprovers		NumDeletedLines	
IsMerged		NumModifiedLinesNew	
		NumModifiedLinesOld	
		NumTotalChangedlines	

Table 3.3: Overview of the features regarding the code reviews retrieved by the data crawler from DevOps.

correlation as the data used does not fit the normal distribution, discovered in the exploration of the data in Section 5.2, and may still include outliers [83]. The Spearman correlation is denoted in APA style as following $r_s[123] = 0.65, p < 0.001$, in which 123 is degree of freedom [84].

In case of the time features we also looked at the reversed percentiles of the features. This gives the percentage of pull requests with a value up to the given value. We reversed the *quantile* function to calculate the percentile values corresponding to a concrete value, for example the percentage of pull requests that are merged within the first hour, via the *ecdf* function in R. We looked into the percentage of pull requests based on the following times: 1 minute, 15 minutes, 30 minutes, 1 hour, 24 hours, 48 hours, 168 hours (7 days) and 672 hours (28 days).

In case of the reviewer interaction we also looked into how many pull requests do contain a code review comment. Subsequently, in how many pull requests the given feedback is actually processed based on whether the pull request has an additional iteration after the feedback is given. Furthermore, we discovered that in practice code reviews are also performed face to face which means this information is not included in the data. However, based on the amount of iterations in a pull requests we tried to estimate the possible amount of pull request triggered by face to face feedback.

3.2.3. Comment data analysis

In the second data analysis we looked into the types of comments given during code reviews. We are interested to find what types of comments actually given during code reviews as earlier research showed that only 20% to 25% of the comments actually addresses functional defects [10, 39]. Additional, by exploring the types of comments we get an insight in the types of comments we might want to pay more or less attention to in the checklist.

At forehand, we created a small list of types of comments we based on the literature [10, 39]. We did not use the full list of categories of the literature as some of the categories became too specific and different programming languages were used requiring other categories. We therefore intended to create the list of categories based on the observation of the comments throughout the classification. We added a new categories if a comment did not fit in one of the existing categories. We ended up with a total of 33 types of categories. The full list of these 33 categories can be found in Figure 3.1.

The categories were classified in multiple batches over time. In each classification we gained new insights on how to classify the comments which lead to inconsistency in the classification. Either we forgot a previously introduced category and assigned the comment to another more general category. Vice versa we could have introduced a category for a comment we previously classified in the more general category and forgot to update the already categorized comments. To overcome the inconsistency and still get a general idea of the distribution of the comments we introduced roughly halfway throughout the classification the following 11 global categories.

1. **Cleanliness and structure:** this category covers comments related to addressing the cleanliness of the code. This means: white spacing, blank lines, indentation and commented code. Additional, the correct usage of CamelCase and snake_slang in the naming of variables and spelling mistakes in the naming.

2. **Clarity and understandability:** includes comments related to the comprehensibility of the written code. In practice, this means the reviewer asking the author for explanation what he or she has done and vice verse the author explaining what he or she has done. In terms of code related comments this includes changes in the naming of variables a more appropriate name for the variable, notice comments regarding the spelling mistakes are part of the previous category.
3. **Code improvements:** this are small improvements to the code, for example the usage of LINQ query, similar to lambda expression or SQL query [85], over foreach loop, or logic that can be replaced with a native method. Also, reversing an if statement to minimize the amount of code lines or improve performance.
4. **OOP concepts:** this are the larger improvements based on the OOP concepts, this means for example duplication and misplacement of code and separation of responsibilities. Furthermore, the usage of key words in the language for example public and private and the other access modifiers. Lastly, it also includes the inheritance of the code and the coupled usage of parameters in the methods.
5. **Architecture:** includes comments related how the different parts of the code should be build, on how these parts should depend on each other. This means how two different services should communicate to each other.
6. **Functional bugs:** are as the name suggest errors leading to unintended behaviour of the program. This also includes comments mentioning the pull request is incomplete, in other words the functional requirements of the PBI are not yet fulfilled.
7. **Test cases:** includes all comments related to the presence of tests, approach of the test and quality of the test cases. However, this does not include the code quality within the test cases itself.
8. **Code security and monitoring:** this are comments related to preventing exceptions and correct handling of those exceptions, for example correct handling of null values. In addition, the category includes comments related to log messages given throughout the code.
9. **Project set-up and configurations:** are comments related to the set-up of the project such as the project file and the correct inclusion of dependencies. In addition, the settings or build files and the usage of environmental variables to assess other services like a database.
10. **Other communication:** this are comment which are not directly related to the code in the pull request. This could also be described as smalltalk or chit-chat. In practice we also included *"fixed"* comments to this category. This are comments describing the issue is solved.
11. **Unknown:** this are the comments we could not fit into one of the other categories.

The introduction of these 11 categories prevented us to go over the already classified categories. Additional, the first 9 categories will reoccur later in the analysis of the effectiveness of the checklist as aspects, further explained in Section 3.4. We were able to divide the initial assigned categories, further called specific categories, to a global category, shown in Figure 3.1. However, we had three specific categories, *naming*, *consistency in code style* and *dead code*, that were assigned to two global categories. In case of the comments in *naming* we went, directly after the introduction of the global categories, over each of these comments and reclassified them to the corresponding global category. In case of enhancing the understandability of the variable name we assigned the comment to *clarity and understandability* and in case of spelling mistakes or wrong usage of capitalization we assigned the comment to *cleanliness and structure*.

We discovered later that we also assigned *consistency in code style* to *Code improvement* and *OOP concepts*, and *dead code* to *cleanliness and structure* and *project set-up and configurations*, however at that moment we had no longer access to the pull requests and could not reclassify the comments based solely on the comment text. Therefore, we kept the results as they were and treated these results with a caution that the distribution might be different. In addition, we still tried to use the specific categories, however with the inconsistency these results should be also handled with a caution. The more specialized categories, we specifically mention this in the results, we treated with the caution *at least x* occurrences and in practice could be more, this means that more general and more frequent used categories would have a lower amount of comments.

In addition, we did not assign three specific categories to global category yet, this are: *false positives*, *missing functionality* and *proposal of another solution*. With the introduction of the global categories we

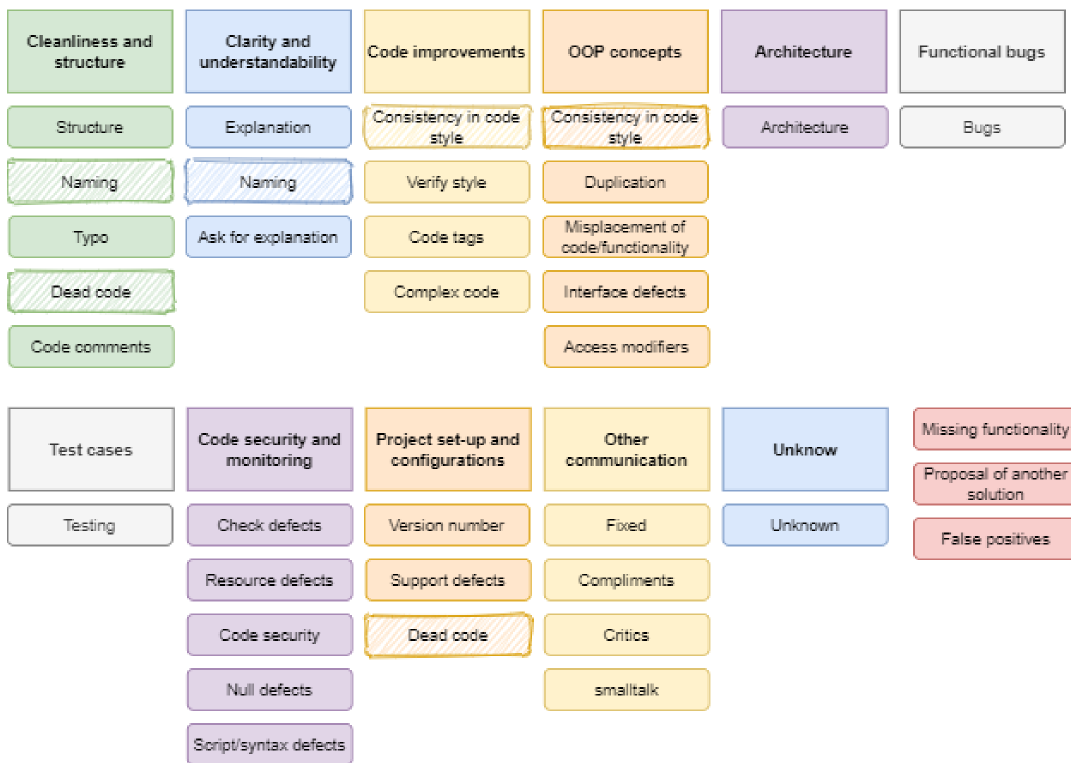


Figure 3.1: Overview of the specific categories grouped to their global categories. Notice the red marked categories are not included into a global category, furthermore the sketched categories are assigned to two global categories.

could indicate to which global category the comments of these specific categories belong. These results will be separately presented in the results in Section 5.3.2.

3.3. Creating a code review checklist

In order to assist the developers by systematizing their code review we have created a checklist. How we created the checklist and how we found the items on the checklist is explained in this section. We choose to create an own checklist over the adaptation of an existing checklist as a checklist should comply to the needs to of the project [8, 69] and different programming languages require its own checklist [39]. This does not mean existing checklist were not used at all, throughout the process we used existing checklist as inspiration sources.

In Section 3.4, the next section, the usage of the checklist and how we measured the effectiveness of the checklist is explained. First we explain our expectations of the checklist in Section 3.3.1. Next, we explain how we discovered the checklist items and created the checklist in cooperation with a dedicated scrum team in Section 3.3.2.

3.3.1. Expectations and intentions regarding the checklist

Our goal for the usage of the checklist is to improve on the structure and clarity in the code review process, based on the findings of Greiler [86]. With the enhanced structure and clarity our hypothesis is that through the usage of the checklist it is clear for the reviewers what is expected from them and that code reviews will happen more thoroughly increasing the quality of the code reviews. Our goal is specifically to clarify the expected coding guidelines within Greenchoice. This will lead to improvements in the code quality, happiness of the developers and increased knowledge sharing. Furthermore, the checklist can be used as guidance during code reviews for new developers. Finally, the checklist should be regularly updated to emphasize the current needs of the team at that moment. An important aspect to keep in mind is that reviewers should not spend additional time without improving the effectiveness, in other words time versus reward should be kept in balance.

3.3.2. Joining a scrum team

In order to create an useful checklist we investigated which factors are important and which can be left out. For this we joined for a month a dedicated scrum team to get familiar with their development process and especially their code review process in detail. We joined a relative young development team consisting of four developers. Throughout our period a developer left the team. This scrum team was chosen in consultation with one of the team leaders. In our observations the team members are already actively involved in the code review process and uses more code review comments compared the other teams. Furthermore, the team is the first pilot team with no dedicated tester. In addition, the team members themselves are open for new ideas and experiments. The team is working on mostly new pilot projects from business which are later transferred to other or newly created teams. Working on the newer features they experience less pressure from business and are familiar with the experimental environment. We did not only focus on the specific team, we kept also an eye on the other teams to discover their needs. Joining the team consisted of attending most daily scrum meetings, sprint plannings, refinement and retrospective meetings. In addition, we paired with the developers, mostly observing, while they were programming and performing code reviews. We held two constructive sessions with the team members to discover the important aspects for creating the initial version of the checklist in detail.

Finding the strengths and weaknesses of the team's code review process

In the first session we focused on the strengths and weaknesses of the code review process of the corresponding team. The session was started with a short introduction with the preliminary results of the questionnaire and interviews, and the goal of the sessions. Next, the strengths and weaknesses of the team were explored, these we later compared to the result of our findings about the strengths and weaknesses from the interviews. We asked the team members to write down their thoughts about code reviews on post-its, green colored for the strengths and pink colored for the weaknesses. Once all team members were finished the post-its notes were presented by the owner and discussed within the team to understand the reasoning behind the post-its. Finally, the session was ended with three examples of checklists [86–88] used elsewhere for the next session.

Finding the checklist items and creating the checklist

In between the two session we have created a long list of checklist items that we thought could be applicable to Greenchoice's current code reviews process based on our observations so far. Most of the items on the long list are based on the existing checklists [86–88]. Additional, several items under the coding guidelines header are retrieved from an older coding guidelines Wiki page of Greenchoice.

In the second session we started with discussing briefly the example checklists [86–88]. The rest of the session was used to explore the factors of checklist items that were deemed important or unnecessary by the team in an open discussion. The long list of checklist items served as inspiration source for us to bring up certain items during the discussion, notice the developers have not seen this list. Once there was concession between the team members regarding necessity of a factor, the factor was written down on a post-it and placed on a large brown paper. After the session we translated these factors into checklist items and narrowed down the long list into a shortlist of checklist items. Based on this shortlist and on the preferences of the team we created the first version of the checklist.

In the last step before actually using the checklist, we sent the checklist to the team and they could give their final feedback. Additional, the initial checklist was presented to the architects and the team leader together with the experimental set-up for the evaluation and usage of the checklist. Based on the feedback of the team and architects we created the final checklist, shown in Appendix G.

3.4. USING AND REFINING THE CHECKLIST

In this section we will explain how we analysed the effectiveness of the usage of the checklist and how we refined the checklist. We used for this a quasi experimental pre post test design [32–34]. The experiment is quasi experimental as we could not randomly sample the participants in our research. We determined in consultation with Greenchoice the teams which were to participate in the experiment. The pre post test design comprises of a pre and a post questionnaire, in the pre questionnaire we gauge the expectations of the usage of the checklist and in the post questionnaire we gather the actual perceived experiences of the participants. Comparing the results of the pre and post questionnaire we can determine which effects the usage of the checklist had. First we explain in detail the general set-up of the experiment in Section 3.4.1 followed by the explanation of the pre and post questionnaire used in Sections 3.4.2 and 3.4.3.

3.4.1. EXPLAINING THE GENERAL SET-UP

In the experiment we included two additional scrum teams making a total of 3 teams using the checklist. Although, the checklist was created in cooperation with only the initial scrum team, we tried to keep in mind that the checklist should also be applicable by additional teams. To achieve this we tried to avoid specific items only applicable to the initial team. Furthermore, the additional two teams were chosen based on similar properties to the initial team. This means similar programming languages, relative young developers and developers being open for experiments. Furthermore, the additional teams consist of 3 developers and a dedicated tester compared to the 4 developers without a dedicated tester of the initial team. However, at the moment of the experiment the initial scrum team had 3 developers. This means that in total 9 developers and 2 testers participated in the experiment.

To measure the effectiveness of the checklist the three teams were asked to use the checklist in every code review during two sessions of two weeks. Before the first session a pre questionnaire was sent to the developers and testers to explore their expectations on the usage of the checklist. Including the testers in the experiment provides an opportunity to gauge whether or not the checklist is also appropriate for the testers having less experience with programming and code reviews. The two usage sessions were aligned with the two weekly sprints of Greenchoice. After the first session a meeting with each team was held to discuss the usage of the checklist. Based on the feedback the checklist was refined and a new version was made and distributed to the teams, the refined checklist is shown in Appendix H. Unfortunately due to the Christmas holidays the second session could not start immediately and the second session was delayed till after the holidays. The second session was identical to the first session, though with the refined checklist and a scrum team decided to stop with the experiment. After the second session a post questionnaire was sent out to evaluate the checklists. The results of the both questionnaire were compared to each other and analysed to discover the effects of using the checklist.

3.4.2. EXPLAINING THE PRE QUESTIONNAIRE

The used pre questionnaire to gauge the expectations on the usage of the checklist consists of 8 sections. The full pre questionnaire is shown in Appendix C. At the moment of the pre questionnaire the teams have not yet seen the checklist except the developers of the initial scrum team who were involved in creating the checklist, though they are also not aware of the latest changes. Before exploring the expectations on the usage of the checklist during code reviews we explored the thoughts related to checklists and code reviews separately. Additional, we looked into the thoughts related to the code quality and which factors are important in de-

termining the code quality. The questionnaire started with an introduction as the first section and has the opportunity for to leave behind their final remarks regarding the questionnaire as last section. Next, in the remaining part of this section the remaining sections in the questionnaire are further explained.

The second section of the questionnaire consists of a single question which explores the personal interest of the respondents on 8 different coding concepts and practices related to code reviews and the usage of a checklist in general as shown below in the enumeration. The affinity measured on a 5 point scale and it is up to the respondent to interpret the scale, at which 1 is the lowest affinity and 5 the highest. We defined affinity as knowledge regarding the subject and attention paid to keep up with the latest trends, features and tricks. This questions gives an insight into the initial thoughts of the respondents regarding these subjects and provides to use a better interpretation on the priority of these subjects.

1. As developer in general
2. Coding guidelines in Greenchoice
3. Language specific features of the programming languages
4. Code quality in general
5. Object-Oriented Programming (OOP) concepts
6. Code architecture
7. Code reviews
8. Usage of checklists

The third section dives deeper into what the respondents qualify as good code quality. We asked two questions; the first question consist of 6 statement regarding the code quality and has a 3 point Likert scale: disagree, neutral and agree. The second questions asks the respondents to select 3 out of the 9 aspects that have the most influence in determining the code quality. The 9 aspects are shown below and were also used as general categories in the data analysis as explained in Section 3.2.3.

1. Cleanliness and structure
2. Clarity and understandability
3. Code improvements
4. OOP concepts
5. Architecture
6. Functional bugs
7. Test cases
8. Code security and monitoring
9. Project set-up and configurations

The fourth section gauges the thoughts towards checklists in general. Starting with the frequency of the usage of checklist within Greenchoice and outside their work, for example grocery shopping list or a cleaning list, and their satisfaction with the usage frequency of checklists also called usage load. Next we asked in the second question their opinion regarding 5 aspects on the usage of a checklist on a 5 point Likert scale. The following aspects including their scale were asked:

1. General experience (from annoying to pleasant)
2. Time spending (from time consuming to time saving)
3. Ease of use (from difficult to easy)
4. Structure, e.g. knowing what to do (from unstructured to structured)
5. precision, e.g. knowing which tasks to do (from imprecise to precise)

The next section covers the thoughts surrounding code reviews consisting of 6 questions. The first question asks whether the respondent omit specific comments during a code review and later encounters it as a challenge in the code base. Next the usefulness of writing comments in DevOps is covered with 5 statements on a 3 point Likert scale. The last questions of the section covers the satisfaction with the used tooling during code reviews and whether specific types of tooling is missing. This is followed up by an optional section of the questionnaire, sixth section, in which the respondent could leave behind suggestions for the specific tooling.

The seventh section and last question section covers the expectation on the usage of the checklist during code reviews. First, 5 statements regarding the expected quality improvement, time management and satisfaction of the respondents are given on a 5 point Likert scale. Next, the expected usage frequency of the checklist is gauged as reviewer and also as author. Followed up with two specific statements with, once more,

a 5 point Likert scale. The statements cover whether they expected to write more comments instead of face to face communication and whether they dare to write more feedback now they have something to fall back on. Second to last the respondents are asked whether they expect to give more feedback due to the usage of the code review checklist for the 9 aspects used as categories in the comment data analysis as explained earlier. lastly, there is the option to give addition expectations on the usage of the checklist.

3.4.3. EXPLAINING THE POST QUESTIONNAIRE

The post checklist questionnaire consists of 4 section and is fully shown in Appendix D. The post questionnaire was used to gauge the thoughts of the respondents on the usage of the checklist during code reviews in practice. Part of the post questionnaire is similar to the pre questionnaire in order to evaluate the expectations to the actual findings. The first section of the questionnaire is a brief introduction and in the last section the respondents had the option to leave behind their final remarks.

The second section consisted of 3 questions of which the first question consisted of 6 general statements regarding the clearness of the experiment, time management and the quality of the checklist. This is measured through a 3 point Likert scale. Next, two open questions were given in which the respondent had to fill in at least one positive aspect and a negative or improvement on the usage of the checklist during code reviews.

The third section consisted of 5 questions. The first question checked the actual frequency of the usage for the checklist as reviewer and as author of a pull request. Next, 8 statements were given regarding the usage of the checklist during as reviewer with a 3 point Likert scale. The statements dived into the approach of the review process, quality and result of the changed process and happiness of the reviewer. This is followed up with 4 similar statements, although now from the perspective as author. Furthermore, the actual perceived change in comment frequency is gauged for the 9 aspects, similar to the 9 aspects explained in the pre questionnaire. Finally, the last question gauges whether the respondents expect to use the checklist in future code reviews.

4

Motivation for code reviews

In this chapter we look into the results of the questionnaire sent to retrieve the motivation for the code reviews process at Greenchoice. Furthermore, with the questionnaire we gain the first insight in the weaknesses and strengths of the code review process. The questionnaire was sent to the full IT development department consisting, at moment, of 58 employees, of which 37 employees filled in the survey, resulting in a response rate of 63, 8%. The survey was sent in the summer vacation period and was open for two weeks. Due to vacations some employees may have missed the survey period.

Before we show the results we will first describe the outline of this chapter. In Section 4.1 we get to know the composition of the department and the experience, frequency and roles with code reviews. Subsequently, in Section 4.2 we show the results of ranking the motivation categories based on three comparison method earlier explained in Section 3.1.3. Furthermore, we compared the motivations based on the global categories explained in Section 3.1. Next, in Section 4.3 we explore the developer's initial thoughts regarding code reviews exposing the first weaknesses and strengths of the process. In Section 4.4 we show several practice examples of actual defects found during code reviews showing the possible effect of code reviews. Finally, in Section 4.5 we discuss the previously showed results and compare them to the literature. In additional, we answer the first research question and show the first observations related to the strengths and weaknesses of the process which are further explored in the remaining research questions.

4.1. Demographic data

In this section we show the results on the demographic questions in the questionnaire in order to get to know the respondents. We show first the function of the respondents in Section 4.1.1, followed by the experience and familiarization with code reviews in Section 4.1.2 and finally in Section 4.1.3 the frequency and roles in the code reviews.

4.1.1. Function

In the first question we asked the respondents for their function. In order to ensure the privacy of the respondents the functions of business consultants (BC) and product owners (PO) were merged together due to the lack of individual answers. Likewise, functions with only one or two employees were grouped into *other*. The number of respondents for each function with their corresponding response rate are shown in Table 4.1.

	Respondents	Employees	Response rate
BC/PO	5	13	38.46%
Developer	19	27	70.37%
Other	8	12	66.67%
Tester	5	6	83.33%
Total	37	58	63.79%

Table 4.1: Number of respondents of the questionnaire grouped by their function, the total amount of employees and their corresponding response rate.

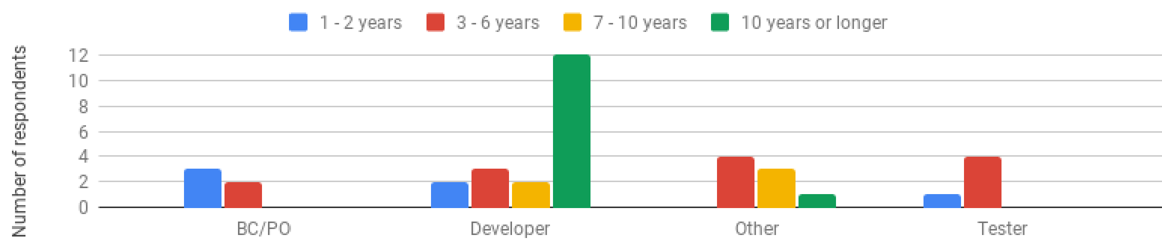


Figure 4.1: Experience of the respondents with code reviews grouped by their function.

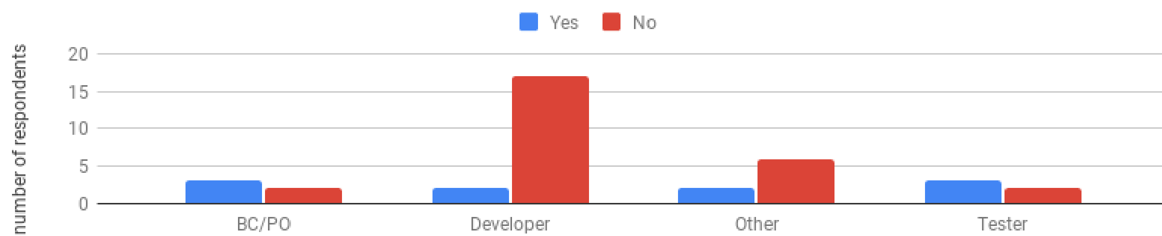


Figure 4.2: Whether or not the respondent came into contact with code reviews for the first time at Greenchoice grouped by the respondents role.

4.1.2. EXPERIENCE AND FAMILIARIZATION

The next considered demographic is years of experience with code reviews. The answers are shown in Figure 4.1, grouped by function to show the differences level of experience between the different functions. Noteworthy is the longer years of experience of the developers against the shorter period of the rest.

The additional experience of the developers is also noticeable in the next question whether or not Greenchoice was the first place where the respondent came into contact with code reviews. The results of this question are shown in Figure 4.2. Each group of functions have in absolute numbers 2 or 3 respondents who encountered code reviewers for the first time at Greenchoice. Hereby, the respondents in *other* and especially developers have in relative numbers a smaller share.

4.1.3. FREQUENCY AND ROLES

In the last two questions of the demographic section we looked into frequency of participating in code reviews of which the results are shown in Figure 4.3. Additional we looked into which types of roles they are involved in during code reviews of which the results are shown in Figure 4.4. Of the respondents 9 of them do not contribute to code reviews, these are all the business consultants and product owners, and 4 respondents of the *other* group having no coding tasks. The developers have a high frequency contributing to code reviews on a daily basis followed by the testers and the respondents of the *other* group having coding tasks.

Most respondents who are actively involved in code reviews do also read other pull request in which they did not contribute. Whereas business consultants, product owners and *others*, depending on their role, are not actively involved in code reviews. Of the product owners and business consultants 2 answered in the optional field that they are in another role active with code reviews, as shown in the following quotes:

> "Depending on the subject, I get to see some code in between to check whether the functional wish is understood. Sometimes I get to see some code afterwards, but certainly not as a review"

> "not, or only because I agree functionality goes to production if it meets the definition of done (and that I know the code has been reviewed)."

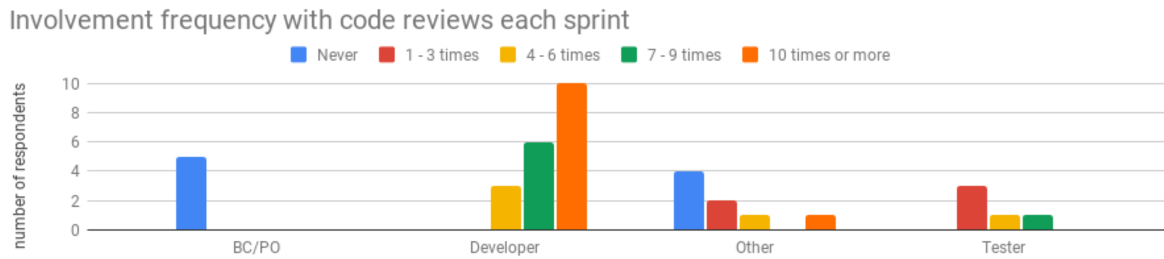


Figure 4.3: Involvement frequency on code reviews in each sprint grouped by function.

Figure 4.4: Roles during code reviews grouped by function.

4.2. MOTIVATION FOR CODE REVIEWS

In this section we show the results of the motivations for Greenchoice's code review process given by the respondents. In the second question the respondents had to give 3 motivations ranked on importance. In the third question the respondents could give 5 additional motivations, these had not to be ranked. The full list of the given motivations classified in their corresponding category and grouped by priority can be found in Appendix B.

In Section 4.2.1 we show the results of the motivations after classification into their corresponding category, these categories were earlier explained in Section 3.1.2. In here we compared the categories based on the three methods explained in Section 3.1.3. Furthermore, we compare in Section 4.2.2 the results in which the motivation categories are grouped into the global categories which were also introduced in Section 3.1.2.

4.2.1. COMPARING THE MOTIVATIONS

The given motivations are categorised in the 11 categories explained in Section 3.1.2 and the full categorisation of the motivations can be found in Appendix B. In order to compare the categories we ranked the motivation categories in three different methods described in Section 3.1.3. In Figure 4.5 we show the first method simply counting the amount of answers for each category per priority. In Figure 4.6 we show the second method, the weighted count for each category per priority. The amount of given answers for each priority is multiplied by its corresponding weight. In Table 4.2 we show for each category the average weight. This is the weighted count divided by the total count. For each category the total count and weighted count of that category is also shown in the table for completeness. The results of the categories are now further explained going over the categories in order of rank in based on the first method, the total count of answers for the category.

The category ranked first counting the answers is *knowledge sharing* with respectively 19 answers. The following two answers give an example of the given answer in this category: *"To learn from each other in the way we write code"* and *"Learn from how other members of the team resolves issues"*. First to notice is that only a single answer was given in priority 1. With 5 motivations in the additional motivation question it has the most optional answers. This is reflected in the weighted rankings where *knowledge sharing* can be found lower in the rankings. For the weighted count it is still ranked third due to the high amount of answers. In the average weight ranking *knowledge sharing* is found at the bottom ranked ninth and solely 0.03 above the lowest ranked and 0.01 above the second lowest.

Based on the count *knowledge sharing* is closely followed by *quality* with 18 answers of which 13 motivations are given as the first priority. With this many answers in priority 1 it is the first ranked category on

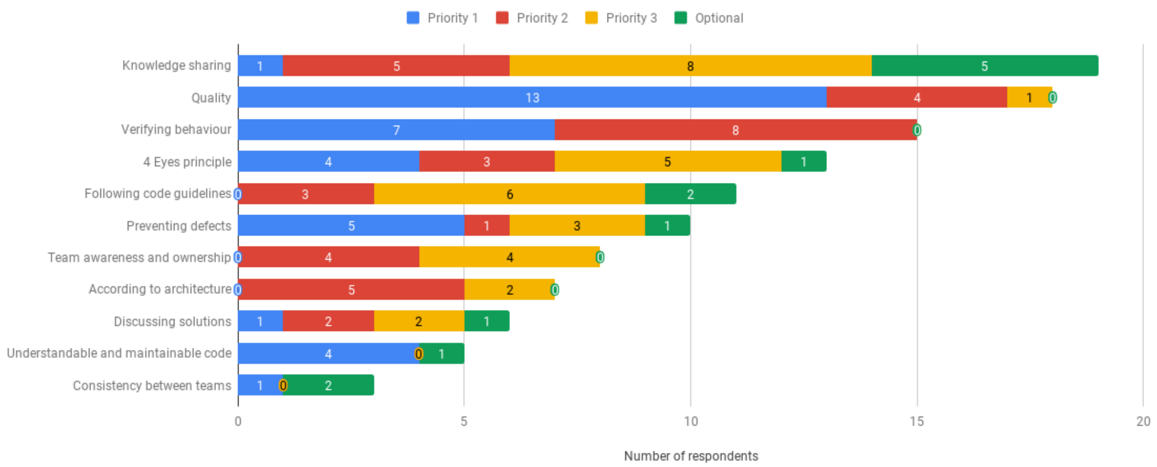


Figure 4.5: Results of the motivation categories ranked by their total count.

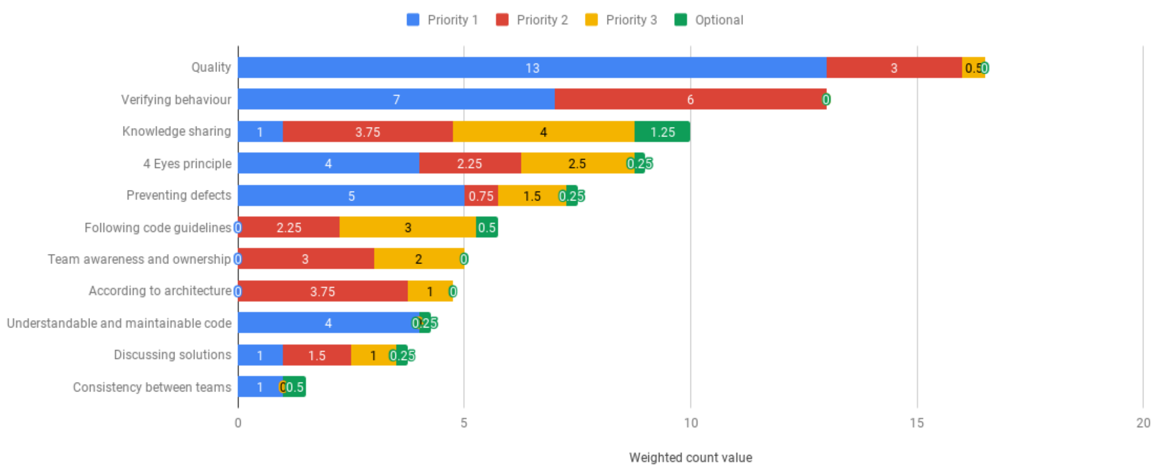


Figure 4.6: Results of the motivation categories ranked by their weighted count.

	Total count	Weighted count	Average weight
Quality	18	16.5	0.92
Verifying behaviour	15	13	0.87
Understandable and maintainable code	5	4.25	0.85
Preventing defects	10	7.5	0.75
4 Eyes principle	13	9	0.69
According to architecture	7	4.75	0.68
Discussing solutions	6	3.75	0.63
Team awareness and ownership	8	5	0.63
Knowledge sharing	19	10	0.53
Following code guidelines	11	5.75	0.52
Consistency between teams	3	1.5	0.50

Table 4.2: Overview of the motivation categories with their corresponding total count, weighted count and average weight.

the weighted count ranking. Having a large gap of 3.5 to the number 2 and has the highest average weight of 0.92. The given answers vary in broad definitions such as: *"Check intention of code, check naming, structure, testability, complexity"* to simpler answers like *"Quality control"*.

With a slightly larger gap *verifying behaviour* finishes the top 3 with 15 answers. The answers are all given in priority 1 and 2 and therefore ranked second in weighted count and average weight with 0.87. Most given answer are in the area of *"Does the code do what it should do"*.

The fourth category by count is *4 eyes principle* with 13 answers. The answers are equally distributed over the priorities and a single answer in optional. It is ranked fourth for the weighted count and fifth for the average weight with 0.69. The given motivations vary from just *"4 Eyes principle"* to *"To prevent that the author has to 'mark his own paper'"*.

The fifth category by count is *following code guidelines* with 11 answers. Most answers were given in priority 3 and no answers were given in priority 1. Still it is ranked sixth in the weighted count, though second last for average weight with 0.52. A single answer was very specific: *"Prevent magic numbers"* most others were more general like: *"Check coding guidelines"*.

The next category by count is *preventing defects* with 10 answers. The ranking of weighted count is slightly higher ranked fifth. This reflects its high average weight due to the lower count with 0.75 ranked at the fourth place. The answered motivations range from *"Sharpness of a developer can sometimes be weakened by distractions, by reviewing you prevent problems earlier than during testing"* to simpler answers *"Check on presence of bugs"*.

The seventh and eighth ranked categories by count are *team awareness and ownership* and *according to architecture* with respectively 8 and 7 answers. *Team awareness and ownership* is also ranked seventh for weighted count and shares the seventh and eighth place for average weight with *discussing solutions* with a score of 0.63. A given answer is: *"The team is responsible for the code, not the individual -> the team feels the ownership of the code"*. *According to architecture* is ranked eight for weighted count and is ranked sixth for average weight with 0.68. The answers are in the trend of *"Code that complies with team / architecture agreements"*.

The last three categories ranked by count are *discussing solutions*, *understandable and maintainable code* and *consistency between teams*. For the weighted count *discussing solutions* and *understandable and maintainable code* switches places. *Understandable and maintainable code* is ranked third for average weight with 0.85 due to 4 out of 5 priority 1 answers. *Discussing solutions* shares place 7 of average weight with 0.63 and *consistency between teams* is last with 0.50. A remarkable given motivation and in our opinion describes the purpose of a code review in a perfect manner is categorized in *discussing solutions* is: *"There is a dialogue about the quality of the solution, does it comply with the guidelines, was a simpler, more manageable way possible, why is this the best. We learn from that dialogue"*, though it touches multiple other motivation categories. An example of given motivation for *understandable and maintainable code* is: *"Coding style (layout, readability, typos, etc.)"*, and for *consistency between teams*: *"Ensuring that the team develops in the same way"*

Summary: In total we found 11 different categories of motivations given by the respondents. The category *knowledge sharing* is ranked first counting the amount of answers given for each motivation, however the answers of this category are given with a lower priority resulting in the ninth rank for the average weight comparison. *Knowledge sharing* is still ranked third in weighted count combining the amount of answer and their priority. The category *quality* is ranked first for weighted count and average weight comparisons and ranked second in the amount of answers. The top 3 of the weighted count is completed by the category *verifying behaviour* on the second place. This category is also ranked third in terms of count. Thus the three most important motivations for code reviews at Greenchoice according to the respondents are: *knowledge sharing*, *quality* and *verifying behaviour*

4.2.2. GLOBAL CATEGORIES

In the classification of the given answers into the categories we observed that some of these categories are closely related to each other. Therefore, we introduced the three global categories: *quality*, *social* and *functional*. In Table 3.1 we showed how the different categories are distributed over the global categories. In Figure 4.7 we show the count per global category per priority and in Figure 4.8 the weighted count. In Table 4.3 we show the average weight and once more for completeness the values of the total count and weighted count are also included.

The global categories are ranked close to each in terms of count. The global category *code quality* has the

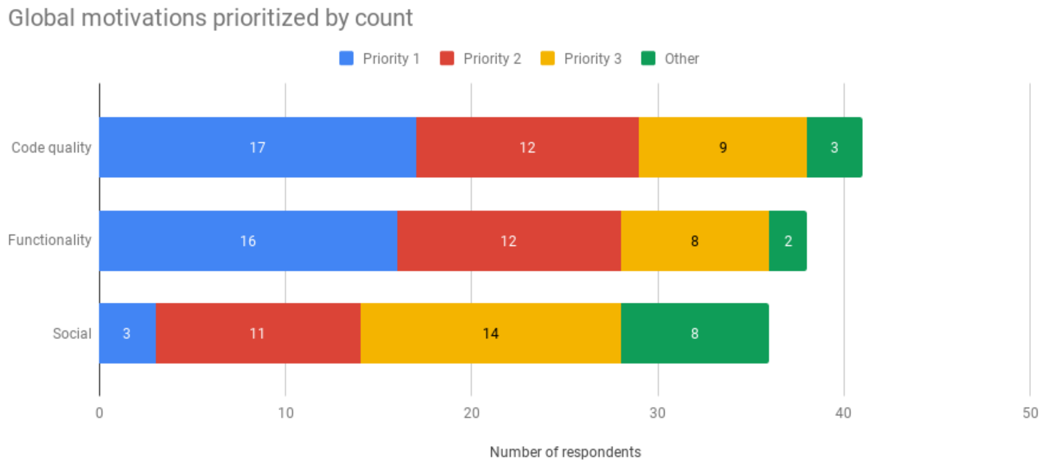


Figure 4.7: Results of the global motivations ranked by count.

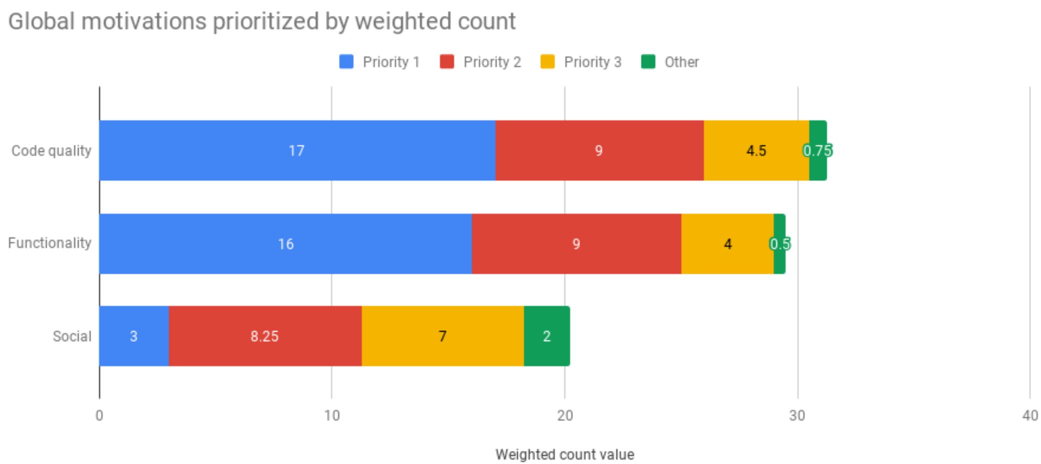


Figure 4.8: Results of the global motivations ranked by weighted count.

most answers with 41 motivations given, succeeded by *verifying behaviour* with 38 motivations and *social* is last with 36 motivations. For the weighted count there is still a small difference between *code quality* and *verifying behaviour*, however *social* loses terrain as there were only a few priority 1 answers given. This is also reflected in the average weight were *social* is last with 0.56 and a gap of 0.2 to the other global categories. *verifying behaviour* is ranked first with 0.78 and overtakes with a small margin of 0.02 *code quality* with 0.76.

Summary: The global categories: *functional*, *code quality* and *social* have similar results in terms of amount of answers given, however including the priority aspect the global category *social* falls behind the other two global categories of which the difference still remains small.

Global category	Total count	Weighted count	Average weight
Functional	38	29.5	0.78
Code quality	41	31.25	0.76
Social	36	20.25	0.56

Table 4.3: Overview of the global motivation categories with their corresponding total count, weighted count and average weight.

4.3. CODE REVIEW STATEMENTS

In the fourth question of the questionnaire we explored the initial thoughts of the respondents related to code reviews in 10 statements. The first 4 statements covered the code review process in general and the statements 5-7 the process from the perspective of the reviewer and statements 8-10 from the perspective of the author. In Figure 4.9 the statements and their corresponding results are shown. First off, two respondents answered all statements with not applicable. Nine respondents answered not applicable to statement 4 and the further remaining statements, these 9 respondents are equal to the 9 respondents who do not contribute to code reviews as we showed earlier in Section 4.1.3. These were the business consultants, product owners and 4 respondents of the *other* group who do not write code.

In the first statement it is good to see that everyone acknowledges, with most of them totally agreeing, that code reviews are an important aspect in the process of developing software. The next two statements concerning the visibility of code reviews and whether or not enough attention paid to code reviews, roughly a fifth of the respondents disagreed on both statements. Additional, the percentage of totally agreed answers shrinks. This indicates that for these two aspects there is potential for improvement. In the fourth statement we see that most reviewers have enough time to perform the code reviews, although few developers indicate that they lack time.

The next three statements cover the experience as reviewer during a code review. First off, 14% of the respondents do not enjoy performing a code review. A greater part 22% also find it difficult to perform a code review. Finally, only a small part 5% thinks that his or her feedback is not useful.

The next three and last statements the situation was turned around to the perspective of the author of the pull request. All respondents enjoy receiving feedback, however a small part 8% find it difficult to receive feedback. Lastly, a small part 5% disagrees that the received feedback is useful.

Summary: Code reviews are considered important in the software development process, however, the visibility of the results of the code reviews is not always clear and more attention could be paid to the code review process. The reviewers find it sometimes hard to review the pull request and do not have issues receiving feedback.

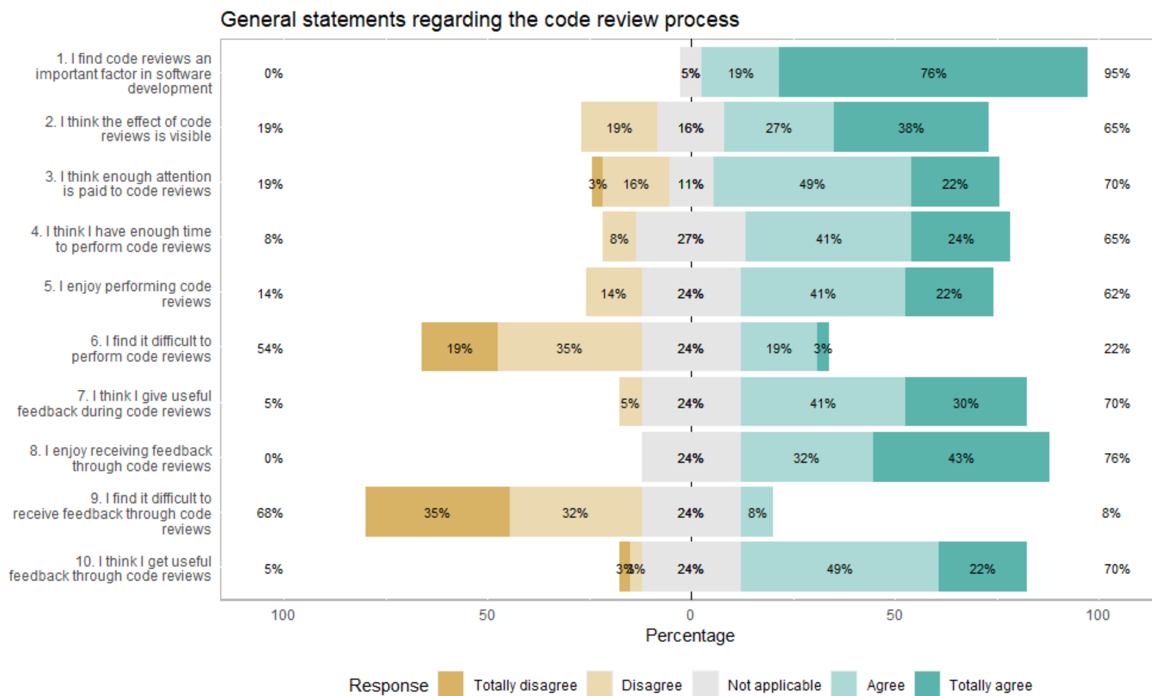


Figure 4.9: Results to the 10 general statements regarding Greenchoice's code review process.

4.4. PRACTICE EXAMPLE

The last question in the questionnaire was whether or not the respondents could remember an example of an actual bug or code smell which was prevented due to a code review. We intended to trigger the respondents to think about the effects of code reviews and also to get some practice examples. Of the 36 respondents 6 answer *no* and 7 *not applicable*. Additional, of the 23 leftover answers with *yes* only 8 of them gave an actual example. A few bugs or code smell which were prevented are:

- Missing null checks or wrong usage of nullable variables causing possible exceptions
- Mistakes in config and build files, such as wrong server address
- A wrong date value was used in the logic

4.5. DISCUSSION

In this discussion we go further into detail on the results of the questionnaire. We will address the observations in the results of the different questions and look into which aspects we will elaborate on further in this research. Additional, we compare our results to the findings of the literature. First we will discuss the demographic results in Section 4.5.1, followed by the motivation categories in Section 4.5.2 and finally the initial thoughts statements in Section 4.5.3.

4.5.1. DEMOGRAPHIC DATA

In this section we discuss the findings in the demographic questions. First we will discuss the lower response rate of the products owners and business consultants. Next, we compare the level of experience and involvement in the code review process to the literature.

Although most teams each have their own product owner and business consultant, only few of them filled in the questionnaire. Although they are part of a scrum team, they do not program themselves. Therefore, they might lack interest in a software engineering related questionnaire or are too busy with their own tasks, hereby the questionnaire was intended to be also suited for non programmers.

In terms of years of experience the developers have by far the longest experience. The group of developers is relative older compared to the other roles and employees at Greenchoice granting them more experience.

Additional, most developers have encountered code reviews before. This can either be during their former work or their study. The experience in terms of years is longer for developers at Greenchoice compared to the findings of Bosu and Carver [41] who found that on average developers have 7 years of experience in open source projects. In terms of frequency in usage the developers are 4 up to 10 or more times involved in code reviews in a two week period of which the majority, 9 developers, are involved in code reviews 10 times or more. Compared to the developers at Google who are involved in three changes a week [35] Greenchoice's developers are slightly more involved in code reviews on a weekly basis.

This does not hold for the testers, business consultants and product owners who mostly encountered code reviews at Greenchoice for the first time. Some of these positions are filled from internal positions with no dedicated prior education. For example a tester could have started at customer service. There is a gap in the participation of testers compared to the findings of Sutherland and Venolia [61] who found that tester together with the developers are the ones most involved in code reviews.

Summary: the developers are on a daily basis involved in code reviews as author, reviewer or co-reader having with 10 years or more the most experience with code reviews. Testers are sporadically involved in code reviews and mostly in the role as co-reader having up to 6 years of experience with code reviews.

4.5.2. MOTIVATIONS

In this section the results of the ranks of the motivation categories are discussed. We also answer the first research questions in stating the motivation behind the code review process. Furthermore, we compare the results of the rankings to the rankings in the literature.

In our results we showed the motivations *knowledge sharing*, *quality* and *verifying behaviour* are the top 3 motivations for code reviews according to the respondents. The greatest difference between these categories is in the average weight comparison, in which *knowledge sharing* has on average 0.34 less priority to the second category *verifying behaviour*. The difference between *verifying behaviour* and *quality* is relative smaller with 0.05. Together with the slightly higher count of *quality*, 3 answers, we can carefully conclude *quality* is the most important motivation. Followed closely by *verifying behaviour* and *knowledge sharing*.

These categories are each assigned to a separate global category: *code quality*, *functional*, *social*. The differences of the count comparison between the global categories are still small, however the answers in the global category *social*, including *knowledge sharing*, have less priority. The global category *social* has on average 0.20 less priority. This reflects the findings similar to the category *knowledge sharing*. The difference between the average weight between the other two global categories is with a small difference of 0.02 in favor of *code quality*.

These findings are new compared to the literature, indicating the main motivation for code reviews is finding defects [8, 9]. In our results we have seen that *verifying behaviour* is more important than *preventing defects*, however also not the main motivation for code reviews. In a recent study Sadowski *et al.* [35] show the initial motivation to start code reviews at Google was to force the developers to write understandable and maintainable code and this motivation changed later on to the educational value. In our results we barely found motivations in the category *understandable and maintainable code*. However, the category *quality* is a broad definition and might include understandable and maintainable code as part of determining the code quality[9].

The specific categories *quality* and *verifying behaviour* are newly introduced categories based on the answer given and did not exist in earlier research in this form. The specific category *quality* is a rather broad term, in the interviews 5.1 we try to clarify what is meant with quality and in Section 7.1.2 we looked further into the definition of code quality. Missing out *quality* in the literature can simply be explained by the more specialized categories used such as: *following code guidelines* and *understandable and maintainable code*. Naturally, these categories are lower ranked in our results due to the lack of answers. The category *verifying behaviour* is the complementary of *preventing defects* in the literature [8, 9]. Instead of finding and preventing defects in the change, the goal of *verifying behaviour* is to verify the correct behaviour of the change. It checks whether or not the acceptance criteria are met of the change.

The check of *verifying behaviour* together with *preventing defects* is closely related to *4 eyes principle*. Hereby *4 eyes principle* differentiates itself by preventing a single developer from harming the system. In chats with the respondents it became clear *4 eyes principle* served two purposes namely: deliberate fraud prevention and preventing accidental mistakes. Compared to the literature *4 eyes principle* is ranked high

at the fourth place in all three comparisons, in the literature it is an unintentional advantage instead of a motivation [35] or not described at all.

In contrast, *discussing solutions* is ranked as second to last motivation in our results, ranked tenth. This is low compared to the literature where it was found as third motivation [9]. The lower ranking might be due to the developers discussing the solution earlier in the sprint refinement and planning documenting this in the PBI. This means that the developer has at forehand already information how to solve the issue. Alternatively, these kind of discussions are held face to face which was observed at the workplace.

The last category we want to further elaborate on is *consistency between teams*, a newly created category, which is ranked last. We were surprised we received these answers due to the lack of clear guidelines within the teams which we found later on in the interviews, see Section 5.1. We doubted due to few amount of answers, only 3, whether we had to merge this category with another category. For example *following code guidelines* or *knowledge sharing*, however we thought the differences between these categories are too big. The given motivations in *consistency between teams* do not specifically mention code guidelines or learning from each other.

We are surprised that we did not encounter motivations related to testing in answers. In addition, other literature [9, 35] which looked into the motivation of code reviews did also not encounter testing as a motivation for code reviews. We are surprised as testing is according to the literature an important aspect in determining the quality of a pull request [37, 89] and according to Gousios *et al.* [65, 66] an agreement between contributor and integrator to assess the code quality. On the other hand Mäntylä and Lassenius [39] explains that using tests as a sole measurement of quality is unfair as testing does not consider the evolvability aspect of the code. Based on our findings and of the other literature [9, 35] testing is not a part of the motivations for code reviews, however, testing can still play an important role in assessing the quality of a pull request [37, 65, 66, 89].

RQ1: What is the motivation behind code reviews?: The *quality* category is the most important motivation for code reviews at Greenchoice. The *quality* category is a very broad definition and can be narrowed down based on the given answers into ensuring the code quality in code reviews. The *quality* category is closely followed by *verifying behaviour* and *knowledge sharing*, the latter category lacks higher priority answers. Grouping the categories to three global categories *code quality* remains the most important motivation followed closely by *functional* and *social*, the latter lacks, once more, higher priority answers.

4.5.3. Code review statements

In this section we will discuss the results of the 10 statements, shown in Figure 4.9, in the questionnaire to gauge the initial thoughts of the respondents on the code reviews process. The respondents could fill in additional open answer fields to support their answer. These answers compared to the results and earlier findings in the literature are discussed here in order of the statements.

Importance of code reviews

In the first statement the importance of code reviews in the software development process is supported by the optional open answer fields. The optional answers in general consist of the advantages of the code reviews such as: keeping the quality high, as author you are critical of your work submitted, knowledge sharing and the earlier defects are found the better. A specific answer given is: *"Agree, submitting code for review makes you more critical and sharper of your own code before creating a pull request"*. Furthermore, an answer included a quote of Robert C. Martin also known as Uncle Bob [90]. It emphasizes the importance of writing readable code as 10 times more time is spend reading the code over writing the code.

> “Indeed, the ratio of time spent reading versus writing is well over 10 to 1. We are constantly reading old code as part of the effort to write new code. ...[Therefore,] making it easy to read makes it easier to write.” [91] (Robert C. Martin)

Effect of code reviews visibility

From the results it became clear that the visibility of the effect of code reviews is not always clear. On the positive side two reactions simply just agreed others described the positive effect of code reviews in more detail: *"Visible in both the little knowledge transfer required within the team and in the SonarQube reports."*.

However, in perspective of a business consultant or product owner the effect is unclear, *"Not for me as a BC/PO. Perhaps for developers?"*. Another BC/PO disagreed to the statement without an explanation, only one BC/PO agreed and the other two answered not applicable. This indicates that the effect of code reviews is unclear to BC/PO. This is not unexpected as most of them are not actively involved in code reviews shown in the demographics results.

Other respondents however are more doubtful mentioning although there are some effects visible there is potential for improvement or more attention needs to be paid in order to see clear effects. Furthermore, the following answer was given on the doubtful side: *"Somewhat. The SonarQube report provides insight into the quality of the code. But sometimes text and explanation is important because it is not as black and white as the report shows. For me, the tester is also an important factor. The more bugs that come out during testing, the more this says about the code review and / or the acceptance criteria."*. Finally a respondent mentioned that the larger the pull request the less effect the code review has. In the interviews 5.1 we look into how the size of a pull request influences the code reviews, how the SonarQube report helps the developers and whether there is a desire for a dashboard to increase the visibility of the effect of code reviews. In additional, we look in the data analyses 5.2 whether we can find a connection between larger pull request and other factors such as time spend.

ATTENTION PAID TO CODE REVIEWS

The next statement considered the attention paid to code reviews. Again the open answers contained simple acknowledgements, while others gave more descriptive answers: *"We pay a lot of attention to the code reviews, we ensure that someone never has to wait long for a code review."* and a BC/PO as following *"Yes, I hear it (code reviews) at least twice a day every day."*. On the opposite side, an answer said they should take it more seriously and *"Too often simple mistakes about the database slip through."*. This is also later discovered in the interviews of which the results are shown in Section 5.1.3.

Another answer included another approach used at a developer's previous work: *"At my previous club, accepting a pull request (the ultimate goal) went way beyond a code review. Here the feature branch was checked out and tested by fellow developers. We didn't have dedicated testers there, so that was the way to keep code quality (functional and technical) high."*. The latter is interesting especially as this different than Greenchoice's approach in the code review process. In addition, the described code review process is more similar to the process we are familiar with. However, in the interviews described in Section 5.1.2, we found that it is not always possible to pull the feature branch locally and run the code.

TIME MANAGEMENT

The fourth statement addressed the time available for code reviews. In the results we saw only 8%, 3 respondents, disagreed. One of the three disagreeing respondents substantiate their disagreement with *"For a global scan [there is enough time], but often not for [a] detailed [scan]."*. Another respondent, who agreed with the statement, gives the following reasoning: *"Sometimes when there is too much (irrelevant) code in a PR to review, it is difficult to indicate the correct amount of time. I think the answer is yes but at the same time we have to make sure that the code to be revised is up to the point and is not overshadowed, for example, by changes that only affect certain naming conventions."*. This latter answer indicates in our perception that larger pull request decreases the effect of code research.

Based on these two answers and the answer of the previous statement we got the impression that code reviews are used mostly as global scans of correctness for the code in the pull requests. In other words they spent 20% of their time to reach 80% of the work, the global scan, however the last 20% of the work requires 80% of the time, the detailed scan, this is also known as the Pareto principle [92]. The question is whether it is feasible to expect the developers being able to perform the detailed scans. In the interviews in Section 5.1 we will look further into whether or not our impression was correct.

REVIEWER'S PERSPECTIVE

Here we will discuss the results of the statement in perspective of the reviewer. In the results we showed that 5 respondents did not find it pleasant to perform code reviews. Furthermore, we showed that 8 respondents found performing code reviews hard. A respondent justifies this with: *"Sometimes it takes me out of my work and sometimes I find it hard to get myself to understand or judge the review."*. Two respondents actually enjoy giving feedback and supports this by the fact that they are also learning from it. Another respondent brings up a new reason why code reviews could be hard: *"Certainly when it comes to functions outside my own direct focus area, it is often a context switch. What does the PBI want, what does the code do. In that sense, it is*

sometimes difficult and assessing a Pull Request also takes longer than I would like or it is at the expense of the quality of the review.". In this answer the context switch is once more mentioned and addressed a new issues reviewing unfamiliar areas of code. This latter was also mentioned by two other respondents. Furthermore, a respondent mentioned that reviews get harder once a pull request contains many files. The challenges of dealing with context switches, larger pull requests which are harder to review and reviewing unfamiliar areas of code are also discovered in the literature [37, 67] and provides the initial subjects to be discussed during the interviews, the results of the interviews are shown in Section 5.1.

In the last statement from the perspective of the reviewer whether the given feedback is useful. The open answers gave us less insight as the previous statements. Either they think their feedback is just useful or we should ask the receivers, which was done in the next statements. We did not look into the what types of comments are considered useful, however in earlier research Bosu *et al.* [36] found that comments addressing functional issues, correct usage of APIs and usage of tooling are considered the most useful comments.

AUTHOR'S PERSPECTIVE

The last three statements were from the perspective of the author of a pull request and addresses the same subjects as the previous three statements. Although, not every respondent enjoys performing a code review every respondent did enjoy receiving feedback. Four answers described that receiving feedback is educational and continuously improves their own knowledge. A respondent gave the following answer: *"In this way I also learn myself."*, another respondent worded it in his or her own way: *"Yes, sometimes it are the small things to keep each other on their toes"*. Some respondents, namely 3, did find it hard to receive feedback via code reviews. Unfortunately, this was not further substantiated with the open answers and we could not find a underlying reasoning for this. Finally, in the last statement we saw most feedback is perceived as useful and is supported with *"Feedback sometimes seems particular, but if the reviewer has an opinion that I do not share, I always have a conversation. Sometimes this results in an adjustment of the code review, but usually this results in an adjustment of the code."*.

Summary: The respondents all acknowledged the importance of code reviews in the development process. Furthermore, we got the impression that most code reviews are performed as global scan and additional time and attention should be investment for a detailed scan, however we have to look further whether this is feasible within Greenchoice. Larger pull requests and unfamiliarity with the area of the pull requests are the first signs of hard to review pull requests. In addition, developers face the challenge to cope with context switches. Lastly, the learning and teaching aspect is appreciated by the respondents.

5

CURRENT CODE REVIEW PROCESS

In this chapter we explore the current code review process at Greenchoice. In order to improve the process it is important to know what the strengths and weaknesses are. To discover these strengths and weaknesses we held 13 semi-structured interviews with the Greenchoice's developers from 7 different scrum teams, of which the results are shown in Section 5.1. In these interviews we get to know the perception of developers and their thoughts on the strengths and weaknesses of the process. Next, we looked into the metadata available regarding the pull requests and code reviews. In Section 5.2 we show the characteristics of the time, size and user interactions features the data provides. Last of the exploration we look in Section 5.3 into the types of comments given during code reviews. Finally, in Section 5.4 we discuss our results and compare them to the literature.

5.1. INTERVIEWS

To explore the current code review process at Greenchoice we have held in total 13 semi-structured open interviews with a length of 45 minutes to a maximum of 1 hour and 15 minutes. The interviews were based on the questions explained in Section 3.2.1. The first question covered the personal code review process of the developer from both perspectives as author and reviewer of a pull request. This is followed by the strengths and weaknesses and concluded with potential improvements of the process.

In Section 5.1.1 we show the results of the preparation of a pull request followed by Section 5.1.2 describing what aspects play a key role in performing code reviews in practice. In Section 5.1.3 we show the strengths and weaknesses of the code process discovered in the interviews. Finally, in Section 5.1.4 we show the potential improvements based on the results of the interviews.

5.1.1. PREPARATION FOR A PULL REQUEST

Coherent to a well executed code review is the preparation of a pull request by the author [67]. During the interviews we found that one of the reoccurring factors is being aware a pull request is ahead. In practice developers give a heads-up at the daily stand-up meeting or throughout the day. If a personal heads-up is not possible online channels, such as Slack or Skype, are used to notify the others. Also, when a pull request is actually created the author notifies the other developers, knowing there is a pull request ready for review.

Another important factor in the literature is guidance throughout the review process [9, 67]. However, we got the impression limited guidance is given as only two developers actually mentioned in the interviews they explain the changes in the description of the pull request. According to one the respondents he or she denotes a list of the changes made in the pull request. In practice it could be that the intention of the pull request is already made clear briefly at the daily stand-up meeting or through the description in the product backlog item (PBI). The company's policy is that a pull request must have at least a single linked PBI and is enforced by the gatekeeper in DevOps, in other words the gatekeeper prevents the pull request to be merged without a linked PBI. In practice it may happen that multiple PBIs are linked to a single pull request. Though, the listing of changes or the description in the PBI does not give the reviewer insight in what the thought process of the developer was, where to look at, how to easily go through the code and what the doubts of the author were.

Six developers specifically mentioned performing a sanity check before creating a pull request or directly

after the creation. Hereby, they go over the changed code and check whether the code is clean and neat. This includes whether there are no warning by the IDE and no ReSharper warnings. Another part of the sanity check is to go over the created pull request to check whether all the necessary and no unwanted files or changes are included in the pull request. In case of testing, only a single developer mentioned to always run the tests locally to check whether or not the tests pass. The others are however dependent on the CI to run the tests and fail once a test does not pass.

Summary: The preparation of a pull requests starts with a heads-up from the author of the upcoming pull request to the fellow developers. Additionally, the author of the pull request performs just before or right after the creation of the pull request a sanity check to check the correctness of the pull request. In the process of creating the pull requests and the sanity check we discovered limited explanation is given in the description of the pull requests to guide the reviewer through the review and the unit tests are sporadically ran locally.

5.1.2. PERFORMING A CODE REVIEW

Once a developer is notified there is a pull request available she might start immediately the review or postpone it to an appropriate moment. In the results of the motivation questionnaire we discovered that some developers find it hard to cope with the context switch related to code reviews. In the interviews a total of 5 developers actually mentioned the context switch. Two of the interviewees found it hard to cope with the context switches, the foremost reason is that the context switch takes them out of their current work. The other 3 developers have less trouble coping with the context switches of which one developer avoids the context switches and postpones the code review to an appropriate moment. This moment is already a context switch itself either the current task is finished or, after a break or meeting.

The remaining 2 developers have less trouble dealing with the context switches and emphasizes it takes little effort and helps a colleague to continue his or her work. Additional, one developer mentioned there is already quite a lot of distraction at the work spaces. The given reason is that the work spaces are quite noisy due to the open design, the teams are located in an open ring with in the center a coffee and lunch area. This area is also used by employees, also from other departments, having a coffee break, lunch break or informal meeting. Furthermore, when someone asks a question, also when directed to a team member, the other team members are easily distracted.

REVIEW TACTICS

Once the review is started we found in the interviews three different approaches being used going over the pull request. The first approach is going in order of the files top to bottom. Secondly, others follow the data flow of the code. The third and last approach is a combination of both, first going through the code to scout what is changed. Followed up by a detailed go through checking the code in more detail, this can either be following the data flow or a personal preference. A single developer mentioned pulling the code locally and check the code in the IDE to ease the review. The IDE gives extra navigation tools, for example go to function, going directly to the declaration of a variable or a method, and the call hierarchy. The call hierarchy gives an overview of which methods are using the variable or a method. In addition, one developer mentioned in the interviews she sporadically pulls the pull request locally, in particular pull requests that are harder to review, and two other developers specifically mentioned they do not pull the code locally.

Although, the changed code is sometimes pulled locally, more sporadically the code is actually ran. This is due to dependencies which only run on the servers and are hard to get them to run local. Additional, we later discovered in a chat with another developer that throughout the sprint the master branch might be unstable. This can make it hard to find a stable version of the dependency to be used local.

During the interviews we were interested how the developers assessed the functionality of the pull request as they cannot run the code locally. The majority, 10 developers, mentioned doing a static code check following the code flow, this furthermore indicates that the other developers also do not pull the code locally. Though, encountering a double if statements or other complex statements the static code check can become tricky. In additional, we discovered the responsibility for functional correctness is for the author and checked by the tester having an own dedicated testing environment.

Although developers are aware of most PBIs and know what the change is supposed to do, it may happen that the developer is not fully aware of the PBI in detail anymore or was not at all. This requires the reviewer

to dive into the linked PBI requiring additional time. Three developers mentioned this is not always possible and reading the PBI is skipped and the pull request is assessed with the best knowledge at hand.

POINTS OF ATTENTION

During code reviews the most important point of attention, emphasized by 11 developers, is: *do I, as reviewer, understand the new code*. In other words is the new code clear and understandable to reviewer. Related to this are the naming of variables, spelling mistakes and usage of variables. For example the code is more understandable, easier and faster to read, if the result of a boolean expression is stored in a variable with a clear name and than used rather than using the expression directly in a conditional statement. In addition, the static code check, mentioned by 10 developers, consists of exposing functional defects and small general code improvements. Of which 4 developers mentioned they try to look in more detail into the technical aspects of the code quality, for example does the new code fit Greenchoice's architectural guidelines. These interviewees were more experienced developers of the teams and have the additional role as tech lead in the team.

GIVING FEEDBACK

The feedback given could be either as comments in DevOps or face to face, depending on the team. Two developers especially value the face to face feedback as this eases the discussions held. Based on the social signals it is easier to see whether a change or explanation is not yet fully understood, either as reviewer or author, and the other can explain it in more detail. This direct interaction is lacking via the communication with comments in DevOps. Vice versa, the comments in DevOps provide to other developers, besides the author or reviewer, also the opportunity to read the comments and learn from the comments. This latter, learning from pull request where a developer was not directly involved in, was mentioned by 3 developers.

> **Summary:** We discovered three approaches going over the changes in the pull request: following the files top to bottom, following the data flow or a combination of both. The code reviews are performed mostly in DevOps itself, meaning the code is sporadically pulled locally for reviewing. However, the feedback could be given through comments in DevOps or face to face. During reviewing the reviewer's main focus is: *do I, as reviewer, understand the code of the change.* Furthermore, the reviewer looks for obvious mistakes either functional as technical defects.

5.1.3. STRENGTHS AND WEAKNESSES

Besides the strengths and weaknesses discovered above in the preparation and execution of a code review, we discovered in the interviews several other strengths and weaknesses of the code review process. These strengths and weaknesses are explained here.

PULL REQUEST SIZE

The size of the pull request is an important factor for an efficient code review according to at least 5 developers mentioning it during the interviews. They experience that larger pull requests are harder to review based on multiple reasons. First off, to the developers it is unclear what has changed in a larger pull request. Furthermore, it is likely that a larger pull request covers more than a single PBI making it harder to follow the data flow. In addition, a reviewer may have limited time for a code review which is enough for an in-depth review for a smaller request. However, encountering a larger pull request provides due to the limited time only the opportunity for a quick scan. The developers mentioned they often do not postpone a larger review to a moment with more time. Thus a larger pull request might not get a thoroughly review and is easier accepted.

Related to a larger pull requests is the risk of scope creep or also known as feature creep. This means the initial goal of the pull requests is overshadowed by other smaller goals. With this the initial goal becomes unclear and it is harder to understand the core goal of the change. Two developers mentioned to use the boy scout rule to improve the code. With this rule you leave the code cleaner than you found it. However, this behaviour can quickly turn in to a scope creep, especially in larger pull requests with more code that could potential be improved on.

BRANCH POLICIES

During the interviews 4 developers mentioned issues related to the usage of branch policies. Additional, we discussed the usage of weaker or stronger branch policies. The branch policies are rules enforcing specific

behavior regarding the branches, in particular for the master branch whether is allowed to push directly to the master branch and when a merge to the master branch is allowed. Greenchoice uses in general three active rules before being able to merge a pull request to the master. The build should pass in the CI, meaning no compile errors and all tests should pass, a PBI must be linked to the pull request and at least a single reviewer must have approved the request. In case of an approval most projects require an approval from a team member of which the project is in ownership. The team is responsible for the project and another team cannot make a change without the knowledge and approval of the team in ownership.

One of the issues the developers face is the required approval from a team member in ownership of the project. It may happen that a developer created a pull request and there is no fellow developer of that team available for a review. They can either be in a meeting or have a day off resulting in a delay for the pull request or being postponed to the next day. However, the developers came up with a creative workaround for this policy. A developer from another team can perform the code review and once satisfied approve the request. However, this does not count as the approval from the team in ownership of the project. Though, being the author of the pull request approving the request themselves does count as the approval from the team in ownership of the project. However, this approval does not count as approval from another developer. Thus, with the self approval together with the approval from a developer from another team the pull request can still be merged without the need of a fellow team member.

Additional, we gauged the option to add another policy enforcing a minimum percentage of test coverage. Most interviewees were not enthusiastic about this idea as this can lead to more frustration once a pull request cannot be merged, especially the fear of preventing an urgent bug fix. Furthermore, they reacted that there is a possibility this works counterproductive and tests are solely written to increase the test coverage instead of actually writing meaningful tests. It is an difficult trade off either loosen the branch policies and give the responsibility to the developers or making the policies stricter enforcing the ideal process. Hereby, Fagan [8] also described that flexibility is key in a well managed process.

ADDITIONAL MERGE FUNCTIONS

The usage of the auto-merge function can also cause some friction in merging a pull request. With the auto-merge function the pull requests will be automatically merged when all branch policies pass. This is useful as a build in the CI requires approximately 30 minutes or even longer, within this time it is possible the request is already approved. Once the build is finished and passes the pull request can be automatically merged without manual intervention.

There is also an option for the reviewers to vote *approved with suggestions*, meaning the pull request has comments deemed not interesting enough preventing the pull request from being merged. Once the author actually thinks no change is needed the author can merge the request immediately, cutting out the back and forward discussion of the comment being irrelevant.

In many cases *approved with suggestions* vote and auto-merge function probably decreases the wait time of the pull request, however the combination also exposes the risks of a pull request being merged before the author can actually read the comments. In most cases this are probably minor improvements and are fixed in a later pull request or added as technical debt. A solution to this, used by at least a single team in our knowledge, is using the feature of enforcing that all comments requires the status fixed, closed or won't fix, in other words the comments should be marked as solved.

CODING GUIDELINES

During the interviews we asked the developers for more explanation regarding the code improvements given as feedback. They mentioned following the general coding principles of the C# language or the specific language they are working with. This means, styling and structure improvements, and using a native method over a self written approach. Hereby, the number of code lines needed can be reduced to increase readability. Though, the fellow developers should still be able to understand the code.

However, a more interesting finding is at least 6 developers mentioned they find it sometimes hard to know where to check on regarding coding guidelines. Therefore, the reviewers tend to focus more on the understandably and functionality aspects and the obvious mistakes rather than the evolvability and maintainability aspects of the code. Hereby, Greenchoice's guidelines are less useful as this are high level guiding principles consisting of: KISS, DRY, YAGNI, SOLID. These guidelines provide no examples on how to apply them in practice. We later discovered there exist a deprecated Wiki page with an extensive list of coding guidelines with examples focusing on the readability and maintainability of the code. We are unaware how familiar the interviewees are with this Wiki page. From the fact that none of the developers mentioned this

Wiki page during the interviews we assume few developers are aware of the existence of the Wiki page and fever are using it.

In additional, due to the lack of clarity in the guidelines at least two developers specifically mentioned they are in occasions uncertain whether or not to leave a comment. They think they lack knowledge assessing what is correct or are afraid of becoming the bogeyman as their comments might come through as nitpicking. Providing a clear list of rules can encourage the reviewer to place the comment as they can fall back on the list with the rules as an agreement made to guarantee the quality. However, at least another developer did not share the feeling of uncertainty and does not mince her words.

Developers who are more familiar with the coding guidelines admitted they are sometimes a bit lax during a code review and could be stricter, especially on the maintainability and evolvability aspects. In general 8 interviewees mentioned reviews could be stricter, however the developers are unknown how much profit is gained against the cost. Another cause for being lax is the pressure for functionality from business. Here, at least two developers mentioned they experience time pressure from business having less time available for checking on code quality when writing code and during reviewing, with as results that the functionality is rushed in the code base.

TESTING

During the interviews there was less attention paid to testing unless we specifically asked about it. In total we ended up in 10 interviews discussing the testing process and how the code review process affects the testing process. In the review tactics we described that the functionality check is mainly the responsibility of the author and the tester. In the introduction we learned most testers have no prior programming experience. Testing itself is mostly done through manual testing and requires additional documenting. However, the testers of Greenchoice are educated in order to automate more of the testing process. In contraction we showed that reviewers also tend to focus on the functionality part. We have to clarify this in the sense that reviewers look for mistakes in behaviour based on the actual code they are reading. The testers focus on the actual correctness of the features through manual testing running the application.

This does not mean developers have no contribution in testing, they should already write unit tests covering the critical parts of the code. The definition of *critical parts of the code* is not well specified and is up to the developers to assess. However, in general it could be said that all logical code which is not obviously correct, such as native methods and third party software, should be unit tested. This does not include parts that are hard to test with unit tests such as establishing network connections and GUI code. An exception to this is the legacy code which mostly does not contain tests. Also, attempts writing tests for parts of the legacy code have not succeeded as it would take too much time to get it functional.

The newer parts of the code have a higher percentage of code coverage and more attention is paid to testing, however it may occur that writing tests is forgotten. This can cause delay to the pull request or the testing is fixed as a separate task later, in most cases the next pull request. Working on hybrid code, new and legacy code, it can be hard to write tests as the legacy code is not suited for unit tests. Additional, the newer code might not always be written with testability in mind. Lastly, at least 5 developers mentioned they barely check on the quality of the unit tests in code reviews and solely look whether tests are present.

KNOWLEDGE SHARING

One of the strengths of code reviews is learning throughout code reviews. Although, it is not at length discussed 9 developers mentioned they are learning through code reviews. Developers learn from receiving feedback and also from giving feedback. A developer mentioned the intention of giving feedback is to teach others to prevent similar mistakes in the future, additional the developer can learn from her own mistakes exposed by others.

We encountered an example that learning from code reviews does not make a developer an expert. A mistake can easily slip through and serve as a wake-up call for the team. For example, a single developer in a team worked mostly on a new part of the code in a new language for the specific team. The other developers were no expert in that language and went quickly over the requests. However, once the other developers also started working on the code and became more familiar, they discovered several issues which had to be refactored.

TEAM COMPOSITION

The scrum teams of Greenchoice have different compositions making the code review process also slightly different for each team. Most teams have 3 developers, a single team has instead of a dedicated tester 4 developers. In addition, the team responsible for the customers portal also includes 3 graphical designers in

addition to the three developers. Teams consisting of all experienced developers are more familiar with each other and are already on the same line of code quality. Therefore, only a quick sanity check is enough as a code review trusting the author of the code. However, a pull request coming from another team gets a more thorough review. Teams having newer or less experienced developers, the experienced developers have to pay more attention to the quality of the pull request and educating the lesser experienced developers. The other way around an experienced developer might need a critical review. It may happen that there is no experienced developer available in that team. In that case the author turns to an experienced developer of another team. The downside is that the other experienced developer has to spend additional time getting familiar with the code, though in most cases an experienced developer with some prior knowledge is available.

SONARQUBE

To measure the quality of the code base and detect code quality issues Greenchoice uses SonarQube [54]. SonarQube assists the developers in detecting functional, maintainability and security defects and provides insight into the quality of the code [93]. Every two weeks a report is mailed to the whole development team with a summary of the code quality for the projects. The sonar analysis is runs every merge build to the master branch, some developers were not aware of this and thought the report was only ran during the nightly build. In additional, most developers looked only at the results through the biweekly report.

The developers being aware of the constant generation of the SonarQube reports mentioned the report are hard assess during code reviews as it is unclear how the report changed according to the new code. This makes the Sonar analysis not very useful during code reviews and teams are reacting on older issues which should preferably be solved early on. In the later stages of our research period Greenchoice switched from SonarQube to SonarCloud [55], this allows for easier integration of the Sonar analysis directly into the code reviews in DevOps. This is further explained in Section 5.1.4 as a potential improvement.

MICROSERVICES

Although this last aspect does not directly affect the code reviews process it plays a role in the intention going in a code review. One of the current goals of Greenchoice is to migrate from large applications to multiple smaller microservices to increase scalability. Knowing an application will be replaced by microservices, new features are temporarily rushed in the old code and code quality is of lesser concern. However, it is unknown for how long the application will be alive and this might be several years or eventually not be replaced at all, though new features were hacked in. It is a difficult trade off whether or not investing time in preserving the quality of the code during code reviews is worth it. In addition, whether solving the earlier introduced technical debt once the lifetime of an application is extended is worth it before the application is definitely phased out. The trade off makes it hard for at least 2 developers to assess through which viewpoint they should assess the code.

An advantage of the smaller microservices is, the teams are no longer dependent on the biweekly sprint release giving them the ability to release their software themselves. This reduces the rush of finishing reviews on time for the company's broad release.

Summary: In the strengths and weaknesses we discovered smaller pull requests are easier to review. Larger reviews are harder to review and might not get the attention needed for an in-depth review. Additional, larger pull requests exposes the risk of scope creep or feature creep overwhelming the reviewer. The branch policies together with features to ease the merge process can assist the developers in the merge process if used correctly, however, can also lead to some frustration especially when a merge is delayed. In terms of coding guidelines we discovered developers tend to focus on the understandability, functionality aspects and obvious defects rather than also maintainability and evolvability aspects. The developers do not always know how to assess the latter two aspects. Additionally, we found some code reviews are performed lax and code reviews could be performed in general more strict. Testing is not addressed often during code reviews and once addressed the presence of test is checked. The interviewees mentioned they value the education aspect of code reviews. Furthermore, we discovered different compositions of experience levels can influence the code review process. Additionally, SonarQube reports are not often used during code reviews to assist the reviewer in assessing the code quality. Lastly, we found the intention of a code review can differ according to the expected life time of an application.

5.1.4. POTENTIAL IMPROVEMENTS

In this section we show three potential improvements we discussed in the interviews. The first improvement is a new subject which we have not mentioned yet, this is the usage of a code review related dashboard. The goal of this dashboard is to improve the visibility of the effects of code reviews. In the results of the first questionnaire we discovered the effect of code reviews is not always visible, shown in Section 4.3. The other two improvements were introduced earlier and consists of the change from SonarQube to SonarCloud and how SonarCloud could be directly integrated in the code reviews process. In addition, the usage of a checklist to assist the developers to know where to look for during code reviews.

CODE REVIEW DASHBOARD

In the interviews we gauged the option to create a dashboard as overview of the results of code reviews to increase the visibility of the code review process based on the recommendations given by Rigby *et al.* [40]. Based on the signals in the metadata related to the code reviews problems in the code review process could be identified [40]. The content of the dashboard consists of basic features as the number of pull requests pending and closed including average time to merge a request. For the pull requests which are still open additional information could be shown, for example how long the pull request is already open and the size of the pull request. Furthermore, additional features can be included such as the change in the quality due to the pull request, number of comments placed and test coverage. In addition, the features could be ranked for each team. The dashboard should also provide an overview which is also understandable by the business consultant and product owners and even by all the employees of Greenchoice. This could help to get an understanding from business to slow down the addition of features for a sprint and focus on the quality of the software.

The developers liked the idea, however they also question the value of the dashboard. Several features, number of pull requests open, are already visible within DevOps itself and the Sonar reports give an insight into the code quality. Furthermore, they do not necessary think others besides the developers and the ones directly involved in the development should be aware of these features. Additional, a developer mentioned the privacy aspect of the authors and reviewers. The dashboard can expose data on the performance of a team or individuals to the rest and is afraid that such a dashboard is used as measurement of performance by the managers. However, we think similar to Fagan [8] code reviews should never be used to measure the performance of developers.

Based on the feedback we did decide to not look further into the usage of a dashboard. The time needed to create such dashboard will take probably more time than it would be considered effective by the developers.

INTEGRATION OF SONAR REPORTS

In the interviews we addressed the usage of the SonarQube reports, however we discovered the report is barely used as source of information during code reviews in practice. In our opinion the integration of the right tooling can provide valuable assistance in code reviews. The change from SonarQube to SonarCloud gives the opportunity to integrate the feedback given in the Sonar analysis directly in the pull request. This means the new warnings and errors detected by the SonarCloud are automatically placed as comment within the pull request. There is also an online overview page available of the reports, including, once set correctly, the difference between master branch and the other branches. In addition, the Sonar report could be added as gatekeeper to prevent a merge once there are new warnings or errors. Including the Sonar warnings as gatekeeper could be useful as Panichella *et al.* [94] showed that 6 to 22% of static analyse warnings are resolved during code reviews. This is slightly higher than Kim and Ernst [95] who found 10% of the warnings are solved.

We did look into the linkage of SonarCloud to DevOps in the code review process, though we did not manage to get it fully operational. We did manage to gain separate reports based on the changes per branch. The inclusion of the warnings and errors in the code reviews as comments in DevOps did not succeed. There are multiple options to link SonarCloud and DevOps and include the analysis in the build process, due to security issues we did not gain access to these settings and lacked important linkage keys. We were dependent on the support team to be able to experiment with the linking of SonarCloud and DevOps, unfortunately they had little time available at that moment we looked into this and we had to create a separate request to include this in the sprint of the support team, however this required additional time and our period at Greenchoice ended.

USAGE OF A CODE REVIEW CHECKLIST

Lastly we looked into the usage of a checklist during code reviews. In one of the interviews a developer mentioned the existence of an old code review checklist, however none is using the checklist anymore. The

checklist was created roughly 6 years ago when Greenchoice started to use code reviews. In the interview we discussed that a similar checklist could be recreated. Furthermore, an already existing checklist from the internet could also be used as checklist, however, a newly made checklist based on the preferences of the teams was preferred and also recommended by the literature[8, 69]. In addition, this checklist could be updated separately to the requirements of each team. The goal of the checklist is to provide guidance during a code review. In 3 other interviews we also discussed the usage of the checklist, in here, increasing the consistency within the team was mentioned and providing guidance in what to look for in code reviews. The latter we discovered earlier in code guidelines as one of the difficulties in the code review process. In the remaining research questions 3 and 4 we will look further into the creation and usage of a checklist.

Summary: In the interviews we discussed three potential improvement to the code review process. The first improvement is to increase the visibility through the usage of a dashboard containing the code review information. However, the required time would probably not outweigh the time investment for creating such a dashboard. The second option is to include the SonarCloud (successor of SonarQube) reports directly in the pull requests as feedback comments. However, we were unable to achieve this during our period at Greenchoice. The third option is to create a code review checklist to assist the reviewer to know where to look for in code reviews. The creation and usage of a checklist during code reviews is further explored in research question 3 and 4.

5.2. METADATA ANALYSIS

To explore the current code review process further we dived into the metadata regarding pull requests and code reviews available via the DevOps API [78]. We created a crawler that gathered a total of 8454 pull requests, the metadata of each pull request contains 35 features as explained in the methodology in Section 3.2.2. The pull requests gathered were opened from 19-07-2017 till 22-01-2020 and closed before 13:00 on 22-01-2020. We excluded pull requests that were opened but not closed yet as the data on those pull requests is not yet complete. In each section we explain the features used in more detail. Based on these features we are interested what the data can tell us about the current code review process.

First we look in Section 5.2.1 into the general information, this includes the evolution of the usage of pull requests over the years, the usage of descriptions and average number of committers, reviewers and approvers. Next, we look into the time features related to the code reviews in Section 5.2.2. In Section 5.2.3 we look how each of the time features correlate to each other. The third group of features we look into are the size features in Section 5.2.4, the size features are divided over the number of lines and files changed in the pull request. To compare the size features we first look how the different size feature correlate to each other in Section 5.2.5, next we compare the size features to the time features in Section 5.2.6. The last group of features we look into in Section 5.2.7 is the user interaction during code reviews, this includes the number of comments placed and the number of iterations used to solve the comments. Finally, we look at the correlation of user interaction features to the time and size features in Section 5.2.8.

5.2.1. GENERAL INFORMATION

First of all, to discover the evolution of the code review process we looked to the number of pull requests per month from the moment Greenchoice started to use DevOps around 19-07-2017 till 22-01-2020 to discover how the intensity of the usage of pull requests evolved over time. In Figure 5.1 we show a time series histogram of the total number of pull requests each month combined with the unique number of authors and projects of that month. From October 2018 onward the number of pull requests started to increase till January 2019. The increase around is explained by the fact that around October 2018 onward the usage of a pull request was enforced through branch policies and thereby also the usage of code reviews.

The number of pull requests started to climb once more, after a small drop, from July 2019 till September 2019. The graph ends with a drop for December 2019 and January 2020. We expect this is due to the holidays and the data for January 2020 is not yet complete. Strangely, the drop around the Christmas and new year holidays is not noticeable in previous years.

We are interested whether the increase in pull requests can be explained by the addition of more developers and how this relates to the number of projects. To further explore this we show in Figure 5.2 the monthly average pull request per author and project. Notice, we added the number of pull requests as guidance divided by factor 25 to scale well with the data. Both features increased heavily from October 2018 till December

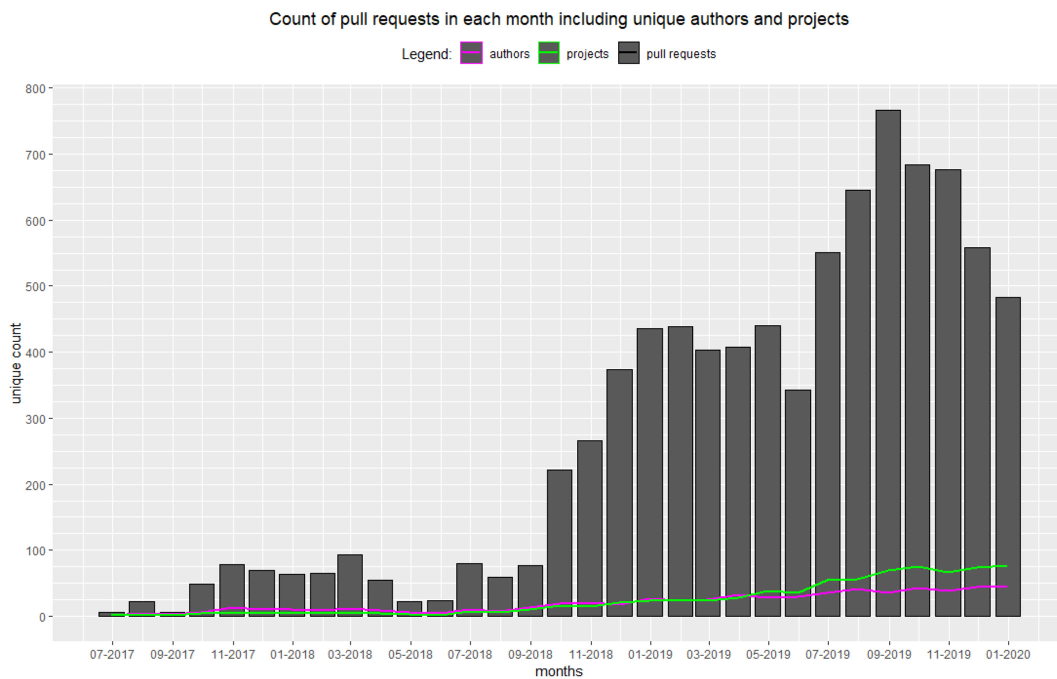


Figure 5.1: Time series histogram showing the total number of pull requests each month together with the average number of unique authors and projects of the corresponding month.

2018. Based on Figure 5.2 the monthly average pull requests per author has steadily increased from that moment ranging 12 to 22 monthly pull requests per author instead of 3 to 8 monthly pull requests. From this moment on the usage of pull requests was obligated through the usage of branch policies.

In addition, during the peak in the number of pull request for the month September 2019, the average number of pull requests per author also increased. We are unsure what caused the peak. We also looked into whether the sizes of the pull requests that month decreased, however based on average size and median size, shown in Figure 5.3 we could not discover any abnormality for that month compared to the other months.

Lastly, in figure 5.2 the average number of pull requests per project decreases from June 2019 onward, this might be the result of splitting the larger projects into smaller microservices.

PULL REQUEST DESCRIPTION

Another interesting aspect we noticed is the description provided in pull requests. Exploring the metadata we encountered a lot of descriptions similar to the title of the pull request or empty descriptions. Additionally, in the interviews we found that only two developers mentioned they gave additional guidance through the description of the pull request. In DevOps the title and description are generated based on the latest commit and are by default similar. Therefore, we counted how many pull requests have an empty description and how many pull requests have a similar title and description. We found that 1691(19.99%) pull requests have an empty description and 5426(64.15%) pull requests have a similar title and description. In addition, 817(9.66%) pull requests contain the title in the description. This means, that at least 84.14% of the pull requests do not contain specific guidance on what has changed and how to go through the pull requests.

Ending the general data we looked into the average number of committers (NumCommitters), reviewers (NumReviewers) and approvers (NumApprovers) for each pull request. An approver voted approved in the pull request where a reviewer could also only left a comment or another vote, thus an approver is also counted as reviewer. We found on average 1.02 committers, 1.62 reviewers and 1.09 approvers for each pull request.

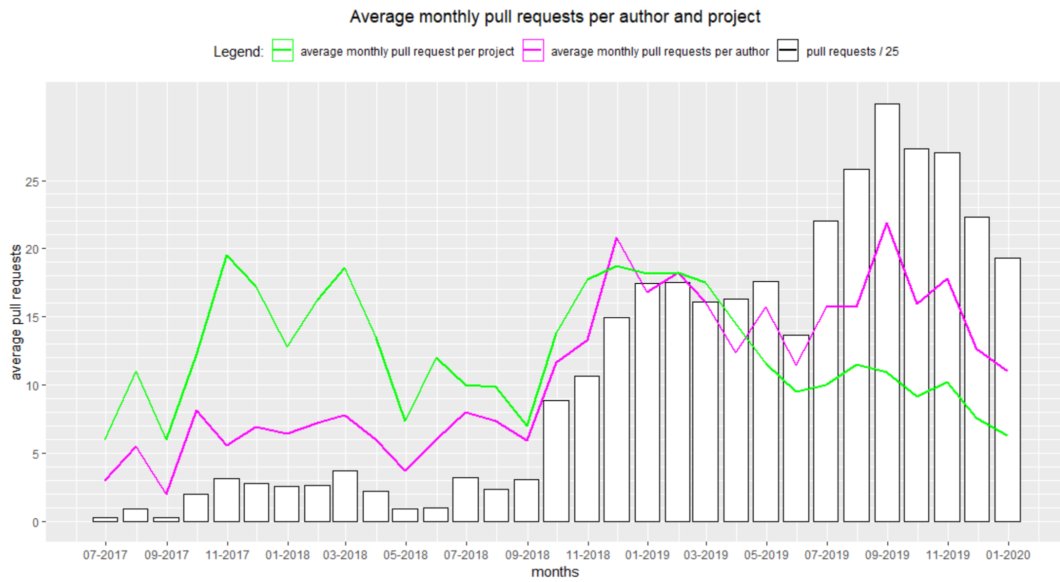


Figure 5.2: Time series histogram showing the average number of monthly pull request per unique author and project. Notice, the total number of pull requests that month is also showed, however divided by factor 25.

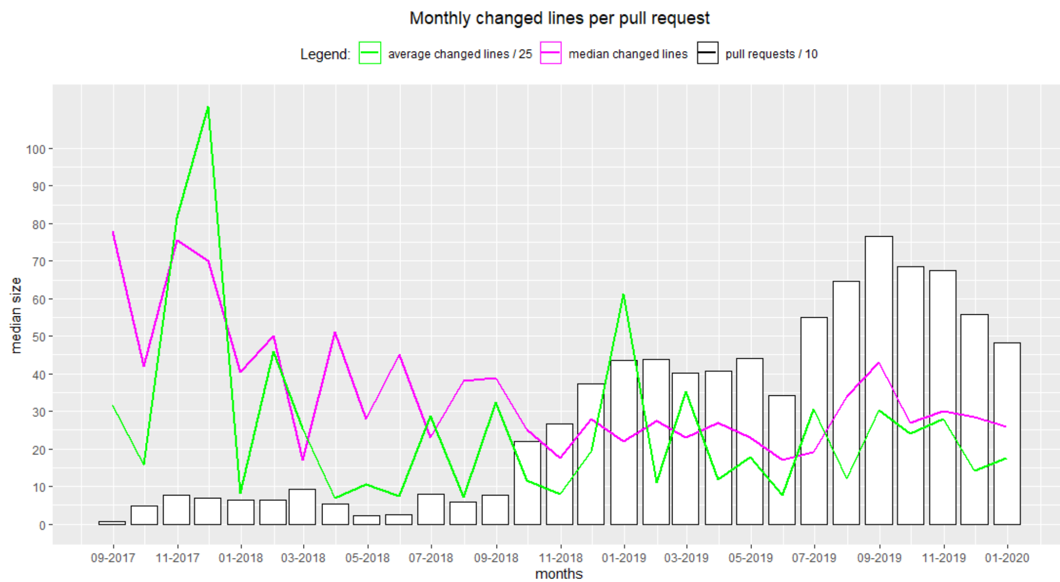


Figure 5.3: Time series histogram showing the monthly average and median changed lines per pull request. Notice, the average amount of changed lines is divided by factor 25 and the number of pull requests per month is divided by factor 10.

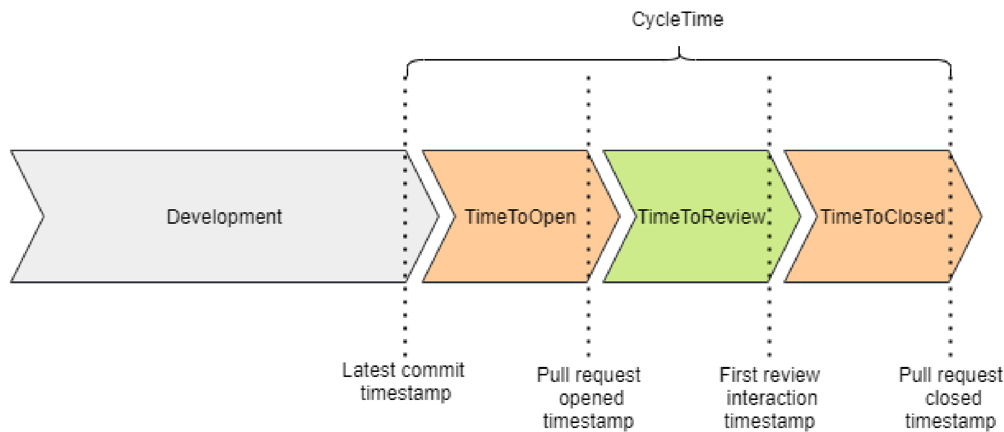


Figure 5.4: Overview of the time features

Summary: From October 2018 Greenchoice adopted its current code review process on DevOps by enforcing the usage of pull requests and thereby the usage of code reviews. From October 2018 each developer made on average 12 to 22 pull requests on a monthly basis. We furthermore discovered that little guidance is given in the description of a pull request of which in 84.14% of the pull requests no description is given at all or the default description. Lastly, we have showed that on average a single developer commits to a pull request, there are on average 1.62 reviewers involved in reviewing a pull request and on average 1.09 approvers for each pull request.

5.2.2. EXPLORATION OF THE TIME FEATURES

One of the most interesting features stored in the metadata is the time spent on a pull request. Most companies aim to reduce the review time for each pull request. With a short review time defects are found early on in the process and reduces the cost of resolving the defects later on [8]. In this section we explore the following features of the pull requests: CycleTime, TimeToOpen, TimeToReview and TimeToClosed.

Hereby, the CycleTime is the time between the last commit before opening a pull request and merging or closing a pull request. The CycleTime can be divided over the other three detailed features: TimeToOpen, TimeToReview and TimeToClosed. TimeToOpen starts at the last commit before creating the pull request and ends once the pull request is created, TimeToReview starts from that point till the first vote is given and TimeToClosed from that point till the actual merge of the pull request. On overview of the time features is shown in Figure 5.4, whereby the time features are calculated based on the gathered timestamps from DevOps.

We gathered the times used in the data analysis in seconds which is hard to interpreted, especially for larger values. We therefore converted the seconds by simply dividing the number of second by 60 or 3600 for respectively minutes and hours, depending on the usefulness of the scale. Initially we sorted the data and manually looked at the minimum and maximum of the features to get a first impression of the range of the data. To explore the distribution of the data we used the 4 different plots and percentiles as explained in the methodology 3.2.2.

CYCLETIME

By manually looking at the minimum and maximum value for the CycleTime we found the shortest CycleTime was 12 seconds and the longest CycleTime took roughly 2666 hours equal to roughly 4 months (roughly 111 days). Next, we explored the data of the CycleTime through the 4 plots shown in Figure 5.5. In the graphs we show the data is clustered within the first hours. However, the distribution within these first hours is unclear in these plots.

In Figure 5.6 we show therefore the distributions in a histogram and Q-Q plot of three subsets of the CycleTime: within the first week, first day and first hour. These plots verifies that the majority of the pull

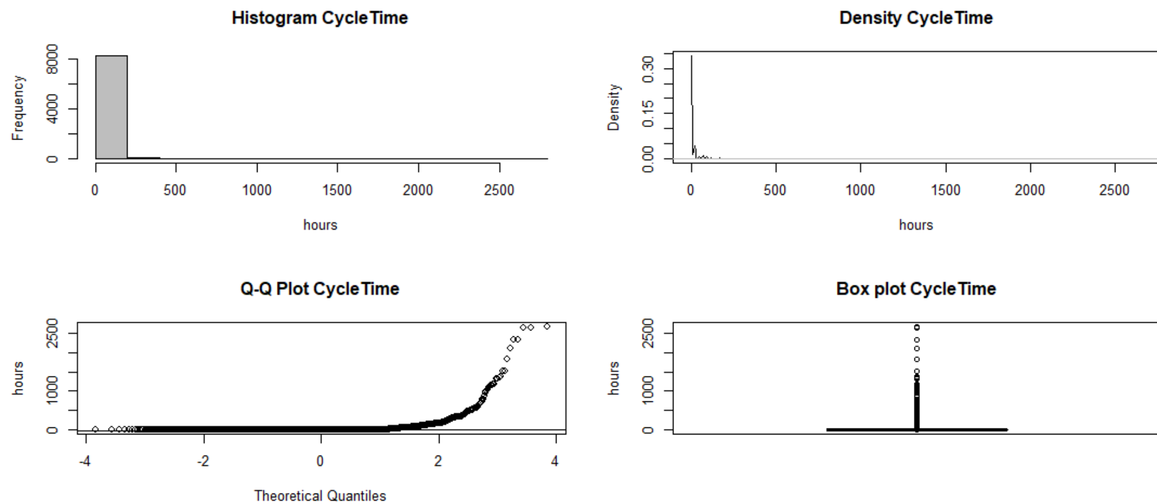


Figure 5.5: The four exploration plots of the CycleTime of the full data set. The plots are histogram plot (upper left), density plot (upper right), Q-Q plot (lower left) and box plot (lower right).

Minutes	**1**	**15**	**30**	**60**	**1440**	**2880**	**10080**	**40320**
Hours	**0.0167**	**0.25**	**0.5**	**1**	**24**	**48**	**168**	**672**
TimeToOpen	50.01%	80.56%	83.14%	85.52%	93.33%	95.06%	98.56%	99.80%
TimeToReview	16.30%	60.07%	69.83%	77.64%	95.41%	96.65%	99.59%	99.96%
TimeToClosed	49.79%	81.40%	85.61%	89.18%	97.46%	98.11%	99.65%	99.94%
CycleTime	2.05%	42.31%	52.35%	61.37%	86.85%	90.44%	97.56%	99.66%

Table 5.1: Reversed percentiles of the full data set of the CycleTime and detailed time features: TimeToOpen, TimeToReview and TimeToClosed. Based on a set period the percentage of pull requests within that period is shown.

requests are merged within the first hour of the latest commit. This is in our perception rather quickly and requires a rather quick form of a review. Diving deeper into the percentiles shown in Table 5.1 we found that roughly 2.5% of the pull requests have a CycleTime within the first minute, 61.4% within the first hour, 86.9% within a day, 90.4% within 2 days, 97.6% within 7 days and 99.7% within 28 days.

Based on the short CycleTime of several pull request we question whether a code review has taken place for each pull request and the form of the code review. We discovered that 8278 of the 8458 pull requests have at least a single approval, indicating that a form of a code review is performed by another developer. The remaining 180 pull requests did not have an approval, for these we are unsure whether a form of a code review was performed. In the next part of this section we show how the CycleTime is divided over the detailed time features. In the remainder of this chapter we furthermore look into the size and user interaction features of the pull requests and later to the types of comments given in the code reviews.

EXPLORATION OF THE DETAILED TIME FEATURES

The exploration of the three detailed time features, TimeToOpen, TimeToReview and TimeToClosed, in general gave similar graphs to the exploration graphs of the CycleTime. The data was once more clustered within the first hours. The only remarkable observation we made during the exploration was that TimeToOpen has more values in the right tail of the Q-Q plot, indicating TimeToOpen has more higher values compared to the other detailed features. This is also noticeable in Table 5.1, in which we also added the detailed time features, as TimeToOpen has roughly 1% more pull requests with a value higher than 168 hours compared to TimeToReview and TimeToClosed. However, We were not able to find a clear reasoning why the creation of several pull requests are delayed this much.

Similar to the CycleTime we explored for each detailed time feature the distributions in more detail based on the three subsets: within the first week, first day and first hour shown in Figures 5.7, 5.8 and 5.9 for respectively TimeToOpen, TimeToReview and TimeToClosed. In the histogram plots we show that the values of the three detailed time features lie, once more, mostly within the first hour. This is also noticeable in the reversed percentiles table, Table 5.1, with 77.64%, 85.52% and 89.18% of the pull requests have a value within the first

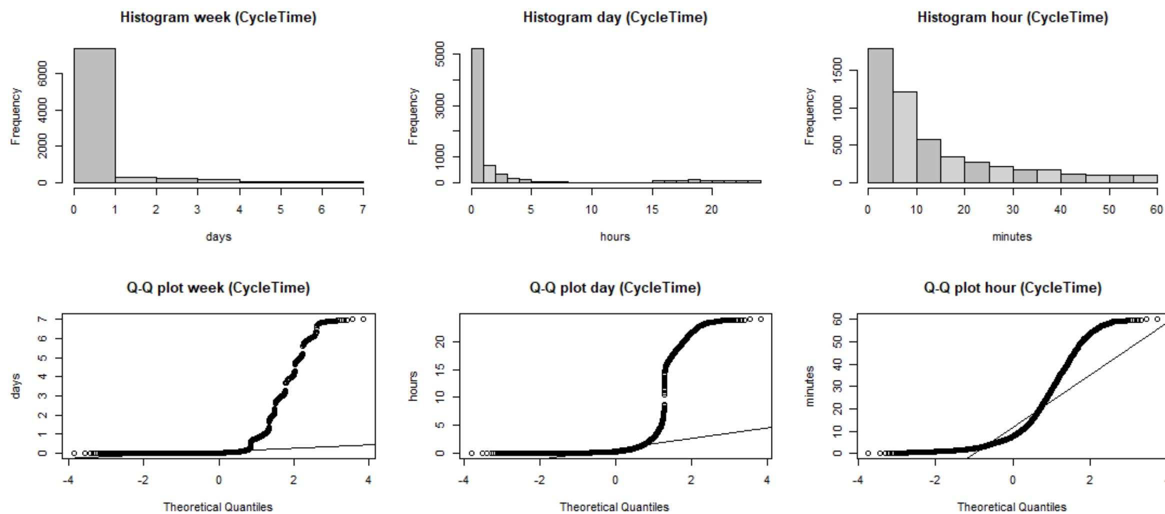


Figure 5.6: Distribution of the CycleTime in three subsets of the data visualized in a histogram plot (upper row) and Q-Q plot (lower row). The subset are a CycleTime under a week (left plots), under a day (middle plots) and under an hour (right plots)

hour for respectively TimeToReview, TimeToOpen and TimeToClosed.

Analysing the detailed time features TimeToOpen and TimeToClosed have the most pull requests within the first hour shown in Table 5.1. Additional, TimeToReview has 16.30% of the pull requests reviewed within the first minute and 60.07%, this is a rather large number of pull requests that is barely reviewed. Though, the TimeToReview values are more spread out over the first hour, based on the histogram and Q-Q plot in Figure 5.8 compared to the plots of TimeToOpen in Figure 5.7 and TimeToClosed in Figure 5.9 in which most values lie within the first 5 minutes. This is somewhat expected as another developers is involved in the review process. In case of the creation of a pull request the author of the latest commit can directly open a pull request once he has pushed the latest commit. In addition, the merge or close of a pull request could also be directly performed by the reviewer once the pull request is approved. Though, in this case it is also possible that feedback should be processed before the merge could be performed, however, this is not noticeable in the data. This indicates few feedback is given and processed in which we will look further into during the analysis of the user interaction features in Section 5.2.7.

Based on our data exploration we made the two hypothesis listed below. In the next section we will look into whether these two hypothesis are correct through analysing the correlation between the time features.

- TimeToReview is responsible for the majority of the CycleTime values, this is based on the observation that TimeToReview is more spread out within the first hour than the other two detailed time features.
- TimeToOpen is responsible for the pull requests with a CycleTime of more than 168 hours, this is based on the observation that TimeToOpen has the most and highest outliers compared to the other two detailed time features.

Summary: The majority, 87%, of the pull requests are merged within the first 24 hours, a day, of the latest commit. A large part, 68%, of the pull requests has a CycleTime under 60 minutes and 52.35% of the pull requests under 30 minutes. Based on the detailed time features the majority, 77.64%, 85.52% and 89.18%, of the pull requests have a value within the first hour for respectively TimeToReview, TimeToOpen and TimeToClosed. The time between the creation of the pull request and the first vote is actually rather quick in which 16.30% of the pull requests are reviewed within the first minute after creation and 60.07% in the first 15 minutes. Based on these rather quick review times we question whether the reviews are not performed too quickly which we will further discuss in the discussion.

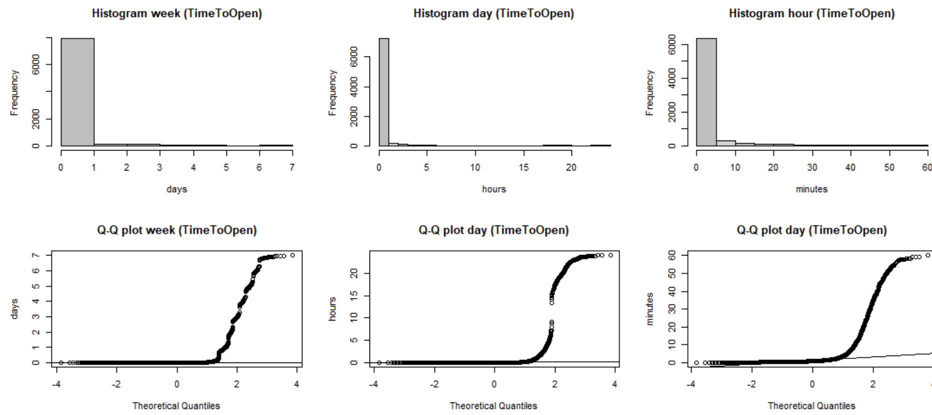


Figure 5.7: Distribution of the TimeToOpen feature in three subsets of the data visualized in a histogram plot (upper row) and Q-Q plot (lower row). The subset are a TimeToOpen under a week (left plots), under a day (middle plots) and under an hour (right plots)

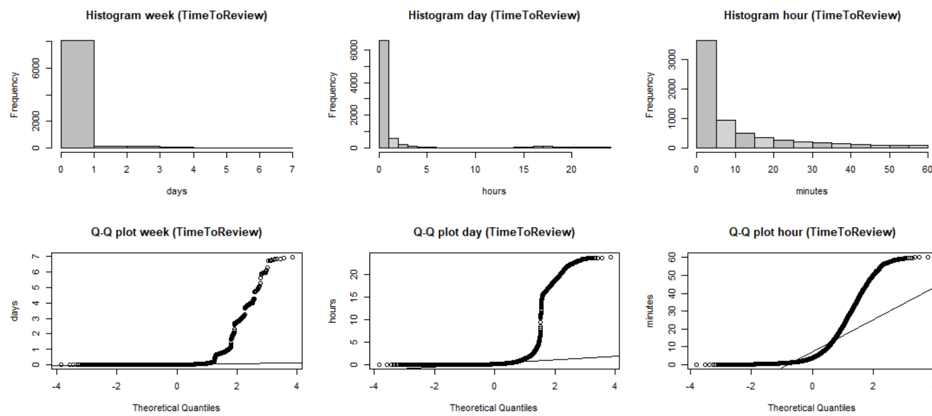


Figure 5.8: Distribution of the TimeToReview feature in three subsets of the data visualized in a histogram plot (upper row) and Q-Q plot (lower row). The subset are a TimeToReview under a week (left plots), under a day (middle plots) and under an hour (right plots)

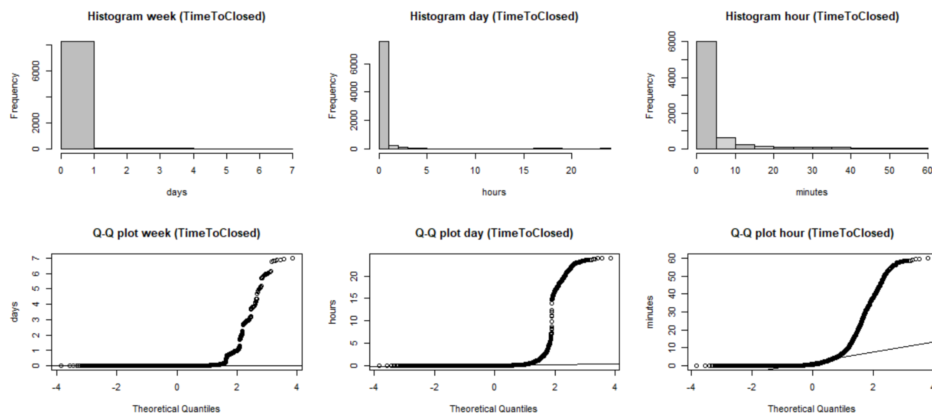


Figure 5.9: Distribution of the TimeToClosed feature in three subsets of the data visualized in a histogram plot (upper row) and Q-Q plot (lower row). The subset are a TimeToClosed under a week (left plots), under a day (middle plots) and under an hour (right plots)

5.2.3. COMPARING THE TIME FEATURES

In the previous sections we explored the individual time features. In this section we explore the correlation of the detailed time features to the CycleTime and whether the detailed time features are correlated to each

other. Our hypothesises, made in the previous section, are that TimeToReview has the highest correlation to TimeToReview, however, TimeToOpen has the highest correlation for pull requests with a CycleTime of more than 168 hours.

We created a scatter plot matrix of the 4 time features, shown in Figure 5.10, including for each scatter plot their corresponding Spearman correlation as explained in the methodology in Section 3.2.2. Additionally, we created two subsets based on the CycleTime in order to answer whether TimeToOpen has the highest correlation for the higher values of the CycleTime. The subsets are separated on the CycleTime up to and including 168 hours, equal to 7 days, and more than 168 hours. The first subset consists of 8252 pull requests with a CycleTime of 7 days or shorter. The other subset contains the remaining 206 pull requests with a CycleTime of more than 7 days. We created similar to the full data set two scatter plots of the time features including the Spearman correlations.

In Figure 5.10 we show the scatter plot matrix of the time features including the Spearman correlation of the full data set. The highest correlation to CycleTime is TimeToReview with a strong correlation of $r_s[8458] = 0.75, p < 0.001$, this indicates a statistically significant positive relationship between CycleTime and TimeToReview. In other words CycleTime grows as TimeToReview grows and vice versa. TimeToOpen follows with a moderate correlation of $r_s[8458] = 0.55, p < 0.001$ and TimeToClosed has a weak correlation of $r_s[8458] = 0.38, p < 0.001$ to CycleTime, both correlations are also considered statically significant. The correlation between TimeToOpen and TimeToReview is low with $r_s[8458] = 0.26, p < 0.001$; the other correlations are lower than 0.2 and are considered very weak. The latter does indicate that a delay in one of the detailed time features does not cause another delay in one of the remaining steps.

To be able to answer the second hypothesis we show in Figure 5.11 the scatter plot matrix of the subset of pull request with a CycleTime longer than 7 days. The correlation between CycleTime and TimeToOpen has a moderate correlation of $r_s[206] = 0.54, p < 0.001$. The correlation between CycleTime and TimeToReview changed to $r_s[206] = -0.02, p = 0.821$. A negative correlation indicates the value of CycleTime increases when TimeToReview decreases and vice versa, though a correlation of 0.02 is very weak. Additional, the correlation between CycleTime and TimeToClosed is very weak with $r_s[206] = 0.04, p = 0.595$. In case of the latter two correlations the p value is higher than 0.05, indicating that the discovered correlations can be caused by coincidence and are not statistically significant.

The correlation of TimeToOpen to the other detailed time features is weak to moderate with $r_s[206] = -0.38, p < 0.001$ and $r_s[206] = -0.41, p < 0.001$ for respectively TimeToReview and TimeToClosed. The negative correlation is logical as having already a higher TimeToOpen value reflects that the other detailed time features should be lower based on the maximum value of the CycleTime.

In Figure 5.12 we show for completeness the scatter plot matrix of the subset of pull request with a CycleTime of 7 days and shorter. For the lower subset the values are slightly different but no changes of importance occurred, this could be expected as the data set is very similar to the full data set.

Summary: We verified our stated hypothesises of the previous section. TimeToReview is the detailed time feature with the highest correlation to CycleTime with a correlation of $r_s[8458] = 0.75, p < 0.001$ in the full data set. Additional, we retrieved weak correlations between the detailed time features themselves, indicating that a delay in one of the features does not lead to a delay in one of the other detailed time features. Furthermore, we showed that TimeToOpen is indeed most responsible for the CycleTime for the subset of pull requests with a CycleTime of more than 168 hours with a correlation of $r_s[206] = 0.54, p < 0.001$.

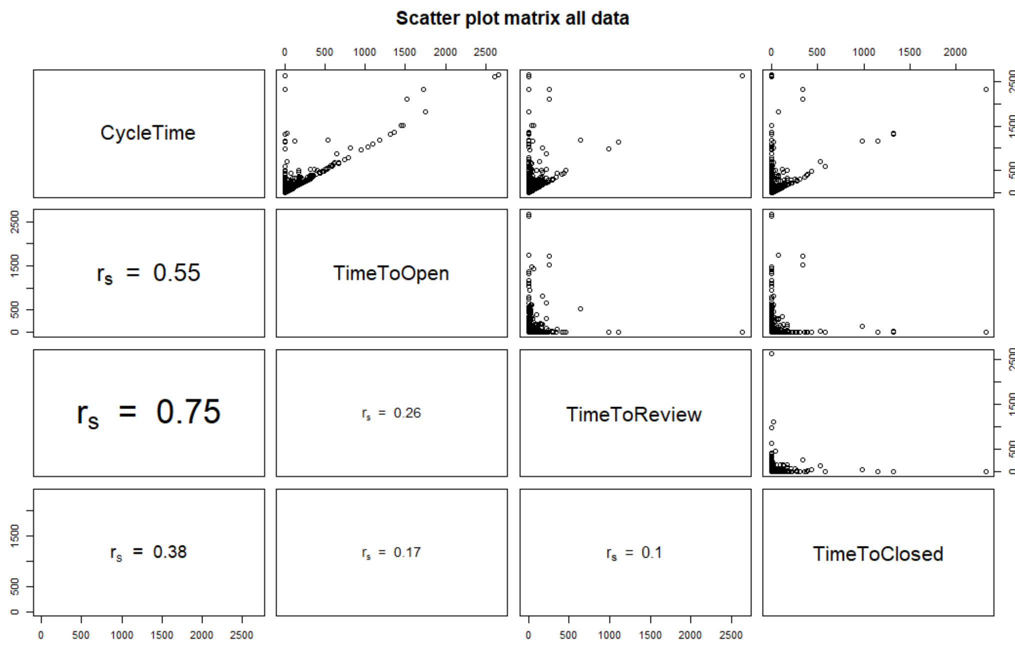


Figure 5.10: Scatter plot matrix of the CycleTime and detailed time features: TimeToOpen, TimeToReview and TimeToClosed consist of the full data set. In the upper right part the scatter plots are shown and in lower left part the Spearman correlation between the corresponding features.

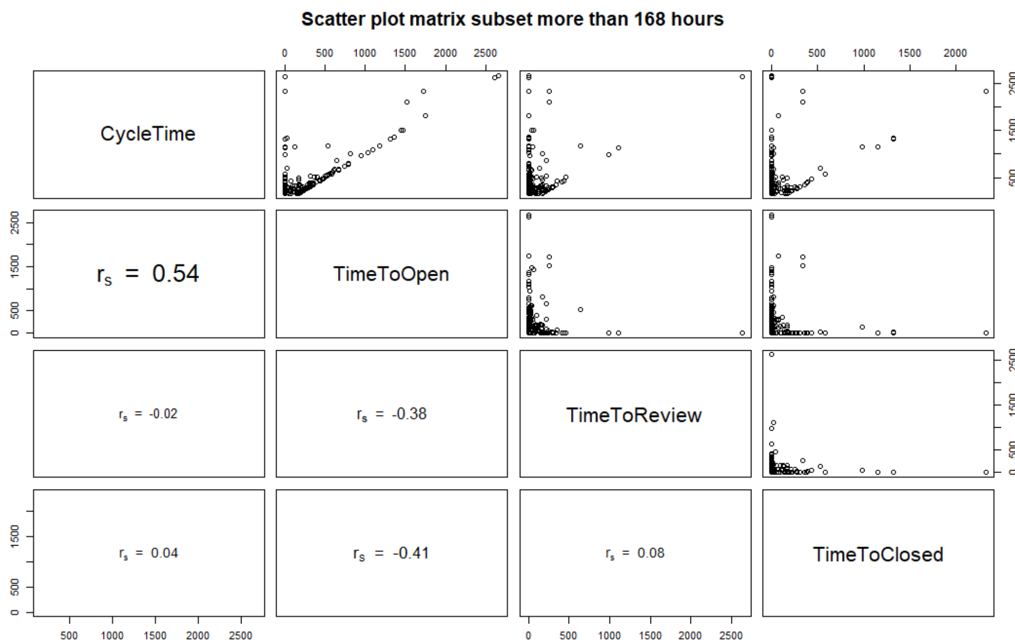


Figure 5.11: Scatter plot matrix of the CycleTime and detailed time features: TimeToOpen, TimeToReview and TimeToClosed. The data shown is data set of pull requests with a CycleTime longer than 7 days. In the upper right part the scatter plots are shown and in lower left part the Spearman correlation between the corresponding features.

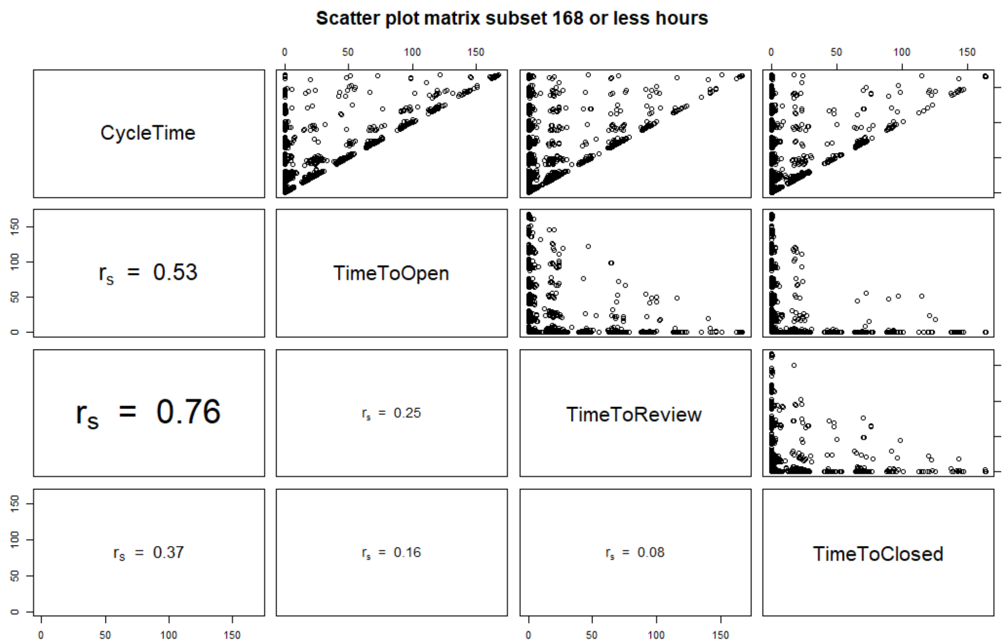


Figure 5.12: Scatter plot matrix of the CycleTime and detailed time features: TimeToOpen, TimeToReview and TimeToClosed. The data shown is data set of pull requests with a CycleTime of 7 days or lower. In the upper right part the scatter plots are shown and in lower left part the Spearman correlation between the corresponding features.

5.2.4. EXPLORATION OF THE SIZE FEATURES

The next aspect we looked into is the size of the pull requests. We have discovered in the interviews that larger pull requests are harder to review and require more time. In the data we captured the size of a pull request in two features summarizing the total number of changes; the first feature is the total number of changed files, NumTotalChangedFiles and second feature is the total number of changed lines NumTotalChangedLines. The feature NumTotalChangedFiles can be divided over the following detailed features: NumAddedFiles, NumDeletedFiles, NumEditedFiles, NumRenamedFiles, NumRenamedAndEditedFiles. The feature NumTotalChangedLines can be divided over the following detailed features: NumAddedLines, NumDeletedLines, NumModifiedLinesOld and NumModifiedLinesNew.

In Figure 5.13 we show the data exploration of both total size features, NumTotalChangedFiles and NumTotalChangedLines, of the full data set and two subsets. The full data set includes a single outlier with more than 200.000 changed lines. We expect this is an initial commit of a project transferring the source code from Greenchoice's previous version control system team foundation services (TFS) to DevOps. In the manual exploration we noticed more of these types of pull requests occurred in the data, therefore we created the first subset excluding this type of pull request in which bulk code is added or deleted.

The first subset consists of pull requests with fewer than 5.000 changed lines and fewer than 100 changed files removing 136 pull requests from the full data set, this means 8322 pull requests remain. The second subset looks in more detail into the denser populated area and consists of pull requests with fewer than 1.000 changed lines and fewer than 60 changed files removing in total 531 pull requests from the full data set, this means 7927 pull requests remain. In the graphs of Figure 5.13 we show that most pull requests, 79.01%, have a size smaller than 200 changed lines and most pull requests, 84.42%, have fewer than 10 changed files, especially visible in the graphs of subset 2.

Furthermore, we calculated for NumTotalChangedLines in the full data set a median of 28 lines, 25th percentile of 6 lines and 75th percentile of 144 lines. Subsequently, we calculated for NumTotalChangedFiles in the full data set a median of 2 files, 25th percentile of 1 file and 75th of 6 files.

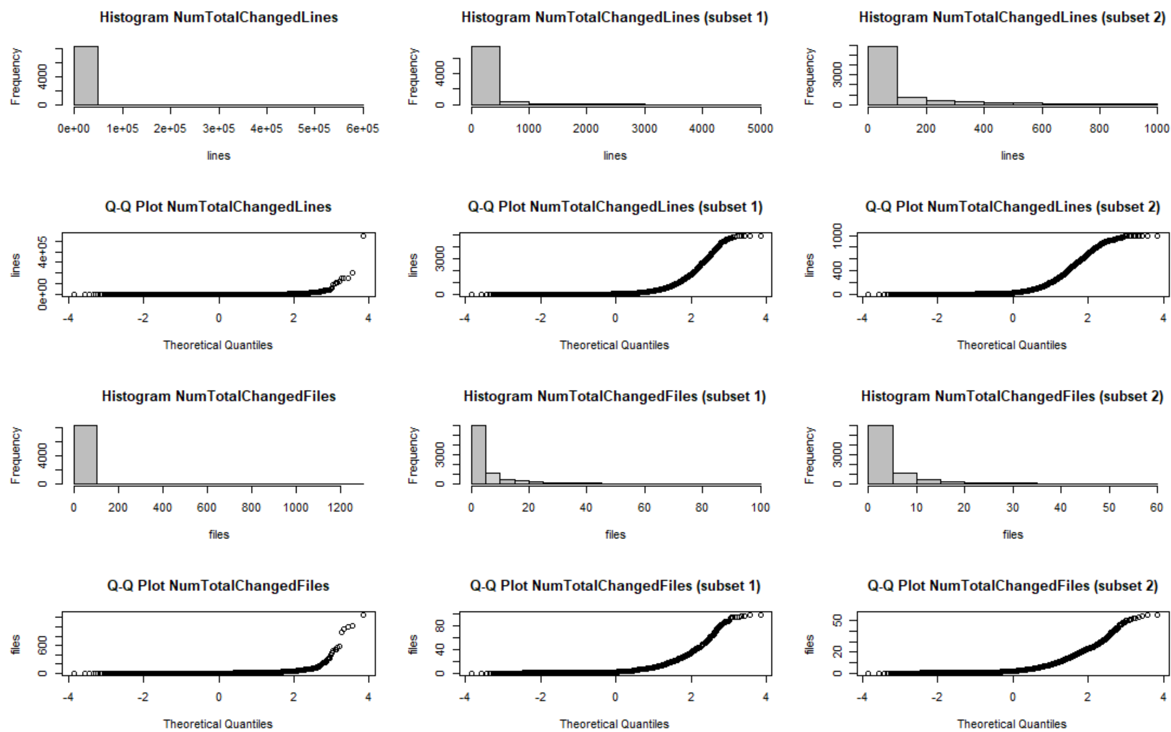


Figure 5.13: Exploration of the size features NumTotalChangedLines (upper two rows) and NumTotalChangedFiles (lower two rows). The plots of the full data set (left column), the first subset consisting of pull request with fewer than 5000 lines and 100 files changed (middle column) and the second subset consisting of pull request with fewer than 1000 lines and 60 files changed (right column).

Summary: We explored the size of the pull requests with the features NumTotalChangedLines and NumTotalChangedFiles. The majority of the pull requests have fever than 200 changed lines, 79.01%, and fewer than 10 changed files, 84.42%. Furthermore, we discovered a median of 28 lines changed and 2 files changed.

5.2.5. COMPARING THE SIZE FEATURES

In this section we show how the size features affect each other. This gives an insight in the type of changes based on the added, deleted and changed lines and files. In the first part we look into the relationship between the total size features and their corresponding detailed features, in the second part and third part we look into the relationship between the detailed files and detailed lines features themselves. Finally, in the fourth part we look into the relationship between the detailed files and lines features. In general we discovered for each correlation that it was statistically significant with $p < 0.001$, due to the large number of pull requests (8458).

First off, the correlation between the total size features NumTotalChangedLines and NumTotalChangedFiles is high with $r_s[8458] = 0.79, p < 0.001$. This indicates if the size of pull request grows both the number of lines changed and files changed increases.

In Table 5.2 we show the correlations of NumTotalChangedFiles and NumTotalChangedLines to their corresponding detailed features. For NumTotalChangedFiles we found a high correlation of $r_s[8458] = 0.88, p < 0.001$ to NumEditedFiles, this indicates that editing a file occurs more frequently than any other file change. In case of NumTotalLinesChanged we found a high correlation of $r_s[8458] = 0.79, p < 0.001$ to NumAddedLines, indicating code lines are mostly added.

COMPARISON OF THE FILES FEATURES

We also calculated the correlation of the files features to each other and the lines features to each other. The correlations between the files features were weak of which the highest correlation was $r_s[8458] = 0.32, p < 0.001$. In other words there is no clear relationship that when a files feature increases another files feature also increases or decreases.

	NumTotal ChangedFiles
NumEditedFiles	0.88
NumAddedFiles	0.55
NumDeletedFiles	0.39
NumRenamedAndEditedFiles	0.26
NumRenamedFiles	0.07

	NumTotal ChangedLines
NumAddedLines	0.79
NumModifiedLinesNew	0.60
NumModifiedLinesOld	0.60
NumDeletedLines	0.57

Table 5.2: Spearman correlations of the total size feature to their corresponding detailed size features.

	NumAddedFiles	NumEditedFiles	NumDeletedFiles
NumAddedLines	0.74	0.45	0.2
NumModifiedLinesNew	0.19	0.72	0.23
NumModifiedLinesOld	0.19	0.71	0.24
NumDeletedLines	0.26	0.54	0.52

Table 5.3: Overview of the Spearman correlations between the detailed lines and files features.

COMPARISON OF THE LINES FEATURES

In the comparison of the lines features we discovered that NumModifiedLinesOld and NumModifiedLinesNew features have a very high correlation to each other $r_s[8458] = 0.96, p < 0.001$. This is not surprisingly as deletion of a line also requires an addition of a new line in order to be counted as a modification. It is possible that two lines are deleted and replaced by a single line, however, this apparently barely decreases the strength of the correlation.

Furthermore, NumDeletedLines has moderate correlations to NumModifiedLinesNew, $r_s[8458] = 0.46, p < 0.001$, and to NumModifiedLinesOld, $r_s[8458] = 0.47, p < 0.001$. This indicates that it is likely that when a line is deleted another line is modified and vice versa. In case of NumAddedLines we discovered only weak correlations to the other lines features, indicating no relationship.

COMPARISON OF THE LINES TO THE FILES FEATURES

We are furthermore interested how the detailed files features are affected by the detailed lines features. The correlations between the files and lines features are shown in Table 5.3. Not surprisingly the correlation between NumAddedFiles and NumAddedLines is high $r_s[8458] = 0.74, p < 0.001$. However, the correlation between NumDeletedLines and NumDeletedFiles is only $r_s[8458] = 0.52, p < 0.001$. This indicates that the deletion of lines is also related to another type of file change, which is NumEditedFiles with a moderate correlation of $r_s[8458] = 0.54, p < 0.001$. NumEditedFiles has furthermore a high correlation to the NumModiefiedLines features, $r_s[8458] = 0.72, p < 0.001$ for New and $r_s[8458] = 0.71, p < 0.001$ for Old, and moderately correlated with NumAddedLines $r_s[8458] = 0.45, p < 0.001$. This indicates that NumEditedFiles is positively correlated to all lines features, whereby editing a line has the highest correlation.

Summary: In this section we showed how the pull requests are structured in terms of changed lines and changed files. In the first part of this section we have showed that when the size of pull request grows both the number of lines changed and files changed increases. Furthermore, NumTotalChangedFiles has the highest correlation to NumEditedFiles, $r_s[8458] = 0.88, p < 0.001$. Whereas the NumTotalChangedLines has the highest correlation to NumAddedLines, $r_s[8458] = 0.79, p < 0.001$.
In the comparison of the detailed size features we made the following observations:

1. The type of files changed tend to be separate from each other, there are only weak correlations between the detailed files features.

2. In terms of lines changed it is likely, $r_s[8458] = 0.47, p < 0.001$, that once a line is deleted a modification to another line is made and vice versa. In terms of added lines there is no clear evidence of a relationship to deleted or modified lines.

3. NumEditedFiles and NumDeletedFiles have both approximately similar correlations to NumDeletedLines with respectively $r_s[8458] = 0.54, p < 0.001$ and $r_s[8458] = 0.52, p < 0.001$. Whereas NumAddedLines has solely the highest correlation to NumAddedFiles, $r_s[8458] = 0.74, p < 0.001$. This means once NumDeletedLines grows it is equally possible this is related to a deletion of a file or editing a file. Whereas, NumAddedLines grows this is more likely related to an addition of a file than editing a file. Though, editing a file still has a moderate possibility.

4. The type of lines changed within an edited file is likely to be modification of lines $r_s[8458] = 0.72, p < 0.001$, however, there is also a moderate correlation to the deletion $r_s[8458] = 0.54, p < 0.001$ and addition $r_s[8458] = 0.45, p < 0.001$ of lines.

5.2.6. COMPARING THE SIZE FEATURE TO TIME FEATURES

Next, we looked into whether the size features of the pull request affect the CycleTime and the other time features. We used the full data set in the comparison, we did also explore the correlation values for the first subset of the size features, however, the correlation values had a difference of at most 0.01 which we considered of no importance. In Figure 5.14 we show the scatter plot matrix of the total size features and the time features with their corresponding Spearman correlation. Once more are the given correlations in this section statistically significant due to the large number of pull requests.

In general we show that the total size features are moderately correlated to the time features, two lower rows in Figure 5.14. With correlations to CycleTime of $r_s[8458] = 0.47, p < 0.001$ and $r_s[8458] = 0.42, p < 0.001$ for respectively NumTotalChangedFiles and NumTotalChangedLines. This indicates that there is some relationship, however no strict relationship, that the CycleTime increases as the size of a pull requests increases and vice versa. Similar results are found for TimeToOpen and TimeToReview, indicating that the total size features also can have a slight influence on these time features. However, for TimeToClosed the correlation is of no importance with a weak correlation of $r_s[8458] = 0.21, p < 0.001$ for both total size features.

We additionally looked into the correlation of the detailed lines and files features to the time features. However, we gained no new insights of based on roughy similar correlations compared to the total size features, of which their correlations to the time features are shown in Figure 5.14. As we gained no new insights with this comparison and the vast number of features involved resulting in an unclear visualization we left out a scatter plot matrix of the correlations.

Summary: In the comparison of the time features to total size features we discovered a moderate relation of the total size features to CycleTime with a correlations of $r_s[8458] = 0.42, p < 0.001$ to NumTotalChangedFiles and $r_s[8458] = 0.47, p < 0.001$ to NumTotalChangedLines. Additionally, slightly lower correlation values were found of the total size features to TimeToOpen and TimeToReview, however to TimeToClosed weak correlations were found. These correlations provide evidence that the size of the pull requests can affect the time features.

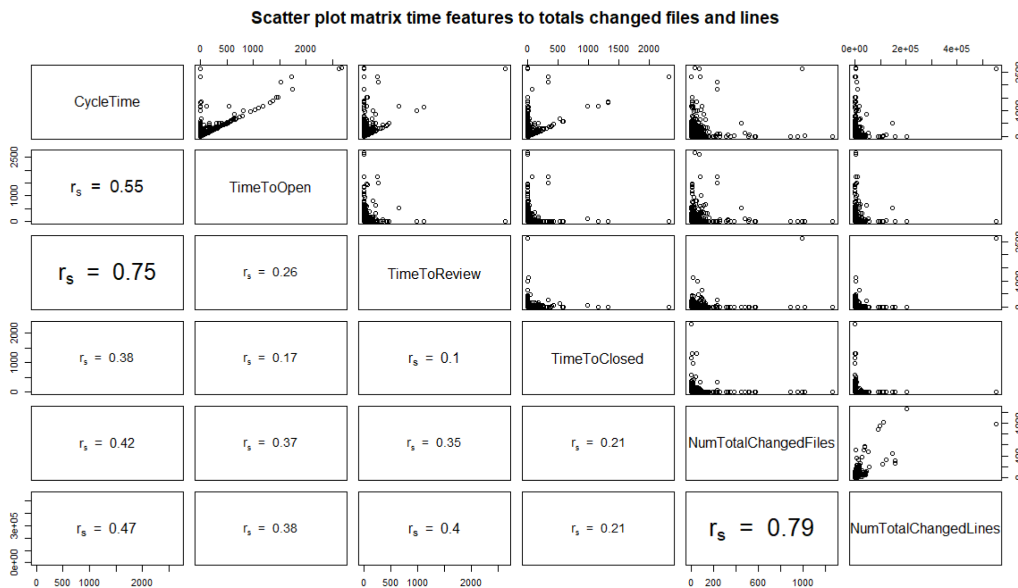


Figure 5.14: Scatter plot matrix of the time features and the total changed files and lines features. In the upper right part the scatter plots are shown and in lower left part the Spearman correlation between the corresponding features.

5.2.7. EXPLAINING THE REVIEWER INTERACTION FEATURES

The last aspect we looked into is the reviewer interaction in the pull requests. Based on the earlier findings of the explanation of the pull request and the fact reviewers give face to face feedback we are interested how the interaction in DevOps looks like based on the data.

We used for this the following features: NumUserThreads, NumUserComments, NumIterations, HasCommitBeforeReview and HasCommitAfterReview. The first two features are threads and comments created by a reviewer. A thread consists of a single or multiple comments reacting to each other, within DevOps it is not possible to have multiple threads within a thread. The features, NumUserThreads and NumUserComments were filtered from the total number of threads and comments in the pull request as the system and a bot also posted comments and thereby creating threads.

The third feature, NumIterations, is the number of pushes to the branch once the pull request is created, starting at 1. The latter two features are whether or not the author has edited the pull request before or after a review interaction, either a vote or comment.

First off, we counted the number of pull request with a user comment and thereby also a user thread. This resulted in 1102 pull requests, 13.03% of the total number of pull requests. In Figure 5.15 we show the distribution of the 1102 pull request for the features NumUserThreads and NumUserComments. In the figure a single outlier stand out with 85 user comments and 73 user threads which is clearly visible in the Q-Q plots. Followed by 5 outliers around the 60 comments, however there is only a single outlier visible around 50 user threads. The distributions look very similar and this is also reflected in the correlation between these two features of $r_s[1102] = 0.87$, $p < 0.001$, shown in Figure 5.17. This indicates that once the number of comments increases the number of threads also increases, this inclines that the average number of comments in threads is close to 1. In practice the average comments per thread is 1.58. Furthermore, on average there are 2.92 user threads and 4.63 user comments in each pull request of the 1102 pull requests.

Next, we look at the number of iterations, NumIterations, in a pull requests. This shows the number of changes happened after creating the pull request and can indicate a possible change based on the given feedback. Pull requests with a single iteration did not process any feedback for sure as no change has happened after creation. In Figure 5.16 we show the distribution of NumIterations of the pull requests having at least a user comment, left plots, and the full data set, right plots. There is a single outlier with 32 iterations and only 4 user threads and 5 user comments. In addition, the next 3 pull requests with a higher number of iterations have also a low number of user comments. This indicates that the changes are not triggered by comments in DevOps, though it is possible the changes were triggered by face to face feedback.

Furthermore, we looked into whether we can find evidence a change has happened after a user com-

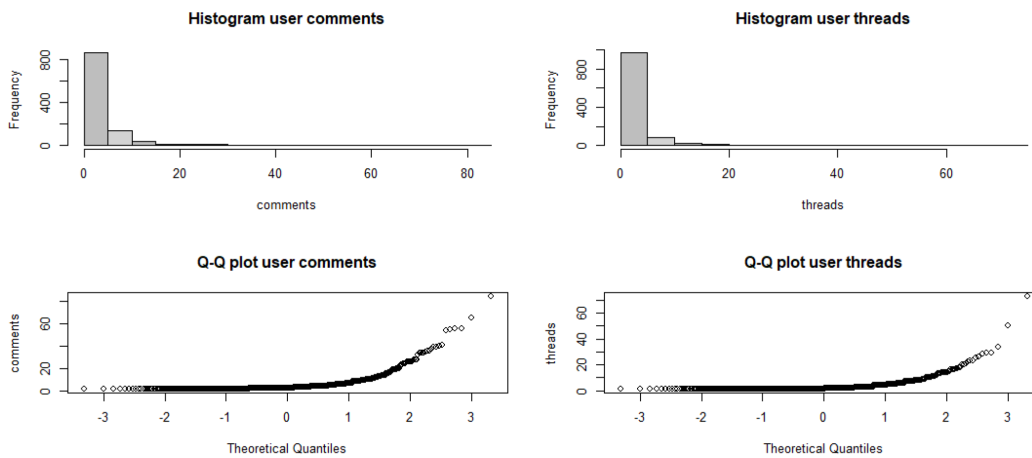


Figure 5.15: Histogram and Q-Q plot to explore the NumUserComments (left) and NumUserThreads futures (right).

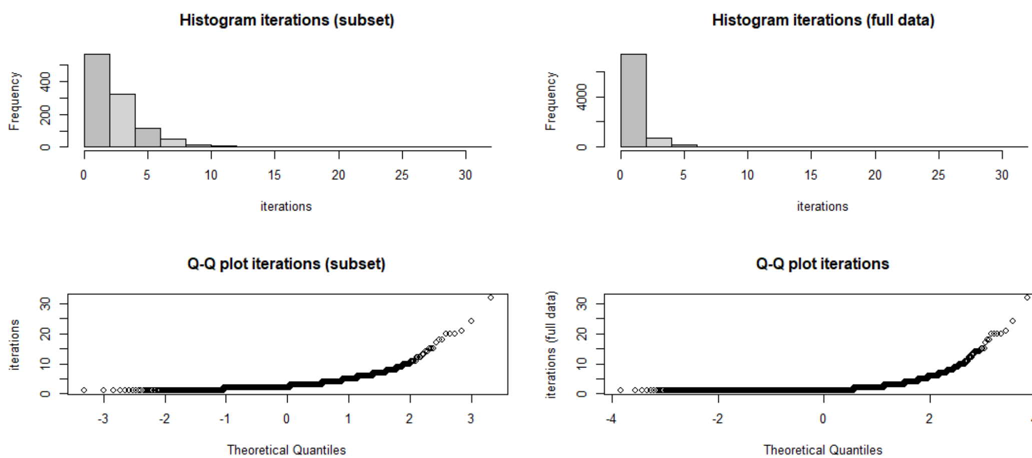


Figure 5.16: Histogram and Q-Q plot to explore the NumIterations of two data sets, a subset consisting of the pull requests with at least a single user comment (left) and the full data set (right).

ment and whether we can predict how many changes are triggered by face to face feedback. For which Beller *et al.* [10] discovered that 10-22% of the changes are not triggered by a review comment. Based on the 1102 pull request with a user comment 889 (80.67%) pull requests had a change after the first comment, indicating a possible change triggered by feedback on DevOps. This leaves out 213 (19.33%) pull requests without a change after the first comment. In addition, of the 1102 pull requests with a user comment, 286 (25.95%) pull requests have a change before the first comment was placed in the pull request. This might indicate the sanity check of the author which we discovered in the interviews. Based on the 7356 pull requests without a user comment 5865 (79.73%) pull requests have a single iteration, this leaves 1491 (20.27%) pull request with a possible change triggered through face to face communication.

Lastly, we calculated for the subset of the pull requests with at least a user interaction the number of reviewers the average, this average increased to 2.01 from 1.62 for the full data set we showed in the general analysis in Section 5.2.1. Though, the average number of approvers decreased slightly to 1.06 from 1.09 for the full data set. This indicates the number of reviewers increases once a user comment is given. Notice the author of the pull request placing comment herself does not count as reviewer. The increased number of average reviewers is somewhat expected as pull requests without comments only requires a single approver and thereby a single reviewer. In case a comment is placed within a pull request another developer might also interact in the discussion or approve the pull request based on the newer changes based on the feedback and the first reviewer is not available to review.

Summary: We discovered that 13.03% of the pull requests contain one or more user comments. On average the subset of pull requests with at least a single user comment has 4.63 comments and 2.92 threads, this means on average a thread has 1.58 comments. Of the pull requests with a user comments 19.33% of the pull requests does not have a change after the first review. In addition, 25.95 of the pull requests has a change before the first review, this might indicate the sanity check we discovered in the interviews. Of the pull requests without a user comments 79.73% of the pull requests has a single iteration, indicating no change has happened. This leaves 20.27% of the pull request without a user comment that could have had a possible change triggered by face to face communication.

5.2.8. COMPARING THE USER INTERACTION FEATURES TO THE TIME AND SIZE FEATURES

Similar to the size features we are interested how the user interaction features affect the CycleTime and other time features. In addition, we looked how these user interaction features affect the size features. We used for this the subset of 1102 pull request with at least a user comment. In Figure 5.17 we show the scatter matrix between the user interaction features, CycleTime and total size features. In general the correlations are weak indicating the features affect each other slightly. The higher correlations between NumUserThreads and NumUserComments, and NumTotalChangedFiles and NumTotalChangedLines we encountered earlier.

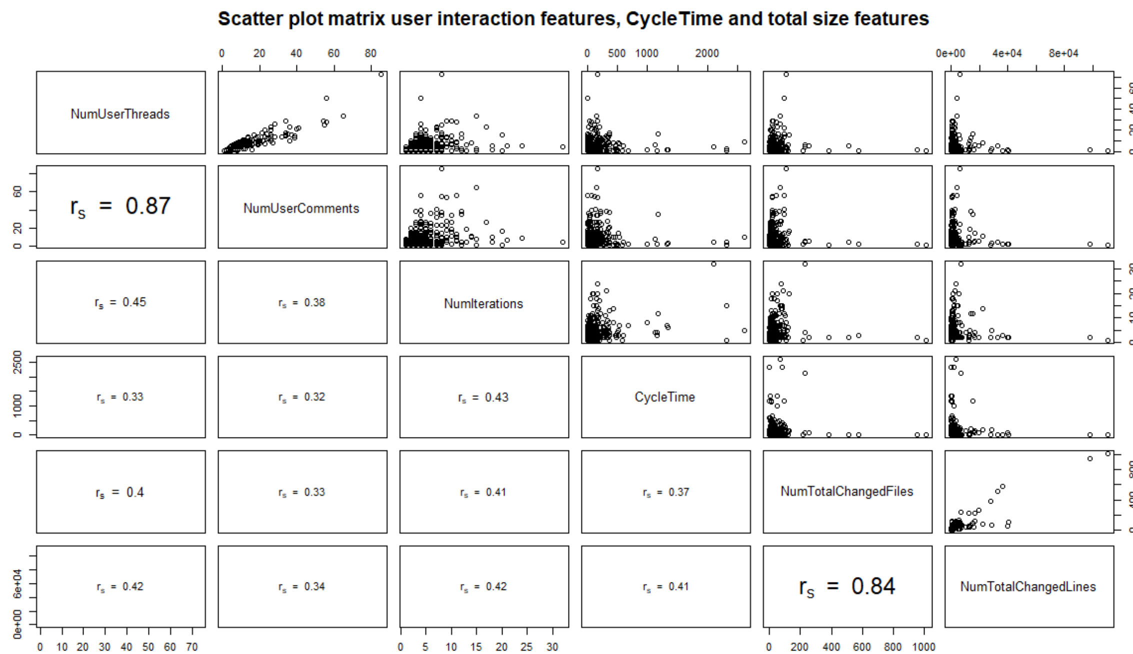


Figure 5.17: Scatter plot matrix of the user interaction features: NumUserThreads, NumUserCommnets and NumIteration to the CycleTime and total size features: NumTotalChangedFiles and NumTotalChangedlines. In the upper right part the scatter plots are shown and in lower left part the Spearman correlation between the corresponding features.

5.3. CODE REVIEW COMMENT DATA ANALYSIS

In order to find the types of code review comments given during code reviews we classified a total of 5100 review comments given in the period 1-1-2018 till 21-01-2020. Although our initial classification did not succeed due to the inconsistency in the usage of specific categories, we could still get a general idea of the types of comments given through the introduction of the global categories as explained in the methodology in Section 3.2.3. Based on the results of the types of comments given we get insight in what type of items should be included in the checklist.

In Section 5.3.1 we present the results of the global categories. In these results we included the distribution across the specific categories which should be conceived with a caution due to the inconsistency as explained in methodology. In the second part we show the results of the specific categories we did not assign to a global category: *false positives*, *missing functionality* and *proposal of another solution*. The results of these categories are still included in the total count of each global category. In the explanation of the distribution of the global categories they a grouped together in *other*. The results and the distribution of these three remaining categories are further explained in Section 5.3.2.

5.3.1. RESULTS OF THE GLOBAL CATEGORIES

In Figure 5.18 we show the total count of comments given in the pull requests for each global category as explained in the methodology 3.2.3. The category with the most comments is *clarity and understanding* with 1694(33.22%) comments. This category consists of the specific categories: *explanation* 895, *naming* 276, *ask for explanation* 429 and *other* 94. The category has, except the naming of clear variables, less to do with actual coding. It mainly covers the discussion between the author and reviewer to get an understanding of the change. In *explanation* the author or reviewer explains a specific subject whereas in *ask for explanation*, as the name suggest, the author or reviewer asks for explanation.

The second category is *other communication* 915(17.94%) consisting of the specific categories: *fixed* 802 and *small talk* 113. This category covers mostly comments that do not directly influence the pull request and contains information regarding the pull request. The *fixed* category is the largest specific category and as the name suggest consist of messages saying *"fixed"* or similar. It are acknowledgements by the author of the pull request mentioning the given feedback is processed. Although it might seem useful to the author to remember which comment still has to be solved, we think marking the comment as fixed or won't fix is more

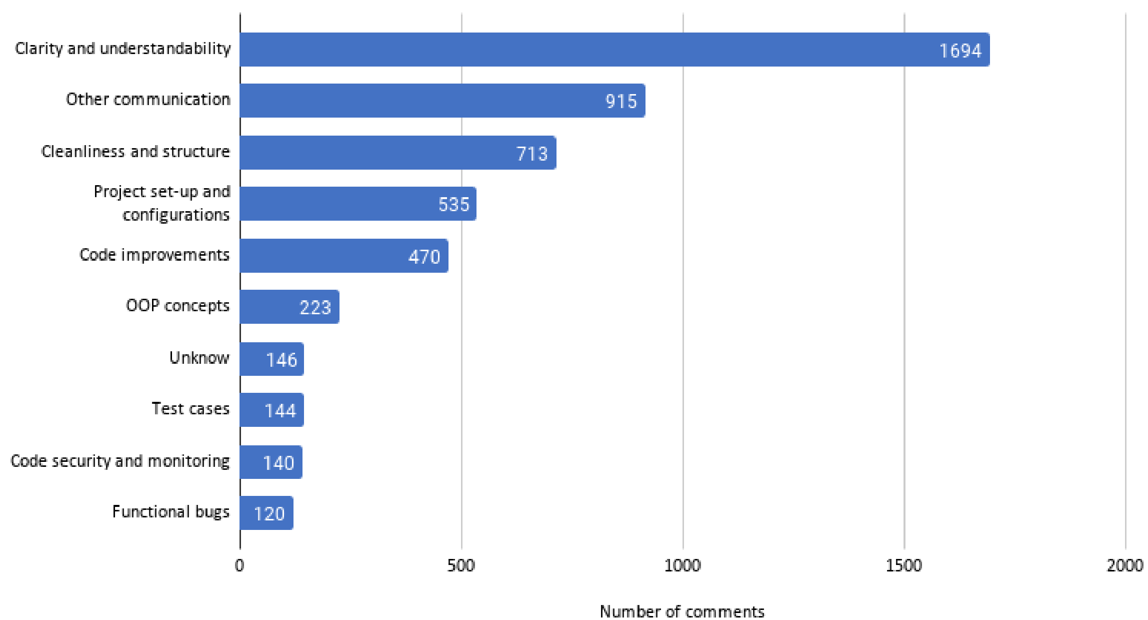


Figure 5.18: The comment type categories and their corresponding total number of comments.

useful. However, a developer mentioned the state of a comment should not be modified by the author of the pull request and should be changed by the reviewer, assessing whether the issue is correctly fixed. The *small talk* category consists of other text message having no direct impact on the pull request.

The third category is *cleanliness and structure* 713(13.98%) consisting of the specific categories: *naming* 142, *structure* 200, *dead code* 225, *typo* 49, *comments* 42 and *other* 55. In contrary to *naming* in *clarity and understandability* these comments address minor mistakes in the spelling of the variable, mostly capitalization, and have less to do with the actual representation of the variable. The *structure* comments are minor improvements in the readability of the code such as brackets, white spacing, blank lines and the usage of unnecessary usings. Additionally, we included *dead code* to this global category as it does not affect the functionality and improves the readability of the code. Finally, we have *typo* and *code comments*, this were specific categories which were not used consistently and should be conceived as *at least*. As the name suggest it are comments regarding the code comments and other spelling mistakes.

Fourth is *project set-up and configurations* 535(10.49%). It consists of the following specific categories: *dead code* 110, *resource defects* 49, *support defects* 209, *version number* 52 and *other* 115. The *dead code* category is somewhat unique not being part of the global category *cleanliness and structure* as we showed previously. There are *at least* the 110 occurrences found of dead code in support files, however in practice there might be even more. The second specific category is also *at least* as using the wrong database name or still using the localhost could also be categorized in other categories, especially the third specific category *support defects*. *Support defects* is the specific category covering most comments regarding usage of tooling, build files, configuration files and environment variables. The latter category, *version number*, is the specific mistake of forgetting to update the version number.

The fifth category is *code improvements* 470(9.22%) consisting of *consistency in code style* 248 and *other* 222. *Consistency in code style* are mostly small improvements to improve the code base. The 222 left over occurrences are within the three specific categories which will be described separately further on.

Sixth is *OOP concepts* consisting of *misplacement of functionality* 85, *consistency in code style* 37, *duplication* 31 and *other* 70. The first specific category is whether the code should be placed elsewhere to fit better in the code base. The latter two categories are both *at least* of which *consistency in code style* covers access modifiers, number of parameters and usage of enum files and *duplication* covers, not surprisingly, duplication of code.

The seventh category is *unknown* with 146(2.86%) comments which we could not classify. Of which a large

part 133 were comments that were classified as *proposal of another solution* which we could not reclassify to the correct global category.

Eight is *test cases* 144(2.82%) solely consisting of the specific category *testing* 80 and *other* 64. This are mostly comments regarding the outcomes and the process of testing the code. This does not cover the code quality of the test cases.

Ninth is *code security and monitoring* 140(2.75%) consisting of *code security and monitoring* 80, *check defects* 8, *null defects* 24 and *other* 28. We initially introduced the first specific category for comments related to security issues such as the handling of user data, correct usage of encryption and the prevention of unexpected crashes. We decided to add the log messages as they help to retrace possible crashes. The latter two categories are, *check defects*, missing cases in an if statements and, *null defects*, missing or unnecessary checks for null values. The latter two should be conceived with *at least* and could furthermore sometimes even be counted as function bug. However, it was hard for us the assess whether a missing case or check would directly lead to an actual bug.

Tenth and last category is *functional bugs* 120(2.35%) consisting of the specific categories *bugs* 53, *resource defects* 11 and *other* 56. During classification it was rather hard to assess whether or not a comment really addressed a functional bug. This could mean we classified potential functional bugs over the other global categories, for example the previous category. Thus, in practice we assume this value would be higher.

This leaves out one global category *architecture*, during the classification we did not encounter comments the we could specifically link to the architecture. In practice, we think we divided the comments regarding architecture over the other categories, especially the global categories *OOP concepts* and *code improvements*. Furthermore, we think that the high level architecture agreements are discussed at forehand in meetings rather than addressing them during code reviews.

Summary: The types of comments given during code reviews are mainly related to the understandability of the change. Next, we found the usage of *"fixed"* comments, acknowledging the feedback in a comment is processed by the author, possesses the second rank. Third ranked we found smaller improvements related to the cleanliness of the code and the structure to increase readability. Project set-up is ranked fourth and is mainly due to specific comments mentioning to update the version numbers and use the exclude from code coverage function. Code improvements is ranked fifth. The categories which were ranked lower are *functional bugs*, *code security and logging*, *test cases* and *OOP concepts*.

5.3.2. RESULTS OF THE REMAINING CATEGORIES

In this section we show the distribution of the 3 remaining categories which we did not yet cover. These are: *false positives*, *missing functionality* and *proposal of another solution*. Due to the introduction of the global categories we could indicate to which global category these specific categories belonged providing addition information regarding the distribution. However, this also increased the inconsistency during the classification and 133 comments could not be reclassified according to the new insights, these 133 comments were added in the *unknown* category. In Figure 5.19 we show the results of the distribution of the remaining 3 specific categories.

The category *false positives* contains comments which were not solved within in the pull request itself. Hereby, it could be the comments were already or would be solved in another pull request. The most false positives are within *clarity and understandability*, this are mostly renaming suggestions or questions for clarification which were not answered in the pull request. Though, it is possible the clarification was given through face to face communication.

The comments of *missing functionality* category are evenly spread over *code improvements*, *functional bugs*, *test cases* and *project set-up and configurations*. In case of *project set-up and configurations* the comments mostly addresses the lack of *exclude from code coverage* in generated code files to exclude the file from being checked by SonarQube. Additionally, missing functionality means the functional requirements of the PBI are not met and counted as functional bug. Other comments were small code improvement that were left out in the pull request.

Lastly, *proposal of another solution*, this are mostly comments proposing a small improvement to the code or an actual alternative solution to the problem. Most comments given are found in the global category *code improvements*. This is due most code improvements are by default suggestion for another approach. This means most comments could fit in here as well as in *consistency in code style*.

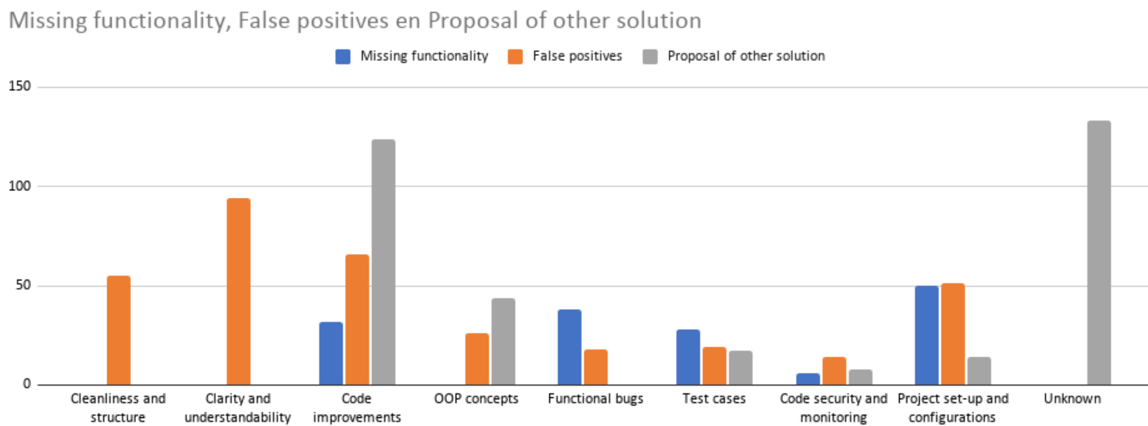


Figure 5.19: Overview of the distribution of the three left over categories: false positives, missing functionality and proposal of another solution

5.4. DISCUSSION

In order to get an general idea of the current code review process at Greenchoice we held interviews, performed data analysis on the metadata related to the pull requests and code reviews, and comment data analysis. In this section we will discuss the results of each of these three methods. In addition, we bring the different results together analysing how the different results relate to each other. Furthermore, we compare the findings in the results to the findings in the literature.

In Section 5.4.1 the results of the interviews are discussed. Next, the results of the metadata analysis are discussed in Section 5.4.2 and finally in Section 5.4.3 the results of the comment data analysis. At the end of each section we answer the corresponding sub research question.

5.4.1. INTERVIEWS

In the interviews we discovered how the developers prepare their pull requests for a code review and how they perform code reviews themselves. Additional, we found the points of attention during code reviews and what the strengths and weaknesses are. During the interviews the interviewees were free to discuss any subject related to code review process. This means that not all subjects were discussed with every developer, therefore we suggest to do a quantitative study, for example an questionnaire, to verify the thoughts of all the developers. Still we got a general idea of the code review process which we will discuss further in this section.

PREPARING THE PULL REQUEST

In the interviews we found the preparation of a pull request it is often well executed. A heads-up is given early on and once the pull request is ready. This provides awareness to the fellow developers so that they can anticipate on the upcoming pull request. During the preparation of the pull request the author performs a sanity check whether or not the pull request is correct. However, one step in the preparation is often neglected and that is providing guidance to the reviewer of what is changed and why the solution is the best solution. We would expect this guidance to be present in the description of a pull request, though we showed in the metadata analysis that roughly 84% has a similar title and description or no description at all. MacLeod *et al.* [67] found that writing descriptions is hard and only a third of their interviewees mentioned they write descriptions for their changes. However, more interviewees mentioned this should be done more often and more thoroughly [67]. In addition, Tao *et al.* [70] showed that review guidance is only given in 5% of the cases.

In terms of understanding the change Bacchelli and Bird [9] found it is the main challenge developers and testers face in the code review process, especially understanding the reason of the change. To resolve the need of understanding the reason behind a change Bacchelli and Bird [9] found that reviewers try to read the change description, run the changed code and ask for clarification via e-mail or face to face communication. In an interview performed by them an interviewee mentioned the description and actual change might contradict each other making the description not useful [9].

Furthermore, Tao *et al.* [70] found that understanding code changes is an important factor in the different development phases, especially in the review process. Understanding the change plays an important role in

assessing the quality of the change and exposing the risk of the change, hereby the risk could be for example breaking the code elsewhere due to the change. To assist the developers in understanding the code change they propose developers need better support to expose the risks of the change. This could be either an explanation by the author, for example which areas are affected by the change. Or methods helping the reviewer discover and go over all the dependencies the change affects, for example go to function and call hierarchy in the IDE. In order to ease understanding the changes Tao *et al.* [70] found that changes should be small and independent.

Process of performing a code review

The most important points of attentions we found during the interviews are whether or not the pull request is a correct solution to the problem and do I, as reviewer, understand the change. Furthermore, are there no obvious mistakes in the code. Less attention is paid to factors such as maintainability and evolvability or in general the code quality. However, the developers and architects also mentioned that a minimum degree of code quality is needed to be able to easily understand the code change. Meaning that the code quality follows automatically with well understandable code. Therefore, there is no need to be discussing over every detail of the code. Furthermore, one does not know at forehand what is needed in the future and not too much time should be spend on discussing endless number of possible scenarios. Additionally, testing is in general more often neglected than they wish to do and could be improved on.

Comparing these findings to the results of the comment data analysis we found that indeed most of the comments, 33.22%, affect the understandability of the change. We previously discovered that understanding the change is important in the code review process [9, 70]. However, in terms of usefulness Bosu *et al.* [36] found comments regarding understanding the change are not considered useful by the author as they do not improve the code. In case of useful comments they found addressing functional bugs, validating potential issues by experts and suggestions regarding usage of external dependencies and code conventions were found useful by the authors of pull requests. Somewhat useful are nitpicking comments regarding the code quality, although they might not be necessary they improve the code in the long run [36]. This are mainly the smaller improvements related to readability such as the indentation, white lines and spelling mistakes that we grouped in our comment data analysis under the category *cleanliness and structure* ranked fourth, 13.49%.

Related to this is one of the weaknesses, we discovered that reviewers could be stricter to the code quality during reviews. Mainly on factors as maintainability and evolvability. In the comment data analysis we consider the categories *code improvements* and *OOP concepts* as categories affecting these factors, together covering 13.57% of the comments in total. However, we also discovered it can be hard to know where to review on. This could be related to the difficulty of keeping the personals skills up to date as developer [37]. Bacchelli and Bird [9] found that several authors complained that reviewers focus on easy and obvious mistakes, styling and formatting issues, and do not address underlying or more important issues. In addition, MacLeod *et al.* [67] found similar thoughts by their interviewees of reviewers focusing on insignificant improvements. This behaviour of focusing on insignificant details rather than more important issues is also called bikeshedding. During the comment data analysis we also encountered several of these discussion, for example whether the name of a variable consists of two words or a single word.

In terms of challenges faced during code reviews Greenchoice has similar challenges to the literature. We already discussed understanding the change and stricter code reviews are part of the challenges. One of the other challenges is the size of pull requests, although according to our interviewees the size of the pull requests are mostly fine it are the occasional larger pull requests. These are hard to review and the quality of the review decreases, this challenge is also found by Kononenko *et al.* [37], MacLeod *et al.* [67], Tao *et al.* [70]. Another challenge we discovered is for the reviewers to cope with context switches, this was also discovered by Kononenko *et al.* [37]. MacLeod *et al.* [67] discovered receiving feedback in a timely manner and selecting the correct reviewers as challenges, however, based on our interviews we have no indication this is also the case at Greenchoice. Another challenge faced by other companies is the distribution of the developers around the globe dealing with different time zones [35], however, having only a single office this does not hold for Greenchoice.

Effect of face to face reviews

During the interviews and throughout observations at the work place we discovered that feedback of code reviews are also given face to face. In a later chat with an architect the term *pair reviewing* came up, similar to the concept of pair programming whereby two developers program together [96–98], the reviewer and

author sit together and go over the pull request. Before the actual sit down for the pair review the reviewer can go over the pull request and already place initial comments as a reminder on what to discuss in the pair reviewing session as preparation. The advantage of this session is the communication is easier and thereby it is also easier to explain the feedback and discuss the why for certain decisions in more detail. In other words it is easier to share the thoughts behind a choice of which the other can learn. The disadvantage is that this takes more time as both the reviewer and the author need to schedule this meeting, also finding an appropriate moment can be hard. However, in practice an informal version of pair reviewing, also called over the shoulder review [99], was mostly performed. Meaning the reviewer would go to the author when he or she is available and go over the pull request. In the ideal scenario the outcomes of the discussions are placed as an answer to the initial comments for the other developers and preserving the historical value.

Furthermore, we showed in the background that McIntosh *et al.* [74] and Thongtanunam *et al.* [72] found that lax code reviews lead to more bugs in practice through analysing the historical comments, in other words fewer bugs are discovered during lax code reviews. In contrary Tao *et al.* [70] found the historical value is of less use in the process according to their interviewed developers. Tao *et al.* [70] found that the developers themselves should be aware and recognize problem areas once these areas reoccur frequently during code reviews. In terms of preserving the historical value Sutherland and Venolia [61] found that only 18% of the data of synchronous reviews is made available. They also found that 49% of the reviews were performed synchronous and 51% asynchronous e-mail based reviews. What the actual distribution is of face to face reviews versus asynchronous reviews via DevOps, what the effects of the different review practices are and what the value is in the preservation of the historical data at Greenchoice requires future work. In addition, whether the stricter code reviews would expose more defects and what types of defects is also future work.

RQ 2.1 How does the current code review process of Greenchoice look like in the perspective of the developers?: The code review process starts by the author of the pull request giving a head-ups to his fellow team members. In addition, the author performs a sanity check for the correctness and completeness of the change just before or right after the creation of the pull request. The reviewers tend to focus on understanding the change and finding obvious mistakes. They mentioned code reviews could be performed stricter improving on the maintainability and evolvability aspects of the change. However, it can be hard to know where to check on for these aspects. The feedback could be either given synchronous face to face and asynchronous in DevOps. The effect of the different methods of providing feedback requires future work. In general the code review process at Greenchoice looks similar to the current modern code review processes described in the literature, however, having its own strengths and facing its own challenges.

5.4.2. METADATA ANALYSIS

In the first part of the metadata analysis we explored whether or not we could explain the CycleTime, the time between the latest commit before opening the pull request and closing the pull request, by the other time features. In the results we were not able to find similar results compared to the results of Davis [82], they found that TimeToOpen has the highest correlation to CycleTime. We found that the TimeToReview has the highest correlation to the CycleTime instead of TimeToOpen. We are unable to determine why we got different results compared to Davis [82] as they lack important information of the process, such as the distribution of the data and the correlation method used.

However, we did discover for pull requests with a CycleTime longer than a week that TimeToOpen has the highest correlation, though only with a moderate correlation. We were already expecting the latter result based on a manual inspection of the data in which a majority of the pull requests with a longer CycleTime also had a longer TimeToOpen. However, we showed only for minor part of the pull requests the higher correlation and does not hold for all pull requests. Based on a later chat with one of the architects we were unable to determine a specific reasoning why several pull requests have a relative long TimeToOpen. Sometimes the longer TimeToOpen could simply be due to the weekend or a holiday period. In other cases it is a change for another team which do not have the time available to deal with the change at that moment and the pull request is delayed in the creation process.

We found that most pull requests were merged within one hour and almost all requests within a week at Greenchoice. We discovered a median of 25 minutes for the CycleTime of a pull request, which is relatively quickly compared to the literature. Sadowski *et al.* [35] found a median of under 4 hours which was already quickly compared to findings of Rigby and Bird [64] finding a median of 14.7 hours up to 20.8 hours for 6 dif-

ferent projects. In terms of size Sadowski *et al.* [35] found a median of 24 lines of code and Rigby and Bird [64] found for the six projects a median of 25 lines of code up to 263 lines of code. In our results of the size features we found a median of 28 lines of code which is compared to findings of the literature relative small. These findings compared to the literature confirms the thoughts of the developers in the interviews of quick and small reviews. In terms of number of reviewers we found an average of 1.62 reviewers and 1.09 approvers for all pull requests and 2.01 reviewers and 1.06 approvers for the pull requests containing at least a single user comment. Sadowski *et al.* [35] found that most reviews required only a single developer, however other literature suggest 2 reviewers Hatton [43], Rigby and Bird [64] is the optimal number of reviewers. Furthermore, we discovered an average of 4.63 comments for the pull requests with at least a single user comment. Rigby and Bird [64] discovered a median of 3 to 4 comments per review for their projects, however, our median over the full data is 0 as only 1102 out of the 8454 pull requests contain user comments.

We conceive in general the size and time aspects are considered positive in Greenchoice's code review process, though on occasions a larger pull request comes by or a pull requests takes more time than usual. In the results we questioned whether the pull requests were not reviewed too quickly. Based on the literature 125-200 lines of code lines is the optimal speed per hour to review [100]. Based on the median of 25 minutes per code review, the pull requests with a median size of 28 lines of code and the optimal review speed the pull requests are reviewed too quickly. However, this leaves limited space for delays and the median of 25 minutes includes wait times and does not represents the actual time spent on the code reviews. In addition, a large part of the reviews are reviewed in a shorter period. The short review time does not necessarily indicate that the pull request is badly reviewed as it could be possible the code is written together or it just a small change. We therefore leave the analysis of the effectiveness of code reviews based on the time spent as future work. Similar to whether stricter reviews will lead to better code reviews detecting more defects and what the effect on the time spent will be.

Based on the analysis how the CycleTime is related to the detailed time features we furthermore explored the CycleTime is affected by the size and user interaction features. In the interviews the developers mentioned that a larger pull request requires more time to review as it is harder to grasp the meaning of the pull request. However, they also mentioned larger pull requests are accepted easier as they require more work and an in-depth analysis is skipped. In the metadata analysis we found a moderate correlation of $r_s[8458] = 0.47, p < 0.001$ between CycleTime and NumTotalChangedLines. This indicates there is a slight relationship between the two features. This is similar to findings of Kononenko *et al.* [37] who found that the review time is influenced by the patch size and reviewers experience.

RQ 2.2 How does the current code review process of Greenchoice look like according to the metadata available on DevOps?: The code review process of Greenchoice is according to the metadata is a rather quick process compared to the literature with a median CycleTime of 25 minutes. In terms of size the process is on the smaller side compared to the literature with a median of 28 lines of code changed. In addition, we discovered moderate correlations between CycleTime and NumTotalChangedLines and NumTotalChangedFiles. This indicates there is a positive relationship between the size and time features. Furthermore, we discovered on average 1.62 reviewers and 1.02 approvers for a pull request, and for the pull requests with a user comment an average 4.63 comments per pull request. In terms of correlation we discovered mostly weak correlations to the time and size features, indicating a slight relationship.

5.4.3. COMMENT DATA ANALYSE

The second analysis we performed is the comment data analysis in which we looked into the comments used during code reviews. This analysis gave us insight in which factors were considered important during code reviews in the past. In the analysis of the comment data we lacked consistency in the usage of the categories. Without the possibility to reclassify the comments we could only retrieve a general idea through the usage of global categories of the types of comments given. Therefore, we could not make an in-depth analysis as we intended to do similar to the analysis of Beller *et al.* [10], Mäntylä and Lassenius [39]. Though, we discovered new findings that were not yet included in the literature. We did not only focus on the technical and functional defects. We also showed the understanding and explanation aspects in the review comments. We are unsure whether Beller *et al.* [10], Mäntylä and Lassenius [39] did not encounter or did not focused on these types of review comments.

In the analysis of the comment data we were surprised by the number of comments, 10.49% , we found

regarding the set-up of the project, configuration files and usage of tooling. Especially two specific types of comments regarding *updating the version number* and the usage of *exclude from code coverage*. These types of comments belongs in both papers [10, 39] in the category support defects and were barely discovered by Beller *et al.* [10] and Mäntylä and Lassenius [39] did not discover any defect of this type. Additional, our category test cases was not included in the papers at all. Thus, our categories *Clarity and understandability*, *project set-up and configurations* and *test cases* are new findings compared to the findings of the research of Beller *et al.* [10], Mäntylä and Lassenius [39], the rest of the categories used by us can be related to one of their categories.

Based on our data we are unable to perform a similar comparison for the ratio between functional and evolvability related comments. In both analysis the definition of functional defects is much broader than we used in our analysis as *functional bug*. Our categories do not have a clear division of functional and evolvability comments, for example the categories *project set-up and configurations* and *code security and monitoring* contains both comments related to the functionality and the evolvability of the code. However, it is likely that our categories *code improvements* and *OOP concepts* are evolvability comments. Still we think we were able to gather valuable information and gain insight in the types of comments used at Greenchoice.

RQ 2.3 What types of comments used are found in the current code review process?: In the code review comment data analysis of 5100 comments we found most, 33.22%, comments are related to understanding and explaining the change. Second is *other communication*, 17.94%, consisting of mainly *"fixed"* messages acknowledging the comment giving feedback is processed. Other categories we discovered are: *cleanliness and structure* (13.98%), *project set-up and settings* (10.49%), *code improvements* (9.22%) and *OOP concepts* (4.37%).

6

CREATING A CODE REVIEW CHECKLIST

In order to improve the code review process at Greenchoice we created a checklist which the developers could use during the code reviews. The goal of the checklist is to improve the code review process providing more structure and assisting the developers to know where to look for during their code reviews. In this chapter we explain the creation process of the checklist, in which the first step in creating the checklist was to join a dedicated scrum team. This is an important step in the creation of the checklist as the checklist should reflect the needs of the project and therefore the team [8]. Together with the scrum team we learned more about the code review process and the development process in general. To get familiar with the strengths and weaknesses in the code review process of the scrum team we held first an exploration session of which the results are shown in Section 6.1. A second session was held to discover which items we should include in the checklist according to the needs of the scrum team, the results of the second session are shown in Section 6.2. After another round of receiving feedback and processing the feedback the final checklist is explained in Section 6.3 and shown in Section 6.4, this checklist will be used during the experiment. We end this chapter with a discussion in Section 6.5.

6.1. THE EXPLORATION SESSION

In the exploration session we looked into the strengths and weaknesses of the scrum team regarding the code reviews with the developers, at that moment 4, of the scrum team we joined. Getting to know the strengths and weaknesses we gained an initial insight on which factors are considered important by the scrum team in the creation checklist. We started the session with the preliminary results that we processed so far to give the scrum team insight in our research progress so far, this are the results of the motivations which are shown in Section 4.2.1.

Next, the real exercise started in which we asked each team member to write down their personal perceived strengths and weaknesses of the code review process on the corresponding color post-it note. The color green was used for the strengths and pink for the weaknesses as explained in the methodology in Section 3.3. Once the developers could not come up with anymore strengths or weaknesses we stopped the exercise, this was after approximately 5 minutes, and we started to reveal the post-its. The post-its were revealed one by one and the corresponding developer explained why she had written down the corresponding strength or weakness. If another developer had also written down that strength or weakness the post-it was also revealed and a discussion was started, especially in the situations of a green and pink post-it. In Table 6.1 an overview of the factors discovered as strength or weakness and their corresponding number of green and pink post-its is shown. Each of the factors will be explained in more detail next.

The first factor discussed is the pull request size with 2 pink post-its, indicating negative experience, and a single green post-it, indicating positive experience. In general the size of the pull requests are perceived within the order of small by the team. This reasoning explains the single green post-it. However, it are the incidental larger pull requests for which the two pink post-its were given.

Related to the larger size of the pull request is the next factor scope creep or also known as feature creep with 2 pink post-its. During scope creep the main goal of the pull request is overwhelmed by less important changes not directly related to the main goal of the pull request. With a larger pull request it is more likely to include smaller changed to improve the code, under the guise of the boy scout rule. However, the team finds

	Green	Pink
Pull request size	1	2
Scope creep	0	2
Merge time	2	0
Build integration	1	2
Testing	1	1
Social knowledge sharing	3	0
Functional knowledge sharing	2	0
Stricter	0	1
Visual Studio warnings	1	1

Table 6.1: Results of the positive and negative factors during code reviews given by a scrum team. The subjects are listed left and the green column correspondents to the number of green post-its given as positive factor, the pink column is similar though as negative factor.

it sometimes difficult to know how many small improvements they can make before they get overwhelmed by them.

A positive factor and accordingly a strength, with two green post-its, is that most pull requests are merged within a relative short time span. A pull request is mostly reviewed within the same day, though when the pull request is created in the late afternoon the pull request will most likely be reviewed the next workday morning. In case of a larger pull request the merge time might increases as more time is needed. Finding a suitable lengthy moment can be hard and sometimes the review has to be split over multiple sessions. Additionally, their experience is that larger requests requires more iterations to resolve all issues before the pull request can be merged.

The next factor is build integration. A single green post-it and two pink post-its were revealed. The green post-it was given due to the fact a passing build is needed before a pull request can be merged. However, the build should be activated and properly set. An example that was given on a newly created repository was that the set-up for the automated build integration was not yet completed. This led to a pull request being merged on the master branch with a failing test. Furthermore, the longer build time was mentioned as a negative factor in the development process.

Subsequently, testing itself had for each color a single post-it. The positive post-it was given for checking the presence of unit tests. The negative post-it covered the testability of the code, sometimes it could happen that a part of the code is hard to test due to the quality of the code or how it is structured. After explicitly questioning it in the discussion a check on the quality of the unit tests is sometimes neglected. Furthermore, it could happen that the unit tests are not directly included in the pull request, however in that case the unit tests are mostly directly after added in a separate pull request.

Social knowledge sharing is a positive factor with 3 green post-its. According to the developers this is reflected in the number of comments given, the discussion which are held during code reviews and the developers experience that they learn a lot from each other. Furthermore, the developers are open to give and receive feedback. This makes it easier to actually give the feedback and if necessary to start a discussion.

Moving on with 2 green post-its is functional knowledge sharing. The first post-it was initially solely given for the check on the functionality of the code. The second post-it contained also the knowledge sharing during the functionality check. The latter post-it emphasizes that during the functionality check the reviewer also learns how the code functions and that it should be easier to make changes in the code if needed. Thus, in other words the code awareness increases. The developer of the first post-it agreed to this explanation, therefore these two post-its are grouped together.

Next, with a single pink post-it is being stricter during code reviews. At certain times, the team members may be a bit harder on each other, for example by being stricter on coding guidelines, the SOLID principles and the earlier discovered testability.

The last factor which was discussed are the warnings in Visual Studio or better said ReSharper warnings, once more with a single green and pink post-it. ReSharper is used as a static analyse tool on code style and useful refactoring options to increase productivity of the developers. Although in most cases these warnings are solved before creating a pull request, it occasional occurs that this is forgotten. In additional, it is company policy that warnings should be treated as errors and should be checked by the automated build.

These results are inline with the earlier findings of the strengths and weaknesses of the code review process discovered in Chapter 5. In the exploration session itself we were able to ask questions to clarify some

of the factors in detail gaining addition insights. However, these insights provide little information on which items to include in the checklist. Based on these findings we extract the need for clear coding guidelines items and items keeping testability in mind for the checklist.

Summary: In the exploration session we discovered the scrum team faces similar challenges as previously discovered in the interviews. These are the risk of larger pull requests and thereby scope creep, however, also the positive side of in general smaller pull requests and pull requests being merged quickly. Next, the testing factor was discussed in which we found testability is not always taken into account. We found that the knowledge sharing is also in this team on point. We furthermore discovered that the code reviews could be stricter. Based on these results we extracted the need for clear coding guidelines items and items keeping testability in mind in the creation of the checklist.

6.2. DETERMINING THE CHECKLIST ITEMS

In this section we look into which factors the scrum team considers important and less important by the scrum team and their first preferred checklist items. Before the second session we have created a long list of checklist items that we thought could be applicable to the current code reviews process used at Greenchoice, shown in Appendix E. We based the items on general findings of the current code review process in Chapter 5 and the results of exploration session shown above. The items we thought were especially useful were the items regarding the coding guidelines, testing of the code and updating the version numbers. Most of the items in the long list we retrieved from existing checklists [86–88]. In order to provide the developers team some inspiration the example checklists were given as reference material after the first session. In addition, we included in the long list most coding guidelines available on an older deprecated Wiki page of Greenhoice.

Starting the second session with the developers of the scrum team, at that moment 3 developers, we went briefly over the example checklists. For the main part of the session we placed a large sheet of brown paper in the middle of the table. The developers could give their preferred factors or items of which they think they are important to be included in the checklist. It was an open group brainstorm session in which they could openly discuss their thoughts and ideas. Furthermore, they could use the sample checklists as reference for ideas. Once a factor or item was finalized it was written down on a post-it and put on the brown paper. Additionally, we opted several items of the long list which were not yet mentioned by the developers to give them new insights and start a discussion.

The developers of the scrum team unanimously agreed to prefer a rather short checklist which emphasizes the items that are deemed imported at that moment by the team, rather than a long complete checklist containing all the items. Based on this preference we aimed to create a checklist that fits on an A5 paper. Therefore, based on the result of the brown paper the long list was narrowed down to a shorter list of checklist items shown in Appendix F. In the appendix of the long list, shown in Appendix E, we have colored the items based on the preferences of the developers. The green items will be put in the short list, orange items are optional and red items will be totally scrapped. It could be possible that several orange items together were rewritten to a single item in the short list. The extensive long list could be put as reference on a Wiki page, though this reference could also be a link to an already existing checklist.

The first part of the shortlist relates to the implementation of the pull request which was found important by the developers of the scrum team. The items comprise the general set-up of the solution provided in the pull request. The general set-up consists of whether the correct solution is provided according to the PBI, the intention of the pull request and whether the code follows Greenchoice's coding principles. The last item we included is whether the necessary versions numbers are updated. In the long list we also had included a part related to whether a better alternative solution was available, however we decided to remove this part based on the feedback of the team. A solution that fits the already stated requirements is a solution which is good enough and no time should be spent on finding the perfect solution. The SOLID principle is added in the short list as a separate section as it consists of multiple principles, these principles can now be checked off separately.

The specific coding guidelines we included in the long list were summarized into general items addressing the readability and maintainability factors. Changing the specific coding guidelines items from the long list to the general items in the shortlist reflects the choice of a small checklist. Additional, these specific guidelines can still be found on the Wiki page, of this Wiki page we based most of the specific coding guidelines. We added a separate check whether the code follows these specific guidelines and referred to the Wiki page.

However, with the general items we still want to keep some distinction between the different coding guidelines factors. This means separated whether the code is easy to read, easy to understand, the code is clean and whether useful code comments are used.

Furthermore, the naming of the variables was separated as its own section which was preferred by the team. The naming of variables can cause some confusion as both Dutch and English names are used. The first items covers whether the names are clear and represents the actual functionality. The second item covers whether the names are in the right language. The latter was especially found important as it can be confusing when to use English and when to use a Dutch name. In general the names should be in English, however specific business terms are written in Dutch. Either an English term does not exist or to keep consistency with the business departments avoiding confusion in terminology.

The next section in the short list is a merge of two sections in the long list which were considerably reduced in length, this are: *code control flow* and *security and privacy*. The code control flow section consists of items related specifically to the handling of exceptions, correct error handling and performance improvements. The only item which the team found important and we put in the short list was clear and useful logging messages. For the security and privacy section, only the items of the correct usage of the access modifier, for example public versus private attributes, and passing through specific values instead of exposing a whole object were deemed interesting by the developers. The other items listed in the long list were not deemed necessary at the current stage of code reviews. In case of security and privacy items regarding authentication, encryption and SQL injection were not deemed necessary. They did not want an extensive checklist to start with and prioritized other items over these items as they thought these items do not affect a greater part of the code reviews, thus the items are too specific and do not occur frequently.

The second to last section in the short list is testing, this was in its entirety taken from the long list. Testing was found very important by the team, which we also discovered in the first session. The section includes whether or not the written code is well testable. Whether there are enough test cases covering the critical parts of the code. The definition of *critical parts of the code* is not well specified and is up to the developers to assess. However, in general it could be said that all logical code which is not obviously correct should be unit tested. This does not include parts that are hard to test with unit tests such as establishing network connections and GUI code. Lastly, the section includes whether the tests are actually correct.

The last section in the short list contains a sole item as an alternative of the *external dependencies* category from the long list. It covers whether an additional review is needed either from a follow team member or another peer with more knowledge. For example, when creating a repository it is common, in addition to a review from a team member, a member of the team responsible for the deployments reviews the pull request. This item was an eye opener for the team for situations where the reviewer does not have enough knowledge. Instead of simply accepting the pull request another developer with more knowledge could be asked to perform a second review.

Summary: A long list of checklist items was created as source of inspiration during the second session. In the second session we explored together with the developers of the scrum team the key factors in the code review process. First of all the developers preferred a rather short checklist over an extensive checklist. Therefore, the long list was reduced into a short list of potential checklist items based on the preferences by the developers of the scrum team. The key factors we discovered were: general set-up of the pull request, a single item mentioning the coding guidelines, naming of variables, code control flow and unit testing.

6.3. EXPLAINING THE CHECKLIST

Based on the shortlist as a result of the second session the list was still too long, we decided earlier that the checklist should fit on a single A5 paper which the developers could easily keep on their desk. Therefore, the list had to be narrowed further down and a feedback session with the architects was held together with discussing the last details of the final experiment. This also lead to new insights from the architects and what they deemed important, the insights are included in the explanation of the final checklist next in this section. Furthermore, we asked the developers of the scrum team for additional feedback on the items included in the shortlist. In the final checklist we reintroduced some factors that were scrapped, summarized existing factors and added new factors. Furthermore, the category headers were removed to save up space and most items became self-contained. The final checklist is shown in Section 6.4 and in Appendix G. Next we will explain

the final checklist and the choices of the items in the checklist.

In the checklist most implementation items are summarized into a single item whether the code is easy to read and understand. Hereby, the coding principles *KISS*, *YAGNI* and *DRY* are listed as reminders to achieve readable and understandable code. The items in the shortlist related to correct functionality are in the perception of the developers obvious and do not necessarily be included in the checklist. The item whether the functionality is implemented at right layer and place is still added to the checklist. Furthermore, upgrading the version number comes back later on in the checklist.

The principles of *SOLID* are still separately listed in the checklist. In general most developers at Greenchoice could increase the familiarity with these principles. Especially the following principles: *single responsibility* as this reduces the complexity of methods and classes. *Open/closed* to also reduce the complexity and in addition prepare the code for future changes in terms of extension. *Dependency inversion* for decoupling the code increasing the testability. The other two principles are useful however they are not commonly found in the code base, though for completeness they were added to checklist.

In the next item on the final checklist we refer to whether the code follows the specific coding guidelines on the Wiki page. The coding guidelines are supported by a newly introduced item based on the input from the architects, checking for SonarQube warnings, in case a SonarQube report is available during the pull request. For most builds a direct SonarQube report is generated, however this does not hold for all projects. This item was added as SonarQube automatically checks for the quality of the code helping the reviewer and also the author to identify defects. Additionally, SonarQube helps in detecting and thereby solving the defects early on instead of fixing them later and accumulate the technical debt. The two naming items of the shortlist are untouched and directly copied into the checklist.

Next item on the checklist is whether relevant information is logged, this item was slightly rewritten to be more clearly including the log level. Based on the input from the architects correct exception handling was reintroduced on the checklist. They considered it important that specific exceptions are used and handled in a correct manner to retrace potential crashes. The items regarding the access modifiers and passing of values instead of full objects are removed as they are too specific and to make space for more important items.

In the next item the three testing items on the shortlist are summarized into a single item. The developers think a single item is enough to be reminded to also check for the test cases. Subsequently, two additional items were introduced based on the input from the architects covering the project file. The first item covers whether no unnecessary dependencies are included in the project file and whether the file is neat. The second item checks whether *treat warnings as errors* is activated in the project file. With the first item the architects want to prevent old unused dependencies stay alive in the project and making the project file messy and slowing it down [101]. The second item enforces the build to only succeed when all ReSharper warnings are solved. ReSharper check mainly code style issues rather than the code smells that Sonar checks on. The usage of *treat errors as warnings* is not always activated, however, it is Greenchoice's policy and therefore included as an item in the checklist. The second to last item in the checklist is whether the version numbers are updated, this is an often forgotten task as we discovered in the comment data analysis. The last item is similar to the last item of shortlist, whether an additional reviewer with more knowledge is required for assessing the pull request thoroughly.

Summary: Based on additional feedback from the developers of the scrum team and the architects the final checklist was created. The items in the final checklist were mostly copied from the items in the short list, however few items in the short list are summarized into a single item in the final checklist or refined for the final checklist. The general set-up items were considered rather obvious and were left out to spare space. Additional, based on the feedback of the architects items related to usage of tooling and project files were added in the final checklist.

6.4. FINAL CHECKLIST

In this section the full checklist is shown as explained in the previous section.

- Is the code easily readable and understandable? (also think of KISS, YAGNY, DRY)
- Is the new functionality implemented at the correct layer and place?

- **Single responsibility,** has a class or method a single solely purpose?

- **Open/closed,** is the open for extension and closed for modification?
- **Liskov substitution,** are objects replaceable by its sub classes without alternating the functionality?
- **Interface segregation,** does the interfaces only contain the used necessary methods?
- **Dependency inversion,** does the code depend on abstractions instead of concrete implementations?

- Does the code comply to the coding guidelines (see the Wiki page)
- Are all SonarQube warnings processed (if available during code review)
- Is the naming clear and obvious, does the naming represents the actual functionality?
- Are the business terms in Dutch and the rest in English?
- Is relevant information logged and at the correct log level?
- Are specific clear exceptions given and handled correctly?
- Are there enough correct tests written covering the critical parts of the code?
- Is there no unused information in the project file, are the references correct?
- Is 'treat warnings as errors' activated, do all files have a green check mark (no ReSharper warnings)?
- Are the necessary versions number updated?
- Is there an additional review from an expert or team member needed?

6.5. DISCUSSION

Together with a scrum team we explored which factors should be included in the checklist for code reviews. In Section 6.5.1 we will discuss first the discovered strengths and weaknesses in the code review process of the scrum team. In Section 6.5.2 we will further discuss how the checklist was created, compare the created checklist to our indented expectations and finally answer the third research question.

6.5.1. FINDING THE STRENGTHS AND WEAKNESSES OF THE SCRUM TEAM

In the first session together with the scrum team we explored the strengths and weaknesses regarding their code review process. Our intention was to discover the first items to include in the checklist, however we ended up with mostly similar results compared to the interviews of which the results are shown in Section 5.1.3. The discovered strengths and weaknesses are mostly process related and are hard to translate into actual checklist items. Though, we discovered in the session that we should pay attention to clear coding guidelines and testing in the checklist.

SIZE OF A PULL REQUEST

The first factor that was discussed was the size of the pull requests. During the session a discussion started when a pull request is too large. There is no single criterion that determines whether a pull request is too large. The outcome of the discussion was that in most cases it actually depends on the situation. In general it is a combination of the number of files, lines of code and the comprehensibility of the change. The comprehensibility of the change is mostly determined by how many tasks or responsibilities the pull request addresses. In the reasoning of the developers it is as a reviewer easy to follow a single path through the code, however once there are multiple paths which are intertwined it gets harder.

The team had recently a larger pull request covering lots of refactoring work denoted in single PBI. It is company policy that this PBI should be completed within a single pull request, therefore it was an anticipated large pull request, however the pull request was still disliked by the team. The refactoring work consisted of multiple smaller tasks that were in a way intertwined according to the team. This situation could either be solved by spending more time to split the PBI into several smaller PBIs or the usage of a feature branch. The usage of a feature branch provides the opportunity to merge smaller incremental changes on the feature branch and recieve regular feedback throughout the refactoring. Finally, a pull request from the feature

branch into the master branch could be created to complete the PBI. The latter merge will still be a larger pull request, however most likely all changes have already been reviewed and should be easier to complete therefor. In addition, using a feature branch has the advantage of keeping the master branch clean of unfinished work. Furthermore, a tool similar to the tool discussed by Barnett *et al.* [68] could be used to split a larger pull request in several smaller independent pull requests.

Furthermore, scope creep was mentioned together with the size of the pull requests. Especially in case of larger pull requests the team sometimes struggles to keep the balance between a small improvement that fits within the pull request or when to create a separate PBI. This is also similar to our findings we discovered in the interviews.

TIME OF PULL REQUEST

In the session we discovered that most pull requests are reviewed within a short time period, mostly within a day. This is comparable to the results we showed in the data analysis in Section 5.2. The developers of the scrum team also indicated that larger pull requests require more time and more iterations. In the data analysis we could not find hard evidence this holds in general, though it could be that this is specifically for this team. In case of the larger refactoring pull request which was described in the previous paragraph it indeed took more time, several days, and required multiple iterations going over the pull request.

TESTING

Next, we gained additional insights on top of the earlier findings in the interviews regarding the usage and review of the unit test cases, especially the testability of the code. The scrum team does not have a dedicated tester, this means that every developer within the team is more responsible and more aware of the test process. The presence of unit tests is important to the team, especially as they develop mostly new code. Based on these observations we consider it important to include test relate items in the checklist to assist the reviewers in the code review process.

KNOWLEDGE SHARING

In the exploration of Greenchoice's motivation for the review process, shown in Section 4.2.1 we have already discovered that knowledge sharing in an important motivation. Additional, in the interviews we also discovered that knowledge sharing is an important factor in the code review process. In the perception of the developers of the scrum team knowledge sharing is also in their team an important aspects.

STRICTNESS

In the interviews we discovered that code reviews are performed sometimes a bit lax. Although, a single developer mentioned the strictness on a post-it the other developers agreed to this in the discussion. However, it is sometimes hard to know what to improve on during the code review. Especially in case it is unknown when and how the code should be modified in the feature. There should be a balance in small incremental steps improving the code quality baseline and nitpicking [37, 67, 102].

6.5.2. CREATING THE CHECKLIST

In the second session we explored which items to include in the checklist. First off, we created a long list, shown in Appendix E, of items that are possibly of interest to include in the checklist based on existing checklists [86–88] and our current findings. The long list included general items on whether or not the pull requests is an appropriate solution to the PBI and whether the code is readable, understandable and maintainable. Furthermore, we included most coding guidelines we could find on the Wiki page of Greenchoice to assist the developers in assessing the code quality. Next, we included several items for code control flow, security and privacy we did not add too many items as we observed little attention is paid to these factors and we did not want to overwhelm the developers with these items and rather focus on the other factors. Though, for our own inspiration we added several items that we could bring up during the second session and start a discussion to explore the thoughts of the developers on these items. Lastly, we added testing items which we thought were relevant based on the results of the first session.

During the second session we discovered that the team preferred a rather short checklist. Based on their preferences the long list was heavily reduced in size into a shortlist, shown in Appendix F, with potential checklist items. Based on additional feedback from the developers after the second session and together with the architects we created the final checklist. In the checklist the implementation items were summarized into a single item, only the SOLID principles were added as separate items. In addition, the specific coding guidelines items of the long list were replaced by a single item referring to the Wiki page with the coding guidelines.

Additionally, we included the usage of SonarQube reports as this tool automatically detects code violations helping the reviewer to asses the code quality. The naming items were held intact and we reintroduced based on the feedback of the architects correct handling of exception and polished the item regarding logging. The three test items were summarized into a single item. Additional, we added items regarding the project file based on input from the architects. The checklist was ended with whether the necessary versions number are upgraded and whether an expert review is needed.

Based on the results of the items in the final checklist less attention is paid to the implementation and coding guidelines as we initially intended. The items regarding the implementation in the shortlist, for example whether the solution is appropriate to the PBI, are in the perception of the developers of the scrum team rather obvious. They substantiate this by the fact they are already performing these checks during a code review. In case of the coding guidelines these items are less obvious as we discovered in the interviews showed in Section 5.1.3 and in the first session with the scrum team. However, the developers of the scrum team preferred a short checklist which leaves little space for specific coding guidelines. This means less structure is given with the current checklist in terms of improving the strictness regarding the coding guidelines. Though, still mentioning that the coding guidelines can be found on a Wiki page and the inclusion of the SonarQube reports in the code review process we still try to improve on the coding guidelines. Additional, including the separate principles of SOLID as items in the checklist we try to improve on the higher level implementation guidelines.

The final checklist is not a complete and extensive list of all items that should be checked on during a code review due to the limited size. This means that the pull requests still should be checked thoroughly based on the common knowledge of the reviewers even once all items in the checklist are checked off. In addition, the checklist does not provide the reviewers with a review strategy on how to perform the code review. The final checklist provides additional guidance in the code reviews by adding specific items related to the discovered key factors during a code review. This means that we included items related to the naming of variables, unit testing and the project file. The latter items are somewhat specific and only applicable when the project file is changed. However, the items addresses issues that are often forgotten during a code review. Similar to upgrading the necessary version numbers and whether a second review from an expert is needed. Including these types of items in the checklist we intent to prevent that these items are forgotten in upcoming code reviews once the situation occurs.

RQ 3 Which factors should be included in a well designed checklist for code reviews?: The first factor to be considered is that a rather small checklist is preferred over a longer extensive list of items. This limits the number of specific items that can be included in the checklist. The factors that should be included are implementation in terms of understandability and readability together with the SOLID principles. In general the coding guidelines supported by the correct usage of the tooling, which are SonarQube and ReSharper. Furthermore, the naming of variables, logging, set-up of the project files, updating the version numbers and a second review from an expert were considered important and therefore included in the checklist. The latter three factors are often forgotten during a code review. In short a compact checklist is desired emphasizing the important factors that are often forgotten during a code review.

7

USING THE CODE REVIEW CHECKLIST

To measure the effectiveness of the checklist created in the previous chapter we performed an experiment based with a quasi-experimental pre post test design [33] as explained in Section 3.4. Initially 3 teams consisting of a total of 9 developers and 2 testers participated in the experiment, however a scrum team left throughout the experiment. The experiment started with a pre questionnaire that was sent out to 3 scrum teams to gather the expectations on the usage of the checklist. The pre questionnaire was answered by all of the 11 respondents of which the results are shown in Section 7.1.

Next, we asked the developers and testers to use the checklist during two periods of two weeks aligned with Greenchoice's sprint planning. In between the two periods we held with each team a meeting to evaluate the usage of the checklist, the results of these meetings are shown in Section 7.2. The testers did not participate in these evaluation session as it turned out they were barely involved in code reviewing and did not use the checklists. Furthermore, during the evaluation sessions one of the teams decided to leave the experiment and focus on their upcoming deadlines. Based on the feedback received in the evaluation sessions we decided to refine the checklist for the second period. The process of refining the checklist and the refined checklist is explained in Section 7.3.

The refined checklist was used in the second period by the remaining 2 teams consisting of 6 developers and a tester. The tester indicated he did not expect that his activity with code reviews would increase for this period. At the end of the second period we sent the post questionnaire to the 6 developers of the two teams as the tester did, once more, not use the checklist in practice. During the second period one of the developers went on vacation and did not fill in the post questionnaire. The results of the 5 developers participating in the post questionnaire are shown in Section 7.4.

Finally, in Section 7.5 we compare the results of the pre and post questionnaires. Furthermore, we elaborate on the observations made by the developers participating in the experiment regarding the usage of the checklist and present our thoughts supported by the literature related to these observations.

7.1. PRE QUESTIONNAIRE

The pre questionnaire was sent out to the 9 developers and 2 testers participating in the experiment which all answered the questionnaire. The questionnaire consists of 8 section, as explained in the methodology in Section 3.4.2, the results of the pre questionnaire are further explained in this section as following. In the first question we explored the personal interest of the respondents in 8 different subjects related to developing software or using a checklist of which the results are shown in Section 7.1.1. Subsequently, in Section 7.1.2 we show the results on the thoughts of the respondents towards code quality and tried to clarify what is meant with the quality motivation found in Chapter 4. Next, in Sections 7.1.3 and 7.1.4 we show the results of the thoughts of the respondents regarding the separate parts of respectively using a checklist in general and using code review comments and tooling during code reviews. In Section 7.1.5 we show the results of the expectations of the respondents on the usage of the checklist during code reviews. Finally, in Section 7.1.6 we show the final remarks the respondents could leave behind.

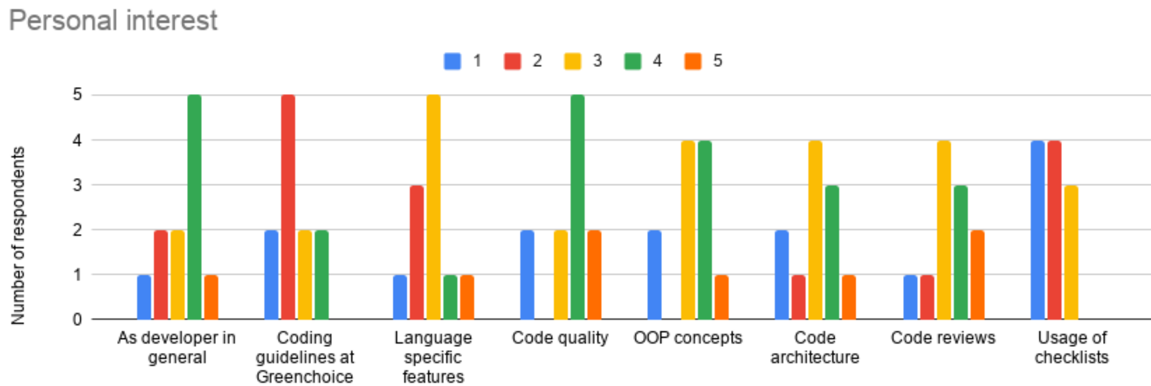


Figure 7.1: The person affinity of the respondents into the given subjects. Whereby 1 is the lowest affinity and 5 the highest.

7.1.1. PERSONAL INTEREST

In the first question of the questionnaire we gauged the personal affinity on 8 subjects of which 7 subjects are related to writing code and the latter subject is the usage of checklists. The used subjects and the results are shown in Figure 7.1. With affinity we imply the personal interest into that subject and for example how active they are keeping up with the latest trends. The affinity is rated on a scale of 1 to 5 in which 1 is the lowest and 5 the highest affinity, it was up to the respondents to interpret their position on the scale.

In the results we show that most respondents have a high affinity as a developer, only a 3 respondents have an affinity of 2 or lower, not surprisingly this includes both testers. However, for the coding guidelines at Greenchoice 8 respondents have a low affinity of 2 or lower with Greenchoice's coding guidelines. Additional, the subject language specific features has also a lower affinity, however in less extend than Greenchoice's coding guidelines. Furthermore, there is a high affinity for code quality and somewhat lower for object-oriented programming(OOP) concepts, architecture and code reviews. For these subjects both testers had, once more, less affinity, this is not surprising as they have less experience with developing in general. Lastly, less affinity is given for the usage of checklists, in our view this gives an opportunity to try out the usage of checklists as a new tool.

The coding guidelines at Greenchoice is lowest rated compared to the other coding subjects. We strongly believe this observation consolidates our findings in the interviews, shown in Section 5.1.3, that Greenchoice's coding guidelines are unclear to the developers. However, other undiscovered factors might also be at play resulting in the lower affinity requiring further research, for example it could be that Greenchoice's coding guidelines are not considered useful and therefore not used.

7.1.2. THOUGHTS TOWARDS CODE QUALITY

In this section we show the findings of the thoughts towards code quality. We explored which factors influences the code quality the most according the respondents. To discover these factors we used two questions; the first question contains of 6 general code quality statements which are further explained in the question. The second question asked the developers to select the 3 most import factors for code quality out of the 9 factors we also used as general categories in comment data analysis, explained in Section 3.2.3. This questions looks further into what the developers meant with the *quality* motivation found in Section 4.2.1.

GENERAL CODE QUALITY STATEMENTS

The 6 general code quality statements covers which factors are important in determining code quality, causes for lower quality code and whether in practice problems are encountered due to lower quality code. In Figure 7.2 we show the statements and the answers to the statements. In the first statement whether or not functionality is more important than code quality a majority of 6 respondents disagreed, however, 2 respondents thought that functionality is more important. Secondly, the respondents were evenly divided whether or not bugs are the result of the carelessness of the author. For the next statement whether messy code is the result of time pressure the 6 respondents and thereby a majority was neutral, though there were 3 more agreeing answers than disagreeing answers given. The respondents unanimously agreed that testing is an important factor in determining the code quality. In the fifth statement a slight majority 6 respondents agreed that they

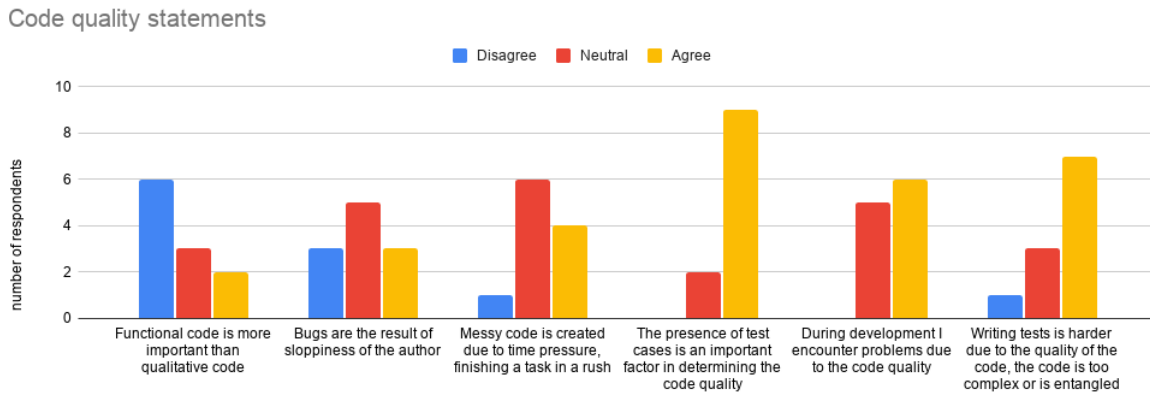


Figure 7.2: Results of the general code quality statements.

factors	Count	Percentage of respondents
Clarity and understandability	10	91%
Cleanliness and structure	7	64%
Test cases	5	45%
Architecture	4	36%
Functional bugs	3	27%
OOP concepts	2	18%
Project set-up and configurations	1	9%
Code improvements	1	9%
Code security and monitoring	0	0%

Table 7.1: The results of the factors that are deemed important in determining the code quality. In the first column the factors are shown and in the second column the count of how many times the factor is mentioned. In the third column the percentage of respondents that has given this answer is shown based on the total of 11 respondents.

run into issues in practice during programming due to the code quality, the other 5 respondents were neutral. Subsequently, in the last statement 7 respondents agrees that writing test is harder due to lower quality code. We should be careful interpreting this question as we did not explicitly mentioned that the developers should encounter the statement in practice at Greenchoice. Although the ambiguity in the formulation, we still consider that writing test is harder in practice at Greenchoice due to too complex code. This is also based on the earlier observation in the first session of creating the checklist that writing test can be hard due to the code quality, described in Section 6.1.

Important factors for code quality

In the second question we asked the respondents to select 3 factors of the 9 factors we used as general categories in the data analysis, explained in Section 3.2.3, of which they think are the most important in determining the code quality. In Table 7.1 we show the results of the most important categories according the respondents. The most important category is *clarity and understandability*, for which only a single respondent did not consider it important. This is inline with the earlier results of the interviews, Do I understand the code, as one of the important checks going over the pull requests. Next, is *cleanliness and structure* with 7 answers followed by *test cases* with only 5 answers. The latter is particular, in the statements we showed that 9 respondents thought it is an important factor in determining the code quality. Furthermore, in the interviews we discovered that testing was given less attention, although they also mentioned during the interviews testing itself deserves more attention. Remarkably, both testers did not include *test cases* in their answers. Furthermore, we received 4 answers for *architecture* and 3 answers for *functional bugs*. *OOP concepts*, *project set-up and configurations* and *code improvements* were all at least answered once. A single category was not mentioned at all this was: *code security and monitoring*. Although the given factors are still rather broad we gained a better insight to the *quality* motivation, found in Section 4.2.1. Hereby the factors *clarity and understandability* and *cleanliness and structure* are two important factors to determine the quality of the code.

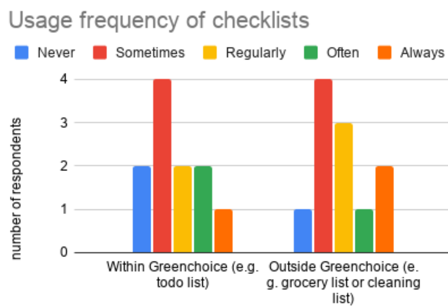


Figure 7.3: Usage frequency of checklist within Greenchoice and outside Greenchoice.

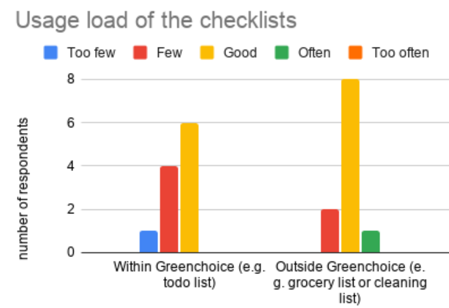


Figure 7.4: Usage load the checklists used within Greenchoice and outside Greenchoice.

7.1.3. THOUGHTS TOWARDS CHECKLISTS

In this section we explored the thoughts of the respondents with two questions towards the usage of checklists in general. The first question addresses the usage frequency and the second question contains 5 statements looking into the expected results in using a checklist.

FREQUENCY

In the first question we explored the usage frequency of checklists within Greenchoice and outside Greenchoice, for example using grocery lists, of which the results are shown in Figure 7.3. Subsequently, we explored their opinion on the usage frequency of using checklists, further referenced as usage load, we show these results in Figure 7.4. We are interested whether the respondents are overwhelmed in the usage of checklists work related and also outside Greenchoice.

The usage frequency of checklists is rather divided over the different answers and the results within and outside Greenchoice are quite similar. In terms of usage load of checklists there is a slightly clearer difference, in personal usage the frequency of checklists is experienced good by 8 respondents and 2 respondents mentioned little and another respondent answered often. For the usage within Greenchoice 6 respondents answered good, the other 5 respondents found the usage load to be little of which one respondent answered too little. Based on the answers we conclude the usage load of checklists within Greenchoice is little to good and outside Greenchoice good.

STATEMENTS

In the second question we gave the respondent 5 statements on the expected advantages and disadvantages of using a checklist. In Figure 7.5 we show the statements and the results to each statement. Each of the statements has two options and its own scale ranging from 1 to 5. At which 1 is totally agree with the first option on the scale, 2 is slight agree to first option, 3 is neutral, 4 is slightly agree to second option and 5 totally agrees to the second option on the scale. This means that for the first statement the first option is annoying and the second option is pleasant.

In the first statement the general experience of using checklists lies according to the respondents between neutral and pleasant. The second statement time spend is centered around neutral, however, with a slight edge with 2 more respondents agreeing to time consuming compared to time saving. In the third statement the respondents experienced that checklists are mainly easy to use and in the fourth statement also provides structure. Whereby structure is defined as following: a checklist gives a clear overview of tasks at hand. Lastly, checklist provide a somewhat higher precision equally divided between neutral and precise and a single totally precise. Precision reflects that all tasks are performed instead of forgetting to perform a task.

In general checklists are pleasant to use and although checklist might require additional time to use, checklists increase the structure and precision of the tasks to perform.

7.1.4. THOUGHTS TOWARDS CODE REVIEW COMMENTS USAGE AND TOOLING

In this part of the questionnaire consisting of 3 questions we explored the thoughts of the respondents towards code review comments in general. The first question dives into whether or not review comments are left out during code reviews which later causes issues when programming. The second question consists of 5 statements regarding the code review comments usage in DevOps. The last question dives into the usage of tooling assisting the developers in the code review process.

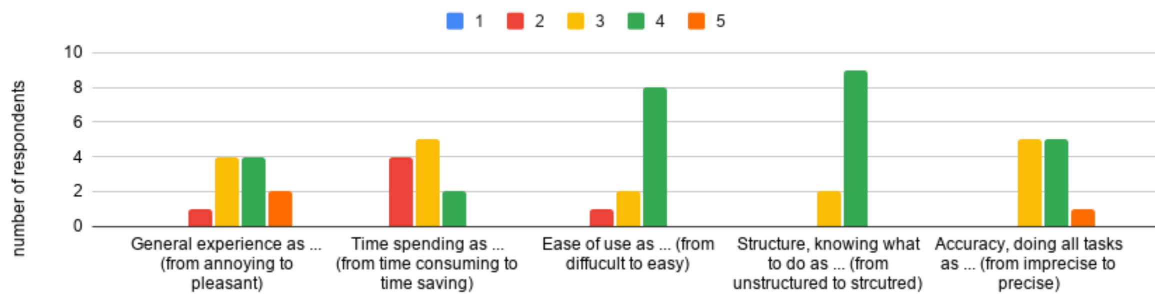


Figure 7.5: General statements on the usage experiences for checklists. Whereby answers 1 and 2 respectively totally agree and agree to the first option, answer 3 is neutral and answers 4 and 5 respectively agree and totally agree to the second option.

LEFT OUT COMMENTS

In the first question we gave the respondents the following statement: *I sometimes leave out comments during a code review, which I later encounter as issue during programming*, on a 5 point Likert scale. We included this statement as we got the impression during several interviews this was the case. This can be caused by the sometimes lax code reviews or developers not wanting to become the bogeyman, as explained in Section 5.1.3.

Of the 11 respondents 2 answered never, 4 sometimes, 4 regularly and a single respondent answered often. With these result we can assume that comments are sometimes left out during code reviews and are later encountered as an issue during programming. However, this question does not clarify the exact reasoning why the comments are left out. The reason could still be being afraid of becoming the bogeyman, the sometimes lax behaviour during code reviews or a combination of both including other factors as well.

COMMENTS USAGE

In the interviews and through observations at the work place we found that the usage of review comments in DevOps is not as trivial as we initially thought and feedback was also given face to face. To explore the thoughts of the developers on the usage of code review comments in DevOps we included the 5 statements shown in Figure 7.6 including their results.

The first statement explores whether the review comments are used as reminder for a later face to face discussion. We got this impression from the architects as they observed it being used in practice and are using it themselves. Four respondents agreed to the statement and the other 7 respondents are neutral.

The second statement has a similar intent to the first statement, however, now as a reminder for the author of the code to resolve the given feedback. We found that a majority of 9 respondents agreed to the statement and the other 2 respondents were neutral. The result of the third statement has a similar result, however covers whether the comments provide awareness to the other developers who are not directly involved in the code review. Having knowledge sharing found earlier as separate motivation in Section 4.2.1, in the interviews in Section 5.1.3 and during the creation of the checklist in Section 6.1 we showed that knowledge sharing is an import factor for code reviews at Greenchoice and this result is not surprisingly.

The second to last statement questioned whether comments can be used to later retrace why and how the code is changed. This can especially be useful in case to discover the cause for defects or even predict upcoming defects [71, 72, 74]. The given answers are mainly neutral with 7 answers and the other 4 answers are equally divided over disagree and agree both having two answers. The last statement whether or not using comments in DevOps is an unnecessary administrative task is answered with a majority of 9 respondent disagreeing and the other 2 respondents were neutral.

In general the usage of review comments in DevOps provides advantages in terms of reminder to process feedback and in less extend to give feedback during face to face conversation. Additional, the usage of code review comments provides awareness to the other developers who are not actively involved in the pull request. The respondents are neutral or divided regarding the historical usefulness of the review comments. Though, acknowledging these benefits the respondents luckily do not think the usage of comments is an unnecessary administrative task.

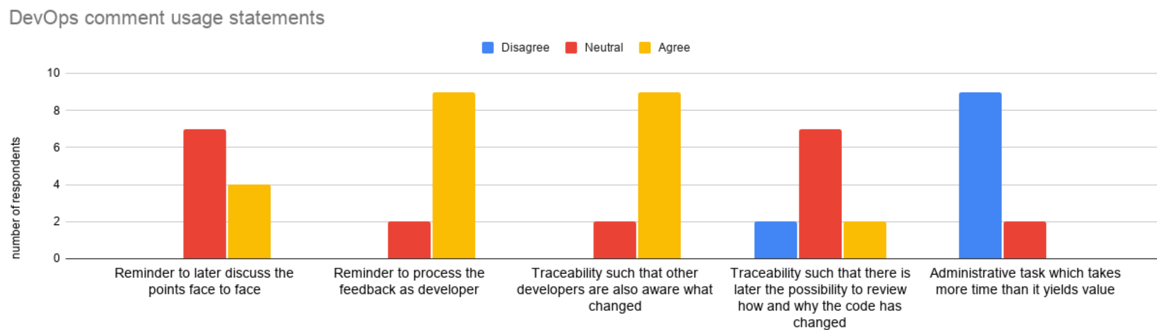


Figure 7.6: General statements regarding the usage of code review comments in DevOps.

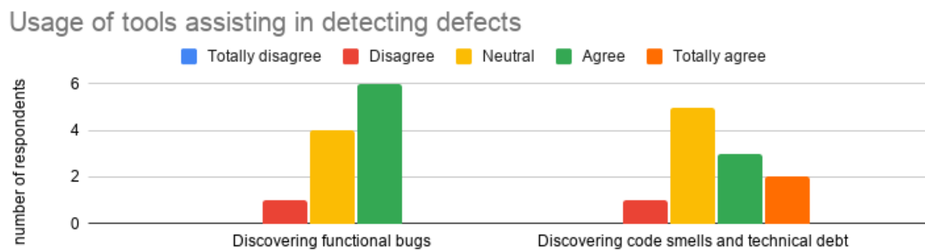


Figure 7.7: Whether tools could be used for discovering functional bugs, and code smells and technical debt

TOOLING

In the remaining questions of this part of the questionnaire we looked into the usage of tooling during code reviews. First we explored for which type of defects tooling is perceived as useful in actually detecting defects. SonarQube can detect different types of defects: bugs, (security) vulnerabilities and code smells (maintainability issues)[93]. We are interested whether there is a difference between the usefulness in detecting bugs and code smells by the tools. Tools like SonarQube are good in detecting patterns in the code that are known to cause issues, however the tools itself cannot interpreted whether the intention of the code is correct. In other words whether the code actually represent the desired functionality in the PBI.

In case of an extensive if-else statement, SonarQube can detect that the amount of cases is excreting a predefined threshold making it hard to maintain the if-else statement in the future. It is rather hard to detect whether equations to determine which part of the code to execute are correct. However, depending on the type of expressions SonarQube can detect that a specific case is missing. Another example, SonarQube can detect the value of variable might be null causing an unexpected null pointer exception preventing a possible bug, however it cannot detect whether the name of the variable is clear to be used later on, missing out a possible maintainability issue.

Based on the different types of defects a tool can find we are interested how useful the usage of tools are to detect the different types of defects. We excluded the vulnerability defects as these play a minor part in the current code review process. In Figure 7.7 we show the results whether tools are perceived to be useful in detecting bugs and code smells. In general the respondents agree tools can be used to detect both defects. There is a small difference that code smells has an additional neutral answer and two totally agree instead of 3 agreeing answers for functional bugs. Furthermore, both types of defects have a single disagree answer.

Furthermore, we gauged whether the developers are missing specific tools, currently Greenchoice uses ReSharper and SonarQube, which was later upgraded to SonarCloud. ReSharper gives the developer additional refactor options and functions as a code style checker. Additional, it can resolve most code styling issues itself. SonarQube, as explained earlier, detects possible bugs, security vulnerabilities and code smells. ReSharper is enforced during a pull request by using the *treat warnings as errors* option in the project file. Greenchoice has a customized style sheet that should be used in every project, we observed that in practice different style sheets were used, however we are unfamiliar whether this is a common issue for all projects. In each build a new SonarQube report is generated and therefore SonarQube is also integrated in a pull request. However, the feedback is not directly integrated in a pull request and the usage of the report is often

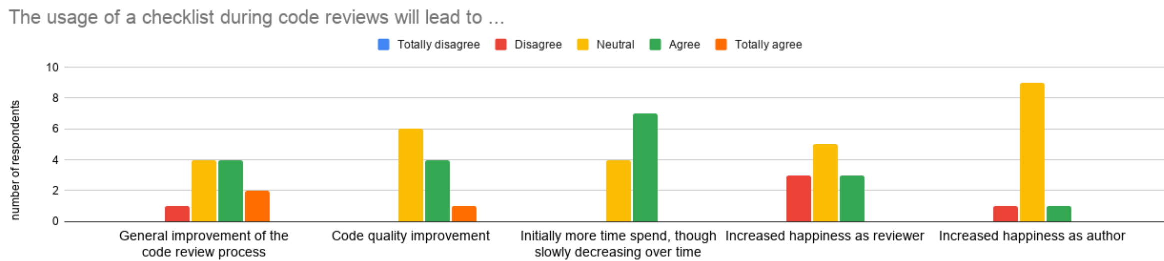


Figure 7.8: General statements whether the usage of a checklist will lead to an improvement given in the statements.

neglected. Developers are either not aware of the generated report or the report provides little information of which new issues are added due to the pull request as we earlier discovered in the interviews, explained in Section 5.1.3.

Only two respondents actually mentioned that they are missing tools during code reviews. The first respondent mentioned the wish of directly integrating the SonarQube feedback in pull requests. With the upgrade from SonarQube to SonarCloud this should be possible. The second answer mentioned multiple tools: *"categorizing code review comments, unit testing linkage to test scenarios. Setting quality gates on checked in code and automated security tests (e.g. anti forgery tokens and encryption)"*. The respondent did not leave a reasoning why these tools should be used. In case of setting quality gates we assume this is proposed to ensure a minimum quality of code in the code base.

We asked furthermore whether the developers are aware of specific tools that could be used in process, however no tools were suggested.

7.1.5. EXPECTATIONS USAGE CHECKLIST DURING CODE REVIEWS

In the final section of the questionnaire we explored in 4 questions the expectations of using a checklist during code reviews and what the expected result could be. The first question consists of 5 statements stating possible improvements as results of using a checklist. The second question gauged how often the respondents expect they will use the checklist. The third question explores the usage of comments instead of only face to face communication and whether they are encouraged to give more feedback with the usage of a checklist. The last question looks into the expectations of the respondents to which aspects the usage of a checklist will lead to an increased comment frequency.

EXPECTED IMPROVEMENT STATEMENTS

In the first question we gave the respondents 5 statements gauging possible improvements of using a checklist during code reviews. The 5 statements together with their results are shown in figure 7.8.

First of all, the first statement gauges whether the respondents think the usage of a checklist will lead to a general improvement in the code review process. Only a single respondents disagreed, 4 respondents were neutral, 4 respondents agreed and 2 respondents totally agreed. Based on these answer we can assume they expect the usage of the checklist will lead to a general improvement in the code review process.

Whether the usage of the checklist also improves the code quality is gauged in the second statement. Here 6 respondents were neutral and the other 5 respondents agreed of which a single respondent totally agreed. This means the respondents also expect that the code quality will increase due to the usage of the checklist.

In the third statement they expect with 7 agreeing answers that the usage of a checklist will initially take more time, however the additional time spent will decrease once they get familiar with the checklist.

The last two statements cover the satisfaction of the respondents either as reviewer or as author. In the role as reviewer there is an equal division between disagree and agree with 3 answers each for both. In the role as author only a single answer for both disagree and agree are given and the other 9 answers are neutral. In case of the role as reviewer the checklist might either decrease or increase the satisfaction depending on the respondent, in the role as author the expected effect on the satisfaction is unknown.

CHECKLIST USAGE FREQUENCY

In the second question we gauged the intended frequency for the respondents of using the checklist. We do not expect that the checklist will be used for every pull request. Using a checklist is a new additional step in

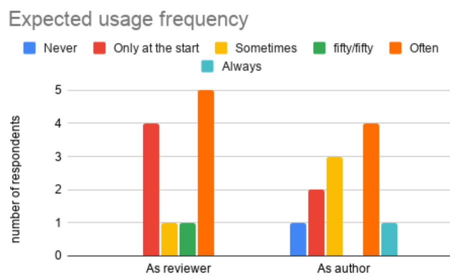


Figure 7.9: Expect usage frequency for the checklist during code reviews.

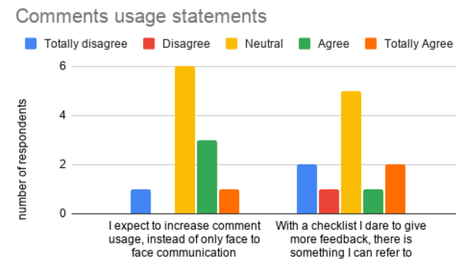


Figure 7.10: Two statements for expected usage of comments during code reviews.

their current code review process and the checklist could be forgotten to be used, however having the checklist printed on their desk we encourage the developers on actually using the checklist during a code review. Furthermore, the checklist might require additional time and the developer does not have the additional time available and skips the use of the checklist. On the other hand the checklist can become trivial after using it for a couple of times, meaning the developer does not need to actually read the checklist in order to remember the items on the checklist. We looked furthermore into the usage frequency of the checklist as author of the pull requests, this to stimulate that the checklist can also be used during the sanity check of the author.

The results of both expected usage frequencies are shown in Figure 7.9. In the role as reviewer the answers are mostly divided between *only at the start* and *often*. In the role as author the answers are divided over *only at the start*, *sometimes* and *often*. In Section 7.4.2 we show the actual usage frequency of the checklist according to the respondents and in Section 7.5.2 we discuss the differences between the expectations and the actual usage.

COMMENTS USAGE

In the third question we dived further into the usage of comments with two statements. The first statement gauged whether the respondents expected to use more review comments in DevOps instead of using face to face communication. The second statement covers whether they are encouraged to give more feedback by the usage of the checklist. In particular whether they dare to give more feedback as the checklist provides an agreement on the guidelines without becoming the bogeyman. Additionally, the checklist provides guidance in where to check on during code reviews.

The statements and the results are shown in Figure 7.10. The first statement has a single totally disagree and 3 agreeing answer of which a totally agree. The second statement is equally divided with two totally disagree or agree and a single disagree or agree.

The first statement whether more comments will be used has a slight majority, however these teams are already actively using the review comments in DevOps. Based on this reasoning we would expect more disagreeing answers as the amount of comments would not increase. We rather think we did misplaced this question as we did not specifically mentioned the code review comments in DevOps.

The second statement whether the respondent dare to give more feedback is rather divided. Furthermore, we also think we misplaced this statement. Based on the current statement it is hard to conclude whether the usage of the checklist actually resolves the issues of being afraid to become the bogeyman and not knowing where to review on.

EXPECTED INCREASE IN COMMENT FREQUENCY

In the last question of the questionnaire we explored the expected increase in the amount of comments for each of the 9 aspects according to the respondents. These 9 aspects are similar to the categories used in the comment data analysis as explained in Section 3.2.3. Notice, the respondents have not yet seen the full checklist, only the scrum team who participated in the creation of the checklist has seen parts of the checklist. Based on the expected increase in the frequency of the comments for each aspect we show in Section 7.4.2 the actual perceived increase for each of the aspects according to the respondents. Furthermore, in Section 7.5.3 of the discussion we will discuss the differences between the expectations and the actual perceived increase.

The results for the 9 aspects are shown in Figure 7.11. The answer *it decreases* was not used and therefore left out in the graph. The only aspect that has more answers for *it increases* than *it stays similar* is *OOP concepts*. Followed by *code improvements* and *test cases* with a single less answer for *it increases*. With, once

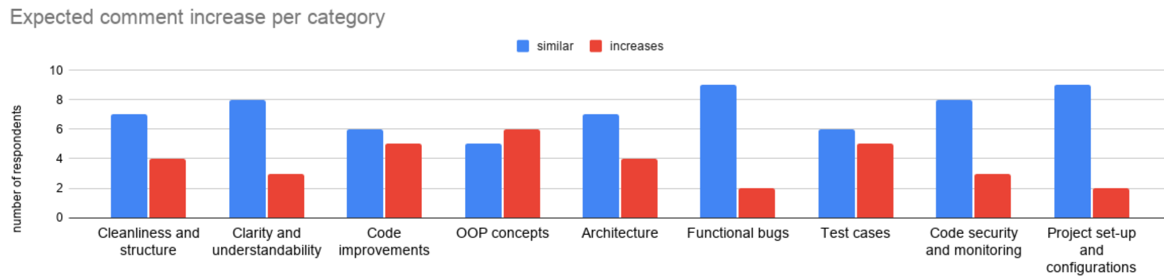


Figure 7.11: Expected increase on the amount of comments for each aspect according to the respondents.

more, a single less answer for *it increases* follows *cleanliness and structure* and *architecture*. Lastly, follow by *clarity and understandability* and *code security and monitoring* respectively 3 answers for *it increases* and *functional bugs* and *project set-up and configurations* with 2 answers.

7.1.6. FINAL REMARKS

Lastly, the respondents had the chance to leave a final remark. There were two remarks given: *"Ideally, you want to have the checklist embedded in DevOps as much as possible, so that it becomes natural to use it and code cannot be merged if it does not meet the requirements"* and *"That as a tester I can contribute more to code reviews than currently, because there are now guidelines for it"*. The latter remark gives the potential that the introduction of the checklist helps in actually involving the tester more into the code review process.

The first remark is rather interesting to look into, whether it could be possible to integrate the checklist within DevOps and possibly as a gatekeeper. The advantage of including the usage of the checklist as gatekeeper enforces the respondents to consciously check off the checklist during a code review, however, the disadvantage is also that it can be counterproductive resulting in delays during hot fixes. Additionally, a similar scenario can occur that the items are only checked as it is required similar to enforcing a minimum test coverage, whereby tests of lower quality are written to achieve the minimum test coverage, explained in Section 5.1.3.

Summary: In the pre questionnaire we looked into the personal interest for several software development subjects. Next, we explored the factors which are considered important in determining the code quality. Subsequently, we explored the expectations of using a checklist in general. Furthermore, we looked into the usage of code review comments in DevOps and the use of tooling during code reviews. Finally, we exposed the expectations of the respondents regarding the usage of a checklist in the code review process.

7.2. CHECKLIST EVALUATION

After the first period in which the three teams have used the checklist shown in Section 6.4 during a period of two weeks. We held with each of the three teams a separate evaluation session. The developers of the team, each team has 3 developers, participated in a roughly 30 minute open discussion. The testers did not participate as they were barely involved in a pull request and did not use the checklist that often in practice. During the evaluation sessions one of the teams decided to stop participating in the experiment, as participating in the experiment required too much effort and they wanted to focus on upcoming deadlines. Based on the feedback discovered in this section we refined the checklist which is further explained in Section 7.3.

In the evaluation sessions we discovered first of all that the structure we intended to give was not fully achieved. All teams mentioned that the items in the checklist are mostly stand alone items and are not applicable to all code reviews. This means that finding the appropriate items for the pull request is a tedious task. The developers would prefer that the items are categorized and based on the categories the developers then can decide whether the category is applicable to the pull request. One of the teams mentioned to include the checklist in DevOps, this team includes the developer who mentioned to integrate the checklist in DevOps in Section 7.1.6. In addition, they mentioned the items in the checklist could be adjusted based on the files in the checklist. This lead to the idea to create the dynamic checklist generator which is further explained in the future work in Section 8.3.1.

Additionally, all teams experienced the items are either too obvious or not applicable to all pull requests. For example, a team mentioned the explanation of the separate SOLID items should be common knowledge and could be replaced by a single item as they still think it is good to be mentioned on the checklist. In case of items being not applicable to all pull requests a team was mostly setting up new projects during the usage period and had therefore many configuration pull requests. The checklists does barely cover these types of pull requests and made the checklist purposeless. Another team had several smaller pull requests and experienced going over the full checklist was not beneficial, though they also mentioned the smaller pull requests are accepted too easily.

Furthermore, the item related to *treat warnings as errors* was mentioned to be overhead by two teams as this affect only the pull requests in which the project file is actually changed. Furthermore, one of the two teams mentioned this item is earlier considered a policy rather than a checklist item. Though, checking this once in a while should do no harm according to the team. This is related to a team mentioning the checks on the Sonar warnings and ReSharper warnings could be included in DevOps. Preferably the warnings of these checks are automatically shown in the pull request as it requires additional steps for the reviewer to check on this.

Vice versa two teams mentioned the checklist lacked items which are often forgotten during a code review, the only item included is upgrading the version numbers. During the evaluation sessions three additional items that are often forgotten were mentioned: the usage of *ExcludeFromCodeCoverge* in auto generated files, forgetting to update temporary values (e.g. localhost and default passwords) and whether there are contracts changed that also affects other teams.

Finally, it was mentioned by a single team that the checklist helps on holding to the agreements that were made. For example, it prevents discussing over whether or not whitespace is necessary to be fixed in the pull request, as stated in the checklist the code should be clean and correct and whitespacing is part of this.

Summary: In the evaluation sessions we have discovered the checklist does provide less structure as intended and includes several items that are either found not applicable or rather obvious in most pull request. Furthermore, we discovered the teams prefer more items which are often forgotten during programming and in the code reviews.

7.3. Refining the checklist

Based on the feedback received in the evaluation sessions we refined the checklist before the checklist is used in the second period. In this section we will explain the refined checklist to what has changed relatively to the firstly used checklist, the refined checklist is shown in Appendix H and in Section 7.3.1.

First of all, we want to improve the checklist in such a way that it provides more structure in the readability to the developers. This means we want to group the items, the groups are denoted by a general header which can be used to check off a group of items quickly as a whole. Underneath the headers the items will still be listed. These items can be followed up with more specific items, further on called explanation sentences, explaining the corresponding item in more detail. Once the first item is unknown or the developer is uncertain she can read the explaining sentences, otherwise these sentences can be easily skipped. To be able to place the additional headers we decided the checklist should fit on a single A4 instead of a single A5. This does not mean the checklist should grow in the amount of items. In the refined checklist we have made 6 groups with a total of 14 items and 24 explanation sentences. In comparison to the previous checklist the new refined checklist has 5 items less and still made an improvement on the depth of the items with the explanation sentences.

Additionally, we removed the items which are rarely relevant in a pull request. Also, we reduced the amount of items that are rather obviously or give a better interpretation to the item. Finally, we want to include more items that are often forgotten. These changes are further explained below as we go over each of the groups one by one.

General set-up

The first group on the refined checklist is general set-up. We have included this category to make a quick check whether the intention of the pull request is clear. This means is it clear what has changed and is the change correctly implemented according the requirements in the PBI. The items and explanation sentences in this group are new compared to previous checklist, however most items and explanation sentences were already introduced in the long list or shortlist of the creation of the checklist as showed in Chapter 6.

Coding guidelines

The next group is coding guidelines in which we also included the coding principles. The SOLID principle of which in the previous checklist each principle was explained separately is combined with the other coding principles as a mere explanation sentence. The first item still focuses on the readability, understandability and maintainability of the changed code. The explanation sentences consist of the familiar coding principles and whether the naming is correct. We added two additional explaining sentences to help the developer look for improvements related to the code quality, however, we still cannot give an extensive list of examples as the Wiki page does provide. The second item focuses on the neatness of the code and Greenchoice's guidelines. In the explaining sentences we included the correct usage of the tooling, instead of the separate items in the previous checklist for ReSharper and SonarQube reports. In case of tooling ReSharper warnings affect mostly the neatness of the code, however, SonarQube also detects issues regarding the maintainability of the code and could been listed as explaining sentence of the previous item. We did however decide to put the tooling together and did consider it to be part of the Greenchoice's coding guidelines. Finally, we gave a short summary of neat code for example no commented code and correct white spacing.

This means we reduced the amount of code quality items and added more depth through the explanation sentences in which we included several sentences that were previously listed as separate items on the checklist. The new structure allows the developers to quickly go over the code quality group, though also providing additional information as explanation sentences.

Testing

The third group is testing of which the first item is a simplified version of the test item on the previous checklist. The additional information of enough and correct tests is further described in the first explanation sentence of that item. Furthermore, we added an additional explanation sentence to the item to stimulate the reviewer to look for missing test cases, for example edge cases. The last item we included is that test cases should also comply to the coding guidelines. It is also important for test cases to be easy readable and maintainable and therefore comply to coding guidelines, especially when they need to be changed or when they fail and the reasoning must be determined.

Logging and error handling

The fourth group is logging and error handling, the group contains two items based on the items on the previous checklist. We also included additional explaining sentences to clarify the items. These explaining sentences are based on the feedback of the architects who deemed these aspects important in the creation of the previous checklist. Additionally, with the increase to the A4 paper we have now enough space to include these explaining sentences.

Project set-up

The fifth group is project set-up, this group contains the items regarding the project file of the previous checklist and a new item whether no hard coded values remain in the setting and configurations files after debugging. Although the existing items were not always applicable to every pull request as we discovered in the evaluation session, the group can easily be skipped when the pull request does not contain changes related to the items. Based on our personal experiences project files can quickly become messy. This observation was also made by Greenchoice's architects who included this in the previous checklist. The developers themselves also mentioned during the evaluation sessions that the project files can become messy, however cleaning up the project files is not always the highest priority. Therefore, we still think it is good to include this item in the refined checklist to keep the project files clean.

Additional, we added whether the of values in setting and configuration files are correct. These types of values are in practice changed for debugging purposes, for example usage of localhost. However, in the comment data analysis we observed that these values are sometimes forgotten to be changed back and had to be corrected based on the feedback in code reviews.

The items of this group could also be part of the next group as these are often forgotten and the next group consists solely of items that are often forgotten. However, we thought it would be more appropriate to place the items together in this group to allow the reviewers to easily skip over the group, thereby increasing the structure and overview of the checklist.

Do not forgettable

The last group is do not forgettable this are items that are often forgotten during programming and code reviewing. Forgetting these items can lead to unnecessary delays or additional pull requests that could have

been prevented. The first item we already encountered in the previous checklist and is whether the version numbers are upgraded. The next two items were suggested by the teams during the evaluation sessions, these are: whether *ExcludeFromCodeCoverage* is used for generated files and whether contracts are edited that affects other teams who need to be informed. Placing *ExcludeFromCodeCoverage* in the generated files excludes them from the analysis by the code quality tools. When a contract is changed and another team using the contract is not notified of the change this can lead to unexpected breakage of the program. This item was suggested by one the teams during the evaluation as they experienced the unexpected breakage in practice due to a changed contract. The final item is copied from the previous checklist and describes whether an additional review is needed from someone with more expertise.

Summary: To increase the structure of the checklist in terms of readability and to improve the applicability of the checklist and therefore also more structure in the review process we have refined the previously used checklist. In general we introduced 6 groups containing the items and these items are optionally accompanied with explaining sentences. The explaining sentences provides to the reviewer additional information related to the corresponding item. The reviewers can now mark the groups quickly as checked and when needed as items separately. When the reviewer is uncertain regarding an item she can read the explaining sentences for more information.

7.3.1. REFINED CHECKLIST

In this section the refined checklist is shown as explained in the previous section.

GENERAL SET-UP

- Does the pull request contain the correct solution for the problem?
  - Does the pull request meet the requirements in the PBI?
  - The solution does not contain additional changes which are not described in the PBI?
  - Is the new functionality implemented at the correct layer and place?
- Is the solution clear and understandable?
  - Is the solution easy to follow and to understand?
  - Is there an (partly) easier solution?

CODING GUIDELINES

- Is the code readable, understandable and maintainable?
  - Does the code comply with the coding principles: KISS, SOLID, DRY, YAGNI?
  - Are the used names clear and according to the guidelines?
  - Are there best practices, tricks or design patters that can be used to increase the readability, understandability or the maintainability? (see also the Wiki page)
  - Does the code contain no anti patters which negatively influence the readability, understandability or the maintainability? (see also the Wiki page)
- Is the code neat and according to Greenchoice's coding guidelines?
  - Does the code no longer contain debug code?
  - Are the code quality tools used correctly? (Green mark for ReSharper and no Sonar warnings)
  - No commented code, unnecessary methods, unnecessary usings, correct indentation, correct accolades and brackets, white spacing, etc.

TESTING

- Are there tests cases covering the critical parts of the new code?
  - Are there enough and correctly written test cases for the functionality?
  - Are there user input or edge cases that need additional testing?
- Do the tests also comply to the coding guidelines?

LOGGING AND ERROR HANDLING

- Is the logging clear and relevant?
  - Are the log lines clear and does it not contain too much information?
  - Are the relevant values included in the log lines?
  - Are the log lines at the correct log level?
- Are exception relevant and clear?
  - Are specific exceptions used?
  - Are the exceptions thrown and catched at the correct level?
  - Is it on basis the exceptions and messages clear what went wrong?

PROJECT SET-UP

- Is the project file correct and neatly?
  - Are there no unnecessary references?
  - Is 'treat errors as warnings' activated?
- Are the values is the setting and configuration files correctly entered?
  - Are there no longer any temporary placeholders?
  - Are there no hard coded values such as localhost and passwords?

DO NOT FORGETTABLE

- Are the necessary version numbers increased?
- Do all migration files contain 'ExludeFromCodeCoverage'?
- Are there contracts edited which also affect other teams and are they informed?
- Is there an additional review required from someone with more expertise?

7.4. POST QUESTIONNAIRE

In the second period of using the checklist for two weeks the refined checklist was used, shown in Section 7.3.1 by two teams consisting of in total 6 developers and a tester. After the second period a post questionnaire was sent out to be able to compare the initial thoughts and expectations of the pre questionnaire, of which the results are shown in Section 7.1. The post questionnaire was sent only to the developers of the teams as the testers, once more, did not use the checklist regularly. This leaves 6 developers for the questionnaire, however one developer went on vacation during the second period and was not able to fill in the questionnaire. This results in a total of 5 respondents for the post questionnaire. This is fever than half compared to the 11 respondents in the pre questionnaire. This means that we have to be aware that these 5 respondents represents only a smaller part of the expectations and they might had deviating expectations.

The post questionnaire is structured in three sections in which in the first section we explore the general experience of the experiment and positive and negative aspects of using a checklist during code reviews. In the second section we look further into the actual usage of the checklist. Hereby we also show the actual results to the expectations in the pre questionnaire. The differences between the expectations in the pre questionnaire and the actual results will be further discussed in Section 7.5. Lastly, in the third section the respondents could leave their final remark regarding the experiment.

7.4.1. GENERAL EXPERIENCE OF THE EXPERIMENT

The first section consist of 3 questions of which the first question contains 6 statements. The statements focus on the general experience of the experiment. The other two questions asks the respondents to give at least a negative and positive aspects as on open question. Using open answer fields we wanted to encourage the respondents to think for themselves instead of us already providing predefined answers.

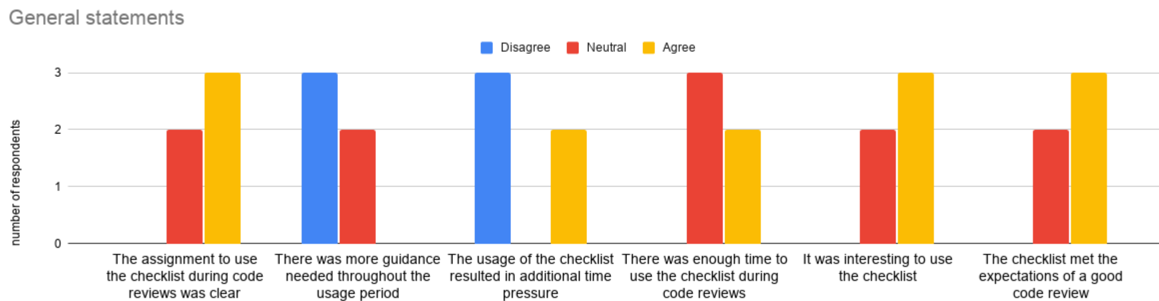


Figure 7.12: General statements regarding the experiment of using the checklist during code reviews.

GENERAL STATEMENTS

To start the questionnaire 6 general statements were given regarding the general experience of the experiment of using the checklists during code reviews. The statements and their corresponding answers are shown in Figure 7.12. In the first two statement we gauged whether the experiment itself was clear to the developers. In both statements 3 of out 5 respondents thought the assignment was clear and no further explanation was needed. In both cases the remaining 2 respondents were neutral, based on the answers this were different respondents for each statement.

The third and fourth statements covered whether the checklist caused additional time pressure and if there was enough time to use the checklist. The third statement, additional time pressure, was answered with 3 disagreeing and 2 agreeing votes. This means some respondents did experienced additional time pressure, however others did not. Additionally, in the next statement 2 respondents agreed that there was enough time available to use the checklist and the other 3 respondents were neutral. We conclude from this that the usage of the checklists has caused additional time pressure, however it was still manageable for the reviewers.

The last two statements were answered both with 2 neutral and 3 agreeing answers. In the first of the last two statements we looked into whether the usage of the checklists was interesting and in the second statement whether the checklists met the expectations for a good code review.

POSITIVE AND NEGATIVE ASPECTS

In the next two questions we asked the respondents in an open question for a positive and negative aspect related to using the checklists during code reviews. The given answers of each respondent are shown in Table 7.2. The positive feedback we could distinguish two aspects: using the checklist as reference work and the checklist provides a clear overview of point to check on during code reviews. The first aspect is given by respondent 1 and hidden in the negative answer of respondent 3. Hereby the checklist functions as reference work to either fall back on during discussions or to look up points of attention. The second positive aspect was given by respondents 2-5, whereby respondents 2-4 mention the clear overview and respondent 5 mentioned the insights in the points of attention during code reviews.

The negative aspects are contradicting each other on the one hand respondent 4 mentioned the checklist should be more extensive to fully cover all the different types of changes. On the other hand the checklist should be shorter and more to the point to the corresponding pull request according to respondents 1-3 and 5.

7.4.2. REVIEW EXPERIENCE

The second section contains 5 questions focusing on the substantive part of the checklist and on the reflection of the pre questionnaire. The first question look into the actual usage frequency of the checklist as reviewer and author. The second question contains 8 statements regarding the usage of the checklist as reviewer. Whereby the third question contains 4 statements regarding the usage of the checklist as author. The fourth question checks for each general category whether actually less, equal or more comments were given with the usage of the checklist according to the perception of the respondents. The last question checks whether or not the respondents think they will use the checklist in the future on for which purpose.

USAGE FREQUENCY

In this question we were interested in the actual usage of the checklist during code review by the respondents. In the pre questionnaire we looked into the expected usage of the checklist where we found the expected

Resp	Positive aspect	Negative aspect
1	Something to fall, together, back on during discussions	It is a bit long and detailed
2	The grouping of the checks allows for an easy lookup, without reading the whole checklist	The coding guidelines block contains a fair amount of text/content/bullets, resulting in that I skip or read it less accurate. Compared to the other blocks on the checklist.
3	It gives a nice overview of points you can check on.	There are too many items, which are not always valid. I find it difficult to use as a checklist. I use it more as a reference for a list of focus points/points of attention.
4	Structure to cover all important points of code quality and error prevention	Checklist is not really applicable to SQL or (very) small PRs
5	Using the checklist made me aware of what is important looking at a pull request	It is difficult to indicate the need for the checklist. Sometimes a pull request is too small for the entire checklist and then you forget it. You might want to oblige it with certain size of a pull request

Table 7.2: The results of a positive and negative aspect regarding the usage of the checklist according the each respondent (resp).

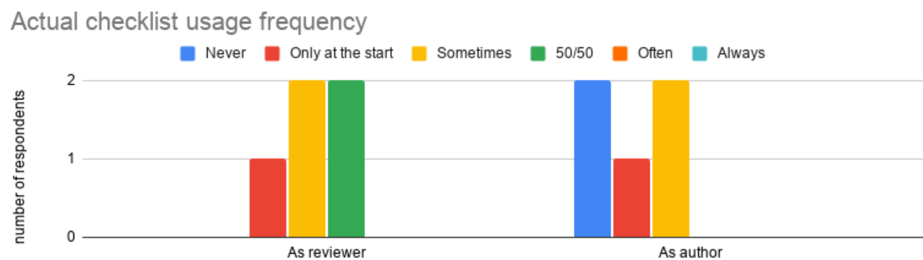


Figure 7.13: Actual usage frequency of the checklist by the respondents.

usage as reviewer would either be *only at the start* or *often* and as author it divided over all answer except *fifty/fifty*. In Figure 7.13 we show the actual usage frequency as reviewer and author according the respondent. First to notice is that the actual usage as reviewer was higher than as author and the usage of the checklist was in general lower than the expectation which will be further discussed in Section 7.5.2.

REVIEWER STATEMENTS

In the second question we gave the respondents 8 statements regarding the usage of the checklist as reviewer. The given statements and their corresponding results are shown in Figure 7.14. In the first statement we were interested whether the introduction of the checklist led to an different approach in code reviewing. Of the given answers 3 respondents agreed to a change in behaviour and a single respondent voted on both disagree and neutral. In the second statement we gauged whether or not the usage of the checklist resulted in better code reviews. The given answers were divided with a single disagree, 2 neutral and 2 agreeing answers. In the third statement with 3 agreeing answers the respondents think they gave more feedback in general, in the fifth question of this section we will look in more detail into the increase in the given code review comments. In the fourth statement we discovered that the checklist has a positive effect on the do not forgettable items with 4 agreeing answers, the amount of do not forgettable items were specifically increased during the refinement of the checklist.

The next two statements are similar divided with a single disagree and agree answer, and 3 neutral answers. In the statements we looked whether the introduction of the checklists gave more structure to the review process and made code reviews more complete. Based on the answers there is no clear indication this was the case.

In the seventh statement the result shows that the usage of the checklist was experienced more as a wake-up call over a real change in behavior with 4 agreeing answers. In the last statement we gauged whether or not

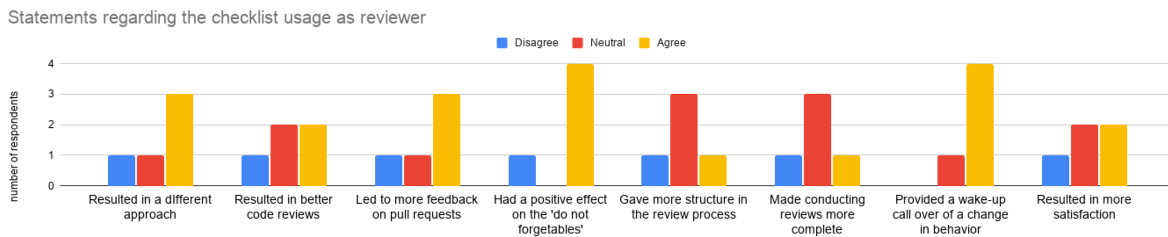


Figure 7.14: Statement results regarding the usage of the checklist as reviewer.

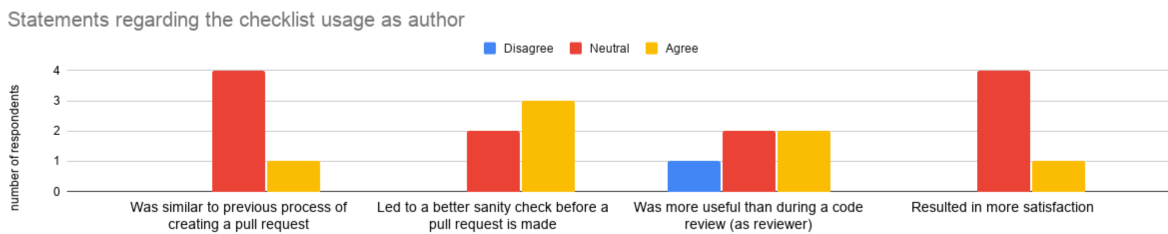


Figure 7.15: Statement results regarding the usage of the checklist as author.

the satisfaction is increased. The given answers are similarly divided to the second statement with a single disagree and 2 neutral and 2 agreeing answers.

Based on the given answers to the statements we conclude that the introduction of the checklist led to an different code review approach and resulted in more feedback. However, the different approach did not result in a change in behaviour and the introduction of the checklist served earlier as a wake up call. Additionally, the introduction of the checklist has had a positive effect on the items that are often forgotten during programming and reviewing, in the refined checklist mainly covered by the do not forgettable group. Unfortunately the checklist did not provide clearly an increase in the structure and and completeness of code reviews as we intended to provide with the checklist.

AUTHOR STATEMENTS

In the third question of this section we gave the respondents 4 statements on the usage of the checklist as the author of the pull request. The given statements and their results are shown in Figure 7.15. In the first statement we gauged whether the creation of a pull request went in a similar way as before. Only a single respondent agreed and the other 4 respondents answered neutral. The second statement gauged whether the checklist increased the quality of the sanity check before the creation of the pull request or shortly after the creation of the pull request. The results shows 3 respondents agreed and 2 respondents answered neutral. In the third statement we were interested whether the usage of a checklist as author is considered more useful than as reviewer. We added this statement based on chats with the developers at the work place mentioning the checklist was especially useful to them in the role as author. In the third statement we got a single disagreeing answer, and 2 neutral and agreeing answers. The last statement had a single agreeing answer and the other answers were neutral. The statement questioned whether the checklist led to increased satisfaction in the role as author of a pull requests.

Based on these answer we conclude the checklist increased the quality of the sanity check. Furthermore, the respondents were divided whether the checklist is more useful during the sanity rather than a code review.

FUTURE USAGE

In the second to last question we asked whether or not the respondents intent to use the checklist in the future and for which purpose. The question was originally a multiple select question with two given answers: *when reviewing pull requests* and *when preparing pull requests*. Furthermore, the respondent could leave their own optional answer, whereby 3 respondents wrote their own optional answer. The answers for each respondents are shown in Table 7.3. The first respondent checked both given answers and additionally wrote an own answer. The second and third respondents only picked one of the given answers and fourth and fifth respondents wrote their own answer. Notice, the numbers used for the respondents in Table 7.3 are different to the numbers used in Table 7.2.

Resp	I will use the checklist in future ...
1	When reviewing pull requests and preparing pull requests. To fall back on in discussions / as a reminder
2	When reviewing pull requests
3	When preparing pull requests
4	During large (unclear) pull requests
5	As a reference work, as a reminder, not as a checklist

Table 7.3: Reactions of the respondents (resp) whether they will use the checklist in future usage.

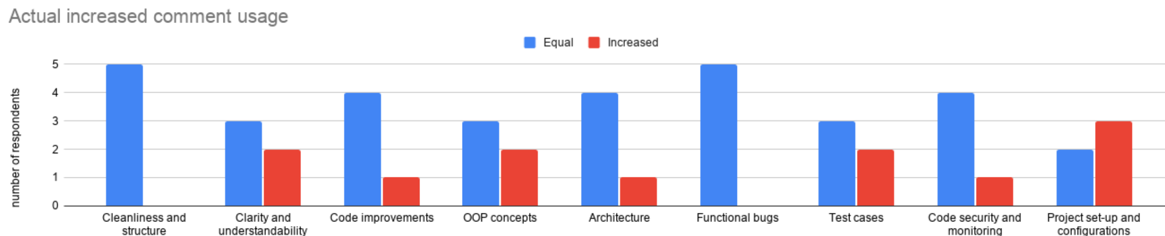


Figure 7.16: The actual increase in comments used for each category according to the respondents.

The fifth answer is rather interesting as this indicates that the checklist is more seen as a reference work than a real checklist. This is also slightly indicated by the first answer as it will be used only during exceptional situations as a large pull request. This answer also indicates that the checklist is not applicable to all pull requests, especially smaller requests.

INCREASED FEEDBACK

In the last question we looked further into whether the comment frequency increased for each of the 9 aspects according to the respondents shown in Figure 7.16. In the pre questionnaire show in Section 7.1.5 we found the respondents expected the comment usage would mainly increase for *code improvements*, *OOP concepts* and *test cases*. In Section 7.5.4 the differences between the expectations and the actual perceived increase will be discussed in detail.

First off, luckily none of the aspects decreased in frequency and the option was therefore left out of the graph. The category with the most increases is surprisingly *project set-up and configurations* of which 3 of the 5 respondents thought their comment usage increased. In the pre questionnaire only 2 of the 11 respondents thought the checklist would increase the comment frequency for this aspect. The following aspects had all 2 respondents who thought their comment usage increased: *clarity and understandability*, *OOP concepts* and *test cases*. Next, for the categories: *code improvements*, *architecture* and *code security and monitoring* a single respondent thought his comment usage increased. For the aspects: *Cleanliness and structure* and *functional bugs* no increase was mentioned.

7.4.3. FINAL REMARKS

Finally, the questionnaire was completed with an open answer field for optional final remarks. In this case we got a single response of: *"The checklist has certainly made me a lot more aware of how we should do code reviews!"*. It is always great to receive this kind of positive feedback.

> **Summary:** In the post questionnaire we looked into the experiences of the respondents in the usage of the checklist in the code review process. First we explored the general thoughts towards the general set-up of the experiment. Subsequently, we showed for each respondent a positive and a negative aspect related to the usage of the checklist in practice. In the latter section of the questionnaire we looked into actual experiences of the usage of the checklist which we are compared in the discussion to the expectations in the pre questionnaire.

7.5. DISCUSSION

In this section we will discuss the results we presented throughout the chapter regarding the experiment of a pre post test design. The experiment started with a pre questionnaire gauging the expectations regarding the usage of the final checklist, shown in Section 6.4, during the code review process. The pre questionnaire was sent to the developers and testers of three scrum teams with a total of 11 respondents. The final checklist was then used for a period of two weeks by the respondents.

Subsequently, we held with each scrum team a meeting to evaluate the usage of the checklist so far. Based on the evaluation sessions we decided to refine the checklist, presented in Section 7.3.1, before the checklist was used for a second period of two weeks. Once the second usage period ended a post questionnaire was sent to the participants to gauge the actual perceived experience in the usage of the checklist during code reviews.

During one of the evaluation sessions a scrum team decided to stop in the experiment as it took too much effort and they rather focus on their upcoming deadlines. This resulted in that we lost 4 participants for the post questionnaire. In addition, the testers were barely involved in any pull request at all and thereby in the experiment of using the checklist. We believe based on the limited participation in code review process of the testers, including the testers in the experiment was too early and requires a better introduction of the testers into the code review process. This resulted in an additional participant of the pre questionnaire who did not participate in the post questionnaire. Together with another developer who was on vacation at the time we sent out the post questionnaire we lacked in total 6 respondents in the post questionnaire compared to the pre questionnaire. The lack of the respondents made it harder to evaluate the results of the pre and post questionnaires. Especially as the respondents of the post questionnaire only represents a slight minority, 5 of the 11 respondents, of the expectations given in the pre questionnaire.

The rest of the discussion is structured as following. In Section 7.5.1 we first discuss the results regarding the general experience of the experiment. Next, in Section 7.5.2 we compare the expected usage frequency to the actual usage frequency of the checklist. Subsequently, we discuss the code review comment usage within DevOps in Section 7.5.3. In addition, in Section 7.5.4 we discuss the expected and actual perceived increase in the amount of review comments given for the different aspects in the code development process. Next, in Section 7.5.5 we discuss in detail the limitations regarding the structure and the applicability of the checklist in the code review process.

7.5.1. USAGE OF THE CHECKLISTS IN GENERAL

In this section we discuss the findings of the respondents related to the usage of a checklist and the experiment in general. In the pre questionnaire we showed that the respondents expect that checklists are easy and pleasant to use. Furthermore, checklists provide structure on knowing what to do and increase the precision, in other words preventing that tasks are forgotten. Though, they also expected using a checklist requires additional time. We also showed in the results of the pre questionnaire that checklists are regularly used by the respondents. We did not find a clear difference between the usage frequency of the respondents within Greenchoice or outside Greenchoice, for example their free time. The discovered difference lies within the usage load, in other words the satisfaction regarding the amount of checklists used. The usage load within Greenchoice is perceived as few to good and outside Greenchoice as good. Based on these results we expect the introduction of an additional checklist during code reviews will not overwhelm the developers.

In the post questionnaire we explored the experience of the respondents of actually using a checklist in practice during code reviews. We showed that the experiment, using the checklist during code reviews, was clear and no additional guidance was needed. Some respondents experienced the additional time pressure which was discovered in the pre questionnaire as others did not, still every respondent had enough time to use the checklist. Furthermore, the respondents thought it was interesting to use the checklist and the items in the checklist met the expectations for a good code review. However, the checklist did not provide the structure and precision compared to the expectations given in the pre questionnaire. Although our initial goal of using a checklist during code reviews was to provide an extensive list of items to guide the reviewer through the process, the created checklists turned out to be short lists of items that were deemed important of which most items are often forgotten during a regular code review. These shorter checklists do not cover all the steps required in the a code review providing less structure and precision. In addition, the listing of the items in the first checklist was unclear to the developers which we improved on in the second checklist. Furthermore, the checklists were limited applicable to the different types of pull requests and therefore it was hard for the developers to determine when it was suitable to use the checklist during a code review. The limited applicability of the checklists is discussed further on in more detail in Section 7.5.5.

7.5.2. Usage frequency

We have also explored in the pre and post questionnaire the expected and actual usage frequency of the checklists during the code reviews according to the respondents. Hereby we looked into the usage of the checklist in the role as reviewer and as author of the pull request. In the role as reviewer the expectations were mostly divided over *only at the start* and *often*, however in practice the actual usage was divided over *sometimes* and *50/50*. In the role as author the expectations were divided over most of the options with the most answers in *often*, however in the post questionnaire the answers on the actual usage were given between *never* to *sometimes*.

This means that in both roles the actual usage of the checklists was lower than the expected usage. Examining the answers of the five respondents that participated in the post questionnaire their expected usage in the pre questionnaire was already at the lower hand. The exact reasoning for the already lower expectations is unclear to us. In addition, we believe the lower usage can be explained by the limited applicability of the checklist, which is further explained in Section 7.5.5. We base this reasoning on the feedback we received during the evaluation sessions with the teams. In here the developers mentioned they skipped the usage of the checklist in code reviews in which the checklist was hard to use or not applicable. Furthermore, we can also imagine that once the items in the checklist are familiar to the developers, it is no longer necessary to specifically look up the items in the checklist.

7.5.3. Code review comment usage

During the start of our research we observed that code review comments are also given face to face. Throughout our research we examined the different thoughts of Greenchoice's developers towards the usage of code review comments in DevOps and giving feedback face to face. In the results of the pre questionnaire we showed that the review comment in DevOps are used as reminders to process the given feedback as author of the pull request. Additionally, the review comments can be used by the reviewer as reminders to discuss the feedback face to face, however, this occurs to a lesser extend in practice.

The importance of reminding which issues to solve in a pull request is given in the following situation we witnessed. The feedback on a pull request was given face to face addressing two separate issues with multiple occurrences. Once the first issue was addressed the author started to resolve the occurrences of the first issue, however, once the second issue was addressed the author immediately switched to resolve the occurrences of the second issue. Once all the occurrences of the second issue were resolved and a new commit was made. However, the remaining occurrences of the first issue were forgotten and the reviewer had to readdress these occurrences and an additional iteration was required to resolve both issues.

Furthermore, we validated in the pre questionnaire that code review comments in DevOps provides awareness to other developers regarding the discussion. The respondents were divided whether or not the review comments in DevOps could be used for historical needs. In the discussion of the interviews in Section 5.4.1 we showed that the literature [72, 74] uses review comments to identify and predict error prone areas of code. However, Tao *et al.* [70] discovered that the developers they interviewed do not require tools to expose these error prone areas of code. In chats with Greenchoice's developers, partly in the interviews, some mentioned that they are using the review comments to retrace bugs, however, others also mentioned this does not occur that often in practice. Based on the review comments placed within DevOps we could perform the data analysis performed in Chapter 5. Furthermore, we were able to extract several checklist items which are often addressed during the code reviews.

In the pre questionnaire we also gauged whether the introduction of the checklist would result in more review comments. Additionally, whether the respondents are more likely to place review comments as there is something to fall back on. In other words whether the checklist removes the uncertainty to place a review comment and the fear of becoming the bogeyman we discovered in the interviews described in Section 5.1.3. In the pre questionnaire we showed that review comments are left out which later causes issues during programming. However, we were unfortunately unable to determine whether this can be related to the uncertainty to place a review comment or the fear of becoming the bogeyman due to unclear statements. We therefore consider it as future work to look further into the relationship between these factors.

7.5.4. Increase in types of comments used

According to the post questionnaire a general increase in code review comments is perceived by the respondents, especially for the *do not forgettable* items. Furthermore we explored in the questionnaires for the 9 software development aspects separately the expected and actual increase in the amount of comments given. Similar to the expected usage frequency of the checklist the post questionnaire has only 5 respondents

and represents only a smaller part of the respondents and the 5 respondents could already have diverging expectations.

In the pre questionnaire the aspects *code improvement* and *OOP concepts* had the highest expected increase. In the post questionnaire the actual perceived increase turned out to be slightly lower than the expectations. However, based on the expectations of the 5 respondents in the pre questionnaire both aspects had 1 less increase in the actual perceived increase. Our initial goal of creating the checklist was to specifically improve on these aspects, however during the creation of the checklist in Chapter 6 we narrowed down most of the specific checklist items related to these aspects. In the discussion in Section 6.5.2 we therefore already lowered our expectations to improve on these aspects.

The respondents did not perceive an increase in the amount of review comments for the aspects *cleanliness and structure* and *functional bugs*. In the pre questionnaire only a single respondent of the 5 respondents expected an increase in *cleanliness and structure* and none for *functional bugs*. The experienced no increase for *functional bugs* is not surprisingly as we did not include specific items related to increase the amount of functional bugs discovered. In addition, the expectations of the 11 respondents in the pre questionnaire were already low. This contradicts the original idea of Fagan [8] to use a checklist to assist the developers in finding functional bugs. However, we furthermore believe that to increase the amount of functional bugs discovered, the developers should spend additional time assessing the pull requests and pull the changes locally to be able to run the change. Once the developers start pulling the changes locally during code reviews the checklist could be adjusted to include items related to running the code locally and discovering functional bugs.

The lack of increase for *cleanliness and structure* is somewhat unexpected and in the pre questionnaire the expected increase was around average. In addition, several items were included in the checklist related to this aspect and especially in the refined checklist. However, these items are also rather trivial and already addressed often in code reviews. In addition, the ReSharper tool can already fix most of the issues related to this aspect.

In addition, the 5 respondents of the post questionnaire expected no increase for *clarity and understanability*, however 2 of the 5 respondents experienced an increase in the usage. This is more surprisingly as less attention is paid to this aspect in the checklist. Furthermore in the comment data analysis described in Section 5.3 we discovered a relative high amount of code review comments are already given related to this aspect.

For the next aspect *test cases* 2 out of the 5 respondents perceived an increase in comment usage, similar 2 out of the 5 respondents also expected an increase. In the first checklist we included a single item related to testing and in the refined checklist we included a specific group dedicated to testing. In the interviews explained in Section 5.1.3 we discovered that during code reviews the check on test cases is often forgotten. The increase in the amount of comments reflects the attention paid to the testing aspect. In addition, Greenchoice held a workshop related to testing, however, this workshop was separate from our research or code reviews at all.

In the pre questionnaire the expectations for *code security and monitoring* and *project set-up and configurations* were in general low. In the post questionnaire 1 of the 5 respondents perceived an increase for *code security and monitoring* which is inline with single respondent of the 5 respondents in the post questionnaire who expected an increase. Furthermore, the expected and actual perceived increase reflects the limited attention paid to this aspect in the checklist.

In case of *project set-up and configurations* it is the only aspect of which the actual usage increased compared to the expectations. In the first checklist we included two items regarding the project file which were even found redundant in the evaluation sessions and the item to upgrade the necessary version numbers. In the second checklist we extended the amount of items related to this aspect, for example the *ExcludeFromCodeCoverage* items in the *do not forgettable* group. We did expect an increase in the usage of this aspect as we included several items related to this aspect, however we and also the respondents did not expect the increase to be this high. We also assume this emphasizes the strength of the *do not forgettable* group in the checklist as most of these items are related to *project set-up and configurations*.

7.5.5. STRUCTURE AND APPLICABILITY OF THE CHECKLIST

In the evaluation sessions we discovered that in general the first checklist lacked structure to be able to easily use the checklist. In addition, the checklist was not applicable to all the different types of pull requests, for this the items are either too profound, the items are not applicable or the items are too obvious for the specific code review. In the open question to list a positive and negative in the post questionnaire we retrieved similar

feedback. Chong *et al.* [46] discovered that students made some mistakes in creating a code review checklist of which the first two common mistakes were unclear and irrelevant questions. We did not look into detail into the clearness of each individual item. However, based on the items that were found not applicable in the checklist we also discovered that relevance of the items are important and made several mistakes.

In terms of structure the respondents desire to check off all items in the checklist. Skipping over items that are not applicable was found annoying by the developers. In the refined checklist we tried to improve on this by grouping the items. The groups of items can now easily be checked off as a whole instead of each item individually. However, the items can still be checked off separately if necessary. Additionally, we introduced optional explaining sentences linked to the items in a group. The items which were experienced rather obviously but also valuable to be mentioned were rewritten into the explaining sentences. Furthermore, the explaining sentences provided additional depth to the items. Receiving positive feedback on the grouping of the items for the refined checklist we think we successfully improved the structure of the refined checklist.

The applicability of the checklist is somewhat related to the structure in terms of length and extensiveness of the checklist. The checklist was experienced lengthy, especially the coding guidelines group in the refined checklist was experienced extensive. Due to the perceived extensiveness of the checklist, the checklist was less applicable to smaller pull requests. However, we also discovered contradicting the extensiveness of the checklist that the checklist lacked items corresponding to specific changes in pull requests. In the evaluation sessions we already received the feedback that the checklist was barely applicable for code reviews related to creating projects. Although we did include a few items regarding this project set-up, it remained very limited. In the post questionnaire it was furthermore mentioned that the checklist did not cover SQL scripts. Similar Halling *et al.* [42] discovered that a checklist does not suit to every role in a team, however, we showed this also holds for the different types of pull requests.

Due to the preference of a short checklist we did not include items related to project creation and SQL scripts. Additionally, we wanted to specifically focus on C# coding files as different programming languages require their own checklist [39]. The focus on c# files excludes the inclusion of SQL related items and for project creation Greenchoice has several Wiki pages. Adding these subjects in the checklist is not optimal as the checklist is already experienced as lengthy.

The coding guidelines group was especially found to be extensive. In the creation of the checklist described in Chapter 6 we already heavily reduced the amount of items related to the coding guidelines. We conclude from this that there is no need by the developers to list specific examples of coding guidelines in the checklist. Furthermore, tools like ReSharper and SonarQube already detect issues related to the coding guidelines. There is no need to be nitpicking over the smallest details and the rules should be flexible [8]. Greiler [44] describes in her blog to automate what can be automated during code reviews and having clear coding style guides is important, however a code review is not the place to discuss the guidelines.

As positive feedback we received that the checklist gave insight into how code reviews could be performed. However, the checklist suits a better role as reference work to look into once having a discussion. In case of reference work the checklist can be extended further including the missing subjects. Based on this reference work a checklist could be created and maintained covering the most important aspects in the reference work at that moment. Furthermore, using a smaller checklist existing of mostly *do not forgettable* related items could be useful during code reviews as a reminder for the most important aspects which are often forgotten.

RQ4 How do developers experience the usage of a checklist during code reviews?: A checklist should not be used as an extensive list of items to direct the code review process. The checklist itself should consist of items which are considered hard in a code review or are often forgotten. These items complement the general knowledge of the reviewers during their code reviews. A longer list of items could be used as a reference work to inspire the reviewers in their code review process. Furthermore, the items in the checklist should be applicable to most pull requests, however, this can be a challenge as different types of changes require different items.

8
Conclusion

In this chapter we conclude our findings that we made throughout this thesis. In Section 8.1 we collect the answers to the introduced research questions in Section 1.3. The research questions were already answered in the discussions of their corresponding chapters. In Section 8.2 we present our recommendations to Greenchoice to improve its current code review process based on our findings. Furthermore, in Section 8.3 we describe interesting follow-up research based on our findings or additional research cases which we could not cover in this study. Finally, in Section 8.4 we describe the possible threats to validity in our research.

8.1. Answers to the research questions

Throughout this thesis we explored the motivation of the code review process at Greenchoice and analysed Greenchoice's current code review process. To analyse the current code review process we held 13 interviews with the developers and created a tool to crawl the data of 8454 pull requests and 5400 code review comments which we later analysed. Furthermore, we created together with a scrum team and input from the architects a checklist which was used for four weeks in two periods of two weeks by respectively 3 and 2 scrum teams. Based on the feedback received regarding the usage of the checklist we created a prototype of a dynamic checklist generator.

This work was all done to answer the research questions introduced at the start of the thesis in Section 1.3. Throughout this thesis we have included for each research question an answer, in the upcoming sections summarize the answers to the research questions, as listed below. In Section 8.1.1 the first research question will be answered, next in Section 8.1.2 the second research question is answered through answering its three sub research questions. Subsequently, in Section 8.1.3 we will answer the third research questions and in Section 8.1.4 the fourth research question.

RQ 1 **What is Greenchoice's motivation behind code reviews?**

RQ 2 **What does the current code review process of Greenchoice look like?**

- ***RQ 2.1*** **What does the current code review process of Greenchoice look like in the perspective of the developers?**
- ***RQ 2.2*** **What does the current code review process of Greenchoice look like according to the metadata related to the code reviews?**
- ***RQ 2.3*** **What types of comments are found in the current code review process?**

RQ 3 **Which factors should be included in a well designed checklist for code reviews?**

RQ 4 **How do developers experience the usage of a checklist during code reviews?**

8.1.1. Motivation for code reviews

In the first research question we looked into the motivation of code reviews at Greenchoice. To discover Greenchoice's motivation for code reviews we sent out a questionnaire to all employees of the software department in which we asked to list their top 3 motivations ranked in priority. In addition, the respondents

could give another 5 optional motivations without any prioritization. The given motivation answers were then grouped into a total of 11 categories. These categories were ranked by 3 equations each of the equations taking either the amount of answers, the quantitative aspect, or the rank of the answers, the priority aspect, or a combination of both aspects into account.

After the evaluation of the answered motivations we derived the following answer to **RQ 1 (What is Greenchoice's motivation behind code reviews?)**: The *quality* category is the most important motivation for code reviews at Greenchoice. The *quality* category is a very broad definition and can be narrowed down based on the given answers into ensuring the code quality in code reviews. The *quality* category is closely followed by *verifying behaviour* and *knowledge sharing*, the latter category lacks higher priority answers. Grouping the categories to three global categories *code quality* remains the most important motivation followed closely by *functional* and *social*, the latter lacks, once more, higher priority answers.

8.1.2. CURRENT CODE REVIEW PROCESS

In the second research question we gained an insight into the actual code review process at Greenchoice. The second research question was answered via three sub questions, for which we interviewed in the first sub question in total 13 developers from 7 different scrum teams. In the second and third sub questions we performed two separate data analysis. The first data analysis covered the meta data regarding the pull requests and code reviews, and the second analysis explored the types of code review comments given. The data of both analyses was gathered by a self made crawler which retrieved the data via the DevOps API and was further analysed in R studio.

WHAT DID WE DISCOVER IN THE INTERVIEWS?

We held 13 semi-structured interviews based on the three questions listed below. In the interviews the interviewee were furthermore free to discuss any code review related subject. In addition, we were able to ask in-depth question in order to get a better understanding.

1. How does the current code review process look like, both in reviewer and coder perspective?
2. What are the strengths and weaknesses of the code review process?
3. What can be improved in the code review process and how?

Based on the discussions in the interviews we derived the following answer to **RQ 2.1 (What does the current code review process of Greenchoice look like in the perspective of the developers?)**: The code review process starts by the author of the pull request giving a head-ups to his fellow team members. In addition, the author performs a sanity check for the correctness and completeness of the change just before or right after the creation of the pull request. The reviewers tend to focus on understanding the change and finding obvious mistakes. They mentioned code reviews could be performed stricter improving on the maintainability and evolvability aspects of the change. However, it can be hard to know where to check on for these aspects. The feedback could be either given synchronous face to face and asynchronous in DevOps. The effect of the different methods of providing feedback requires future work. In general the code review process at Greenchoice looks similar to the current modern code review processes described in the literature, however having its own strengths and facing its own challenges.

META DATA ANALYSIS

In the metadata analysis we analyzed 33 data features of 8454 pull requests whereby we mainly focused on the time, size and user interaction aspects. The different features were analysed and we calculated the Spearman correlation between the different features to discover possible positive or negative relationships. In which we were especially interested in the relationship between the time and size features.

Based on the metadata analysis we derived the following answer to **RQ 2.2 (What does the current code review process of Greenchoice look like according to the metadata related to the code reviews?)**: The code review process of Greenchoice is according to the metadata is a rather quick process compared to the literature with a median CycleTime of 25 minutes. In terms of size the process is on the smaller side compared to the literature with a median of 28 lines of code changed. In addition, we discovered moderate correlations between CycleTime and NumTotalChangedLines and NumTotalChangedFiles. This indicates there is a positive relationship between the size and time features. Furthermore, we discovered on average 1.62 reviewers and 1.02 approvers for a pull request, and for the pull requests with a user comment an average 4.63 comments per pull request. In terms of correlation we discovered mostly weak to moderate correlations to the time and size features.

COMMENT DATA ANALYSIS

In the code review comment data analysis we manually classified 5100 comments from the 8454 pull requests over 11 global categories. We looked into the types of code review comments used during code reviews. Based on the results we gained the following answer to **RQ 2.3 (What types of comments are found in the current code review process?)**: In the code review comment data analysis of 5100 comments we found most, 33.22%, comments are related to understanding and explaining the change. Second is *other communication*, 17.94%, consisting of mainly *"fixed"* messages acknowledging the comment giving feedback is processed. Other categories we discovered are: *cleanliness and structure* (13.98%), *project set-up and settings* (10.49%), *code improvements* (9.22%) and *OOP concepts* (4.37%).

8.1.3. CREATION OF THE CODE REVIEW CHECKLIST

In the third and fourth research questions we explored the creation and usage of a code review checklist. In the third research we focused on creating the checklist and which factors are deemed important and which unnecessary. The checklist was created together with a scrum team to explore to needs of the scrum team. The created checklist is shown in Appendix G and we retrieved the following answer to **RQ 3 (Which factors should be included in a well designed checklist for code reviews?)**: The first factor to be considered is that a rather small checklist is preferred over a longer extensive list of items. This limits the number of specific items that can be included in the checklist. The factors that should be included are implementation in terms of understandability and readability together with the SOLID principles. In general the coding guidelines supported by the correct usage of the tooling, which are SonarQube and ReSharper. Furthermore, the naming of variables, logging, set-up of the project files, updating the version numbers and a second review from an expert were considered important and therefore included in the checklist. The latter three factors are often forgotten during a code review. In short a compact checklist is desired emphasizing the important factors that are often forgotten during a code review.

8.1.4. USAGE OF THE CODE REVIEW CHECKLIST

The created checklist was used by 3 scrum teams in the first period of two weeks. After this period we held with each team an evaluation session and based on the feedback we refined the checklist, the refined checklist is shown in Appendix H, which was then used by 2 scrum teams for a second period of 2 weeks. To get an understanding of the effect of using the checklist we used a quasi experimental pre post test design. To evaluate the effect of using the code review checklist we sent out a pre and post questionnaire. Furthermore, the evaluation sessions with the different scrum teams provided us with additional insights.

Based on the observation we derived the following answer to **RQ 4 (How do developers experience the usage of a checklist during code reviews?)**: A checklist should not be used as an extensive list of items to direct the code review process. The checklist itself should consist of items which are considered hard in a code review or are often forgotten. These items complement the general knowledge of the reviewers during their code reviews. A longer list of items could be used as a reference work to inspire the reviewers in their code review process. Furthermore, the items in the checklist should be applicable to most pull requests, however, this can be a challenge as different types of changes require different items.

8.2. RECOMMENDATIONS TO GREENCHOICE

In this section we present our recommendations to Greenchoice for their code review process and the usage of code review checklists in general. In order to improve Greenchoice's code review process we recommend to evaluate regularly, at least yearly, their current code review process. In the evaluation, the strengths and weaknesses of the process should be addressed and determine whether adjustments are needed. A first step in the improvement of the code review process could be to look in detail in the difficulties encountered by the developers in assessing the code reviews in terms of code quality, in other words performing stricter code reviews. In our study we have discovered that the developers experience these difficulties, however, we were not able to clearly identify these difficulties and the reasoning for them. We retrieved indications that it could be related to unclear coding guidelines, not knowing what to look for in code reviews as reviewer, or uncertainty or lack in knowledge.

In the assessment of the code quality in code reviews the usage of tools could be used and these tools should be preferably be integrated in the code process in DevOps. In a recent chat with an architect we were notified that the Sonar warnings are included in the code review process in DevOps. The Sonar warnings are even together with a minimum test coverage enforced as gatekeeper in the code review process, this means

that a pull request cannot be merged with open Sonar warnings or a too low test coverage. However, the tools do not detect all defects in the code and manual code reviews remain necessary to detect the remaining defects. In the manual code reviews the use of checklist could be used. The used checklist should be adjusted to mainly include items which are often forgotten, which are important at that moment and which could not be addressed with tooling. Some examples are given in the refined checklist of *updating the necessary version numbers* and included *ExcludeFromCodeCoverage*, or performance issues with LINQ queries which was mentioned in the chat with the architect. The downside of these items is that these items become quickly too specific and the number of items too extensive. Therefore the usage of the dynamic checklist generator, which is discussed in the future work in Section 8.3.1, could be considered.

8.3. FUTURE WORK

Throughout our study we encountered several findings that required further study or we gained new ideas based on our results which could not address in this thesis. In this section we will summarize the discovered future work throughout this thesis.

First off, we discovered Greenchoice uses both synchronous face to face reviews and asynchronous reviews through DevOps. However, we did not look into the distribution of the synchronous and asynchronous reviews and what the differences are in terms of results. In future work the distribution of the synchronous and asynchronous reviews could be explored. Furthermore, the difference results of synchronous and asynchronous reviews could be explored. For example, whether there are different types of defects discovered and the quality of the reviews.

During the data analysis we questioned whether the reviews are not performed too quickly and more time should be used to perform a more in depth review. Furthermore, we questioned whether a more in depth review or a stricter review would increase the number of defects found and the severity of the defects. In other words, whether investing more time in the code review process also outweighs the costs. In future work the severity and costs of the defects detected could be analysed. Furthermore, a separate or follow-up study could be to report the defects detected post release and check whether code reviews could have prevented these defects in stricter code reviews.

Based on the usage of the checklist and perceived improvements in the number of comments by Greenchoice's developers we discovered that the checklist performed well with items that are often forgotten during code reviews. Additional, the checklist did not perform well for the intended specific coding guidelines. We consider it as future work to test whether these findings also holds in other projects and companies and whether new findings could be discovered which we did not encounter.

8.3.1. DYNAMIC CHECKLIST GENERATOR

The usage of the code review checklist resulted in contradicting feedback of an too extensive checklist and also missing specific items for the different pull requests. A solution to this could be to create multiple smaller checklists for the different types of pull requests. However, this opposes the challenges to select the correct checklist for the specific pull request. In addition, multiple checklists could be required for a single pull request. To solve the latter challenges we created a prototype of a dynamic checklist generator, this tool generates a code review checklist based on the changes in the pull request. The tool currently uses the items in the refined checklist as database, however, in theory it is possible to create an extensive list of checklist items in the database. This is highly recommended as the current used items is still a limited set of items and should be extended to increase the effectiveness of the tool. For example, items related to the creation of a project and SQL scripts could be included as mentioned in the feedback. Additionally, the algorithm used to determine the checklist items could be refined to better predict which items a relevant. For example, besides the file types in the pull request the actual changed lines of code could be analysed in addition.

The dynamic checklist generator provides flexibility and applicability to the code review checklist. The advantage of using this tool is that the generated checklists can include rather specific items without increasing the standard length compared to a static checklist. The generated checklists are furthermore more applicable to the different types of pull requests. In other words, a smaller checklist is generated for a small and trivial pull request, however, a longer more specific checklist is generated for a larger pull request.

We were unfortunately only able to give several demos to a few developers and we were not able to perform a proper analysis of the usage of the tool due to limited time. However, the preliminary results on the demos were promising. The general feedback was positive in terms of possible applications of the tool. We leave therefor the analysis on the effectiveness of the tool as future work. Furthermore, the integration of the

tool in DevOps is also left as future work.

In addition, Gonçalves *et al.* [45] propose a study of which the results are currently unknown on the effect in terms of functional defects share and functional defects found over the review time of the different types of guidance used during a code review. They did research the following three methods: no guidance, a checklist and strategy. The latter is similar to a checklist based review strategy, however, it only shows the relevant parts to the piece of code that is currently reviewed. In other words this strategy is rather similar to our approach with the dynamic checklist generator.

8.4. THREATS TO VALIDITY

Our study also has its threats to validity and these will be explained further in this section.

First off, the research is performed at a single company and whereby the exploration of the motivation for code reviews, getting insight in the actual code review process and the data analyses were performed over the full software development department. The research being conducted in a single company exposes the threat to external validity that these finding are company specific and a similar study at other companies could result in different results.

Additional, we interviewed only a part of the developers exposing the risk that interviewing a different group of developers would result in different findings. However, we minimized the possibility of different findings by interviewing developers from different scrum teams. The inclusion of different developers from different teams minimizes the risk of specific findings related to a single scrum team being uncovered. Furthermore, the checklist was created together with a single scrum team and was evaluated initially by 3 scrum teams and later 2 scrum teams. This also means that these findings can be different to other teams within the company and for the teams in other companies. However, we do also recommend based on our own findings and the literature [8, 69] that a checklist should be created based on the needs of the project and preferably of the team.

In the set-up of our study we have also discovered some threats to construct validity. We noticed throughout our study that feedback of the code reviews was also given through face to face communication which was not considered during the set-up of our study. This means that the results of the comment data analysis is only based on the comments given in DevOps as the comments given through face to face communication are not documented.

Next, the majority of questions used in the pre and post questionnaires were closed questions, especially in the pre questionnaire we included questions to verify earlier findings. However, when the results to these findings turned out different than expected we could not explain these results as we lacked the reasoning of the respondents. This could be solved by adding (optional) open questions to the questionnaire to extract the reasoning. Furthermore, we could other methods such as an interview to extract the reasoning behind the choices. However, this exposes a treat to the internal validity as the results of the questionnaires were evaluated much later in time compared to the distribution of the questionnaires. This resulted in that we were no longer active at Greenchoice and we were not able to address our questions to the responsible developers.

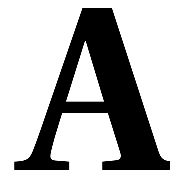
Finally, we have the internal threats to validity in which the internal process of the study affects the results. In which the most difficult internal validity is better known as the Hawthorne effect [103], which means as long as extra attention is paid to a process the process will improve. However, The question is whether the process itself is actually improved or the people involved are stepping up as they are observed. We were the first, since the adoption of the code review process at Greenchoice, who actively looked in detail into the process. The used evaluation of the checklist is quasi experimental design, this indicates that no control group was used and different factors besides the usage of the checklist could also be at play affecting the results. Furthermore we discovered that the usage of the code review checklist served as a wake-up call, this might indicate that the developers themselves also paid more attention to the code review process and thereby improving the process.

In the interviews we were not able to address each subject in every interview. This means we could have missed out the opinions of the interviewees in the interviews we did not discuss the subjects. Furthermore, we did not record the interviews and made a summary of each interview as a reference work, this exposes the risk of leaving out minor details that were later considered important.

Next, the code review comment data analysis itself should also be interpreted with a caution as we did not clearly define the categories up front and the categories created based on the comments were not used consistently.

Lastly, in terms of the analysis of the usage of the code review checklist the usage period of the checklist

was shorter than intended with a total of 4 weeks. We intended to use the checklist at least for 8 weeks, however due to the Christmas and new year holidays and rescheduling of the sprint planning by Greenchoice we missed out 2 sprints. The additional 4 weeks of usage could provide additional results on the usage of the checklist and also expose long term effects.



Motivation questionnaire

A.1. Demographic questions

The first part of the questionnaire consists of demographic questions to get started and getting to know the respondent.

1. What is your function?
   (a) Business consultant
   (b) Developer
   (c) Product owner
   (d) Tester
   (e) Other...
2. How long do you have been familiar with the term "code review"?
   (a) This is the first time
   (b) less than a year
   (c) 1 - 2 years
   (d) 3 - 6 years
   (e) 7 - 10 years
   (f) longer than 10 years
3. Was Greenchoice the first place were you came into contact with code reviews?
   (a) Yes
   (b) No
4. How often are you on average involved with code reviews per sprint?
   (a) Never
   (b) 1 - 3 times
   (c) 4 - 6 times
   (d) 7 - 9 times
   (e) 10 times or more
5. In which role are you involved at code reviews? (multiple answers are possible)
   (a) Creator (I wrote the code and created the pull request)
   (b) Reviewer (I reviewed the pull request)
   (c) Reader (I looked at the pull request, did not contribute to the pull request)
   (d) Other...

A.2. MOTIVATION FOR CODE REVIEWS

The second part consist of the main question, what is the motivation for code reviews. Each respondent had the option to give 3 motivations and had to rank them by priority.

1. Briefly give 3 different motivations why code reviews take place in your view. Prioritize the answers. Assume a general situation where all aspects are considered.

   Feel as free as possible giving the motivations and try to be as specific as possible.

   On the next page there is an option for additional motivations, these do not need to be prioritized.

   (a) Priority 1
   (b) Priority 2
   (c) Priority 3

A.3. MOTIVATION FOR CODE REVIEWS (ADDITIONAL ANSWERS)

Third part was optional to give up to 5 additional motivations. These did not need to be prioritized.

1. Extra possibility to give motivations

   (a) Extra option
   (b) Extra option
   (c) Extra option
   (d) Extra option
   (e) Extra option

A.4. CODE REVIEW STATEMENTS

The fourth part consisted of 10 statements. The respondents had to give their opinion for the code reviews statements. Additional the respondent had the option to leave a remark for each statement. The respondent could choose one from the following answers.

- Strongly disagree
- Disagree
- Agree
- Strongly agree
- Not applicable

THE FOLLOWING 10 STATEMENT WERE GIVEN

1. I find code reviews an important factor in software development.
2. I think the effect of code reviews is visible.
3. I think enough attention is paid to code reviews.
4. I think I have enough time to perform code reviews.
5. I enjoy performing code reviews.
6. I find it difficult to perform code reviews.
7. I think I give useful feedback during code reviews.
8. I enjoy receiving feedback through code reviews.
9. I find it difficult to receive feedback through code reviews.
10. I think I get useful feedback through code reviews.

A.5. CODE REVIEWS IN PRACTICE

The fifth section contains of two questions to see the practical effect of code reviews.

1. Can you remember a situation where a bug or code smell was prevented via a code review?
    (a) Yes
    (b) No
    (c) Not applicable
2. If yes, give an example of a bug or code smell that was prevented via a code review.
    (a) Open answer field.

A.6. FINAL REMARKS

In the final section there was a possibility to leave the e-mail address, this way we could contact the respondent if we had any questions. Also there was the option to leave final remarks, questions or suggestions about the questionnaire.

B

Motivations for code reviews

B.1. 4 eyes principle

Priority 1

- Check colleague
- Four eyes principle, quality control
- To get feedback from my colleagues
- To prevent that the auditor has to 'mark his own paper'

Priority 2

- Any security vulnerabilities introduced should normally be noted in this four-eyes principle
- Control / traceability (no code that can / may be implemented unmonitored)
- Integrity check

Priority 3

- 4 Eyes principle
- 4 Eyes principle
- 4 Eyes principle
- Four eyes principle
- Traceability

Additional

- Increase safety

B.2. According to architecture

Priority 2

- Assess intent
- Code that complies with team / architecture agreements
- Compliance with architecture and coding guidelines
- Enforcement of architecture
- Equivalent design of the code

Priority 3

- Does code comply with architecture/guidelines
- Is the solution in line with the architecture, if not, we coordinate it or make it right

B.3. Consistency between teams

Priority 1

- Standardization

Additional

- Consistent codebase
- Ensuring that the team develops in the same way

B.4. Discussing solution

Priority 1

- There is a dialogue about the quality of the solution, does it comply with the guidelines, was a simpler, more manageable way possible, why is this the best. We learn from that dialogue

Priority 2

- Dialogue about choices that have been made
- To create consensus on the end result

Priority 3

- Exchange ideas
- Solution direction (other insight)

Additional

- Share perspectives

B.5. Following code guidelines

Priority 2

- Have it tested whether the written code meets the coding guidelines
- Prevent magic numbers
- To follow the standards

Priority 3

- According to guidelines
- Check code conventions, recognize technical debt
- Check coding guidelines

- Does the code comply with certain rules / agreements
- Enforcement coding convention
- Not too much code

ADDITIONAL

- Avoid duplicate code
- SOLID

B.6. FUNCTIONALITY

PRIORITY 1

- Check correctness of the code
- Check whether the code meets all agreements (both architecture and coding guidelines and naming conventions)
- Code passes accept criteria in PBIs
- Does the code do what it should do
- Submit sound and working code
- The changed code does what is supposed to do –> the required functionality is delivered
- Whether it is functionally correct

PRIORITY 2

- Check whether the code seems functionally do what has been described
- Code meets general requirements
- Completeness
- Control algorithm (edge cases, error handling, etc)
- Correct implementation
- Does the code do what is expected?
- Static testing
- To check whether the code does what we expect it should do (functional)

B.7. KNOWLEDGE SHARING

PRIORITY 1

- To learn from each other in the way we write code

PRIORITY 2

- Knowledge sharing
- Knowledge transfer
- Knowledge transfer / knowledge sharing
- Learn from each other
- Sharing knowledge

PRIORITY 3

- Knowledge sharing
- Knowledge sharing
- Knowledge sharing
- Knowledge sharing within the team
- Learn
- learn from each other
- Learn from how other members of the team resolve issues
- Transmission of knowledge

ADDITIONAL

- Improving coding knowledge
- Learn from your mistakes (as a developer)
- Learning from other people's work (as a reviewer)
- Stimulate each other to become even better
- To learn from each other

B.8. PREVENTING DEFECTS

PRIORITY 1

- Check on presence of bugs in code
- Prevent (production) issues
- Prevent technical/function design flaws
- Quality control (bugs can be found early)
- To check whether there are any errors in the code

PRIORITY 2

- Sharpness of a developer can sometimes be weakened by distractions, by reviewing you prevent problems earlier than during testing

PRIORITY 3

- Bug prevention
- Discovery of possible bugs in work
- To prevent technical and / or logical errors

ADDITIONAL

- Discovering (new) technical debt

B.9. QUALITY

PRIORITY 1

- Check code quality, standardization and consistency
- Check intention of code, check naming, structure, clarity, testability, complexity
- Code quality
- Guarantee code quality
- Guarantee code quality
- Improve code quality
- Improve quality
- Quality
- Quality control
- Quality control
- Quality control
- Quality control/increase
- Review of quality

PRIORITY 2

- Guarantee code quality
- Quality check
- Quality of code
- The code is of good quality

PRIORITY 3

- To guarantee the quality of the code (code guidelines, architecture, standards etc.)

B.10. TEAM AWARENESS AND OWNERSHIP

PRIORITY 2

- Be aware of each other / knowledge sharing
- Knowledge transfer of software under ownership of the team
- Notification colleagues
- Stay informed / knowledge sharing

PRIORITY 3

- [Making] Changes in codebase insightful for teammates who did not work on it
- Fellow developers understand what is coded for product ownership
- Knowledge sharing. By having the code reviewed by a peer, he gains insight into the code that someone else has written and can maintain it in this way.
- The team is responsible for the code, not the individual -> the team feels the ownership of the code

B.11. UNDERSTANDABLE AND MAINTAINABLE CODE

PRIORITY 1

- Clearly
- Code must be neatly written and assessed by others so that another person also understands what it says and what is meant and its purpose
- Coding style (layout, readability, typos, etc.)
- Readability of the code, by another developer

ADDITIONAL

- Is code easy to maintain

C

PRE QUESTIONNAIRE

C.1. INTRODUCTION

Besides a short introduction of the questionnaire the respondent was asked to leave his or her name behind. This way we could compare the results with the according post questionnaire results.

C.2. PERSONAL INTEREST

The first real question gauged the personal affection of 8 mostly code or a single checklist related aspects. The question was as following formulated: How active are you personally in applying and improving the following aspects. Are you aware of the latest features and how actively do you try to apply them? Interpret the scale according to your own interpretation where 1 is the least and 5 the most.

THE FOLLOWING 8 ASPECTS WERE GIVEN

1. As developer in general
2. Coding guidelines in Greenchoice
3. Language specific features of the programming languages
4. Code quality in general
5. Object-Oriented Programming (OOP) concepts
6. Code architecture
7. Code reviews
8. Usage of checklists

C.3. ATTITUDE AGAINST CODE QUALITY

The third section consisted of two question to clarify what code quality is.

1. The first question consisted of 6 statements were the respondent could either answer: disagree, neutral or agree.

   ### THE FOLLOWING STATEMENTS WERE GIVEN

   (a) Functional code is more important than qualitative code
   (b) Bugs are the result of sloppiness of the author
   (c) Messy code is due to time pressure, finishing a task in a rush
   (d) The presence of test cases is an important factor in determining the code quality

(e) During development I encounter problems due to the code quality

(f) Writing tests is harder due to the quality of the code, the code is too complex or is entangled

2. In the second question the respondent had to pick 3 categories which have the most influence on determining the code quality.

THE FOLLOWING CATEGORIES WERE GIVEN

- Cleanliness and structure
- Clarity and understandability
- Code improvements
- OOP concepts
- Architecture
- Functional bugs
- Test cases
- Code security and monitoring
- Project set-up and configurations

C.4. THOUGHTS TOWARDS CHECKLISTS

This section looks into the attitude towards the usage of checklists.

1. How often do you use a checklist ...? The options were: never, few, regular, often and always.

   (a) within Greenchoice

   (b) outside Greenchoice

2. What to do you think about the frequency of the usage of a checklist ...? The options were: too few, few, fine, often, too often

   (a) within Greenchoice

   (b) outside Greenchoice

3. The last 5 question of this section consisted of statements with 5 point Likert scale. The statements all started with: "The usage of a checklist I experience in terms of...?".

   (a) general usage as ...? Annoying to pleasant.

   (b) time spending as ...? Time consuming to time saving

   (c) ease of use as ...? Difficult to easy

   (d) structure (I know what I have to do) as ...? Unstructured to structured

   (e) precision (I do all the tasks) as ... ? Imprecise (I do not do any task) to precise (I do all tasks)

C.5. THOUGHTS TOWARDS CODE REVIEWS

This section looks into the attitude towards code reviews. We dived into what categories are useful quality control during code reviews, usefulness of writing down comments and tooling during code reviews.

1. I sometimes leave out comments during a code review, which I later encounter as issue during programming. Measured by a 5 point Likert scale from never to too often.

2. Next 5 statements are given to get an insight about using comments over face to face communication. The respondent could either disagree, be neutral or agree.

   Adding comments in DevOps during code reviews I see as ...

THE FOLLOWING STATEMENTS WERE GIVEN

(a) reminder to later discuss the points face to face

(b) reminder to process the feedback as developer

(c) traceability such that other developers are also aware what changed

(d) traceability such that there is later the possibility to review how and why the code has changed

(e) administrative task which takes more time than it yields value

3. Next two questions are 2 statements about the usage of tooling, measured once more with a 5 point Likert scale from totally disagree to totally agree.

   (a) Tools are an important method for discovering bugs

   (b) Tools are an important method for discovering code smells and technical debt

4. Do you miss specific tooling during code reviews within Greenchoice?

   (a) Yes

   (b) No

5. If yes, which tooling do you miss within Greenchoice?

   (a) Open answer field

6. Are there any tools that you would recommend to use within Greenchoice?

   (a) Yes (continue to an additional question)

   (b) No

C.6. TOOLING (OPTIONAL SECTION)

This is an additional section if the respondent has any tool(s) she or he recommends.

1. What is name of the tool(s)?

   (a) Open answer field

2. Why are you recommending the tool(s)?

   (a) Open answer field

C.7. EXPECTATIONS ABOUT THE USE OF A CHECKLIST DURING CODE REVIEWS

This section discovers the expectations of the checklist during code reviews.

1. Starting with 5 general statements on the usage during code reviews. The following answer could be given.

   - Totally disagree
   - Disagree
   - Neutral
   - Agree
   - Totally agree

THE FOLLOWING STATEMENTS WERE GIVEN

(a) The use of a checklist during code reviews results in a general improvement on the review process

(b) The use of a checklist during code reviews results in an improvement in code quality

(c) The use of a checklist costs initially more time but decreases over time

(d) The use of a checklist improves satisfaction in the role of reviewer

(e) The use of a checklist improves satisfaction in the role of developer

2. The next two question dives into the expected use frequency. The following answer options were given.

   - Never
   - Only at the beginning
   - Sometimes
   - fifty/fifty
   - Often
   - Always

   I expect as ... to use the checklist [frequency].

   (a) reviewer

   (b) author of the code

3. Followed by two more specific statements, again with the 5 point Likert scale from the first question.

   (a) I expect to write more comments during code reviews, instead of only discussing them in person.

   (b) With a checklist I dare to give feedback more often as I can fall back on the checklist and is not only based on personal opinion, there are clear guidelines.

4. The second last question uses, once more, the 9 aspects similar to the first question of the fifth section. The respondent had to answer if they expect to give more feedback on each category due to the checklist.

   I expect to give ... feedback for [category].

   - less
   - equal
   - more

5. I Have additional expectations for the usage of the checklist during code reviews, namely ...

   (a) Open answer field

C.8. REMARKS AND FEEDBACK

Lastly a thank you message was given and the option to leave their final comments behind that be could not be given elsewhere.

1. Have you any remarks or feedback left?

   (a) Open answer field

D
POST QUESTIONNAIRE

D.1. INTRODUCTION
A short introduction of the questionnaire is given. Additional the name of the respondent is asked to compare the results to the pre questionnaire.

D.2. EXPERIENCE USAGE OF CHECKLIST
In this section we dived into the experiment set-up and general usage experience of the checklist during the experiment.

1. The first question consist of 6 statements about the general set-up of the experiment on which the respondent could either disagree, be neutral or agree with.

   #### THE FOLLOWING STATEMENTS WERE GIVEN

   (a) The assignment to use the checklist during code reviews was clear
   (b) There was more guidance needed throughout the usage period
   (c) The usage of the checklist resulted in additional time pressure
   (d) It was interesting to use the checklist
   (e) The checklist met the expectations of a good code review
   (f) There was enough time to use the checklist during code reviews

2. Give at least one positive aspect on the usage of the checklist

   (a) Open answer field

3. Give at least one negative aspect on the usage of the checklist

   (a) Open answer field

D.3. REVIEW EXPERIENCE
1. The next two question dives into the actually usage frequency. The following answer options were given.

   - Never
   - Only at the beginning
   - Sometimes
   - fifty/fifty
   - Often

- Always

I have as ... used the checklist [frequency].

(a) reviewer

(b) author of the code

2. The next 8 statements are looking into the actual perceived results during code reviews from the perspective of the reviewer. The respondent could either answer disagree, neutral (NA) or agree.

The usage of the checklist as reviewer ...

THE FOLLOWING STATEMENTS WERE GIVEN

(a) resulted in a different approach of a review

(b) resulted in better code reviews

(c) led to more feedback on pull requests

(d) had a positive effect on the 'do not forgetables'

(e) gave more structure in the review process

(f) made conducting reviews more complete

(g) provided a wake-up call over of a change in behavior

(h) resulted in more satisfaction

3. This followed up by additional 4 statements, however this time in the perspective of the author of the pull request.

The usage of the checklist as author ...

THE FOLLOWING STATEMENTS WERE GIVEN

(a) was similar to the current process of creating a pull request

(b) led to a better self check before a pull request is made

(c) was more useful than during a code review (as reviewer)

(d) resulted in more satisfaction

4. The next question checks whether or not the respondents have given more feedback on the specific categories.

THE FOLLOWING CATEGORIES WERE GIVEN

- Cleanliness and structure
- Clarity and understandability
- Code improvements
- OOP concepts
- Architecture
- Functional bugs
- Test cases
- Code security and monitoring
- Project set-up and configurations

I feel like I gave ... feedback for [category]

- less
- equal

- more

5. I keep using the checklist in the following situation

    - During reviewing a pull request
    - During preparation of a pull request
    - Other, namely ...

D.4. REMARKS AND FEEDBACK

Lastly a thank you message was given and the option to leave their final comments behind that be could not be given elsewhere.

1. Have you any remarks or feedback left?

    (a) Open answer field

E

Long list of checklist items

To create the short list we have colored the items on the long list indicating whether or not they should be placed on the short list. The colors are however not the actual truth and might be changed throughout the process based on new feedback or information, Though, it is still a good indication for the short list. Green marked items will be put in the short list for creating the checklist, however it may happen that multiple green items are merged into a single item on the short list. Orange marked items are options for in the short list, but most likely rewritten, merged with other items into a general item or not considered important enough for the short list. Red marked items will be totally scrapped for the short list based on the preference of the team, either they are already covered by other items or deemed unnecessary.

Review set-up

- Is the pull request not too large?
- Is it clear what changes and why?
- Are there additional files which do not belong in the pull request?
- Is the correct merge branch selected?

Implementation

- Does the code do what it is supposed to do, does it comply to the PBI or task?
- Does the change add unwanted compile-time or run-time behaviour?
- Is there a package or library used which should not be used?
- Is there a package or library available which simplifies the code?
- Is the solution not unnecessary complex? (KISS)
- Are there parts of the functionality already used in other places which could be reused? (DRY)
- Is the new functionality implemented in the correct layer and place?
- Is there another solution available which increases the readability, understandability, maintainability, performance or improves the security?
- Does the code comply to the architecture agreements?
- Does the code comply to the coding guidelines?
- Are the necessary version numbers increased?

Coding guidelines

- Do all static analyse tools pass?
- Is the code easily readable?
- Are there parts of the code that need to be rewritten to increase the readability, think about smaller methods
- Is the naming clear and obvious, are the business terms in Dutch and rest in English?
- Is the code placed at the right place?
- Is the code flow understandable and correct?
- Are comments if needed in Dutch and the technical terms in English?
- Does every class have its own file?
- Are functional names given, thus Button1 becomes SendButton?
- No unnecessary pre or post fixes and abbreviations are used unless it is used in business terms?
- Is a data model used if a method needs more than 5 parameters?
- Are two different methods used for boolean parameters? Based on avoiding usage of booleans as parameter this mostly implicates an if else statement.
- Are decimals used instead of doubles or floats?
- Are the right access modifiers used?
- Are there no magic numbers or strings?
- Is there a need to use an enum for values?
- Are there complicated boolean expressions which should be written as a separate variable for readability?
- Are unused usings cleaned up in the code?
- Are there always brackets used for each statement, also single line statements?
- Is there no (unused) commented code in the files?
- Are unmanaged resources correctly disposed?
- Is LINQ used when possible?
- Do the comments explain why and less what happens, the latter is already in the code?

Code control flow

- Are there exceptions thrown instead of error values returned?
- Are there specific exception thrown and caught instead of general exceptions?
- Is the logging clear and complete?
- Does the code have a good performance, for example string builder is used?

Security and privacy

- Are there no new or older security vulnerabilities in the code?
- Are the authentications handled correctly?
- Is sensitive data handled securely, for example directly encrypted?
- Is the right encryption used?
- Is user data handled correctly and checked for SQL injection?
- Is the code safe for external issues, for example an outage of a cluster?

External dependencies

- Are administrative tasks regarding the pull request completed, for example updating the scrum board?
- Do other teams, either other scrum teams or business teams, need a heads-up before the merge is fulfilled?

Testing

- Is the code written keeping the testability in mind? (mocking is prevented as much as possible
- Are there enough test written covering the critical parts of the code?
- Are the tests that have been written actually correct?

F

Shortlist of checklist items

Implementation

- Is it clear what has changed in the pull request and why?
- Is the code programmed according to the specifications in the PBI? Does the code comply to the function and technical requirements?
- Is the solution not unnecessary complex? (KISS)
- Does the code not contain additional functionality which is not described in the PBI? (YAGNI)
- Are parts of the functionality already used elsewhere which could be reused? (DRY)
- Is the new functionality implemented in the correct layer and place?
- If existing functionality is affected, is that functionality still correct?
- Does the code comply to the architectural agreements?
- Are the necessary version numbers updated?

SOLID

- **Single responsibility,** has a class or method a single solely purpose?
- **Open/closed,** is the open for extension and closed for modification?
- **Liskov substitution,** are objects replaceable by its sub classes without alternating the functionality?
- **Interface segregation,** does the interfaces only contain the used necessary methods?
- **Dependency inversion,** does the code depend on abstractions instead of concrete implementations?

Coding Guidelines

- Is the code easily readable?
- Is the code easily understandable?
- Is the code neat and clean, no unnecessary comments, commented code, correct indentation, correct braces etc.
- Do the code comments describe why and not what happens in the code?
- Does the code comply to the companies coding guidelines, see the Wiki page?

Naming

- Is the naming clear and obvious, does the naming represents the actual functionality?
- Are business terms in Dutch and the rest in English?

Code flow and security

- Is the logging clear and on point?
- Are there no unnecessary public attributes which could be made private?
- Are there no unnecessary objects passed through which could alternatively be replaced by passing the values?

Testing

- Is the code written keeping the testability in mind? (mocking is prevented as much as possible)
- Are there enough test written covering the critical parts of the code?
- Are the tests that have been written actually correct?

External review

- Is there a review required from an external programmer or expert for a thorough review?

G

Final checklist

- Is the code easily readable and understandable? (also think of KISS, YAGNY, DRY)
- Is the new functionality implemented at the correct layer and place?

- **Single responsibility,** has a class or method a single solely purpose?
- **Open/closed,** is the open for extension and closed for modification?
- **Liskov substitution,** are objects replaceable by its sub classes without alternating the functionality?
- **Interface segregation,** does the interfaces only contain the used necessary methods?
- **Dependency inversion,** does the code depend on abstractions instead of concrete implementations?

- Does the code comply to the coding guidelines (see the Wiki page)
- Are all SonarQube warnings processed (if available during code review)
- Is the naming clear and obvious, does the naming represents the actual functionality?
- Are the business terms in Dutch and the rest in English?
- Is relevant information logged and at the correct log level?
- Are specific clear exceptions given and handled correctly?
- Are there enough correct tests written covering the critical parts of the code?
- Is there no unused information in the project file, are the references correct?
- Is 'treat warnings as errors' activated, do all files have a green check mark (no ReSharper warnings)?
- Are the necessary versions number updated?
- Is there an additional review from an expert or team member needed?

Refined checklist

General set-up

- Does the pull request contain the correct solution for the problem?
  - Does the pull request meet the requirements in the PBI?
  - The solution does not contain additional changes which are not described in the PBI?
  - Is the new functionality implemented at the correct layer and place?
- Is the solution clear and understandable?
  - Is the solution easy to follow and to understand?
  - Is there an (partly) easier solution?

Coding guidelines

- Is the code readable, understandable and maintainable?
  - Does the code comply with the coding principles: KISS, SOLID, DRY, YAGNI?
  - Are the used names clear and according to the guidelines?
  - Are there best practices, tricks or design patters that can be used to increase the readability, understandability or the maintainability? (see also the Wiki page)
  - Does the code contain no anti patters which negatively influence the readability, understandability or the maintainability? (see also the Wiki page)
- Is the code neat and according to Greenchoice's coding guidelines?
  - Does the code no longer contain debug code?
  - Are the code quality tools used correctly? (Green mark for ReSharper and no Sonar warnings)
  - No commented code, unnecessary methods, unnecessary usings, correct indentation, correct accolades and brackets, white spacing, etc.

Testing

- Are there tests cases covering the critical parts of the new code?
  - Are there enough and correctly written test cases for the functionality?
  - Are there user input or edge cases that need additional testing?
- Do the tests also comply to the coding guidelines?

Logging and error handling

- Is the logging clear and relevant?
  - Are the log lines clear and does it not contain too much information?
  - Are the relevant values included in the log lines?
  - Are the log lines at the correct log level?
- Are exception relevant and clear?
  - Are specific exceptions used?
  - Are the exceptions thrown and catched at the correct level?
  - Is it on basis the exceptions and messages clear what went wrong?

Project set-up

- Is the project file correct and neatly?
  - Are there no unnecessary references?
  - Is 'treat errors as warnings' activated?
- Are the values is the setting and configuration files correctly entered?
  - Are there no longer any temporary placeholders?
  - Are there no hard coded values such as localhost and passwords?

Do not forgettable

- Are the necessary version numbers increased?
- Do all migration files contain 'ExludeFromCodeCoverage'?
- Are there contracts edited which also affect other teams and are they informed?
- Is there an additional review required from someone with more expertise?

Bibliography

[1] Y. Bassil, *A simulation model for the waterfall software development life cycle,* arXiv preprint arXiv:1205.6904 (2012).

[2] S. Balaji and M. S. Murugaiyan, *Waterfall vs. v-model vs. agile: A comparative study on sdlc,* International Journal of Information Technology and Business Management **2**, 26 (2012).

[3] N. B. Ruparelia, *Software development lifecycle models,* ACM SIGSOFT Software Engineering Notes **35**, 8 (2010).

[4] L. Rising and N. S. Janoff, *The scrum software development process for small teams,* IEEE software **17**, 26 (2000).

[5] K. Beck, M. Beedle, A. Van Bennekum, A. Cockburn, W. Cunningham, M. Fowler, J. Grenning, J. Highsmith, A. Hunt, R. Jeffries, *et al.*, *Manifesto for agile software development,* (2001).

[6] G. Bavota and B. Russo, *Four eyes are better than two: On the impact of code reviews on software quality,* in *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)* (IEEE, 2015) pp. 81–90.

[7] L. Pascarella, D. Spadini, F. Palomba, and A. Bacchelli, *On the effect of code review on code smells,* arXiv preprint arXiv:1912.10098 (2019).

[8] M. Fagan, *Design and code inspections to reduce errors in program development,* IBM Systems Journal **15**, 182 (1976).

[9] A. Bacchelli and C. Bird, *Expectations, outcomes, and challenges of modern code review,* in *Proceedings of the 2013 international conference on software engineering* (IEEE Press, 2013) pp. 712–721.

[10] M. Beller, A. Bacchelli, A. Zaidman, and E. Juergens, *Modern code reviews in open-source projects: Which problems do they fix?* in *Proceedings of the 11th working conference on mining software repositories* (2014) pp. 202–211.

[11] *Cambridge dictionary - Review* ((last accessed 10 May 2021)).

[12] X. Su and T. M. Khoshgoftaar, *A survey of collaborative filtering techniques,* Advances in artificial intelligence **2009** (2009).

[13] *Netflix prize* ((last accessed 10 May 2021)).

[14] J. Bennett, S. Lanning, *et al.*, *The netflix prize,* in *Proceedings of KDD cup and workshop*, Vol. 2007 (Citeseer, 2007) p. 35.

[15] G. Linden, B. Smith, and J. York, *Amazon. com recommendations: Item-to-item collaborative filtering,* IEEE Internet computing **7**, 76 (2003).

[16] J. Webster and R. T. Watson, *Analyzing the past to prepare for the future: Writing a literature review,* MIS quarterly , xiii (2002).

[17] R. W. Wright, R. A. Brand, W. Dunn, and K. P. Spindler, *How to write a systematic review,* Clinical Orthopaedics and Related Research® **455**, 23 (2007).

[18] E. Aromataris and A. Pearson, *The systematic review: an overview,* AJN The American Journal of Nursing **114**, 53 (2014).

[19] P. Cronin, A. M. Kelly, D. Altaee, B. Foerster, M. Petrou, and B. A. Dwamena, *How to perform a systematic review and meta-analysis of diagnostic imaging studies,* Academic radiology **25**, 573 (2018).

[20] B. Kitchenham, *Procedures for performing systematic reviews,* Keele, UK, Keele University **33**, 1 (2004).

[21] B. Kitchenham, O. P. Brereton, D. Budgen, M. Turner, J. Bailey, and S. Linkman, *Systematic literature reviews in software engineering–a systematic literature review,* Information and software technology **51**, 7 (2009).

[22] G. Gousios, M. Pinzger, and A. v. Deursen, *An exploratory study of the pull-based software development model,* in *Proceedings of the 36th International Conference on Software Engineering* (2014) pp. 345–355.

[23] M. Fowler and M. Foemmel, *Continuous integration,* Thought-Works) http://www. thoughtworks. com/Continuous Integration. pdf **122**, 1 (2006).

[24] B. M. Hales and P. J. Pronovost, *The checklist—a tool for error management and performance improvement,* Journal of critical care **21**, 231 (2006).

[25] A. Gawande, *The checklist manifesto: How to get things right,* Journal of Nursing Regulation **1**, 64 (2011).

[26] D. M. Conley, S. J. Singer, L. Edmondson, W. R. Berry, and A. A. Gawande, *Effective surgical safety checklist implementation,* Journal of the American College of Surgeons **212**, 873 (2011).

[27] S. Russ, S. Rout, N. Sevdalis, K. Moorthy, A. Darzi, and C. Vincent, *Do safety checklists improve teamwork and communication in the operating room? a systematic review,* Annals of surgery **258**, 856 (2013).

[28] A. Degani and E. L. Wiener, *Cockpit checklists: Concepts, design, and use,* Human factors **35**, 345 (1993).

[29] D. Boorman, *Today's electronic checklists reduce likelihood of crew errors and help prevent mishaps,* ICAO Journal (2001).

[30] L. G. Portugal, D. R. Adams, F. M. Baroody, and N. Agrawal, *A surgical safety checklist for performing tracheotomy in patients with coronavirus disease 19,* Otolaryngology–Head and Neck Surgery **163**, 42 (2020).

[31] M. Grelat, B. Pommier, S. Portet, A. Amelot, C. Barrey, H.-A. Leroy, and R. Madkouri, *Patients with coronavirus 2019 (covid-19) and surgery: guidelines and checklist proposal,* World neurosurgery **139**, e769 (2020).

[32] E. R. Babbie, *The practice of social research* (Cengage learning, 2020).

[33] A. D. Harris, J. C. McGregor, E. N. Perencevich, J. P. Furuno, J. Zhu, D. E. Peterson, and J. Finkelstein, *The use and interpretation of quasi-experimental studies in medical informatics,* Journal of the American Medical Informatics Association **13**, 16 (2006).

[34] S. J. Stratton, *Quasi-experimental design (pre-test and post-test studies) in prehospital and disaster research,* Prehospital and disaster medicine **34**, 573 (2019).

[35] C. Sadowski, E. Söderberg, L. Church, M. Sipko, and A. Bacchelli, *Modern code review: a case study at google,* in *Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice* (2018) pp. 181–190.

[36] A. Bosu, M. Greiler, and C. Bird, *Characteristics of useful code reviews: An empirical study at microsoft,* in *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories* (IEEE, 2015) pp. 146–156.

[37] O. Kononenko, O. Baysal, and M. W. Godfrey, *Code review quality: how developers see it,* in *Proceedings of the 38th International Conference on Software Engineering* (2016) pp. 1028–1038.

[38] A. Dunsmore, M. Roper, and M. Wood, *Object-oriented inspection in the face of delocalisation,* in *Proceedings of the 22nd international conference on Software engineering* (2000) pp. 467–476.

[39] M. V. Mäntylä and C. Lassenius, *What types of defects are really discovered in code reviews?* IEEE Transactions on Software Engineering **35**, 430 (2008).

[40] P. Rigby, B. Cleary, F. Painchaud, M.-A. Storey, and D. German, *Contemporary peer review in action: Lessons from open source development,* IEEE software **29**, 56 (2012).

[41] A. Bosu and J. C. Carver, *Impact of peer code review on peer impression formation: A survey,* in *2013 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement* (IEEE, 2013) pp. 133–142.

[42] M. Halling, S. Biffl, T. Grechenig, and M. Kohle, *Using reading techniques to focus inspection performance,* in *Proceedings 27th EUROMICRO Conference. 2001: A Net Odyssey* (IEEE, 2001) pp. 248–257.

[43] L. Hatton, *Testing the value of checklists in code inspections,* IEEE software **25**, 82 (2008).

[44] M. Greiler, *A Code Review Checklist – Focus on the Important Issues* (2020, (last accessed 10 May 2021)).

[45] P. W. Gonçalves, E. Fregnan, T. Baum, K. Schneider, and A. Bacchelli, *Do explicit review strategies improve code review performance?* in *Proceedings of the 17th International Conference on Mining Software Repositories* (2020) pp. 606–610.

[46] C. Y. Chong, P. Thongtanunam, and C. Tantithamthavorn, *Assessing the students' understanding and their mistakes in code review checklists—an experience report of 1,791 code review checklist questions from 394 students,* arXiv preprint arXiv:2101.04837 (2021).

[47] H. Ibrahim, A. Ilinca, and J. Perron, *Energy storage systems—characteristics and comparisons,* Renewable and sustainable energy reviews **12**, 1221 (2008).

[48] H. Chen, T. N. Cong, W. Yang, C. Tan, Y. Li, and Y. Ding, *Progress in electrical energy storage system: A critical review,* Progress in natural science **19**, 291 (2009).

[49] D. Akinyele and R. Rayudu, *Review of energy storage technologies for sustainable power networks,* Sustainable Energy Technologies and Assessments **8**, 74 (2014).

[50] M. O. Ahmad, J. Markkula, and M. Oivo, *Kanban in software development: A systematic literature review,* in *2013 39th Euromicro conference on software engineering and advanced applications* (IEEE, 2013) pp. 9–16.

[51] V. Lenarduzzi, F. Lomio, N. Saarimäki, and D. Taibi, *Does migrating a monolithic system to microservices decrease the technical debt?* Journal of Systems and Software , 110710 (2020).

[52] *Azure DevOps Services* ((last accessed 10 May 2021)).

[53] R. C. Martin, *Design principles and design patterns,* Object Mentor **1**, 597 (2000).

[54] *SonarQube* ((last accessed 10 May 2021)).

[55] *SonarCloud* ((last accessed 10 May 2021)).

[56] *ReSharper* ((last accessed 10 May 2021)).

[57] M. Fagan, *Advances software inspections ieee transactions software engineering vol,* SE-12 (1986).

[58] F. Shull and C. Seaman, *Inspecting the history of inspections: An example of evidence-based technology diffusion,* IEEE software **25**, 88 (2008).

[59] L. G. Votta Jr, *Does every inspection need a meeting?* in *Proceedings of the 1st ACM SIGSOFT Symposium on Foundations of Software Engineering* (1993) pp. 107–114.

[60] P. C. Rigby and M.-A. Storey, *Understanding broadcast based peer review on open source software projects,* in *2011 33rd International Conference on Software Engineering (ICSE)* (IEEE, 2011) pp. 541–550.

[61] A. Sutherland and G. Venolia, *Can peer code reviews be exploited for later information needs?* in *2009 31st International Conference on Software Engineering-Companion Volume* (IEEE, 2009) pp. 259–262.

[62] A. Tsotsis, *Meet Phabricator, The Witty Code Review Tool Built Inside Facebook* (2011, (last accessed 10 May 2021)).

[63] N. Kennedy, *Google Mondrian: web-based code review and storage* (2006, (last accessed 10 May 2021)).

[64] P. C. Rigby and C. Bird, *Convergent contemporary software peer review practices,* in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering* (2013) pp. 202–212.

[65] G. Gousios, A. Zaidman, M.-A. Storey, and A. Van Deursen, *Work practices and challenges in pull-based development: The integrator's perspective,* in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering,* Vol. 1 (IEEE, 2015) pp. 358–368.

[66] G. Gousios, M.-A. Storey, and A. Bacchelli, *Work practices and challenges in pull-based development: the contributor's perspective,* in *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)* (IEEE, 2016) pp. 285–296.

[67] L. MacLeod, M. Greiler, M.-A. Storey, C. Bird, and J. Czerwonka, *Code reviewing in the trenches: Challenges and best practices,* IEEE Software **35**, 34 (2017).

[68] M. Barnett, C. Bird, J. Brunet, and S. K. Lahiri, *Helping developers help themselves: Automatic decomposition of code review changesets,* in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering,* Vol. 1 (IEEE, 2015) pp. 134–144.

[69] T. Gilb and D. Graham, *Software inspections* (Addison-Wesley Reading, Masachusetts, 1993).

[70] Y. Tao, Y. Dang, T. Xie, D. Zhang, and S. Kim, *How do software engineers understand code changes? an exploratory study in industry,* in *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering* (2012) pp. 1–11.

[71] M. Staron, J. Hansson, R. Feldt, A. Henriksson, W. Meding, S. Nilsson, and C. Höglund, *Measuring and visualizing code stability–a case study at three companies,* in *2013 Joint Conference of the 23rd International Workshop on Software Measurement and the 8th International Conference on Software Process and Product Measurement* (IEEE, 2013) pp. 191–200.

[72] P. Thongtanunam, S. McIntosh, A. E. Hassan, and H. Iida, *Investigating code review practices in defective files: An empirical study of the qt system,* in *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories* (IEEE, 2015) pp. 168–179.

[73] E. S. Raymond, *The cathedral and the bazaar,* (1997).

[74] S. McIntosh, Y. Kamei, B. Adams, and A. E. Hassan, *The impact of code review coverage and code review participation on software quality: A case study of the qt, vtk, and itk projects,* in *Proceedings of the 11th Working Conference on Mining Software Repositories* (2014) pp. 192–201.

[75] M. Ferreira, M. T. Valente, and K. Ferreira, *A comparison of three algorithms for computing truck factors,* in *2017 IEEE/ACM 25th International Conference on Program Comprehension (ICPC)* (IEEE, 2017) pp. 207–217.

[76] A. Joshi, S. Kale, S. Chandel, and D. K. Pal, *Likert scale: Explored and explained,* Current Journal of Applied Science and Technology , 396 (2015).

[77] K. Louise Barriball and A. While, *Collecting data using a semi-structured interview: a discussion paper,* Journal of advanced nursing **19**, 328 (1994).

[78] *Azure DevOps Services REST API Reference* ((last accessed 10 May 2021)).

[79] *Is there a way to get the amount of lines changed in a Pull Request via the Dev Ops Service REST API?* ((last accessed 10 May 2021)).

[80] *RStudio* ((last accessed 10 May 2021)).

[81] *Time series visualization with ggplot2* ((last accessed 10 May 2021)).

[82] N. Davis, *What Analyzing 180,000 Pull Requests Taught Us About Shipping Faster* (2018, (last accessed 10 May 2021)).

[83] E. Marshall and E. Boggis, *The statistics tutor's quick guide to commonly used statistical tests,* Statstutor Community Project , 1 (2016).

[84] *Reporting Statistics in APA Format* ((last accessed 10 May 2021)).

[85] *LINQ – Lambda Expression vs Query Expression* ((last accessed 10 May 2021)).

[86] M. Greiler, *A Code Review Checklist – Focus on the Important Issues* ((last accessed 10 May 2021)).

[87] S. R. GUTHA, *Code Review Checklist – To Perform Effective Code Reviews* (2015, (last accessed 10 May 2021)).

[88] L. Boltneva, *Code Review Checklist Infographic* (2015, (last accessed 10 May 2021)).

[89] D. Spadini, M. Aniche, M.-A. Storey, M. Bruntink, and A. Bacchelli, *When testing meets code review: Why and how developers review tests,* in *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)* (IEEE, 2018) pp. 677–687.

[90] R. C. Martin, *Clean code: a handbook of agile software craftsmanship* (Pearson Education, 2009).

[91] *Robert C. Martin quote* ((last accessed 10 May 2021)).

[92] R. Sanders, *The pareto principle: its use and abuse,* Journal of Services Marketing (1987).

[93] *SonarQube user guide: issues* ((last accessed 10 May 2021)).

[94] S. Panichella, V. Arnaoudova, M. Di Penta, and G. Antoniol, *Would static analysis tools help developers with code reviews?* in *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)* (IEEE, 2015) pp. 161–170.

[95] S. Kim and M. D. Ernst, *Which warnings should i fix first?* in *Proceedings of the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering* (2007) pp. 45–54.

[96] L. A. Williams and R. R. Kessler, *All i really need to know about pair programming i learned in kindergarten,* Communications of the ACM **43**, 108 (2000).

[97] L. Williams, R. R. Kessler, W. Cunningham, and R. Jeffries, *Strengthening the case for pair programming,* IEEE software **17**, 19 (2000).

[98] L. Williams, *Integrating pair programming into a software development process,* in *Proceedings 14th Conference on Software Engineering Education and Training.'In search of a software engineering profession'(Cat. No. PR01059)* (IEEE, 2001) pp. 27–36.

[99] M. Greiler, *Code Reviews at Google are lightweight and fast* ((last accessed 10 May 2021)).

[100] A. L. Ferreira, R. J. Machado, J. G. Silva, R. F. Batista, L. Costa, and M. C. Paulk, *An approach to improving software inspections performance,* in *2010 IEEE International Conference on Software Maintenance* (IEEE, 2010) pp. 1–8.

[101] *Visual Studio - Why should I remove unused references?* (2015, (last accessed 10 May 2021)).

[102] M. Greiler, *Proven Code Review Best Practices from Microsoft* ((last accessed 10 May 2021)).

[103] J. McCambridge, J. Witton, and D. R. Elbourne, *Systematic review of the hawthorne effect: new concepts are needed to study research participation effects,* Journal of clinical epidemiology **67**, 267 (2014).