FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Improving the Software Quality Assurance Process in Academic Software Development with Gamification and Continuous Feedback Techniques

Diplomarbeit

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering and Internet Computing

eingereicht von

Georg Ernst Moser
Matrikelnummer 1026578

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuer: Thomas Grechenig
Mitwirkung: Andreas Mauczka

Wien, 22. Februar 2019

(Unterschrift Verfasser) (Unterschrift Betreuer)

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.ac.at

Improving the Software Quality Assurance Process in Academic Software Development with Gamification and Continuous Feedback Techniques

submitted in partial fulfillment of the requirements for the degree of

in

Software Engineering and Internet Computing

by

Georg Ernst Moser
Registration Number 1026578

elaborated at the
Institute for Information Systems Engineering
Research Group for Industrial Software
to the Faculty of Informatics
at the Vienna University of Technology

Advisor: Thomas Grechenig
Assistance: Andreas Mauczka

Vienna, February 22, 2019

Statement by Author

Georg Ernst Moser
Marktstraße 9, 7521 Eberau

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

I hereby declare that I am the sole author of this thesis, that I have completely indicated all sources and help used, and that all parts of this work – including tables, maps and figures – if taken from other works or from the internet, whether copied literally or by sense, have been labelled including a citation of the source.

(Place, Date) (Signature of Author)

Kurzfassung

Zum Themenbereich Softwarequalitätssicherung in akademischen Softwareprojekten wurden bisher nur wenige wissenschaftliche Arbeiten publiziert. Auch wenn agile Entwicklungsmethoden im Kontext von akademischen Studenten-Softwareprojekten in Publikationen beschrieben wurden, lag der Fokus selten auf Qualitätssicherung. Ansätze von Gamification im Bereich der Qualitätssicherung werden in der bestehenden Literatur ebensowenig beschrieben.

In der vorliegenden Arbeit wurden für ein typisches Studentenprojekt geeignete Methoden der Qualitätssicherung, ausgearbeitet, wenn möglich mit Konzepten der Gamification angereichert und anschließend evaluiert. Als Fallbeispiel wurde ein Kooperationsprojekt zwischen der Montanuniversität Leoben und der Forschungsgruppe INSO der Technischen Universität Wien, das sogenannte Mineralbay-Projekt ausgewählt. Im Rahmen der Arbeit wurde das komplexe Gebiet der Qualitätssicherung in 3 Kategorien zu unterteilen: Entwicklungsprozess, Entwicklungstätigkeit und Dokumentation.

Zunächst wurde eine genaue Analyse des Qualitätssicherungsprozesses, dessen Eigenheiten und insbesondere Schwächen durchgeführt. Mittels Umfragen und Interviews wurden die Ansichten der einzelnen Projektbeteiligten in Bezug auf Qualitätssicherung erhoben, um potentielle Verbesserungen auszuarbeiten und in Forschungsfragen zu formulieren. Weiters wurden die relevanten Aufgabenstellungen im Bereich der Qualitätssicherung durch Elemente der Gamification attraktiver zu machen.

Um den Ergebnissen der Arbeit allgemeine Gültigkeit und Übertragbarkeit zu geben, wurde Mineralbay nach Expertenbefragungen mit Leitern akademischer Studenten-Softwareprojekte als stellvertretendes, typisches Studentenprojekt gewertet, sodass die Ergebnisse der vorliegenden Arbeit auch auf andere Studentenprojekte übertragbar sein sollten.

Die erarbeiteten Anpassungen am Entwicklungsprozess wurden im Rahmen eines Experiments evaluiert, um in weiterer Folge die Forschungsfragen beantworten zu können. Die Resultate der mittels Interviews und Fragebögen gewonnenen Einsichten wurden in Form eines Leitfadens für Qualitätssicherung in zukünftigen Studentenprojekten zur Verfügung gestellt.

Keywords

Qualitätssicherung, Gamification, Agile Softwareentwicklung, Studenten Experimente, Informatik Ausbildung

Abstract

Software Quality Assurance (SQA) in academic software development has not been subject to a substantial amount of research so far. While subjects like agile practices in academic software projects have been covered in detail, the focus was never placed on the development process as a whole. While gamification has been applied to various aspects of software development, research on applications in the field of SQA are scarce.

Throughout this thesis, the author evaluated several practices aimed at improving SQA in a case study. For the purpose of making Quality Assurance (QA) related tasks more appealing, the author used concepts of gamification. Representing a typical student software project, the Mineralbay project, started by the Montanuniversität Leoben in cooperation with the research group Industrial Software (INSO) of the Vienna University of Technology (TU), built the underlying case. The goals of the thesis were formulated as several research questions dividing the complex subject of quality assurance into the levels of process, coding and documentation.

Initially, a thorough analysis of the development process as a whole was performed, highlighting deficits and bad smells in terms of SQA. Interviews and surveys were held, gaining insights on the developers view of SQA in the given setup. The result was a multitude of possible improvements which was elaborated and subsequently expressed as a list of research questions.

In order for the results to gain general validity, the representativeness of the underlying case was assured by expert interviews, where domain experts identified Mineralbay as a typical student software project.

The proposed measures were evaluated and data was collected using interviews, participant observation as well as surveys.

The gained results suggested that high software quality in student projects is a product of multiple factors. An easy to understand, working and non-frustrating build process forms a solid basis. A profound set of tests should be executed on each change, preventing regression. An unreliable or hard to maintain build process will not be followed. A tailored review process (on code, but also configuration basis) supported by easy to use tools has been identified as another key factor of quality assurance, as well as for knowledge transfer. Introducing code style guidelines will help for better readability and maintainability. Integrated Development Environment (IDE), integrated tools for checking code compliance and auto correction are important for developers in this context. Static code analysis will help uncovering bad code, improve code quality as well as developer knowledge. Gamification has furthermore been identified as a way of making otherwise unrewarding quality assurance tasks more attractive. The elaborated and evaluated gamification tools did increase motivation to attend these kind of tasks.

Based on the gathered conclusions, a list of suggestions on how to implement quality assurance in a typical student software project was compiled.

Keywords

Software quality, gamification, agile software development, student experiments, computer science education

Contents

List of Figures

List of Tables

List of Abbreviations

API Advanced Programming Interface

BDD Behavior Driven Development

CD Continuous Delivery

CI Continuous Integration

CSS Cascading Style Sheets

DACH Germany, Austria, Switzerland

DB Database

DevOps Development And Operations

DoD Definition of Done

IDE Integrated Development Environment

IID iterative and incremental development

INSO research group Industrial Software

JS JavaScript

LOC Lines of Code

MV* Model View Whatever

NPM Node Package Manager

OOP Object Oriented Programming

QA Quality Assurance

SCM Source Control Management

SE Software Engineering

SQA Software Quality Assurance

TDD Test Driven Development

TU Vienna University of Technology

VCS Version Control System

VoIP Voice over IP

WAR Web Archive

1 Introduction

1.1 Motivation

In a software project, the delivery of high quality software is essential for both the developers (the software company) as well as the customer. Low quality software is usually not only hard to maintain but in addition results in loss of trust and bad reputation for the developers, whereas the customer potentially faces high costs. Software engineering programs at university aim at providing the necessary education for their students to graduate with solid skillset, needed to produce high quality software. While equipping students with the essential theoretical foundation to build upon, more practical courses usually set itself the target of preparing them for their future profession as engineers. This includes urging students to learn how to use new technologies fast. However, such courses usually address problems only in the scope of manageable sized exercises, which are custom tailored for allowing the students to complete them within a few weeks. Although perfectly suitable for strengthening technical skills, they usually fail to educate participants in the task of proper software engineering processes (programming-in-the-large) [104].

Adding the possibility to participate in long running projects to the curriculum, enables students to take part in a development process closer to real industry projects. Unlike the usual university project of modes size, proper development processes, combined with the necessary toolchain and infrastructure form a crucial part.

In most university courses, software quality of the practical exercises is not an issue since the code is written, has to work in order to get a grade and is most times abandoned afterwards. Long running projects on the other hand require students to write maintainable code, especially since there is a fluctuation of participating students, which confronts new members with the task of working with existing code.

This thesis will provide a case study on the analysis of existing and introduction of new QA measures in an agile development process in the scope of a (cross) university project (of the long running type), developed by students solely.

Consulting the literature, work in this field is hard to find. While research in the field of student projects has been done, the focus was usually set on experiencing with agile methods, enhancing the learning experience or similar, but not QA. In the field of QA on the other hand, the focus in related work was usually set on industry projects.

In general, long running student projects have not frequently been subject to research. Hence future projects will strongly benefit from a refined QA process, tailored for exactly this kind of project.

1.2 Scenario Description

The goal of the underlying software project is to solve a problem in the tunnel driving domain:

When driving a tunnel, loads of raw-material is excavated from the ground. These are called tunnel spoil and are composed from a set of different kinds of primary raw materials. Of course their proper disposal has to be guaranteed by the tunnel driving company, since these materials are declared as waste by several Austrian laws [88] [89] [87]. Hence tunnel construction companies are tasked to analyze these materials and deliver them to the correct waste disposal [28] [27]. However what disposal centers usually do is prepare these raw materials as products and sell them on.

Summarized, tunnel driving companies have to pay for the disposal and removal of excavation materials, although they could sell them on directly, making a (potential) profit.

Mineralbay therefore aims at showing that the direct reuse of tunnel spoil would have economic and ecological benefits. To tackle this problem a web platform will be built enabling tunnel driving companies, to offer their excavation material online. Possible customers can place a search and find the materials fitting their needs on Mineralbay and are given the opportunity of obtaining them directly from the excavation site. This search will take parameters like geographic location and suitability into account. Hence Mineralbay will provide a platform for construction companies to find buyers for their tunnel spoil and vice versa.

The idea for this platform was brought to life by the Montanuniversität Leoben who sought contact with the TU as a partner for doing the actual implementation, thus creating the inter-university project Mineralbay. As a result of this cooperation, the software will be developed by students of the TU who have been given the opportunity to participate on the project in the scope of a practical course. The domain knowledge necessary for building the online platform will be provided by the Montanuniversität Leoben.

1.3 Aim of the Work

Since student software projects differ widely from regular industry projects (in terms of setup, infrastructure, developer availability, etc.) establishing a feasible QA process bears special difficulties. The overall goal of this thesis is to analyze the existing QA measures in the development process of the Mineralbay student software project and introduce new ones. Due to the characteristics of a student developed software, fine tuning of a QA process proves a challenging task.

In a first step the status quo of the process will be determined and possible weaknesses identified. The gained information will build the foundation for future improvements, covering subjects like reviews, software testing, automated tests, build process, gamification in QA, etc. The potential improvements will be compiled in a set of research questions, dividing the subject of QA in three levels: process, coding and documentation. Afterwards, these questions will be addressed in the scope of a controlled experiment, altering the development process by introducing the elaborated QA measures. Data will be collected during and after the experiment, for the purpose of further analyses.

The outcome of this work will be a deeper insight in the current process as well as a blueprint QA process for future projects to come.

1.4 Structure of the Work

The structure of this thesis is made up by the following sections:

- *Fundamentals:* Cover the theoretical frame for the study
- *Problem description:* Introduces the case and states the research questions.
- *The Case Study:* Describes the conducted case study.

- *Results:* Presents the methods of data collection as well as the collected results.
- *Discussion:* Critically analyzes the gained results and relates them to the research questions asked.

2 Fundamentals

This chapters introduces all theoretical concepts which build the foundation of this thesis. It starts with introducing the principles of empirical studies and case studies in particular. Furthermore basics in the field of quality, software quality and QA are covered as well as how to measure quality and Gamification fundamentals.

2.1 Empirical Studies

Empirical research is based on observed phenomena and acquiring knowledge from real experience rather than from theory:

> An empirical study is really just a test that compares what we believe to what we observe [81, p. 1]
>
> An empirical study is a defined, scientific, method for posing research questions, collecting data, analyzing the data, and presenting the results [82, p. 1]

Since this thesis is based on an empirical study, reasoning why this type of study is suitable for software engineering problems will be provided:

2.1.1 In Software Engineering

Empirical studies have become an important part of software engineering research and practice. [105, p. 1]

Software engineering is an intensely people-oriented activity and hence it does not lend itself to analytical approaches. It involves real people working in real environments and hence is very much governed by human behavior through the people developing software [117, p. 6].

While all software processes aim at producing software, the underlying process, the tools and technology used and the rationale for doing so varies widely [81, p. 3] Accordingly to understand software engineering, one should study software engineers as they work, study real practitioners as they solve real problems [14, p. 8] [105, p. 9] [64, p. 1]. Empirical methods such as controlled experiments, case studies, surveys and post-mortem analyses are essential to the researcher and help to evaluate and validate the research results. [14, p. 7] Wohlin et. al describe them as "needed so that it is possible to scientifically state whether something is better than something else [14, p. 7]." Thus, empirical methods provide one important scientific basis for software engineering. The essence of an empirical study is the attempt to learn something useful by comparing theory to reality and improve our theories as result [81, p. 4].

Empirical research seeks to explore, describe, predict, and explain natural, social, or cognitive phenomena by using evidence based on observation or experience. This involves obtaining and also interpreting evidence which can be done for example by experimentation, surveys or systematic observation. Empirical research can make use of both *qualitative* and *quantitative* methods for collecting and analyzing data [108, p. 3].

- **Qualitative Methods**: Collect material in the form of text, images or sounds drawn from observations, interviews and documentary evidence, and analyze it using methods that do not rely on precise measurement to yield their conclusions, but on interpretation. [50][17] [108, p. 3] [14, p. 8] [37, p. 15f].
- **Quantitative Methods**: Collect numerical data and analyze it using statistical methods. Often aim to identify a cause-effect relationship. [50][17] [108, p. 3] [14, p. 8] [37, p. 15f].
- **Mixed Methods**: Take advantage of using multiple ways to explore a research problem, i.e. quantitative and qualitative [17].

Empirical studies are an important input to the decision-making in an improvement seeking organization. Since the introduction of new techniques, methods, or other ways of working usually aims at improving the current situation, before doing so, an empirical assessment of the impact following such changes can be performed [117, p. 24]. In Software Engineering improvement is very hard to evaluate without direct human involvement. Unlike a regular product, for software development processes it is not possible to build a prototype which can be evaluated in a safe environment. The only real evaluation of a process or process improvement proposal is to have people using it, since the process is just a description until it is used by people [117, p. 5]. This is exactly where the benefit of empirical method lies, since they are important in order to get objective and quantifiable information on the impact of changes [117, p. 24]. They fit into the context of industrial and academic software engineering research, as well as in a learning organization, seeking continuous improvement [117, p. 24]. Orchestrated correctly, they can provide deeper knowledge concerning understanding, improving, controlling and predicting the software process.

2.1.2 Types of Empirical Studies

Experiment

An experiment in software engineering is an empirical enquiry that manipulates one factor or variable of the studied setting, while the other variables are controlled at fixed levels. A variable in an experiment should correspond to the main activities, which have to be performed by a participant in order to achieve the goal [74, p. 2]. The effect of the manipulation is measured, and based on this a statistical analysis can be performed [14, p. 9] [117, p. 11] [6, p. 1]. Experiments are often highly controlled (hence the name "controlled experiment") and run in a laboratory environment [14, p. 9] [117, p. 10].

When experimenting, subjects are assigned to different treatments at random. [14, p. 9]

Experiments can be categorized in

- human-oriented experiments: Humans apply different treatments to objects
- technology-oriented experiments: different technical treatments are applied to different objects

In some cases it may be impossible to use true experimentation, i.e. to perform random assignment of the subjects to the different treatments. Such experiments are referenced as quasi-experiment [14, p. 9] [117, p. 11].

Case Study

Normally a case study is conducted studying a real project (i.e. a Software Project). It provides a deeper understanding of the the object under study. Case studies are used for monitoring projects, activities or assignments. Data is collected for a specific purpose throughout the study. A case study is an observational study while the experiment is a controlled study [14, p. 9] [117, p. 10] [94]. Case Studies are discussed further in 2.2.

Survey

The survey, sometimes referred to as research-in-the-large (and past [57, p.2]), since it is possible to send a questionnaire to or interview a large number people covering whatever target population needed [14, p. 10]. A survey is usually conducted when e.g. a tool or technique (the subject of study), has been in use for a while and thus, an investigation performed in retrospect [83] (retrospective study). As a result the researcher has no control over the activity which is under study. Unlike other types of studies (case studies or controlled experiments) variables cannot be manipulated during a survey research. A situation is recorded and can then be compared with other ones [75, p. 137].

Information is usually gathered by interviews or questionnaires which are performed with a sample that is representative from the population to be studied. The results from the survey are then analyzed to derive descriptive and explanatory conclusions. They are then generalized to the population from which the sample was taken [14, p. 10][117, p. 10].

2.2 Case Studies

Case studies as a type of empirical study have already been introduced in 2.1. Reviewing the literature, work focusing on the design and execution of a case study can be found. The theoretical frame of reference which did build a solid basis for the design of this thesis underlying case study (described in detail in 3.4) is presented in the following section:

2.2.1 Definition

Every now and then the term "case study" is part of the title of software engineering research papers. What these papers have in common is that they study a specific case, in contrast to a sample from a specified population [94]. According to Runeson et al. [94] this is where the similarities end, because some of the papers regard the studies of toy examples in a university lab (in vitro) as Case Studies, whereas others conduct well-organized studies in the field of operations (in vivo).

Case Studies have been frequently used as research methodology in areas such as psychology, sociology, political science, social work, business, and community planning [90, p. 150], aiming not solely at observing and increasing knowledge, but also at introducing changes in the studied phenomenon [94].

These goals, being to reach a better understanding and gain knowledge about the phenomenon under study which is than used to introduce improvements, can also be applied to software engineering research.

Unfortunately, reviewing the literature several definitions of the term "case study" can be found:

- Benbasat [9, p. 2]

  > A case study examines a phenomenon in its natural setting, employing multiple methods of data collection to gather information from one or a few entities (people, groups, or organization). The boundaries of the phenomenon are not clearly evident at the outset of the research and no experimental control or manipulation is used.

- Robson [90, p. 150]

  > Case study is a strategy for doing research that involves an empirical investigation of a particular contemporary phenomenon within its context using multiple sources of evidence.

- Robert K. Yin [118, p. 18]

  > an empirical enquiry that investigates a contemporary phenomenon within its real-life context, especially when the boundaries between phenomenon and context are not clearly evident.

Considering these definitions, all three of them define a case study as en empirical research methodology. Robson and Benbasat add the necessity of multiple sources and since in a software project there are many factors that influence the outcome, Yins addition of the boundaries between phenomenon and context not being clearly evident can be considered fitting too.

Taking these definitions into account, Runeson et al. derive their own definition specifically for software engineering [94, p. 27]:

> Case study in software engineering is an empirical enquiry that draws on multiple sources of evidence to investigate one instance (or a small number of instances) of a contemporary software engineering phenomenon within its real-life context, especially when the boundary between phenomenon and context cannot be clearly specified.

2.2.2 In Software Engineering

Experimentation in software engineering has clearly shown that there are many factors impacting the outcome of a software engineering activity, for example, when trying to replicate studies [118]. Case studies offer an approach that does not need a strict boundary between the studied object and its environment [117, p. 24]. Due to their dexterity, case studies have become a popular research method in software engineering and are frequently used to understand, explain or demonstrate the capabilities of a new technique, method, tool, process, technology or organizational structure [82, p. 1].

2.2.3 Case Study Process

Analog to most other empirical studies, when conducting a case study, there are five major process steps to be walked through [93, p. 137f] [94, p. 23ff] [42, p. 1] [117, p. 58].:

1. Case study design
2. Preparation for data collection
3. Collecting evidence
4. Analysis of collected data
5. Reporting

Case study design

Altough case study research is of flexible type, each of the steps gone through throughout the process is planned from the outset of the study and does not come about through serendipity. [82, p. 1] [93, p. 138] [13, p. 28].

Having a well prepared plan for a case study is crucial for its success and should at least contain the following elements [90, p. 155]:

1. Objective - what to achieve?

   The objective of a case study may for example be of exploratory, descriptive, explanatory, or improving nature. Although defined in the beginning, the objective is initially more like a focus point which evolves during the study. [4, p. 127f] [93, p. 139]

2. The case - what is studied?

   According to Yin the case may be anything which is a "contemporary phenomenon in its real-life context" [118]. In software engineering it often is a software project, but could also be (a group of) people, a process, an event, etc. However it must not be a "toy" program or project, due to lack of real-life context [93, p. 139] [94, p. 26]. It is possible for a case study to have multiple cases (i.e. to show parallels between them).

Yin distinguishes between [118, p. 50]:

- *Holistic* case studies: The case is studied as a whole (single unit of analysis). The goal is to study the global nature of the phenomenon.
- *Embedded* case studies: Multiple units of analysis are studied within a case. There can be for example a main unit and several smaller units. The goal may be to look for consistent patterns of evidence across units.

3. Theory - frame of reference

   Defining the theoretical frame of reference of the case study makes the context of the research clear, and helps both those conducting the research and those reviewing its results [16, p. 118] [94, p. 29] [93, p. 140].

   In order to build a solid foundation for the oncoming case study an extensive review of the literature in advance is recommended. The literature review should consider all previous significant work in the area. Only after having done this, the researcher can argue and come up with a set of statements describing the necessity for further investigation in the area of study [114, p. 2].

4. Research questions - what to know?

   In a case study, the research questions, reflect the focus of the study. There is no infallible way of generating research questions [90, p. 26] [12, p. 1f]. They are created by refining objectives into questions about the situation or problem to be studied and the purpose of the study. It is likely that this will take several iterations [4, p. 127f] [91, p. 19]. Caused by the nature of case study research questions will commonly be of the "why" and "how" type [9, p. 2]. It is important that these questions have not been addressed before and that they are formulated precise and unambiguous [114, p. 3] A research question may be related to a hypotheses, i.e a proposed explanation for a (an aspect of) phenomenon [94, p. 30].

5. Methods - how to collect data? (Discussed in detail in 2.2.3)

   The basic methods of data collection are defined at design time for the case study and rendered more precisely later. They are influenced by the sources of data known to be available. In a software project data is often automatically collected, for example via a ticketing system, issue tracking, version control systems, emails, automatic test suites, etc. [94, p. 32]

   Lethbridge et al. define three categories of methods: *direct* (e.g. interviews), *indirect* (e.g. tool instrumentation) and *independent* (e.g. documentation analysis)[65, p. 313ff]. Additionally one can distinguish between *qualitative* and *quantitative* methods. It is generally a good approach to combine qualitative and quantitative methods, in order to take advantage of the strengths of both [102, p. 3].

6. Selection strategy - where to seek data?

   Contrary to, for example, surveys and experiments, in a case study, he case and the units of analysis (groups, individuals, entire organization, specific project or a decision) should be selected intentionally. The case to draw data from should be "typical", "critical”, "revelatory” or "unique" in some respect, in practice however, many cases are selected based on availability [9, p. 5ff]. Since a case study might be replicated, case selection is particularly important [94, p. 61ff].

Yin also composed a list of five elements which are especially important for the research design of case studies [118, p. 27]:

1. a study's questions
2. its propositions, if any
3. the unit(s) of analysis
4. the logic linking the data and the propositions
5. the criteria for interpreting the findings

Case Study Protocol

All design decisions and field procedures for carrying through the case study are written down in the *Case Study Protocol*. This document is living documentation and keeping it up to date is important, since it helps the researcher to keep track of the study. Furthermore the Case Study Protocol helps making the research concrete, when written during the planing phase. It also helps to determine data collection sources, as well as when and what kind of data to collect [117, p. 60] [93, p. 141].

Ethical considerations

Empirical research is build on trust between all participating parties and ethical aspects must be taken into consideration, in order for the researcher not to destroy this foundation.

Before taking part in the study all subjects and organizations must be informed about the study's purpose, as well as which data will be collected and how will it be used. Afterwards the researcher must let them explicitly agree to participate in the case study, i.e. give informed consent.

Data used for publication must be made anonymous, which sometimes requires more than simply removing the subjects names, since they may still be identified by their characteristics if the set is small enough.

If the case study includes dealing with confidential information in an organization (e.g. software development practices, opinions not meant for superiors, etc.), special care must be taken [117, p. 33] [106, p. 105ff] [93, p. 141].

Collecting Data

Cases studies in software engineering usually provide large amounts of collectible data (for example automatically created data produced along the way), which sometimes must still be refined after collection. The researcher can also act as an active agent during data collection where evidence is defined. The data sources must be selected carefully, the raw and refined data structured in a way so that it is possible to find appropriate data [91, p. 22] [94, p. 47].

Deciding which data to collect is an important task, especially since in the early phase of the study, the researcher does not know all the details about the case yet. It is therefore common to adapt the data collection as the study progresses (flexible design for case studies). Furthermore data can be collected in an iterative way (for example using two interview iterations) [94, p. 49].

Case studies in comparison to other research methods have the advantage that evidence can be collected from multiple sources. The practice of using several data sources, and several types of data source, in order to limit the effects of one interpretation of one single data source is referred to as data triangulation. Evidence from different sources is used to corroborate the same fact or finding [94, p. 49] [90, p. 168] [91, p. 23] [65, p. 312] [11, p. 18].

According to Denzin [21, p. 300 ff.] 4 different types of triangulation exist:

1. Data triangulation: More than one method of data collection is used (e.g. observation, interviews, etc.)
2. Observer (Investigator) triangulation: More than one observer is used in the study.
3. Methodological triangulation: Combining qualitative and quantitative methods.
4. Theory triangulation: Multiple perspectives or theories are used

Some of the data collection methods named in Listing **??** will now be discussed in detail:

Interviews Interview based Data collection is one of the most important and frequently used sources of data in case studies in software engineering. Much of the knowledge that is of interest to the study exists only in the minds of the people working in the case being investigated. Interviews can help to collect exactly this kind of knowledge (historical data from the memories of interviewees. opinions or impressions about something). Generally, Interviews can be used for primary data collection or as validation of other kinds of data, like documents, automatically collected data, etc. [94, p. 50] [117, p. 62] [102, p. 15] [62, p. 229].

During an interview essentially a researcher talks to an interviewee and asks a set of predefined questions (interview questions) that the interviewee attempts to answer. The interview questions usually are based on the research questions [94, p. 50] [117, p. 62].

According to Robson [90, p. 288f], questions can be

- *open* i.e. inviting a broad range of answers and issues from the interviewed subject, while not providing any restrictions.

  Advantages: Broad range of answers. Interviewee will probably give information about a wider range of topics.

- *closed*, i.e offering a limited set of alternative answers.

  Advantages: Easy to analyze the results

- *scale*, i.e ask for a response in the form of degree of agreement or disagreement.

  Advantages: Easy to analyze the results

Interviews can be divided into *unstructured*, *semi-structured* and *fully structured* interviews [90, p. 290] [66, p. 268].

In an *unstructured interview*, questions are of the open type and reflect general concerns and interests of the researcher. The conversation is allowed to develop based on the interest of both the interviewee and the researcher. Unstructured interviews are most feasible for exploratory studies [94, p. 50] [117, p. 62] [102, p. 15] [90, p. 290f].

In a *semi-structured interview*, although questions are planned in advance, they are not necessarily asked in the same order as they are listed. The way the conversation develops can influence the order in which the different questions are presented to the interviewee. A list might be kept to ensure, that all questions are handled. Furthermore the wording used to express each question can vary as well as the amount of time and attention given to different topic. Improvisation and exploration of the issues raised in the conversation is welcome. [94, p. 51] [117, p. 62] [102, p. 15f] [90, p. 290f]. Robson recommends these type of interviews if the "interviewer is closely involved with the research process (e.g. in a small-scale project, when the researcher is also the interviewer)" [90, p. 290].

In a *fully structured interview* all questions are planned in advance and all questions are asked in the same order as in the plan. They closely relate to questionnaire-based surveys and make frequent use of closed questions [94, p. 51] [117, p. 62] [102, p. 15] [90, p. 290f].

Runeson et al. describe different principles, on the order of asking different kinds of questions [94, p. 52].

While planning an interview, it is important to decide on whom to interview. Runeson et al. advise to select interviewees based on differences instead of trying to replicate similarities [94, p. 51]. In order to validate the questions, a pilot interview can be conducted. It is important, that questions are not formulated in a leading way.

The *interview session* is started by the researcher presenting the aim of the interview and its study, optionally asking for permission to record the conversation, as well as how the collected information will be used. Afterwards a set of introductory questions are typically asked about the background of the subject, etc. to break the ice. It is essential to establish an atmosphere of trust between the interviewee and the researcher, especially since sensitive questions might be asked [43, p. 7]. Only then the main interview questions, which take up the largest part of the interview, will be presented. The researcher makes sure that the interviewee's opinions have been properly understood by summarizing them. This way also additional feedback can be gained. It is crucial not to interrupt the subject, or not to give the interviewee enough time to think about answers, since information might be lost [94, p. 52f].

Taylor et al. provide a number of points for creating and maintaining a good atmosphere during the interview [111, p. 116f]:

1. Be Nonjudgmental: The interviewer must not make negative judgments about the interviewee or put him down. Otherwise they wont open up about their feelings.[10, p. 140]
2. Let People talk: It is crucial to let the conversation flow, once people start talking about something important.
3. Pay Attention: Communicate interest in what the interviewee is saying. Do not drift off.
4. Be Sensitive: Do not pressure the interviewee when he feels uncomfortable about a matter.

During the interview sessions, it is recommended to record the discussion (audio or video), since it is nearly impossible to record all details and especially decide what is important to record [66, p.272] [117, p. 63].

An Interview can be conducted with every single subject, but it is also possible to conduct group-interviews. Such interviews are called **Focus Groups** [94, p. 54f].

Kontio et al. point out the benefits of focus groups as [59, p. 101] [60, p. 6]:

- *Discovery of new insights*: Due to different backgrounds and the interactive setting, interviewees are encouraged to react to other participants points and compare them with their own experiences.
- *Aided recall*: Other participants are able to confirm (or reject) facts that would have been hidden in an individual interview.
- *Cost-efficiency*: Multiple subjects can be interviewed at the same time.

Advantages according to [64, p. 320] [90, p. 299]:

- High level of interaction, group dynamics
- Natural quality control through participants (they check each others answers)
- Questions can be clarified
- Probing for unexpected responses
- Good atmosphere can be build to improve the quality of responses

Disadvantages according to [64, p. 320]:

- Time and cost intensive
- Scheduling and travel
- (Optional) Need of transcription

Observations A general approach in observation is to record the subjects under study and analyze the recording later. [94, p. 56] Since most of a software engineers work is done in the head, plain observation might not lead to the expected results. *Think aloud protocols*, which require the subject to verbalize his or her thoughts during process, enabling the researcher to understand it, provide an alternative.

Participant observation, as defined in [111, p. 55f], refers to "research that involves social interaction between the researcher and informants in the milieu of the latter, during which data are systematically and unobtrusively collected." Observations can be conducted to capture firsthand behaviors and interactions (for example software engineers conducting a certain task) that might not be noticed otherwise [102, p. 5].

In the Participant-Observer (Joining the Team) method, the researcher essentially becomes part of the team and participates in key activities. This way the researcher develops a high level of familiarity with the team members and the tasks they perform. The subjects under study are comfortable with the researchers presence and tend not to notice being observed [64, p. 327].

Advantages according to [64, p. 327]:

- Participants usually feel more comfortable if the researcher is part of the team.
- Researcher is able to develop a deeper understanding of the tasks

Disadvantages according to [64, p. 327]:

- Time consuming
- Loss of perspective if researcher becomes too involved.

Analysis of Electronic Databases of Work Performed In almost all bigger software development environments the work performed by developers is carefully managed using problem reporting, change request and configuration management systems. The data produced by such systems is a rich source of information for researchers.

Advantages according to [64, p. 330]:

- Often automatically generated
- Data is stable, not influenced by researchers preferences

Disadvantages according to [64, p. 330]:

- Manually entered data (for example in forms) incomplete, (and) or filled in different ways by different developers
- Potentially lots of data must be processed
- Subjects might not remember old records (and wont be able to provide additional information)

Coding

In order to process qualitative data using software, it is often necessary to extract numerical information. The process of *coding* extracts values for quantitative variables from qualitative data (often collected from observations or interviews). Therefore it is important to understand that qualitative data (information expressed as words or pictures) is not necessarily subjective and neither is quantitative data (data represented as numbers or other discrete categories) objective. Coding can therefore be used to translate qualitative data into quantitative data, without affecting its subjectivity or objectivity [102, p. 25].

Analysis of collected data

Data analysis as a separate phase starts after the data has been collected, but can also be done through the collection process. The main objective is, to derive conclusions from the data. Afterwards the reader should be able to follow the derivation of results and conclusions from the collected data (i.e. the chain of evidence must be kept) [118, p. 132ff].

Usually during a case study, a multitude of different evidence from different sources will be collected. Data analysis of this rich resource therefore proves a time consuming task [91, p. 24] [64, p. 335]. Generally it is often the case, that both quantitative and qualitative data has been collected. Data analysis is conducted differently for each type of data [117, p. 65] [93, p. 150].

Validity

The validity of a study denotes "the trustworthiness of the results, and to what extent the results are not biased by the researchers subjective point of view" [94, p. 71]. Of course concerns regarding validity must be raised throughout the whole research process and cannot be ticked off in a single step.

In Yin [118, p. 40f] uses a classification scheme which distinguishes between the following four aspects of validity:

1. *Construct Validity*: Identifying correct operational measures for the concepts being studied. For example: Are the constructs discussed in the interview questions interpreted in the same way by the researcher and the interviewed persons [94, p. 71]?

   Tactics:

   - Using multiple sources of evidence to see if they converge.
   - Building a solid chain of evidence.
   - Handing a case study report over to key informants, for them to review for it accuracy.

2. *Internal Validity*: This aspect of validity applies only for explanatory or causal studies, not for descriptive or exploratory studies. It aims at establishing a causal relation (how and why event X leads to event Y), whereby certain conditions are believed to lead to other conditions. However the researcher must be careful, because relationships can be spurious: e.g. when investigating whether one factor affects an investigated factor there is a risk that the investigated factor is also affected by a third factor.

   Tactics:

   - Pattern matching
   - Explanation building
   - Address alternative explanations
   - Use of logic models

3. *External Validity*: Are the outcomes of the study generalizable beyond the current case study or are they interesting to other people.

   Tactics:

   - use theory in single-case studies
   - use replication logic in multiple-case studies

4. *Reliability*: Demonstrating that the operations of a study do not depend on the researcher and can be repeated with the same results. This includes: Data collection procedures, coding of collected data, comprehensibility of questionnaires or interview questions, etc.

   Tactics:

   - Use case study protocol
   - Develop case study database

In [90, p. 171ff] Robson has listed some ways to increase the validity of the study (e.g. triangulation, peer debriefing, negative case analysis, etc.)

Reporting

The report of a study communicates its findings but is also the main source of information for judging the quality of the study [93, p. 154] [117, p. 69]. In order to do so, it is recommended to additionally report other aspects of the case study, such as the case study protocol and prepare collected artifacts (e.g. interview data) for dissemination [94, p. 77].

Yin [118, p. 165ff] and Rowley [91, p. 24] both describe the task of reporting to be one of the most challenging activities when doing case study research. An effective report must consider the target audience and therefore be tailored regarding its content. As a consequence a case study might result in multiple reports. In order to ease the task of reporting it is advised to start writing while the case study is still in process [90, p. 495] [91, p. 24].

In [90, p. 487ff] Robson defines a *set of characteristics* that a case study report should have. These have been summarized by Runeson in [94, p. 82]:

- Describe what the case study was about: Reporting the case study objectives and research questions, as well as possible changes that happened over time.
- Communicate a clear sense of the studied case: Interesting details like internal characteristics (size of the studied unit, average age of the personnel, etc.), should be included.
- Provide a "history of the inquiry" so the reader can see what was done, by whom and how: The sequence of actions and roles taken in the case study must be reported, without loosing oneself in details.
- Provide basic data in a focused form, so the reader can have confidence that the conclusions are reasonable.
- Articulate the researcher's conclusions and set them into a context they affect.

Concerning the *structure* of a case study report, Yin [118, p. 170ff] suggests several alternative formats:

- Linear Analytical: The traditional research report format - It starts with the issue or problem being studied, followed by a review of the literature. The subtopics then proceed to cover the methods used, the findings from data collected and the conclusion.
- Comparative: The case is repeated twice or more times, comparing alternative descriptions or explanations of the same case study.
- Chronological: The sequence of chapters might follow the early, middle, and late phases of case history. The researcher must be careful not to give disproportionate attention to the early events and insufficient attention to the later ones
- Theory-Building: The sequence of chapters or sections will follow some theory-building logic in order to constitute a chain of evidence for a theory.
- Suspense: Inverts the linear-analytical structure - conclusions are reported first and then backed up with evidence.
- Unsequenced: The sequence of sections or chapters assumes no particular importance.

Table 2.1 shows which types of structure can be applied for which type of research.

Academic reporting of case studies usually makes use of the linear analytical structure [93, p. 156] [68, p. 124].

Type	Explanatory	Descriptive	Explanatory
Comparative	X	X	X
Chronological	X	X	X
Theory-Building	X		X
Suspense	X		
Unsequenced		X	

Table 2.1: Structure types for case study reports based on Yin [118]

2.3 Action Research

Action Research tries to "influence or change some aspect of whatever is the focus of the research" [90, p. 199]. Since the author of this study actively participated in the underlying software project and did not restrict himself to passive observation, his methodological approach can be categorized as action research.

2.3.1 Definition

Robson [90, p. 199ff] describes Action Research as "primarily distinguishable in terms of its purpose, which is to influence or change some aspect of whatever is the focus of the research" . It further "adds the promotion of change to the traditional research purposes of description, understanding and explanation." Moreover he states that "Improvement and involvement are central to action research". In particular Robson lists improving:

- a practice of some kind
- the understanding of a practice by its practitioner
- the situation on which the practice takes place

In Action research, the researcher(s) are not limited to passive observation. On the contrary, collaboration between the researcher(s) and those who are the focus of the research, and their participation in the process, are "central to action research" [90, p. 200]. **Participatory Action Research** which "involves the investigation of actual practices and not abstract practices" [54, p. 277], highlights this aspect.

Action research is often depicted as a self-reflective cyclic process [54, p. 276] ,[90, p. 201]) Kemmis & McTaggart describe the spiral of self-reflective cycles as follows [54, p. 276]:

- Planning a change
- Acting and observing the process and consequences of the change
- Reflecting on these processes and consequences
- Replanning
- Acting and observing again
- Reflecting again, and so on ...

2.4 Agile Methods

Agile methods have been adapted widely in software engineering. The software process used in the project under study is based on agile methods as well. The following section is aimed at providing the necessary basics about agile software development, recommended for reading this thesis.

2.4.1 Agile Manifesto and Agile Development

The term "agile software development" was first described by Kent Beck et al. in the "Manifesto for Agile Software Development". It was authored 2001 as a reaction to classical, plan-driven software development processes and describes the key values of agile software development [8] as:

Individuals and interactions over processes and tools
Working software over comprehensive documentation
Customer collaboration over contract negotiation
Responding to change over following a plan

Adding that "while there is value in the items on the right, we (the authors) value the items on the left more."[8]

Along with these values, the authors further derive 12 principles for agile software development, which support the definition and continuing evolution of many software development methods [63, p. 24] [8].

Although Agile Development was disruptive at its time, some of its key concepts were already present in iterative and incremental development (IID). IID is an approach of building software (or anything) in which the overall lifecycle is composed of several iterations in sequence. Each iteration is a self contained mini project [63, p. 9f].

2.4.2 SCRUM

According to the FORBES magazine SCRUM is the most popular Agile methodology today [1]. Furthermore "The 2015 State of Scrum Report" states that 82% of the almost 5,000 people surveyed stated, that "Scrum improves quality of work life" and 95 % plan to continue using scrum. The "framework for developing and sustaining complex products" [101, p. 3] can therefore be seen as quite a success.

Definition

SCRUM is not a standardized process, defining a set of sequential steps that are guaranteed to produce, in time and in budget, a high-quality product that delights customers, but a framework for organizing and managing work. [92, p. 13].

According to Ken Schwaber and Jeff Sutherland SCRUM is a "framework within which people can address complex adaptive problems, while productively and creatively delivering products of the highest possible value." [101, p. 3].

The authors describe SCRUM also as:

- Lightweight
- Simple to understand
- Difficult to master

Origins

The SCRUM process was created by Jeff Sutherlan in 1993. However the term "scrum" itself has its origins in an analogy put forth in a 1986 study by Takeuchi and Nonaka, published in the Harvard Business Review [110, p. 137].

The authors state, that in "today's fast-paced, fiercely competitive world", where "speed and flexibility are essential", the sequential approach "won't get the job done". Instead they propose a "holistic method" for product development, which they referred to as the "rugby approach", because the ball gets passed within the team as it moves as a unit up the field [110, p. 137]:

The approach was based on six central characteristics [110, p. 138ff]:

1. *Built-in stability*
2. *Self-organizing project teams*
3. *Overlapping development phases*
4. *Multilearning*
5. *Subtle control*
6. *Organizational transfer of learning*

[1] as of 2017

When Jeff Sutherland and his team created SCRUM as a software development process at the Easel Corporation in 1993, they followed this example, comparing high-performing, cross-functional teams to the scrum formation used by Rugby teams.

What they did was combining approaches from Takeuchi and Nonaka with elements of Object Oriented Programming (OOP), empirical process control, IID, software process and productivity research, and complex adaptive systems [92, p. 34], resulting in the first paper on SCRUM "Scrum Development Process. OOPSLA'95 Workshop on Business Object Design and Implementation" being published in 1995 [99]. In 2001 followed "Agile Software Development with Scrum" by Ken Schwaber and Mike Beedle [100].

Description

Figure 2.1 depicts an overview of the SCRUM process: The product owner, having the big picture of the product breaks it down into a set of features, a process called *grooming*. These are collected and prioritized in the *product backlog*. During the sprint planing, the SCRUM team assembles the *sprint backlog* from features of the *product backlog*. These features must be completed in the *sprint execution*. Each day during the sprint the team members meet to talk about work done, work to be done, impediments,.. an activity called *daily scrum*. The outcome of a sprint should be a potentially shippable product, which is inspected by the stakeholders and SCRUM team in the *sprint review*. A sprint is concluded by a *sprint retrospective*, where all team members reflect on the SCRUM process being used to create the product, possibly resulting in adaptions being made. [38, p. 387f] [96, p. 62f] [100] [92, p. 17f] [101]

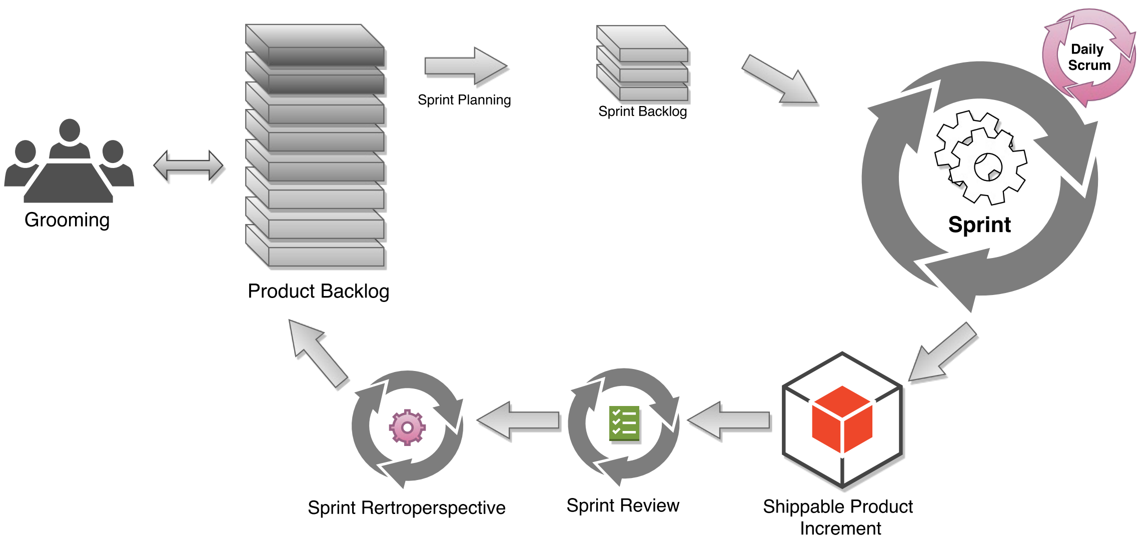


Figure 2.1: Scrum Framework according to [92, p. 17]

Roles

The Scrum framework stipulates three roles [92, p. 14ff]:

- *Product Owner:* Is responsible for the product and serves as the liaison between the development team and its customers. He manages the product backlog.
- *Scrum Master:* Is responsible for ensuring that SCRUM "values, practices and rules are enacted and enforced" [100]. He also represents the SCRUM team to the outside world.

- *Development Team:* Is a cross-functional group of people able to fulfill their given tasks.

Artifacts

An artifact is something of historical interest that deserves to be looked at again.

- *Product backlog:* A prioritized list of tasks yet to be done: "an evolving, prioritized queue of business and technical functionality that needs to be developed in a system" [100].
- *Sprint backlog:* A subset of the product backlog, consisting of the features to be implemented in the current sprint.
- *Product increment:* A list of tasks that have been completed during the last sprint and all previous ones, reflecting how much progress has been made.
- *Burn-down:* A visual representation of the amount of work that still needs to be completed.

Events

- *Grooming:* The activity of extending, maintaining and prioritizing the Product Backlog.
- *Sprint:* A time box (from about one week to one month) depicting an iteration in the SCRUM process.
- *Daily Scrum:* Daily meeting, aimed at facilitating communication and helping monitoring the progress towards meeting the sprint goal.
- *Sprint Planning:* A meeting in which the SCRUM team decides what will be part of the next springs backlog. At the end of the sprint all tasks must be done.
- *Sprint Review:* Demonstration of the results to stakeholders and interested parties.
- *Sprint Retrospective:* Team reflects on the previous sprint, ensuring continuous improvement of the process.

2.4.3 Lean Software Development

Lean Software Development is derived as the software equivalent of lean manufacturing which has its origin in the Toyota Production System [73] [84].

The term "Lean Software Development" itself was introduced by Mary Poppendieck and Tom Poppendieck in their book "Lean Software Development: An Agile Toolkit" [86]. The authors see their adaption as part of the agile development practices, a view shared by Ken Schwaber, who described Lean Software Development as an "agile process" and a "more understandable, robust, and everyday framework for understanding the workings of agile processes" [86].

According to [86], lean thinking is about 7 core principles [85] [86]:

Eliminate waste

Waste is anything not adding value to the customer, e.g a feature, functionality or process step. 2.2 shows examples of waste:

Manufacturing	**Software Development**
A component sitting on the shelf collecting dust	collected requirements in a book gathering dust
A manufacturing plant producing more items than immediately needed	Developers implement more features than immediately needed
Moving products around unnecessarily	Handing a task over to another team

Table 2.2: Waste in Manufacturing and Software Development [86, p. 13]

In [103] Sedano et al. identify waste typically occurring in software development processes

- *Building the wrong feature or product*: i.e. Parts that do not address user or business needs
- *Mismanaging the backlog*: Duplicating work, expediting lower value user features, delaying necessary bug fixes
- *Rework*: Altering already delivered work that was faulty
- *Unnecessarily complex solutions*: Creating unnecessarily complex solutions, neglected simplifications, etc.
- *Extraneous cognitive load*: Unneeded expenditure of mental energy i.e. due to inefficient tools, complex or large stories, technical debt
- *Psychological distress*: Burdening the team with unhelpful stress i.e. caused by "rush mode", conflicts, etc.
- *Waiting/multitasking*: idle time, often hidden by multi-tasking
- *Knowledge loss*: re-acquiring information that the team once knew
- *Ineffective communication*

Amplify learning

Lean production practices can not be translated directly to software development. The reason for this is that generating good software is not a production process, but a development process.

The Poppendiecks compare software development with the process of a chef creating a recipe and production with the execution of this recipe 2.3.

Development - of Recipe	**Production** - of Dish
Quality is fitness for use	Quality is conformance to requirements
Variable results are good	Variable results are bad
Iteration generates value	Iteration generates waste (rework)

Table 2.3: Development vs. Production according to [86][85]

According to [86] "The best approach to improving a software development environment is to amplify learning."

Decide as late as possible

The later a decision is made, the more knowledge might have been collected (the future is closer and easier to predict), helping in making the right decision. Accordingly development practices that allow for late decision making are effective in domains that involve uncertainty.

Deliver as fast as possible

In Software development, the discovery cycle is critical for learning: Design, implement, feedback, improve. Rapid delivery brings the advantage of shortening these cycles, which leads to early feedback, better learning and communication within the team. Furthermore speed assures that customers get what they need now, not what they required yesterday; a point becoming more and more important in the era of rapid technology evolution.

Empower the team

Involving developers in decision making is fundamental to achieving excellence, since nobody understands the details better than the people who actually do the work.

Build integrity in

Integrity in a product means that it excels in all ways, has no flaws [76]. Conceptual integrity means that the system's central components work well together as a whole with balance between flexibility, maintainability, efficiency, and responsiveness and it is a critical factor in creating perceived integrity Integrity comes from wise leadership, relevant expertise, effective communication and healthy discipline.

See the whole

Experts in an area (e.g UI or Business Logic) tend to care most about the part of the product representing their own specialty. If software is developed by multiple companies, people will want to maximize the performance of their own company. Quite often these patterns let the common good suffer. Furthermore complex systems with integrity, are not the sum of their parts, but also the product of their interactions. That's why overall performance should be measured instead of individuals or organizations.

Source	**Definition**
Oxford English Dictionary	The standard of something as measured against other things of a similar kind; the degree of excellence of something
ISO 8402-1986	The totality of features and characteristics of a product or service that bears its ability to satisfy stated or implied needs.
Crosby [18]	Conformance to the requirements
Juran [52]	Fitness for use
Peter Drucker [24]	Quality in a product or service is not what the supplier puts in. It is what the customer gets out and is willing to pay for.

Table 2.4: Definitions of Quality

2.5 Software Quality

Shipping and producing high quality software is essential in software development and also strongly tied to the goal of this thesis. The following section introduces basic definitions and concepts in the field of quality with a particular focus on software quality.

2.5.1 Quality

"A product is not high quality because its hard to make and costs a lot of money... this is incompetence" - Peter Drucker [24]

Giving a universally applicably definition of quality is far from easy. Unfortunately, quality is "hard to define, impossible to measure, easy to recognize" [58]. Moreover its "generally transparent when present, but easily recognized in its absence" [36, p. 3].

The definitions provided in the literature, may they be formal or informal, usually are context-depended as shown in 2.4.

Gilles [36, p. 3] provides some insights about quality and states that it:

- Quality is not absolute, but means different things in different situations.
- Quality cannot be measured upon a quantifiable scale like for example the room temperature.
- Some Aspects of Quality can be measured, some not.
- Quality is multidimensional, meaning that it has many contributing factors.
- Quality is subject to constraints.
- Quality is about acceptable compromises. Where quality is constrained and compromises are required, some quality criteria may be sacrificed more happily than others. A ranking, identifying critical attributes (least likely to be sacrificed) will come apparent.
- Quality criteria interacts with each other, causing conflicts. In a car for example fuel economy might conflict with top speed.

From a commercial point of view it is necessary to consider the constraints upon excellence. Although a company, only focusing on the excellence of their product, while neglecting customer expectations will not last, providing long term, high quality solutions is an integral part of success [36, p. 5].

Within a technical context however, the definition provided by the ISO is more relevant. It associates quality with the ability of the product to fulfill its function and recognizes that this is achieved through its features and characteristics. Quality is defined by having the required range of attributes, whilst achieving satisfactory performance within each attribute.

ISO9001:2008 expands this view, by stating that:

"The quality of an object can be determined by comparing a set of inherent characteristics against a set of requirements. If those characteristics meet all requirements, high or excellent quality is achieved but if those characteristics do not meet all requirements, a low or poor level of quality is achieved. So the quality of an object depends on a set of characteristics and a set of requirements and how well the former complies with the latter."

2.5.2 Software Quality

While all definitions provided in 2.4 may be applied to software quality, it makes sense to define the term "software quality" separately.

According to IEE [49] software is:

"Computer programs, procedures, and possibly associated documentation and data pertaining to the operation of a computer system."

Following this definition it can be deducted that high quality software depends on the quality of all 4 components. *Computer programs* (i.e. the code) are needed to instruct the hardware, *procedures* define how to use them, *documentation* is needed e.g. for maintenance and usage instructions and *data* is used for configuration, testing etc. [20, p. 15f]

When comparing software products with other industrial products (keeping the definitions in mind), fundamental differences can be found [20, p. 4f] [36, p. 7]:

- Product complexity: Can be measured for example by the number of states. Whereas an industrial machine can have up to a few thousand modes of operation, a software product usually exceeds millions of software operation possibilities.
- Product visibility: Defects in software are usually hidden, whereas e.g. a missing door on a car is easily spotted.
- Product development and production process: In software projects defects can only be detected in the development phase.
- Software has no physical existence.
- Change of client needs over time and high expectations: Customers expect software to be up to date, but car manufacturers for example wont update your car to meet the newest emission levels.

Furthermore the definition of software quality is hindered, since usually perspectives and expectations influence peoples minds, concerning what their understanding of software quality is. Accordingly different people would have different views and expectations, based on their roles and responsibilities [113, p. 15].

Kitchenham and Pfleeger [56, p. 13f] state five major views of software quality:

1. Transcendental: Quality is hard to define, but can be recognized when present. Associated with properties that delight users.
2. User: Fitness for purpose or meeting user's needs.
3. Manufacturing: Conformance to process standards.
4. Product: Quality correlates with internal quality indicators.
5. Value-based: Customer willingness to pay for the software.

Having established that software quality is not something that can be measured upon a quantifiable scale, is subject to many different views, multidimensional and hard to define, the problem of its definition can be tackled by describing its characteristics [36, p. 9] [77, p. 6].

While some definitions start with terms like defect rate, reliability and customer satisfaction [53, p. 4], consulting the literature, many attempts to capture the extent of software quality, which gauge towards more specific attributes can be found:

- *CUPRIMIDSO*: Capability (Functionality), Usability, Performance, Reliability, Installability, Maintainability, Documentation/Information, Service, Overall
- *FURPS*: Functionality, Usability, Reliability, Performance, Serviceability
- *CUPRIMDA*: Capability, Usability, Performance, Reliability, Installability, Maintainability, Documentation, Availability.
- *ISO 9126 Quality characteristics*: Functionality, Reliability, Usability, Efficiency, Maintainability, Portability

The ISO 9126 standard has been revised and replaced by ISO/IEC 25000, which considers additional quality factors (Security, Compatibility) and aggregates them to a quality model 2.2. This model categorizes the product quality into characteristics and sub-characteristics:

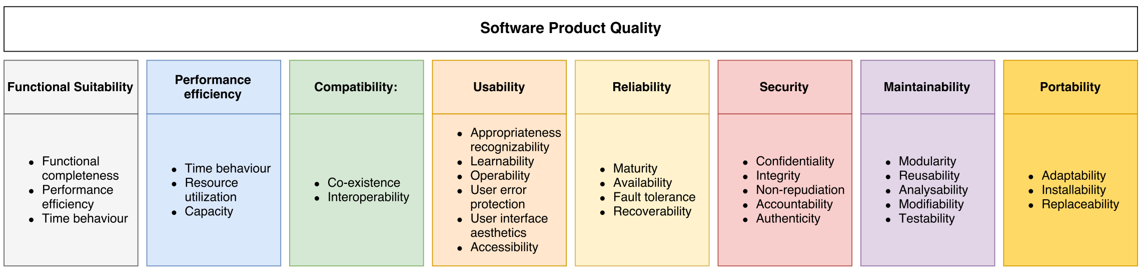


Figure 2.2: ISO/IEC 25010 Quality Model

1. **Functional Suitability:** represents the degree to which a product or system provides functions that meet stated and implied needs when used under specified conditions
   - *Functional completeness:* Degree to which the set of functions covers all the specified tasks and user objectives.
   - *Functional correctness:* Degree to which a product or system provides the correct results with the degree of precision needed.

- *Functional appropriateness:* Degree to which the functions facilitate the accomplishment of specified tasks and objectives.

2. **Performance efficiency:** represents the performance relative to the amount of resources used under stated conditions

   - *Time behaviour:* Degree to which the response and processing times and throughput rates of a product or system, when performing its functions, meet requirements.
   - *Resource utilization:* Degree to which the amounts and types of resources used by a product or system, when performing its functions, meet requirements.
   - *Capacity:* Degree to which the maximum limits of a product or system parameter meet requirements.

3. **Compatibility:** Degree to which a product, system or component can exchange information with other products, systems or components, and/or perform its required functions, while sharing the same hardware or software environment.

   - *Co-existence:* Degree to which a product can perform its required functions efficiently while sharing a common environment and resources with other products, without detrimental impact on any other product.
   - *Interoperability:* Degree to which two or more systems, products or components can exchange information and use the information that has been exchanged.

4. **Usability:** Degree to which a product or system can be used by specified users to achieve specified goals with effectiveness, efficiency and satisfaction in a specified context of use.

   - *Appropriateness recognizability:* Degree to which users can recognize whether a product or system is appropriate for their needs.
   - *Learnability:* degree to which a product or system can be used by specified users to achieve specified goals of learning to use the product or system with effectiveness, efficiency, freedom from risk and satisfaction in a specified context of use.
   - *Operability:* Degree to which a product or system has attributes that make it easy to operate and control.
   - *User error protection:* Degree to which a system protects users against making errors.
   - *User interface aesthetics:* Degree to which a user interface enables pleasing and satisfying interaction for the user.
   - *Accessibility:* Degree to which a product or system can be used by people with the widest range of characteristics and capabilities to achieve a specified goal in a specified context of use.

5. **Reliability:** Degree to which a system, product or component performs specified functions under specified conditions for a specified period of time.

   - *Maturity:* Degree to which a system, product or component meets needs for reliability under normal operation.
   - *Availability:* Degree to which a system, product or component is operational and accessible when required for use.
   - *Fault tolerance:* Degree to which a system, product or component operates as intended despite the presence of hardware or software faults.

- *Recoverability:* Degree to which, in the event of an interruption or a failure, a product or system can recover the data directly affected and re-establish the desired state of the system.

6. **Security:** Degree to which a product or system protects information and data so that persons or other products or systems have the degree of data access appropriate to their types and levels of authorization.
    - *Confidentiality:* Degree to which a product or system ensures that data are accessible only to those authorized to have access.
    - *Integrity:* Degree to which a system, product or component prevents unauthorized access to, or modification of, computer programs or data.
    - *Non-repudiation:* degree to which actions or events can be proven to have taken place, so that the events or actions cannot be repudiated later.
    - *Accountability:* Degree to which the actions of an entity can be traced uniquely to the entity.
    - *Authenticity:* Degree to which the identity of a subject or resource can be proved to be the one claimed.
7. **Maintainability:** Degree of effectiveness and efficiency with which a product or system can be modified to improve it, correct it or adapt it to changes in environment, and in requirements.
    - *Modularity:* Degree to which a system or computer program is composed of discrete components such that a change to one component has minimal impact on other components.
    - *Reusability:* Degree to which an asset can be used in more than one system, or in building other assets.
    - *Analysability:* Degree of effectiveness and efficiency with which it is possible to assess the impact on a product or system of an intended change to one or more of its parts, or to diagnose a product for deficiencies or causes of failures, or to identify parts to be modified.
    - *Modifiability:* Degree to which a product or system can be effectively and efficiently modified without introducing defects or degrading existing product quality.
    - *Testability:* Degree of effectiveness and efficiency with which test criteria can be established for a system, product or component and tests can be performed to determine whether those criteria have been met.
8. **Portability:** Degree of effectiveness and efficiency with which a system, product or component can be transferred from one hardware, software or other operational or usage environment to another:
    - *Adaptability:* Degree to which a product or system can effectively and efficiently be adapted for different or evolving hardware, software or other operational or usage environments.
    - *Installability:* Degree of effectiveness and efficiency with which a product or system can be successfully installed and/or uninstalled in a specified environment.
    - *Replaceability:* Degree to which a product can replace another specified software product for the same purpose in the same environment.

Term	Definition
Error	A mistake made by a person resulting in a defect in the product.
Defect, Fault	A flaw in the product resulting from the commission of an error.
Failure	The experienced manifestation of a defect being encountered in the product.

Table 2.5: Software errors, faults and failures [41], [113, p. 20], [20], IEEE 610.12

While all listed approaches differ in quantity, naming and detail of its described characteristics, commonalities can still be found, strengthening the image of software quality as a multidimensional construct.

Gilles describes a polyhedron metaphor, depicting quality as a three- dimensional solid, where each face represents a different aspect of quality. Using this metaphor he furthermore tries to close the cycle towards multiple views on software quality [36, p. 9]. Gilles polyhedron metaphor can be combined with the ISO 9126 standard, resulting in an adaption of the quality cube, using the ISO 9126 Quality characteristics as faces 2.3.

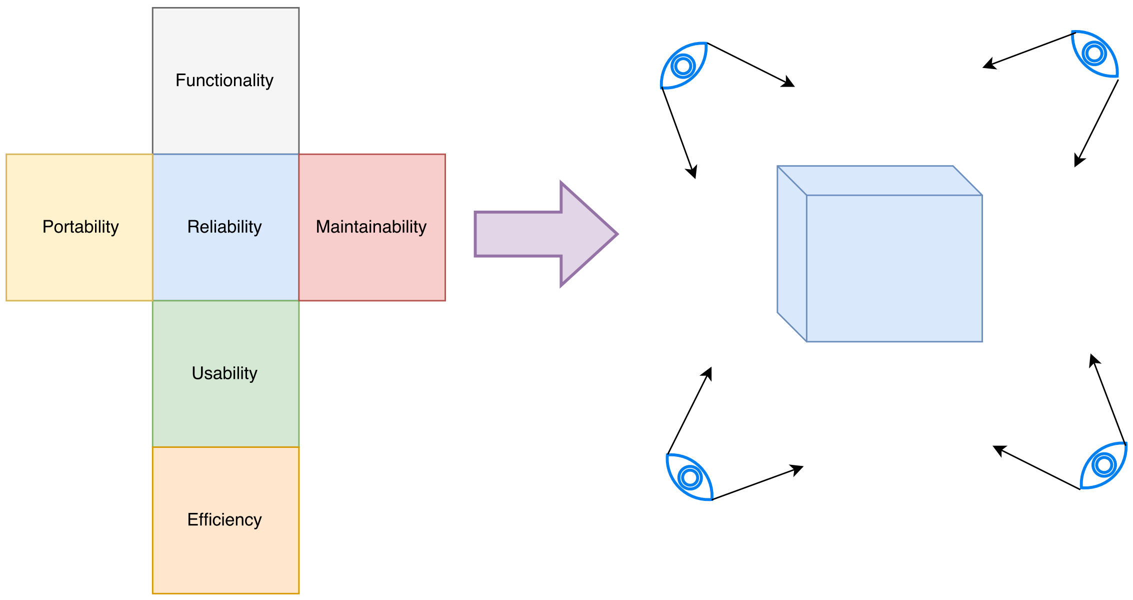


Figure 2.3: Dimensions of Quality (Adapted from [36, p. 9])

When defining quality enhancing procedures, tailoring processes for quality assurance and talking about metrics, one fundamental fact is often forgotten: Software quality is about people [36, p. 15] It is people who develop software, carry out quality assurance and do the testing. All procedures aimed at enhancing quality can only be of value, provided that people are capable and willing towards their effective use.

2.5.3 Software Quality Assurance

While quality is something that can (or can not) be present in software to a certain degree, software quality assurance is the road that leads to high quality software.

One of the most commonly used definitions of SQA is offered by the IEEE Glossary. It has matured over several iterations (since 1984) and now (IEEE 730-2014) describes SQA as:

> A set of activities that define and assess the adequacy of software processes to provide evidence that establishes confidence that the software processes are appropriate for and produce software products of suitable quality for their intended purposes. [48]

Despite the fact that the current IEEE definition tries to be as open as possible it still neglects maintenance, timetable and budget issues. [47] [20, p. 26]

In [20, p. 27] Galin proposes and extended definition

> Software quality assurance is: A systematic, planned set of actions necessary to provide adequate confidence that the software development process or the maintenance process of a software system product conforms to established functional technical requirements as well as with the managerial requirements of keeping the schedule and operating within the budgetary confines.

In his definition Galin emphasizes that SQA (a) must not end at the release date, but must go on during maintenance, (b) comply to budgetary constraints (c) keep the schedule. Accordingly, quality assurance is meant to minimize the costs of quality by introducing a variety of activities throughout the development and maintenance process in order to prevent the causes of errors, detect them, and correct them in the early stages of development. [20, p. 31].

Quality control is a set of activities carried out with the main objective of withholding a product from shipment if it does not qualify. Its actions are therefore only a subset of the total range of quality assurance activities. SQA relates to quality control in a way, that its measures substantially reduce the rate of non-qualifying products [20, p. 29, p. 31].

2.6 Agile Software Quality

Agile software development comes along with new benefits and challenges concerning software quality. The following chapter tries to discuss problems of QA in plan driven approaches as well as characteristics of QA in agile projects:

> "one of the main challenges in agile projects it is the lack of knowledge of agile practices" [115].

2.6.1 Problems with QA in Plan Driven Approaches

In classic plan driven development processes, for example the waterfall model quality assurance, is usually done in a separate phase of the process. Naturally this phase must follow only after the implementation is done [107].

This approach can lead to problems when keeping in mind that software is developed according to a plan, against a deadline. One phase can only start if the preceding one is finished. If i.e. implementation takes longer than calculated in the plan, the remaining phases (i.e. testing) loose precious time. Instead of delaying the release it might be the case that the quality assurance period is squeezed [107].

Even if QA can start on time, if the number of found defects is high, the product is returned to the development team to fix the defects and another testing phase is kicked off. This cycle might repeat itself causing unnecessary stress.

In agile iterative development on the other hand, each iteration must deliver working software that has been fully tested. Each story implemented by the developing team is tested subsequently. The story is not complete until it passes all of its tests and is accepted by the customer. This allows the customer to actually see progress and eases project planning [71].

In [44] Huo et al. provide a comparative analysis concerning quality, between the Waterfall model as a representative of the traditional methodologies approaches, and the agile group of methodologies. They conclude that "agile methods do have practices that have QA abilities", "the frequency with which these agile QA practices occur is higher than in a waterfall development" and "agile QA practices are available in very early process stages due to the agile process characteristic".

2.6.2 Quality Assurance in Agile Development

The use of agile practices with the objective of achieving high software quality, makes QA process immanent in agile software development [72] [95]. Mnkandla et al. tried to link agile techniques to their quality assuring capabilities by evaluating a predefined set of quality assurance factors 2.6 in relation to the corresponding agile techniques that implement the factors [72].

Name
Correctness
Robustness
Extendibility
Reusability
Compatibility
Efficiency
Portability
Timeliness
Integrity
Verifiability
Validation
Ease of use
Maintainability
Costeffectiveness

Table 2.6: Set of quality assurance factors

As a result, for each of the listed factors agile techniques that help ensuring them could be found.

Fortunato et al. evaluated and analyzed relevant studies on quality assurance practices in agile software development [31].

Their key finding were that:

1. The use of people with a generalist professional profile, who perceive teamwork and good communication between team members as a philosophy, are adept of constant knowledge improvement, favor the perfectionism and support the adoption of agile methods in projects. This statement is also backed by the Chaos Report 2015, which states that "one of the factors that contributed to the success of the projects were the use of proficient agile people. [112]"
2. The use of good programming practices is regarded as one of the pillars of quality assurance, making the training and retraining of the team in techniques such as coding standards, refactoring, creating unit tests, review code, pair programming and use of continuous integration tools are important factors for achieving the quality assurance.
3. Software testing proved as the factor, most likely to be incorrectly performed.

Again it can be concluded that the application of agile practices leads to better software quality.

According to [1] adopting agile development methodologies has a positive impact not only on quality but also on productivity:

The results of a survey showed that the application of agile practices like: knowledge sharing, active stakeholder, participation, self organizing teams, reduced documentation, responding to change, team size, flexible design and trainings improve quality and productivity [1]

While having "build in" QA measures, agile software development also allows for the easy integration of new measures. Generally agile processes like SCRUM do not represent direct work instructions for QA, but merely "empty buckets" which need to be filled with situation specific SQA practices and processes [51].

2.7 Monitoring and Measuring Quality

"Measure what is measurable, and make measurable what is not so." - Galileo Galilei

"You can't control what you can't measure" - Tom DeMarco (1982)

2.7.1 Measuring Quality

In order to manage software quality (i.e. state improvements, point out weaknesses), it has to become measurable. However, when making things measurable one must be careful not to distort the situation under measure and assure that the derived figures do indeed reflect the property under scrutiny [36, p. 41].

Quality measurement is usually expressed in terms of metrics.

2.7.2 Software metrics

Metrics provide the means to measure software quality and can be defined for every type of quality, e.g product, customer satisfaction, process, etc.

Before going into detail, the term metric has to be defined:

Definition

IEEE provides two alternative but complementary definitions describing quality metrics as a category of SQA tools:

"A function whose inputs are software data and whose output is a single numerical value that can be interpreted as the degree to which software possesses a given attribute that affects its quality. [46]"

"A quantitative measure of the degree to which an item possesses a given quality attribute. [49]"

Whereas the first definition by IEEE refers to the process that produces quality metrics the second refers to the outcome of the above process.

Gilles [36, p. 41] describes a quality metric as a "measurable property which is an indicator of one or more of the quality criteria that we are seeking to measure."

It must

- be clearly linked to the quality criterion under measure
- be sensitive to the different degrees of the criterion
- provide objective determination of the criterion that can be mapped onto a suitable scale.

However metrics must be distinguished from direct measures. A direct measure (i.e. the expansion of mercury in a thermometer in order to measure temperature, or the number of lines of code) does not depend upon a measure of any other attribute. It can be seen as a function whose domain is only one variable [36, p. 41].

Take the traditional fairground trail of strength as an example: A contender must use a hammer to punch a plate, resulting in a score represented by the height reached on the column. Here, strength is not measured in a absolute and verifiable way. The reached height depends only on the force exerted upon the plate at the bottom, which however is affected by multiple factors other than strength (i.e. swing technique, etc.).

The score of this experiment is therefore like a metric of strength, i.e. clearly linked to the criterion it seeks to measure.

Objectives

The main objectives for software quality metrics according to [20, p. 414] are:

- To facilitate management control as well as planning and execution of the appropriate managerial interventions.
- To identify situations that require or enable development or maintenance process improvement in the form of preventive or corrective actions introduced throughout the organization

Comparison provides the practical basis for management's application of metrics and for SQA improvement in general [20, p. 414].

Performance data can be compared with quality targets, standards, other projects, other teams, etc.

Requirements for good metrics

In order to come up with useful metrics, Galin [20, p. 415] provided requirements (general and operative), an applicable and successful metric must provide.

General Requirement

- *Relevant*: Related to an attribute of substantial importance
- *Valid*: Measures the required attribute
- *Reliable*: Produces similar results when applied under similar conditions
- *Comprehensive*: Applicable to a large variety of implementations and situations
- *Mutually exclusive*: Does not measure attributes measured by other metrics

Operative Requirement

- *Easy and simple:* The implementation of the metrics data collection is simple and is performed with minimal resources
- *Does not require independent data collection:* Metrics data collection is integrated with other project data collection systems: employee attendance, wages, cost accounting, etc. In addition to its efficiency aspects, this requirement contributes to coordination of all information systems serving the organization

- *Immune to biased interventions by interested parties:* In order to escape the expected results of the analysis of the metrics, it is expected that interested people will try to change the data and, by doing so, improve their record. Such actions obviously bias the relevant metrics. Immunity (total or at least partial) is achieved mainly by choice of metrics and adequate procedures

Classification

The literature provides multiple classifications of software quality metrics:

A general distinction can be made whether a metric is predictive or descriptive [36, p. 44] [70, p. 104] [5, p. 138]: A *predictive metric* can be used to make predictions about the software later in its lifecycle. An example is structuring: A well structured software will be easier to maintain later in its life. A *descriptive metric* describes the system at the time of measurement. As an example reliability can be described by the number of crashes in a given period of time.

Another way of categorizing metrics is by what they are measuring, either external or internal characteristics. *External characteristics* are quality factors (like described in the ISO/IEC 25010 Quality Model). *Internal characteristics* are objective measures like size, control flow complexity, inter-module coupling and modular cohesion [39].

Product metrics (related to software maintenance) are defined as snapshots of a product state, whereas process metrics (related to the software development process) are defined as measures over time [20, p. 412f] [39].

2.7.3 List of Metrics

Agile Metrics

The upcoming of agile methods in software development also called for new metrics to be introduced in order to deal with agile practices [61] [75]. Their purpose was to quantify the projects progress in essential parts of agile development like testing and Continuous Integration (CI) [51].

Number of Tests: Since agile development encourages developers to use test driven development, the total number of tests can be a helpful metric.

Test Coverage: Measures how much of the code is covered by tests. There are different measurements like line coverage and branch coverage. The test coverage gives a general impression on how well the code is tested.

Test Growth Ratio: Measuring the test coverage only can prove less expressive, if the project lacks test at its current state and the decision to go test driven for example was only made recently. In such a scenario it makes more sense to measure the growth of tests in relation to the growth of the source code. Of course this metric can also be falsified if the source code is reduced e.g. due tu refactoring, or after removing functionality.

Number of Broken Builds: If in a CI setup, a build is triggered and fails for project specific reasons and therefore is not merged in the code basis, it is called "broken build". This usually happens if a quality requirement was not met (syntax error, unable to compile, tests failed, etc.) by the code that should have been build. A broken build might have been caused by a software bug that otherwise might have ended up in production. Fixing this build usually has highest priority in the team. Again one must be careful since in some project setups broken builds can be blamed on faulty tests (indeterministic) or infrastructure problems.

2.8 Gamification

Gamification techniques will be applied to the field of QA in the course of this study. The overall goal is to boost motivation on the developers side, in terms of doing software quality related tasks.

The following section will provide background information on the concepts of Gamification:

2.8.1 Definition

The term "Gamification" has its origins in the digital media industry and dates back to 2008 [22].

Consulting the literature several definitions have been proposed:

Deterding et al. "the use of game design elements in non-game contexts" [22, p. 2ff]

Pereira et al. "applying game-design thinking to non-game applications to make them more fun and engaging" [80, p. 744].

Huotari et al. "a process of enhancing a service with affordances for gameful experiences in order to support user's overall value creation." [45, p. 3]

Zichermann et al. "The use of game thinking and game mechanics to engage users and solve problems" [119]

2.8.2 Elements of Gamification

Gamification is founded in the fundamentals of human psychology and behavioral science, and rests on three primary factors: motivation, ability level and triggers.

Gamification uses certain elements/mechanics of games to motivate people [116, p. 72ff] [19, p. 85]:

- *Achievements*: Given to the player on the completion of a behavior.
- *Points*: Are allocated for specific high value behaviors and achievements.
- *Badges*: Represent certain achievements of the user.
- *Levels*: Increases as the users reach a certain number of points.
- *Quests*: The tasks the player has to complete are presented as a quest, with additional game elements (such a story) that make it more attractive.
- *Voting*: Players can vote on another player's behavior. The votes themselves represent the rewards obtained by each player.
- *Leaderboards*: A ranking with the top players is presented to all players to increase competitiveness.

2.8.3 Gamification in Software Engineering

Dale et al. state that "When done correctly, gamification provides an experience that is inherently engaging and, most importantly, promotes learning." [19, p. 85], a feature most important to software engineering.

Francisco-Aparicio et al. conclude that "Gamification can be used as a tool to improve the participation and motivation of people in carrying out diverse tasks and activities that generally could not be very attractive." [32], which does apply to various software QA related tasks. Furthermore Pedreira et al. describe gamification as a "promising field which can help to improve the daily engagement and motivation of software engineers in their tasks." [79, p. 158]

Several well established examples for gamification in software engineering already exist:

- *StackOverflow* [2]: question-and-answer site
- *RedCritter* [3]: software that enables enterprises to manage, showcase, and reward employee achievements.
- *Codecademy* [4]: Online Programming Courses with reward system.

[2] https://stackoverflow.com/
[3] https://www.redcritterconnecter.com/home
[4] https://www.codecademy.com

3 Problem Description

The following chapter describes the thesis's underlying problem in terms of scope, scientific relevance as well as its unique characteristics. The study's case will be presented in detail and a list of research questions compiled, which will guide the research throughout this thesis. Subsequently the studies design will be presented, concluding with the attempt of establishing case representativenes.

3.1 Academic Software Project Specifics

Software projects developed by students in an academic environment, by their nature, have unique characteristics which will be described in the following chapter, alongside their impact on daily software engineering business.

3.1.1 Differences to industry projects

Software projects developed by students in the context of an university course differ greatly from industry projects. In the following the most significant differences will be listed:

1. *Costs are irrelevant*: It does not matter how much time is spent by the students, since their work comes at no cost at all.
2. *Focus*: The focus of academic projects is to maximize the learning experience of students, not to solve a problem and deliver a working product. Additionally, issues people are facing in industry projects, like e.g. time pressure or pressure to deliver usually do not apply.
3. *High Developer Fluctuation*: It might be the case that a student joins the project and leaves after 4 months, or even earlier.
4. *Developer Availability and Scheduling*: Students usually attend other university courses as well, or are even in a working relation. As a result the amount of time they can spent on the project is limited, differs from week to week and person to person. [15, p. 4]
5. *Developer motivation*: Motivation is the driving power in student projects [109, p. 416]. However, motivation and project participation might differ largely from student to student and on a day to day basis. Additionally, unlike professional software developers, students are not compensated for their work by payment or other bonuses, but only a degree at the end of their work, which might impact motivation as well [15, p. 3].
6. *Limited management options*: The attendees are here by their own will. They are not bound to their work like an employee in a company. Furthermore, unlike in an industrial setting, project supervisors cannot easily fire or hire developers. [15, p. 5]
7. *Quality vs. features*: Students usually prefer to add new features instead of instead of enhancing software quality [35, p. 40] While this might apply on developers in a professional background as well, regulation is harder in academic software projects because of the limited management options.

8. *Different Skill Level*: Students with different levels of education or practical experience might work on the same team.

9. *Working environment*: Students usually do not develop software in an office and their SCRUM teams are not provided with the best possible tool, as suggested by Schwaber and Beedle [p. 39f][100]. Communication is mostly done over Voice over IP (VoIP) [98]

10. *Quality and maintainability*: Academic software is often written to demonstrate a solution for a given problem in a certain context. Once the solution is presented (e.g. to receive a grade), the code itself has served its purpose and is no longer maintained. This consequently leads to code quality being of low priority.

Now that the differences between university and industry projects have been described, their influence on the development process can be elaborated:

1. *Costs are irrelevant*: Because there are no financial and only limited temporal restrictions, the focus can be set on the development process and learning experience. This of course aids the author with his thesis in terms that conducting experiments does not cause conflicts with business interests.

2. *Focus*: Since the main focus lies on gathering experience and knowledge, this must also be taken into account when designing the development process. On the one hand it is important for developers not to rush through the developing process, on the other hand the process must not cause too much overhead.

3. *High Developer Fluctuation*: The biggest issue here is, that knowledge has to be preserved. Developers might implement crucial parts of the system and take all their knowledge about these parts with them, when they leave the project. Furthermore developers usually need a considerable amount of time, until they get productive. An experienced developer is usually tasked with introducing newly joined members into the development process, again, at the cost of developing speed.

4. *Developer Availability and Scheduling*: Since students usually only spend a few hours per week on the project, their tasks have to be small enough to be manageable in this amount of time.

5. *Developer motivation*: Since developers do not receive monetary compensation for their efforts, they have to find their motivation somewhere else. Too much work that they deem unnecessary can easily frustrate them.

6. *Limited management options*: Students cannot be forced or urged to do a certain task. The workload generally has to be well balanced for each student and the given tasks have to catch his interest.

7. *Quality vs. features*: Tasks related to QA, like repairing defects or eliminating bad code smells might not be regarded as rewarding by students. Motivating students to do such work therefore proves to be difficult.

8. *Different Skill Level*: Since the course in whose framework the development on Mineralbay is done, is available to a broad range of students (i.e. bachelor and master students from different programs), their level of skills differs greatly.

9. *Working environment*: Meetings usually have to take place in public places, which most likely are not perfectly suited for software development, or online, using VoIP and screen sharing tools.

10. *Quality and maintainability*: The amount of time students spent on the project is limited (e.g. they reached their compulsory amount of hours to receive a grade). Therefore the focus might not be set on high quality, since once they part from the project they wont have to maintain the code they produced.

3.2 The Case - Project Details

In the following section this thesis's underlying case will be described in detail.

3.2.1 Synopsis

The process of driving a tunnel, beside its main purpose, results in great amounts of raw-material being excavated from underground construction sites. These are called tunnel spoil and are composed from a set of different kinds of primary raw materials.

In the Germany, Austria, Switzerland (DACH) region alone, tunnel construction projects (underground rail, urban rail, rail, road, sewers and utilities) with a total excavated volume of about 120 m^3 are currently under construction or at the design phase (as of 12/2014).

The Mineralbay project was born to achieve higher utilization rates for the excavated materials by providing a material management and merchandising software system for the exchange and trade of rock excavation material. These materials, known as tunnel spoil are declared as waste by Austrian law tunnel [88] [89] [87], which leaves tunnel driving companies with the responsibility of their disposal. From a technical point of view however, this "waste" is actually raw material. Disposing it, only for disposal centers to resell them, makes no sense from an ecological and economical perspective [28].

Concluding, direct recycling of tunnel spoil would provide several benefits [27] [28]:

- Economic advantages: Selling the tunnel spoil will result in earnings, while the removal of disposal fees will result in savings.
- Ecological advantages: Unnecessary transport, etc. will be avoided,

This is where Mineralbay comes in:

Parties in need of material will be able to find it on Mineralbay in its best fitting form, as close as possible (keeping transport costs low), while construction companies will be able to sell their tunnel spoil conveniently.

In order to provide the best service Mineralbay will process data on material, mass and time parameters for the mined rock e.g. in the form of online material analysis results collected from tunnel drilling machines.

This enables selected information being made accessible to a wide circle of interested parties simultaneously. The resulting value added chain is extending far outside the construction industry and thus enables successful upcycling [29].

Brought to live as a university project by the Montanuniversität Leoben, Chair for Subsurface Engineering and INSO at TU, Mineralbay offers a motivating setting for participants in an academic environment. The domain specific knowledge, as well as the requirements are provided by the representatives of the Montanuniversität, while students of the TU implement the software.

Team

The developing team is built from students of the INSO at the TU and consists of both master and bachelor students.

There are several reasons why students join the Mineralbay team:

- *Bachelor Thesis:* Students are given the opportunity to write their bachelor thesis in the context of the Mineralbay project. They usually tackle specific problems in the development process. Alongside the theoretical part of the thesis they have to work 300 hours practically on the project.
- *Master Thesis:* Students are able to do their master thesis in the context of the Mineralbay project. Contrary to the bachelor thesis writing students they are not obligated to do practical hours, but do so anyway in order to get familiar with the project.
- *Practical course (project):* Students can also apply for doing a practical course in the Mineralbay project. They have to work a predefined set of hours on the project to get their certificate.

Students can take on different roles in the Mineralbay project:

- *Developer:* Developers are tasked with implementing features of Mineralbay. Every developer has to work on all layers of the stack (Database, Backend, Frontend). Writing tests (Unit, Integration and browser/acceptance tests) is also a task of the developers.
- *Tester:* Testers perform manual testing, administer test case, are responsible for the acceptance of features, report bugs in the form of tickets, etc.
- *Requirements Engineer:* Requirements Engineers communicate with the customer (representative from the Montanuniversität Leoben) and extract features from the requirements.

3.2.2 Technical Description

The basic architecture of the software follows the REST [1] principle. A web client, implemented as a browser application is communicating with a remote backend server, providing all data.

The software is implemented using open source tools solely and had its kickoff in 2014. In the following the technological stack will be described:

Database

For development purposes a H2 Database (DB) [2] is used. Being an in memory database H2 brings the advantage of easy setup and teardown, without having to install any database software on the developing machine.

In production persisting the data is managed by a PostgreSQL DB [3] instance. PostgreSQL is a powerful opensource database which offers high compatibility and tool support and does allow queries on geographical data too.

For the purpose of keeping DB schemata consistent, DB versioning is done by Liquibase [4]. This software allows to describe each database change in a file and auto prepares DB instances on boot-up to comply with the newest schema.

[1] https://en.wikipedia.org/wiki/Representational_state_transfer
[2] http://www.h2database.com
[3] https://www.postgresql.org/
[4] www.liquibase.org

Backend

The backend is implemented using Java [5], a programming language used in most basic programming courses at the TU [6] and the Spring Boot [7] Framework, which is a widely spread all-purpose web framework.

Apache Maven [8] is used for building the application and for dependency management, because of its prevalence and high plugin availability.

Test are written using the JUnit Framework, which is well integrated in the given stack and known by most of the students.

Frontend

The web application, executed in the browser, is developed using JavaScript [9] and the Model View Whatever (MV*) framework AngularJS [10], which, in the rapidly changing world of JavaScript (JS) frameworks, seemed promising in terms of long time support. The growing community additionally offered lots of pre-written components that could be used directly and accelerated development.

The visual representation is handled by the Cascading Style Sheets (CSS) framework Bootstrap [11], which offers easy styling of the website using predefined CSS classes and responsive behavior out of the box.

JS dependencies are managed by the Node Package Manager (NPM) [12] and GRUNT [13] is used for building the web-application.

Acceptance Testing

Since it was the goal to write acceptance tests for each use case, an appropriate browser test framework had to be selected.

Ultimately the Groovy [14] based Geb[15] browser automation solution was chosen in combination with the Spock[16] testing and specification framework which also supports the Groovy language. In comparison to Java Groovy is less verbose and not bound to several restrictions which make Java cumbersome for testing (e.g. method names,..)

Development Tools

- *Redmine*: The OpenSource software Redmine is used as ticket managing system and issue tracker.

[5] https://www.java.com/
[6] curriculum 2017
[7] https://projects.spring.io/spring-boot/
[8] https://maven.apache.org/
[9] https://www.javascript.com
[10] https://angularjs.org/
[11] getbootstrap.com
[12] https://www.npmjs.com
[13] https://gruntjs.com/
[14] http://groovy-lang.org/
[15] http://www.gebish.org/
[16] http://spockframework.org/

- *git*:used as Version Control System (VCS) and SCM
- *gerrit*: used for code reviews
- *Jenkins*: used to build the software
- *Squash TM*: used to to maintain the test plan and keep track of the manually executed tests.

3.2.3 Project Description

Mineralbay is developed using a SCRUM like software development process:

Current Development Process

The Mineralbay development process however significantly adapts the SCRUM blueprint, making it more appropriate for student projects.

The biggest difference is the removal of dailies. Implementing these daily meetings would be almost impossible, since developers (being students themselves), have vastly differing timetables, making it hard to find a timeslot.

However weekly Jour Fixes take place, which try to compensate for the lack of dailies, but tackle other tasks as well.

- *Status:* Students present their progress on the tasks they are working on, discussing occurring problems
- *Sprint planing:* (if a sprint has ended): Tasks are taken from the backlog, teams are assembled and given tasks.
- *Sprint retros:* Retrospectives for the purpose of finding out what could be improved, etc.
- *Feature estimations:* New features are presented by the requirements engineers and estimated by the developers.

Sprints usually differ in length depending on features being developed.

Developers form teams of twos and always develop in form of pair programming. Some students prefer to meet physically, while others use chat tools, VoIP and screen sharing, meeting in a virtual way.

3.3 Research Questions

In order to gain a result the following research question will be answered:

As basis for the following research questions 3 levels of the software development process (inspired by [20]) will be described:

- **Process level**
  Elements and sub-processes of the specific software development process.
- **Coding level**
  Errors (according to the definition in [20, p. 16]) are made by programmers. More specifically, they make these errors during the process of writing code, which is covered by this level.
- **Documentation level**
  Definitions, instructions, documentation and all other kinds of knowledge that accumulate during the project.

Table 3.1 provides an overview of the research questions which will be discussed in detail in the following:

Level	**Improves Software Quality?**
Process	Introducing reviews?
	Setting up software versioning?
	Self assessments for team building?
	Test software in a production like environment?
	Gamification as tool for improving software quality?
Coding	Unit testing as review criteria?
	Test quality and coverage?
	Successful local test run before pushing?
	CheckStyle without errors?
	Introduce code quality gates?
Documentation	Acceptance tests provided with requirements?

Table 3.1: Research questions overview

Status quo

At the moment the quality gates in the development process are defined in a Definition of Done (DoD) (i.e. a checklist of goals) for each feature. In the current process however, as early evidence suggests, developers often chose not to comply to (parts of) this DoD because of the additional workload. In the following, measures aimed at improving the overall software quality will be proposed, categorized by the levels described. The provided measures will seek to increase the compliance to the DoD on the one hand and introduce new quality-boosting factors on the other:

3.3.1 RQ1: On Process level

Which advanced techniques aimed at software QA can be applied and implemented in a typical student software project?

1. Reviews
   Current State: The current process includes peer and management reviews: Every commit should be reviewed by another developer before it is accepted in the code base. Before merging into the master branch, another review is performed by an assistant. However both reviews are usually neglected, resulting in un-reviewed patchsets in the codebase.
   Hypothesis: A sound and easy to follow review process will help knowledge transfer as well as improve code quality.
   Evaluation: A controlled experience will be performed, testing the proposed setup. Participant observation combined with a subsequent interview and survey will identify its impacts.

2. Versioning
   Current State: In the current development process the number of the project version is never increased. This often leads to confusion concerning software versions deployed on the production or test server and local environment.
   Hypothesis: Introducing a tailored versioning scheme and additionally visualizing development progress by displaying the software version number on the website, will enhance traceability and enable users and developers to map certain features, bugs and changes to versions of the product.
   Evaluation: Stakeholders will be interviewed about the implemented versioning process.

3. Self assessments for team building
   Current State: At this time team building is done mostly by the project leader with occasional exchange of team members.
   Hypothesis: Performing self assessments rating a developers skills in different areas of development as basis for team building (i.e. teams will be built by developers complementing each other) will improve learning experience and code quality.
   Evaluation: Team members will be interviewed about experiences with this kind of team building technique as well as learning experience, and the progress made during the development sessions.

4. Test software in a production like environment
   Current State: This is important, because in the production environment a different database is used and frontend code is uglyfied/minified. However developers often neglect this step because they lack knowledge on how to do so, or simply due to convenience.
   Hypothesis: Production mode tests will be part of the DoD and review process. This will

reduce the number of bugs, which are specifically dependent on environment and might slip into production.
Evaluation: Number of errors that only occur in production will be measured and compared to previous data. Developers will be interviewed about their opinion concerning the testing effort and impact.

5. Gamification as tool for improving software quality
Hypothesis: Tasks dedicated to improving software quality usually aren't attractive to students. Using a gamification approach for quality related tasks, rewarding positive actions, will lead to better overall quality.
Evaluation: Team members will be interviewed whether they felt motivated by the gamification tool or not.

3.3.2 RQ2: On Coding level

How to include quality enhancing methods in the process of the student writing code in a typical student software project?

1. Unit tests written and successfully executed
Current State: Currently, unit testing (Part of DoD) is handled as a kind of a gentleman's agreement. If time runs out during the sprints it is often neglected.
Hypothesis: Checking the submitted source code for sufficient tests will be part of the review process. Unit testing the software will improve quality and reduce the number of errors found during testing and in production.
Evaluation: Bug reports which are linked to faulty calculations will be analyzed.

2. Test quality and coverage
Current State: The current goal of 50 % line coverage has not been reached. Coverage scans are done scarcely and complex code can reach production without being unit tested properly. Furthermore code coverage alone does not necessarily lead to well tested code.
Hypothesis: Introducing meaningful test criteria (i.e. mutation testing for test suits) will improve quality and reduce the number of errors found during testing and in production. A Sonar server will be integrated in the build chain ensuring that code coverage is met.
Evaluation: Bug reports that are linked to faulty calculations will be analyzed.

3. Successful local build and test run
Current State: Developers often skip the local test execution (Part of DoD) and abuse the build server as test execution server, thus blocking this resource unnecessarily and causing unnecessary build failures.
Hypothesis: A tool that creates a ranking on how often a single developer breaks the build, thus utilizing gamification to address this matter, will lower the number of broken builds.
Evaluation: After conducting an experiment, developers will be interviewed, whether they felt motivated by the ranking. Furthermore the number of broken build will be compared.

4. CheckStyle without errors
Current State: There is no style guide for the Mineralbay project (although part of DoD), allowing the developers to, for example switch between CamelCase and snake_case which has already caused trouble, by breaking tests.
Hypothesis: Implementing a code style guideline and including it in the developers IDEs as well as the code quality server will improve readability, fault tolerance, etc.
Evaluation: Developers will be interviewed about their opinion concerning the code style guideline and its impact on readability, fault tolerance, etc.

5. Introduce code quality gates
   Current State: The only code analysis done in the current process is by hand during the code reviews.
   Hypothesis: Defining and enforcing adequate quality gates by a code quality analysis server will improve code quality and maintenance.
   Evaluation: Developers will be interviewed about their opinion concerning the code quality gates and their impact.

3.3.3 RQ3: On Documentation level

How to improve software quality of a typical student software project by enhancing documentation and sharing knowledge?

1. Acceptance tests written and fulfilled
   Current State: Acceptance tests are written by developers, which therefore also decide about covered use cases. As a result test coverage usually is not as high as intended and regression is not detected.
   Hypothesis: Acceptance criteria should be defined in requirements. These will improve the tests and help developers understand the given task.
   Evaluation: Interviews will reveal if the developer felt supported by provided acceptance criteria.

3.4 Study Design

In this section, the chosen research method, being a case study will be elaborated in detail, as well as why this type of study has been chosen. Furthermore the case study design used in this thesis will be presented, as well as the methods used for data collection and analysis combined with the handling of validity threats.

3.4.1 Case Study Rationales

In order to answer the research questions the case study approach in combination with participatory action research was chosen as fundamental research methodology for this thesis.

According to Yin [118] (covered in detail in 2.2) a case study is an empirical method for "investigating contemporary phenomena in their context" where "the boundary between the phenomenon and its context may be unclear". Furthermore unlike experiments, no control of behavioral events is required [118]. According to [94] this makes it well suited for a complex, ongoing phenomenon like a software engineering project.

Yin states that case studies are particularly suitable to answer "how" and "why" questions [118, p. 9] and Schramm notes that they "illuminate a decision, or set of decisions: why they were taken, how they were implemented and with what result" [97].

Accordingly the case study approach can be seen as well fitting to answer the established research questions 3.3, as they mainly are of the "how" type and their subquestions ask for "how" and "why" as well.

Moreover a participatory action research approach was taken too, since the author actively participated in the underlying software project and did not restrict himself to passive observation.

3.4.2 Single-Case Design Rationales

Yin lists several scenarios when a single-case study is an appropriate design [118, p. 47]:

1. *Critical case:* The case under study is meeting all of the conditions for testing a well-formulated (has specified a clear set of propositions as well as the circumstances in which the propositions are believed to be true) theory.
2. *Extreme or unique case:* The phenomenon is so rare that any single case is worth documenting and analyzing.
3. *Representative or typical case:* The study of a typical, standard example of a wider category (a "typical" project among many different projects, etc.).
4. *Revelatory case:* The phenomenon was previously not accessible for scientific investigation.
5. *Longitudinal case:* The same single case is studied at two or more different points in time.

The Mineralbay project is both typical for a long running university project (rationale 3 - 3.5), as well as unique in its constellation (rationale 2):

While some of the characteristics that make the Mineralbay project extreme and unique are already described in 3.1, additional ones can be listed:

- Mineralbay is industry-related: Besides developing skills, domain knowledge is also required.
- Real customer: The project is used as a show-case to real domain experts.

Propositions

According to [118, p. 37ff], propositions are not mandatory for merely exploratory case studies. However specific propositions increase the likelihood of researchers being able to place limits on the scope of the study and direct the attention on topics of interest. They may also reveal any bias on the part of the researcher and thus improve objectivity [7, p. 10] [118, p. 37ff].

Consequently based on the theoretical frame of reference mentioned in chapter 2 and personal experience a set of propositions was compiled. In the following these propositions are linked to the related research questions and sub-questions, respectively:

- **RQ1: Process level**

  4. **Test software in a production like environment:** Developers will not neglect testing the software in a production like environment if it requires minimal effort.
  5. **Gamification as tool for improving software quality:** Gamification motivates developers to do otherwise tiresome code quality related tasks.

- **RQ2: Coding level**

  1. **Unit tests written and successfully executed:** A sound CI setup helps improving code quality.
  1. **Unit tests written and successfully executed:** Developers will accept the CI setup
  3. **Successful local build and test run:** Gamification can be utilized to prevent students from unwanted behavior by introducing a ranking board.
  4. **CheckStyle without errors:** Developers will accept provided tools and standards for development

- **RQ3: Documentation level**

  1. **Acceptance tests written and fulfilled:** It is possible to define acceptance criteria in the requirements.
  1. **Acceptance tests written and fulfilled:** Acceptance criteria helps developers understand their given task.

3.4.3 Case Study Design Overview

As described in Chapter 2.2 conducting a case study requires careful planning. In detail, the case study process (2.2.3) starts with the case study design. For this study a holistic single case design was chosen (the reasoning provided in 3.4.2), with Mineralbay being the case under study. In the following the study design will be described for this thesis:

Yin [118] and Robson [90] both provided guidelines for designing a case study. The case study design will be described based on these guidelines following Robsons naming, but referencing Yins work as well.

Objective - what to achieve?

The objective is to provide enhancements for the QA process in agile student projects, based on the Mineralbay case.

The main purpose of this thesis according to the classification of [90] will be an improving one. I.e. trying to improve the QA process in the studied case. Secondarily exploratory and descriptive goals will be covered.

The propositions, have been listed in chapter 3.4.2

Research questions - what to know?

A list of research questions has been compiled. A detailed description is given in Section 3.3.

The case - what is studied?

The case will be the Mineralbay project, including software, people and process. The case study will be of holistic nature since the case is studied as a whole [118]. A precise description of the case can be found in Chapter 3.2.

Theory - frame of reference

An extensive review of the literature has been performed. Its results are captured in chapter 2.

Methods - how to collect data?

Most of the data collected will be of qualitative nature (group interviews, one-on-one interviews, participant observation), but also quantitative information (questionnaires, generated data, Code Metrics, etc.) will be gathered. Whenever possible data (source) triangulation, i.e considering multiple data sources will be used to come to an result. Data collection is described in detail in 3.4.4.

Selection strategy - where to seek data?

The unit(s) of analysis for this case study are the developers of Mineralbay as well as the produced artifacts. A detailed description is given in 4.4.

3.4.4 Data Collection

Several types of data collection were used during this study, for the purpose of triangulation. The methods of data collection, based on the definitions provided in 2.2.3 are listed in the following:

- *Questionnaire*: This method proved to be convenient for data collection, since the participants could take part at any time from any place. Furthermore due to partitioning, assumptions can be made based on partitions, not individuals.
- *One-on-One Interviews*: Semi structured one-on-one interview were used to gain insights inside the subjects individual experiences.

- *Focus Groups / Group Interviews*: Certain questions could benefit highly from the characteristics of group interviews (2.2.3), especially the interviewees complementing each other.
- *Code Metrics*: The source code of the software, parts of it, or just a single commit can be analysed and metrics generated. A description of the used metrics is provided in 2.7.3
- *Analysis of Electronic Databases of Work Performed*: Software systems like the ticketing system in use (Redmine), or Static Code Analysis Tools (SonarQube), etc. provide useful data, ready for analysis.

3.4.5 Threats to Validity

Validity in case studies, determines the trustworthiness of results, as outlined in 2.2.3.

The classification scheme of Yin [118, p. 40f] distinguishes between four aspects of validity: *Construct Validity*, *Internal Validity*, *External Validity* and *Reliability*.

In the following section the validity of this study will be addressed according to Yins classification scheme:

Construct Validity

Identifying correct operational measures for the concepts being studied

Throughout the process of this study, the following issues were identified as the biggest threats to construct validity:

Subjective Bias:

Subjective bias describes "decision making or evaluation based on personal, poorly measurable, and unverifiable data or feelings" [26, p. 119]. In the context of a case study this is most likely the bias towards confirming the researchers assumptions. In [30] Flyvbjerg states that case studies are often alleged that they "allow more room for the researcher's subjective and arbitrary judgment than other methods"[30, p. 18], but ultimately comes to the result that case studies contain "no greater bias toward verification of the researcher's preconceived notions than other methods of inquiry" [30, p. 21].

In this thesis *Interviewer bias*, being the "systematic difference between how information is solicited, recorded, or interpreted" [78, p. 3], was found to be a possible issue. Recording and subsequent transcription and analysis, as well as follow up questions/interviews were used as countermeasures. Making use of observer triangulation (i.e. the use of different investigators), was not possible due to a lack of resources.

Furthermore in order to mitigate the risk of subjective bias, datasource triangulation (considering multiple datasources), along with theory triangulation (i.e. multiple perspectives on the same data set) were applied.

Yin also recommends establishing a clear chain of evidence, an advise the author tried to follow throughout this thesis.

Response Bias:

Non-response to questionnaires (as well a non participation in interviews) may potentially bias the results. This is can be explained, as those who participate in such interviews or respond to such questionnaires may differ in some systematic way from those who don't (non-respondents) (e.g. particular groups are unrepresented in the sample) [40] [67].

Countering this risk the participants were segmented based on a set of general demographic questions:

- Skill level on relevant technologies
- Project experience
- Work experience
- Attended Study Program

Language Barriers :

Since most of the students were enrolled in a masters program, which is held in English, some of them were not fluent in German and sometimes even had difficulties explaining their point of view in English. However, as pointed out in [67, p. 3], such language barriers must not result in the exclusion of the subject, as this can bias the results.

In order to allow participation for all project members, the questionnaires were written in English and interview questions asked depending on the interviewee.

Project Member Fluctuation:

Being one of the project immanent specifics, as outlined in 3.1, developer fluctuation is quite high in the project. Long term project members of course are ideal subjects for this study, since they can reflect on previous iterations and compare the situation with the current one. However developers usually tend to leave the project when having reached this level of experience, because they have completed their work quota.

Again segmentation can be used to avoid dependency on a certain attendee. Frequent collection of data can also lower the risk.

Data Collection

Finding the correct data collection techniques for each inquiry is not trivial.

Since all subjects participate in their spare time and do not benefit from participation it is important not to be too demanding with data collection and overstretch their patience.

A tradeoff will have to be made, in order to collect as much data as possible, without costing participants too much time and effort.

However methods of data collection have to be evaluated carefully for each inquiry. Triangulation will be used for the purpose of not relying on a single data source.

Internal Validity

Since according to Yin [118] internal validity is only an issue for explanatory or causal studies and not for descriptive or exploratory studies and the underlying study being from the latter type (i.e. exploratory) it does not have to be considered.

External Validity

Assuring external validity or generalisability is always an issue with single case studies.

Statistic generalization can not be applied, instead generalizing the results is recommended to be done by analytic generalization [118].

Furthermore according to Eckstein single case studies attempt to generalize by testing hypotheses [30, p. 10], which have been described for this study 3.4.2

In order to allow findings to be generalized for a typical student software project, an expert evaluation will be performed, for the purpose of identifying the case as a typical student project.

Reliability

Reliability of the study is assured by collecting study protocols, providing and versioning all collected data and document each step carefully. All data necessary for reconstructing the study will be kept available.

3.5 Establishing Case Representativeness

3.5.1 Description and Goal

In order for the thesis to gain general validity, it has to be assured that the underlying case, i.e. the student project Mineralbay, resembles a typical student project, i.e. is a representative or typical case 3.4.

For this purpose, advisors, teaching at software engineering classes at the TU were interviewed, aiming at categorizing Mineralbay as a typical student project, with all its problems and characteristics A.8.

3.5.2 Expert Interviews

The experts participating in the interviews had gathered experience in supervising student projects for 6 - 20 semesters with an average of 9 semesters.

In a first step the interviewees were questioned about the characteristics of a typical student software project in their experience: They stated that in a typical student software project:

- The teams are made up from 4-6 students
- Used software methodologies are agile, iterative paradigms, like SCRUM.
- Communication is done mostly online, but physical meetings take place too.
- There is a weekly meeting with the supervisor, but students also meet on their own to develop or do project planing
- The customer is most likely fictional and represented by an university employee in bachelor courses. In advanced courses however it usually is a "real" customer, in most cases somehow accustomed to one or more of the team members. It might be also the case, that the students develop their own software , e.g. when planning to do a startup.
- students do the requirements engineering. In basic courses the requirements are sometimes given and precompiled.
- The decision of code reviews being carried through depends largely on the group and on the supervisor. Some mentors require the students to do code reviews, some groups do so by themselves. Other interviewees however stated that this is rarely the case.
- Versioning is often predefined and determined by given milestones.
- Pair programming is done by some groups but usually not mandatory. Some supervisors encourage their groups to do so.
- Testing depends on the specific group and the course as well. Sometimes Test Driven Development (TDD) is mandatory, sometimes Behavior Driven Development (BDD) is done and in other cases only unit testing is required.
- Goals concerning code coverage are usually not dictated. If so they are between 70 and 80 percent of the business logic.
- Groups have to pick a code style for their project and follow this style throughout development.

- There are students that greatly care about high quality code. Interviewees however stated that the majority of the developers care most about working software. Code quality becomes less of a goal towards deadlines at the end of the project.
- Time usually is not an issue. However, especially at the beginning students have difficulties estimating the amount of work they can do in a sprint. Additionally bad time management, resulting in "crunshing" before deadlines is often an issue.
- The teaching efforts of the courses are aimed at learning team software development skills, trying out new technologies, but also teaching practical software engineering knowledge and providing insights into the work of a software engineer in a team.
- The usual amount of work lies between 200 and 250 hours per semester.
- Students usually participate because they need the degree, or the course is mandatory.
- Skill distribution inside teams varies largely and depends on multiple factors, e.g. if the students know each other (and how well), if the group is put together randomly, etc.
- Groups consisting of both, bachelor and master students are quite common.

After collecting benchmark data of a typical student project for the purpose of building a reference, the Mineralbay project was described to the interviewees in detail. Project setup, development process and practices were presented and the participants remaining questions answered. In the following, domain experts were asked which aspects of a typical student software project were covered by Mineralbay and which were not.

All of the interviewees stated that, based on its characteristics, Mineralbay generally represents a typical student project. Nevertheless multiple factors have been remarked that make this project unique (and thus differs from a typical student software project):

- "The project is about 1,5 to 2 times the size of an average student software project"
- "The situation that the customer is a total stranger to the students adds difficulty"
- "Requirements engineering will take more time and effort in Mineralbay"
- "There is more work in terms of requirements engineering"
- "Students will not be familiar with the domain."

3.5.3 Conclusion

Based on the results of the interviews it can be concluded that Mineralbay can be categorized as a representative of a typical student software project.

Although some differences have been pointed out by the participating supervisors between Mineralbay and the student projects familiar to them, none of them threaten the validity of this case study.

4 The Case Study

The following chapter describes the case study's scope and execution for the purpose of documenting how results were achieved. In the beginning an analysis of the underlying software project (focusing on QA) prior to the actions taken in the course of this study will be given. Another section will focus on the measures that have been elaborated in order to improve the QA process. These will be described in detail as emphasizing their impact on software quality. The chapter concludes with a summary on how data was collected.

4.1 Status Quo

The following section aims at describing the project situation, prior to the changes introduced in the course of this thesis, both from a technical quantitative view, as well as from a subjective qualitative view. The informational background of the project will not be discussed, as it is already provided in Chapter 3.2

4.1.1 Project Evaluation

The following section offers insights in the software project concerning its size in numbers.

Project Size

As of October 2017 the software project had 29000 Lines of Code (LOC):

- 19000 Java
- 10000 Javascript

Configuration and translation files are ignored in these numbers.

The following charts show the project growth since its beginning. These charts were generated evaluating the git repository, using the OpenSource tool gitstats [1].

[1] https://github.com/hoxu/gitstats

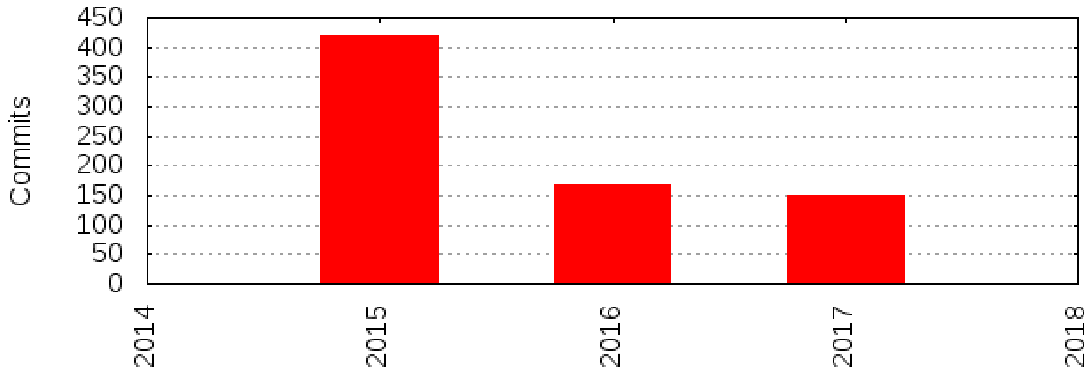


Figure 4.1: Project commits by year

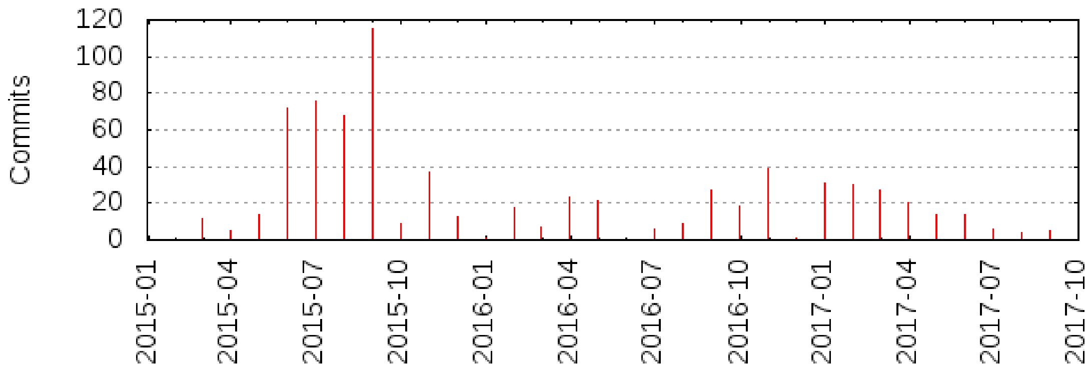


Figure 4.2: Project commits by year and month

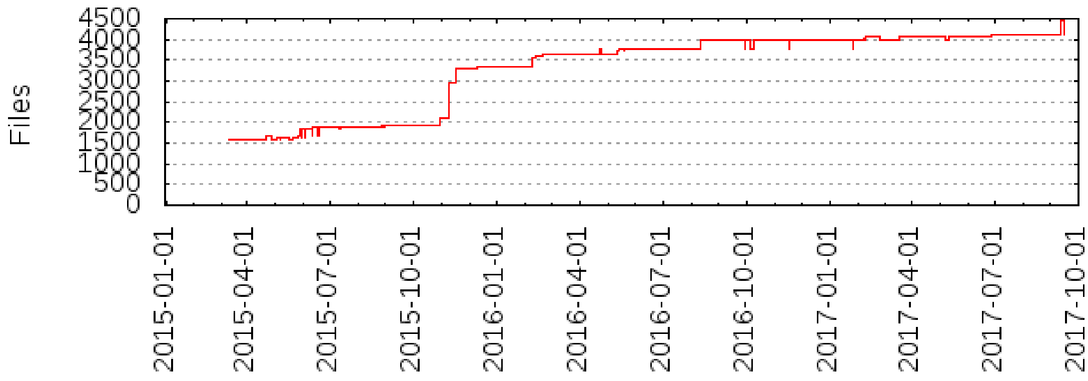
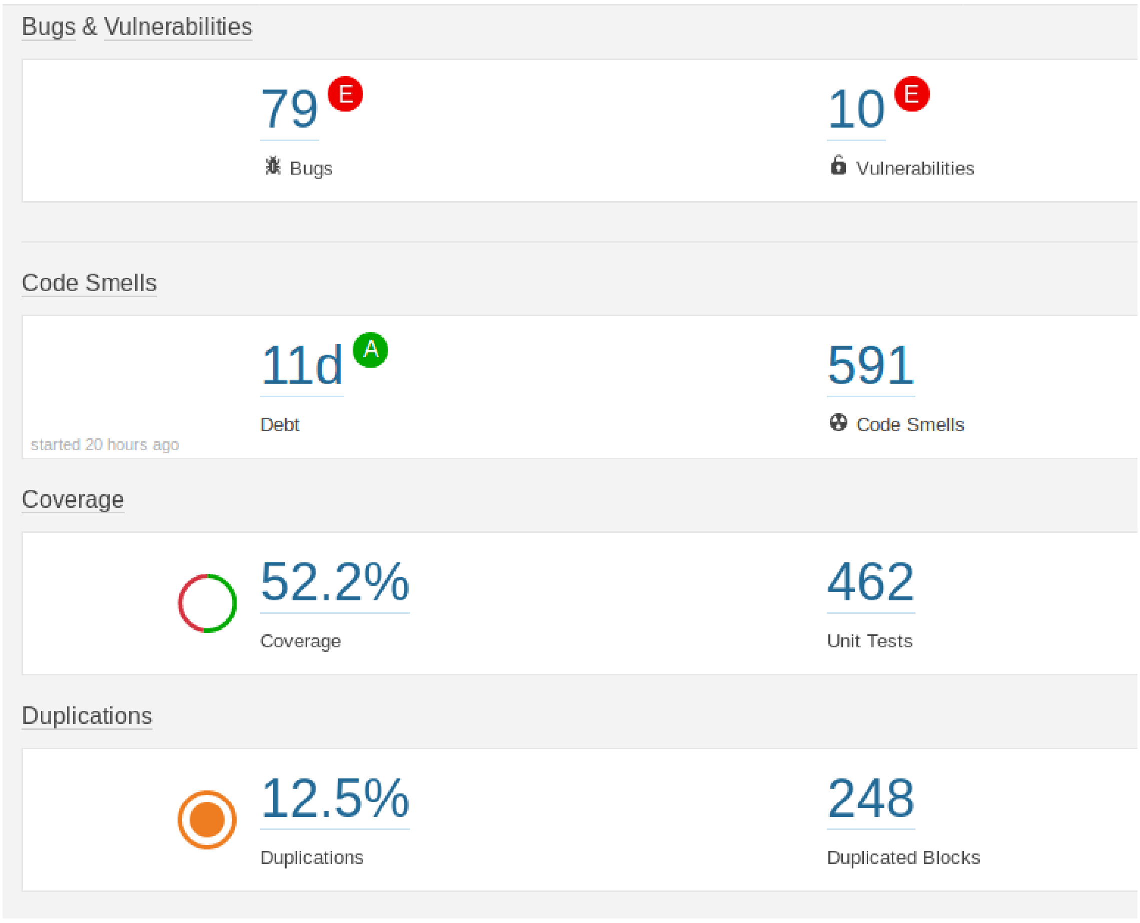


Figure 4.3: Project Number of Files by date

4.1.2 Static Code Analysis Results

For the purpose of achieving a solely technical point of view on code quality a one time static code analysis was performed. The tool SonarQube [2] revealed the following results, after executing an analysis with the source code of the software:


Bugs & Vulnerabilities

79 E
Bugs

10 E
Vulnerabilities

Code Smells

11d A
Debt

591
Code Smells

started 20 hours ago

Coverage

52.2%
Coverage

462
Unit Tests

Duplications

12.5%
Duplications

248
Duplicated Blocks

Figure 4.4: SonarQube Report

The code coverage is not measured by SonarQube directly. For the purpose of doing so, the Open-Source Tool JaCoCo [3] is utilized, calculating the Line Coverage of the source code. Furthermore, since only Java code is tested on a code basis, other languages, like JS are ignored in the results.

4.2 Initial Observations towards Quality

In order to identify possible quality deficits and collect the individual opinions of project members, in the beginning of this study developers were asked to take part in a questionnaire (listed in Appendix A.1), concerning software quality of the project.

Afterwards a follow-up semi structured interview was conducted, gathering additional insights.

[2] Used in its version 6.5 http://www.sonarqube.org
[3] JaCoCo - Java Code Coverage Library http://www.jacoco.org

4.2.1 Quality Questionnaire

The questionnaire started with generic demographic questions, enabling segmentation of participants later on. The second part focused on the participants thoughts about the current software and process quality, whereas the third part asked about their opinion concerning possible improvements.

The form was managed on Google Forms [4], which allowed to send invitations directly via mail.

Evaluation

The results of the questionnaire provided interesting insights concerning the developers point of view on software quality and moreover confirmed some of the authors hypotheses on the topic as well.

All of the participants were enrolled in a "Software Engineering" program, 60 % in a masters and 40 % in a bachelors program. Most of the students (70 %) were employed next to their studies, having gathered between 1 to 8 years of programming experience.

Concerning overall code quality frontend code was rated considerably worse than backend code 4.5:

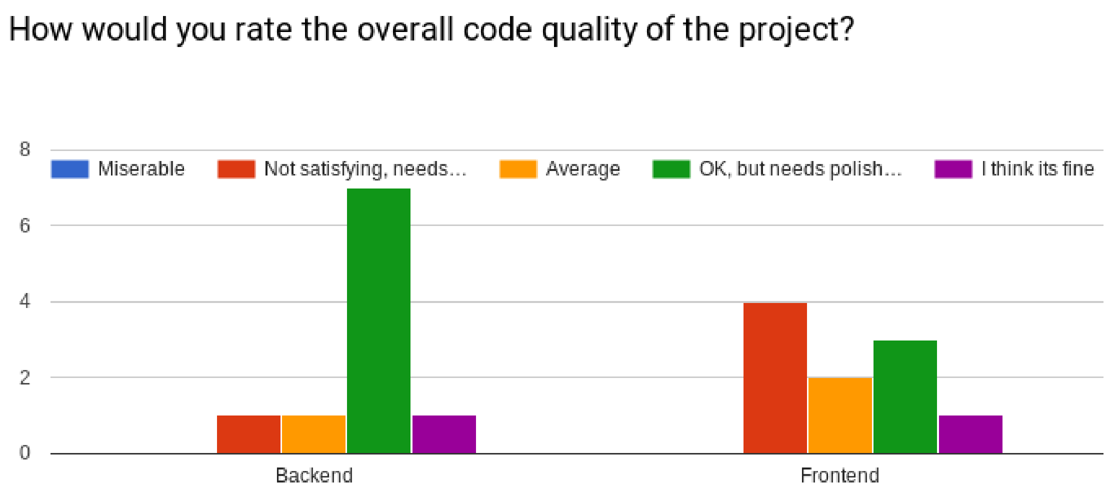


Figure 4.5: Overall Code Quality

It was observed that more experienced students rated the overall code quality [5] worse than less experienced ones. Moreover a possible connection between the number of semesters a student was actively participating in the project and his judgment on code quality could be identified. More experienced developers tended to rate the quality worse than less experienced ones. A similar connection could be established between the quality ratings and the number of years of programming experience the student had already gathered.

[4] https://docs.google.com/forms
[5] Calculated as the arithmetic mean of frontend and backend quality ratings

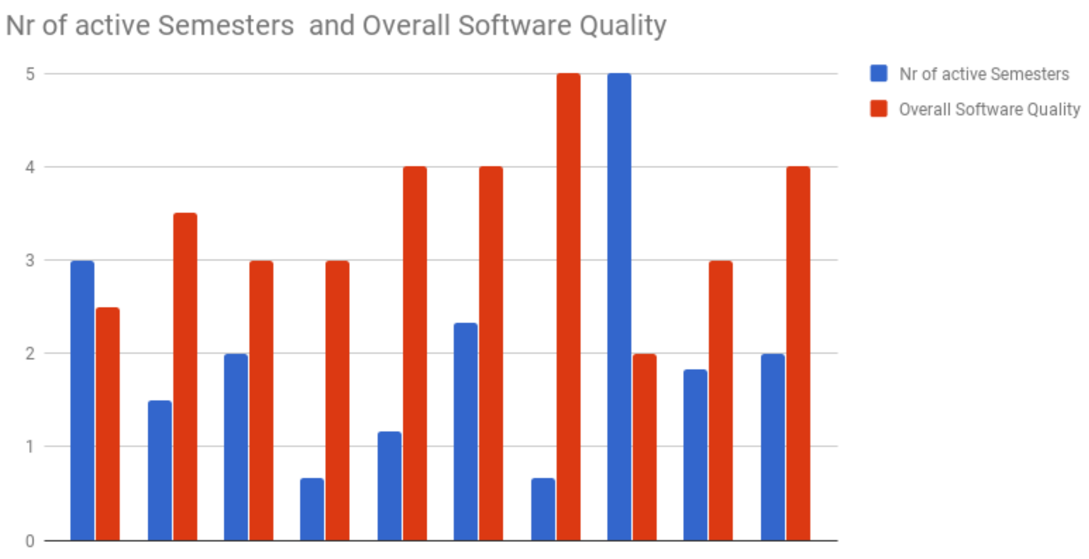


Figure 4.6: Overall Code Quality in Relation to Active Semesters in Project

Test quality was rated rather well, although results on the frontend test were widely spread.

Two thirds of the developers stated that "coding" is responsible for the most defects in the software. All subjects of this subset were experienced developers, employed next to their studies, currently in their masters studies and long term members of the project (average number of semesters being part of the project: 14.5 Semesters).

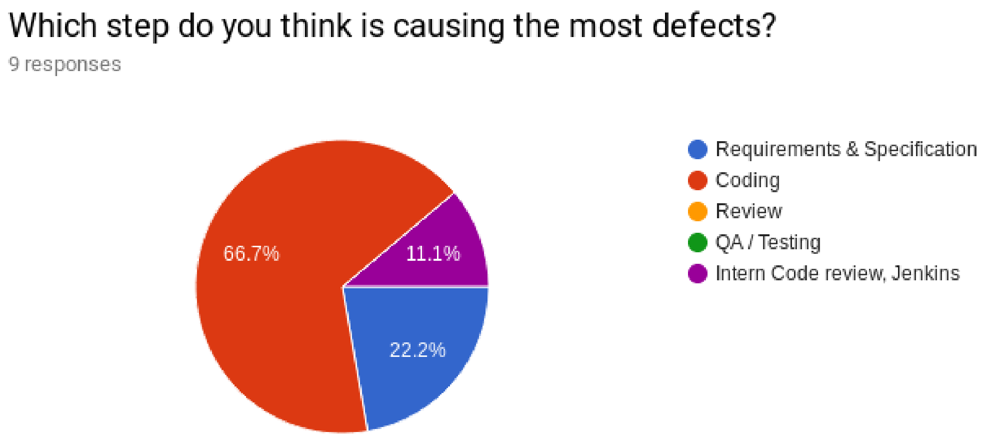


Figure 4.7: Identifying the step causing the most defects

The results regarding a question asking the participants "which feature of the software is in which version" were ambiguous. Answers varied between 1 to 4 on a scale of 5, whereas 5 was equal to knowing perfectly well. No obvious relation could be found between project experience, level of education or any other demographic information and the results of this question.

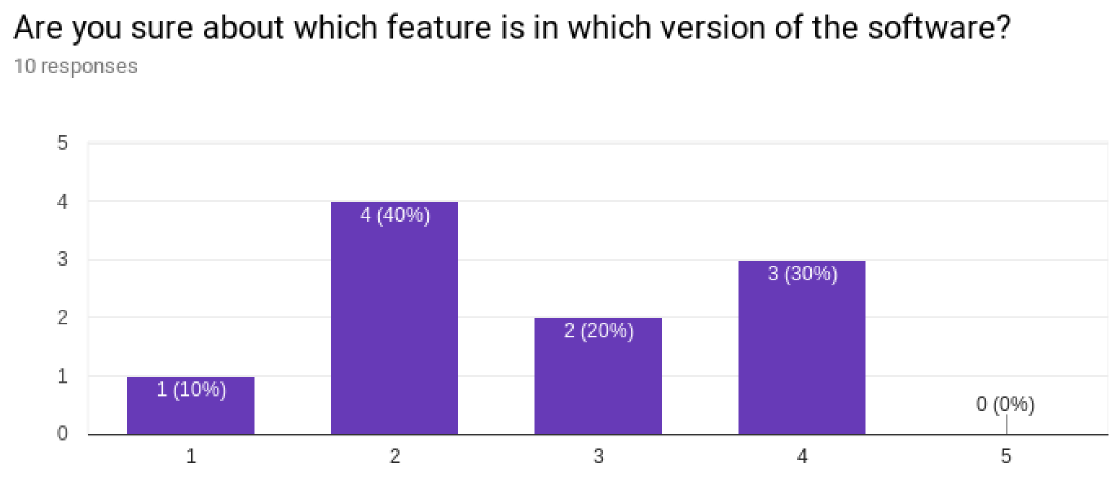


Figure 4.8: Confidence about which feature is in which version of the software

Another interesting insight emerged from questions about testing skills and assessment of own tests. While nearly all participants were confident about their testing skills, many were not happy with the way they are testing their own code.

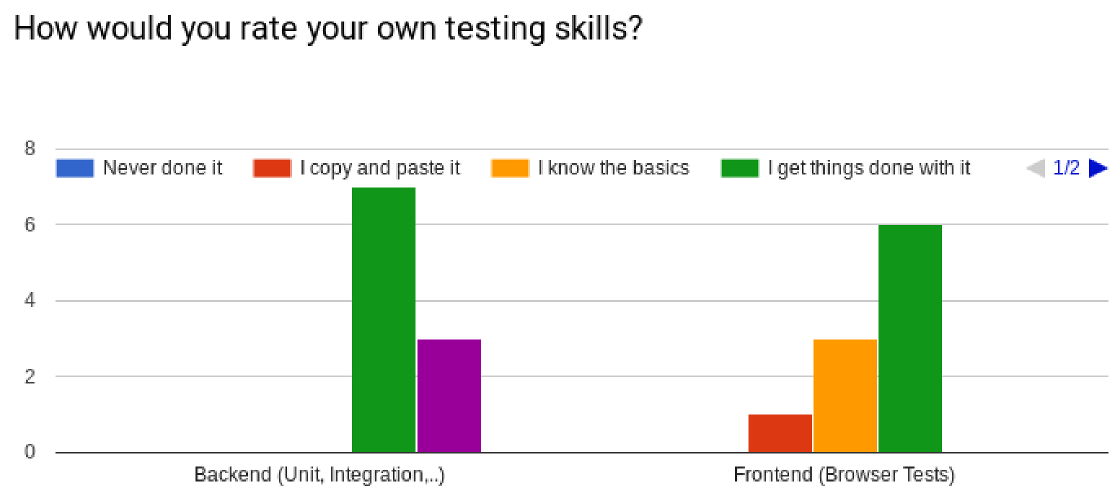


Figure 4.9: Rating of Testing Skills

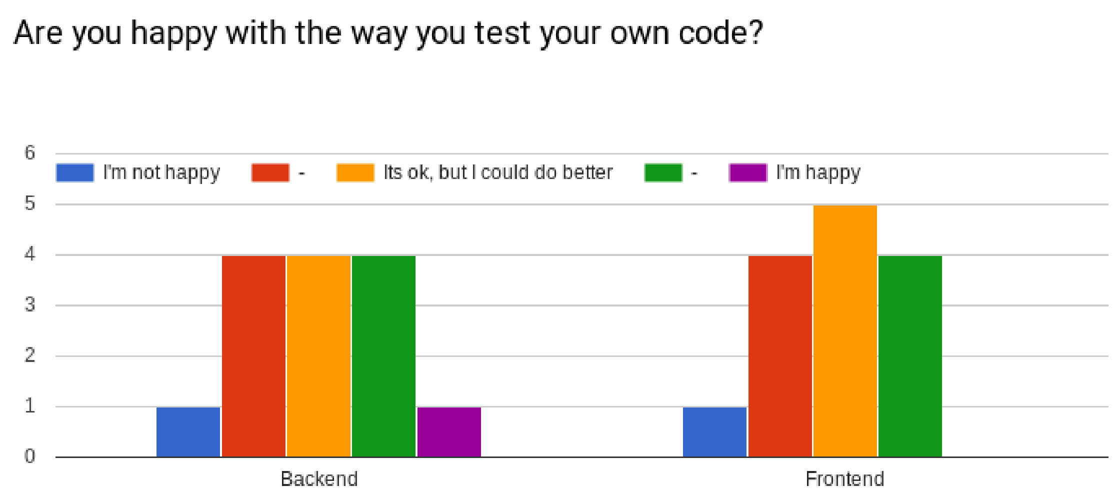


Figure 4.10: Self Assessment of written Tests

Furthermore two thirds of the developers chose to ignore parts of the DOD before pushing their code into the SCM:

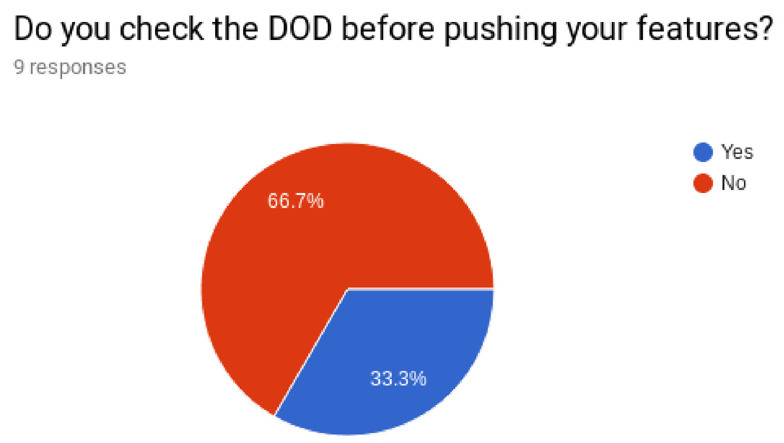


Figure 4.11: Checking of DOD before pushing into SCM

The responses to the question if code quality would benefit from continuous code analysis, were solely positive.

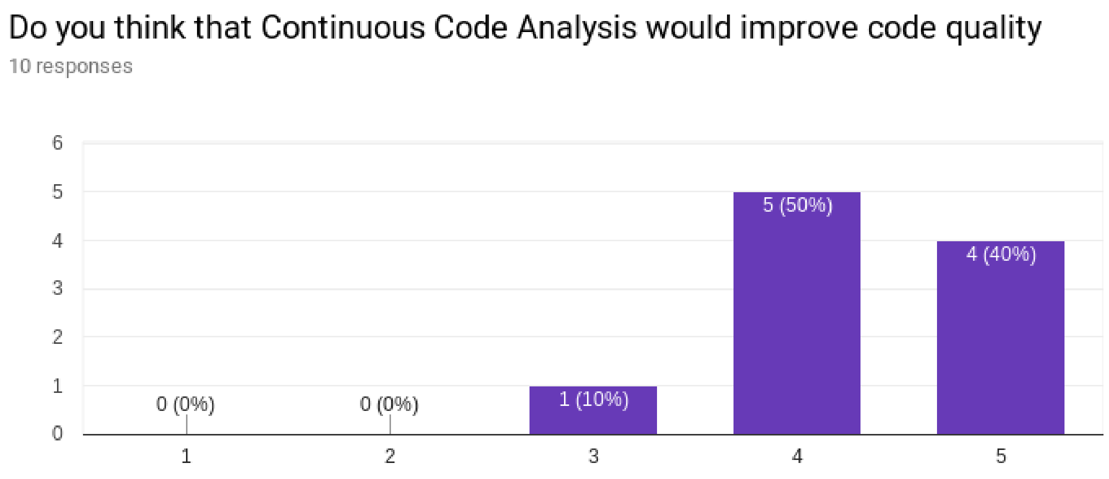


Figure 4.12: Positive effects of static code analysis

4.2.2 Quality Group Interview

After evaluating the initial questionnaire the students were invited to join a group interview/focus group. The main reason for conducting this interview session was that the results of the questionnaire rose some additional questions which needed further explanation.

The interview was carried out within a focus group since the benefits of this type of data collection (as described in Chapter 2.2.3) were suited well for the type of questions asked. Moreover the explanatory character of focus groups was promising in terms of discovering new insights on the matter of quality issues.

A semi structured (described in Chapter 2.2.3) interview setup with open questions was chosen, because event though the author had a number of questions prepared, valuable information due to unexpected turns in the interview, resulting from the flexibility brought by semi structured interviews was sought.

The attendees were selected from the group of developers based on availability. The resulting group can be seen as homogeneous [6] [90, p. 301] The interview was prepared based on the theoretical background described in Chapter 2.2.3 as well as the practical tips found in [90, p. 288]. The whole interview guideline is attached in the Appendix A.2.

Evaluation

After an introduction, the author started the interview session with asking the participants about current development impediments. This offered new insights which were covered in the interview later on in more detail, but also made the whole setting feel much more relaxed and natural immediately.

The 4 developers which took part in the interview will be referenced as P1, P2, P3, P4 in the following.

[6] Have a common background, position or experience

- *Problems in the current code base:*
  - Lots of different coding styles (P2, P1)
  - Refactorings started, but never finished (P1)
  - Different patterns used for same problems (P1)
  - Solutions copied, but also the non satisfying ones (P1, P4)
  - Too many branches (P1)
- *Why are errors introduced:*
  - No more peer programming, due to scheduling issues (P2, P3)
  - Code reviews have been neglected (P2, P3, P4)
- *Problems with versioning:*
  - Developers do not know which versions are deployed (P2, P3)
  - No versioning at all (P2, P3, P4)
- *Current impediments:*
  - Not enough developers (P1, P3)
  - Missing privileges on build server to fix the build
- *Testing:*
  - No new tests written, only adapting existing (P3, P4, P2)
- *Problems with DoD:*
  - Build on CI Server is not working properly. Parts of the DOD are skipped therefore

The interview session revealed many problem and impediments in the current development practices.

The build server has been causing problems frequently and the developers lack the privileges to fix it. This causes them to neglect the build results, possibly leading to regressions, since there is no mechanism to detect them before being merged into the SCM anymore.

The absence of pair programming and the neglect of code reviews might result in low quality code, bugs or other issues finding their way into production.

Since there is no proper versioning and as a result of the developers being unaware of which version is deployed in production, several issues might arise: Foremost of all, bugs and features can not be tracked.

The lack of tests covering newly added code is also an issue.

4.3 Measures

Based on the identified deficits the author compiled and implemented, a set of measures for the purpose of enhancing the QA process and answering the research questions. The following section introduces each measure in detail, describing its implementation as well as the addressed research questions:

4.3.1 SCM Branching Model

This section describes how the branching model was refined.

Rationale

In the current process, due to the limitations and characteristics of a long running student software project, it is often the case, that some features the team has committed itself to in a sprint, were not finished in time. Originally the sprint was simply "extended" until all features were implemented. However this approach is unsuitable for versioning and prolongs time to customer (i.e. prevents delivery of finished features to the customer even though they are ready).

Addressed Research Questions

Establishing a sound branching model is essential for software quality in a development process. Each commit in the SCM contains changes which potentially improve or decrease overall quality, hypothetically even preventing the artifact from functioning at all.

It therefore is mandatory to keep commits, which have not been tested properly from entering the stable code base. Additionally, a branching model must support development and not cause too much overhead for the developers to accept it. It can also be used to enforce quality gates (e.g. stricter gates for each step closer to a production branch) and ease parallel development.

The research questions regarding code quality impact concerned, in particular are:

- Establishing a code review process (Description in Chapter 3.3 RQ 1)
- Establishing a versioning process (Description in Chapter 3.3 RQ 2)
- Testing in a production like environment (Description in Chapter 3.3 RQ 4)

Description

The proposed branching model (as depicted in 4.13) introduces branches for each sprint. These sprint-branches are branched off the master which itself is considered stable and ready for deployment at any given time. Based on the sprint branches, feature branches are created, containing the changes necessary for the implementation of a feature. Upon merging branches back to their origin, code quality gates (e.g. automated tests, reviews) have to be passed. The closer a branch is to the master, the stricter these quality gates become.

Bugfixes will, depending on their size, be merged directly into the sprint branch or a dedicated dedicated bugfix branch, which is treated like a regular feature branch will be created.

If a feature is not finished in time, it will not be merged into the branch of the sprint it was supposed to be part of. Instead the feature will be postponed to the next sprint and thus merged in the according sprint branch upon its completion. This way the information of which feature is included in which sprint is preserved. Time to customer is reduced too, because the sprints output will be merged sooner and delays eliminated.

Before each sprint a developer will be named responsible for overseeing that branching is done according to this model and the feature branch is merged at the end of the sprint.

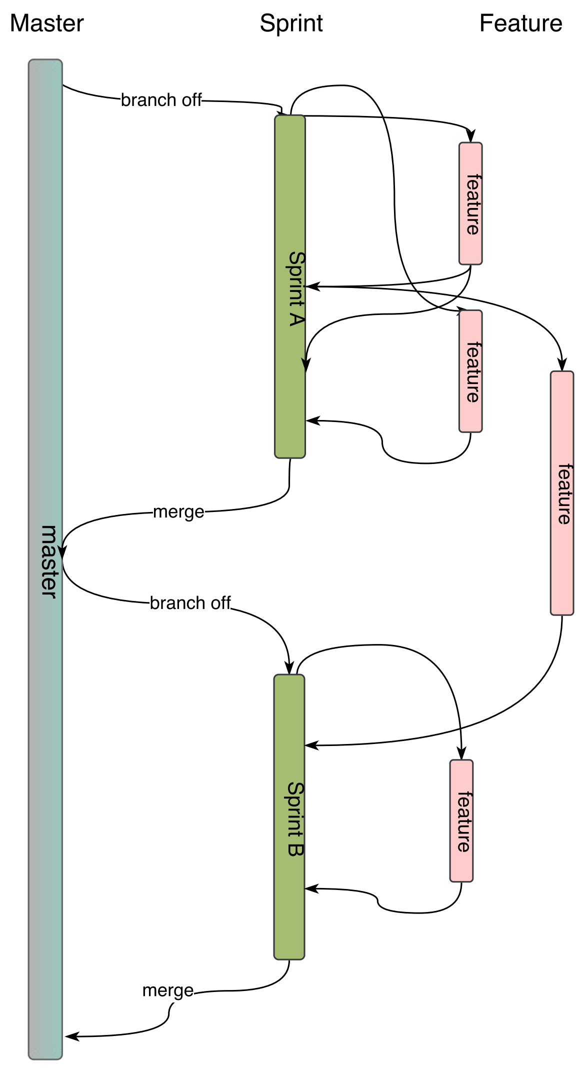


Figure 4.13: SCM Branching Model

4.3.2 Build Migration And Automated Testing

The following section describes how the build infrastructure was migrated from Gerrit/Jenkins to Gitlab/Gitlab CI for the purpose of easing development and improving QA processes.

Rationale

The software project started with a Jenkins build server responsible for building each commit and running tests.

However the server often became unresponsive, or builds unexpectedly stagnated and did not terminate. Furthermore the Jenkins build nodes were not maintained by the development team. This was a major issue, since software required for building the application has to be maintained on the build node as well. Additionally not being able to alter the build nodes bared the way to a desired Development And Operations (DevOps) practice.

Considering the initial build setup, especially the automated frontend tests caused problems, because due to web browsers and frameworks evolving quickly, the driver software on the nodes has to be kept up to date. Since the developers were not authorized to tackle these updates themselves, the infrastructure team had to be tasked, which proved cumbersome.

In the course of this thesis, the build systems was migrated to Gitlab CI, allowing developers to handle the CI setup themselves.

Addressed Research Questions

A sound build process is necessary for producing high quality software. It can be the base for numerous quality gates. Having deficits here, like i.e. slow or unstable builds, might lead to various problems like frustrated developers.

The addressed research questions, in particular are:

- Establishing a code review process (Description in Chapter 3.3 RQ 1): Code reviews are carried out using Gitlab.
- Testing in a production like environment (Description in Chapter 3.3 RQ 4): Gitlab is used for building images for each major version of the software.

Build Pipeline Setup

The configured pipeline is depicted in 4.14 and consists of multiple stages. It is triggered each time a developer pushes new changes into the SCM:

1. **Frontend Build**:
   Description: First the frontend AngularJS app is build in a nodejs environment.
   Result: A bundled Javascript application.
2. **Backend Build**:
   Description: The resulting artifacts of the frontend build are passed to the backend build which compiles the java application and bundles it together with the frontend.
   Result: The result is a deployable Web Archive (WAR) file

3. **Tests**:

   Tests are executed parallel in order to achieve a faster build.

   - Unit Tests:
     Description: Unit tests of the backend application.
     Result: A test report
   - Browser Tests (Multiple Runners):
     Description: Browser tests of the web interface. Since execution can take a while, multiple runners are used, each executing a different test set.
     Result: A test report

4. **Publish Container**: The application is packed in a container and published on the internal registry.

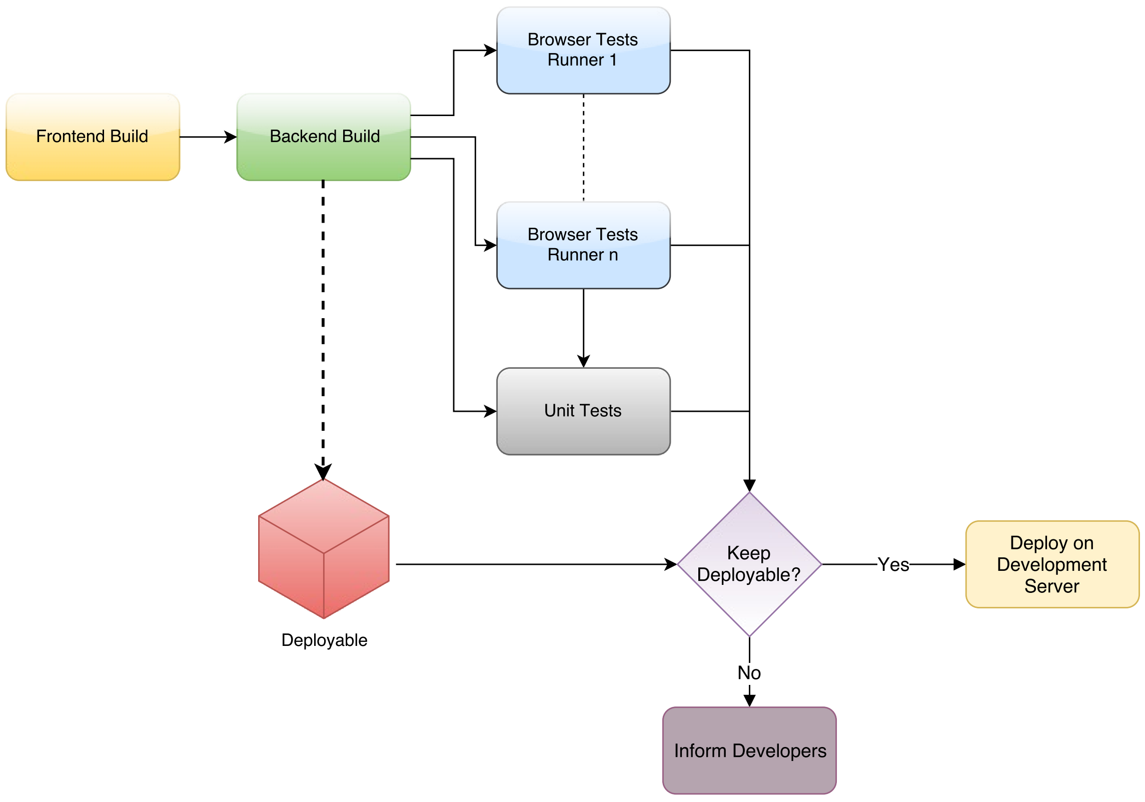


Figure 4.14: Build Pipeline

If the pipeline succeeds the resulting artifact is considered stable, otherwise the developers are informed about a failing build via email.

A future goal is to deploy the release on the development server automatically.

Build Verification

Build Verification prevents erroneous, or even non-compiling code from being submitted into the SCM. Each commit is build on the CI server automatically and checked for defects using automated tests.

The Mineralbay Software is tested using unit tests as well as automated browser tests.

However executing the browser test suite is taking a lot of time. Therefore it has been decided to split the test into multiple sets. These subsets can be run individually and are triggered in different scenarios:

- *Master and Sprint Branch*: All tests are executed. In this case it is most important that the merged features must not introduce any defects or lead to regression.
- *Regular Branch*: Only a core set is executed. The basic functionality of the software must be assured. The general build verifier is depicted in 4.15

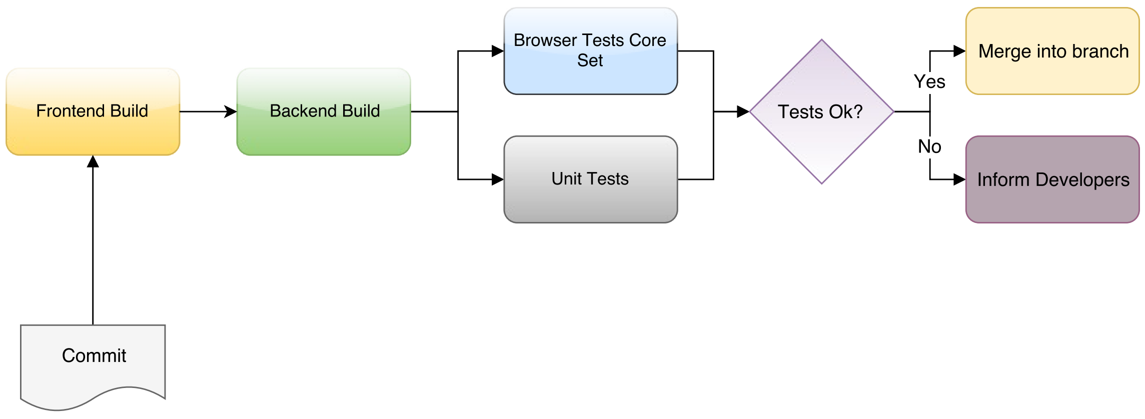


Figure 4.15: Regular Build Verifier

In the previous verification setup, only the backend unit tests were run, missing potential severe errors in the frontend build.

4.3.3 Static Code Analysis

This section describes how static code analysis was introduced into the development setup.

Rationale

Static code analysis is a process best introduced early the a project. In the case of Mineralbay however this was neglected.

Especially since due to being a long-term student project, many different developers were working on the software, resulting in many different code styles and tastes too. Parts of code containing bad smells or styling were often copied and reused, thus spreading throughout the software.

A consistent code style was never enforced. Consequently, performing static code analysis on the project reveals several issues:

- Executing the Java Tool Checkstyle [7] and checking the Mineralbay project against the official Java Codestyle, 9408 errors are reported [8].
- Running JsHint [9] on the JavaScript part of the project[10] revealed 1011 errors

Code quality can be improved if code is formatted in a uniform way and the static source code is analyzed permanently.

The following section will describe how these measures have been implemented.

Addressed Research Questions

The research questions covered by this chapter are:

- Code coverage goals (Description in Chapter 3.3 RQ 2): Code Coverage is measured using SonarQube.
- Introducing code style standards (Description in Chapter 3.3 RQ 4): Code style compliance is determined by SonarQube.
- Introducing code quality gates (Description in Chapter 3.3 RQ 5): Code quality metrics are calculated using SonarQube.

Integration of SonarQube

In order to check the existing sourcecode for weaknesses, the static code analysis tool SonarQube has been set up and included in the development process.

Each time a branch is merged into the master, the sonar task is executed, rating the code and listing improvements or deteriorations.

[7] http://checkstyle.sourceforge.net/
[8] As of 12.11.2017, only Java Code was checked
[9] http://jshint.com/about/
[10] find . -name '*.js' -print0 | xargs -0 jslint

SonarQube allows for quality gates to be configured. These are a set of rules (i.e. line coverage must be higher than 60 %, no bugs of level "blocker" present), based on the metrics established by SonarQube. If a build does not pass these quality gates it can be considered failed.

In a newly setup project the utilization of this tool might help to prevent

- bad smells
- untested code
- code not complying to the style guidelines

from being introduced into the source code. This way, the project code will always stay "clean".

However in a project that has been under development for a relatively long time, like Mineralbay, which has not been making use of such tools, loads of bugs, quality issues or other bad smells might have been introduced already.

In such a scenario enforcing harsh code quality gates from one day to the other, does not make a lot of sense. These goals will be missed instantly and many hours must be spent reworking the software until it satisfies the defined gates. It is however more likely, that developers start to simply ignore the targeted goals.

The chosen approach in Mineralbay therefore is the following: After each iteration (sprint) of the software, the calculated metrics should not deteriorate. Comparing the analysis results, the current build must score better or, at worst, equal to the previous one.

After each merge into the master branch a review about code quality will be held. It will be discussed why certain improvements could be achieved, as well as why certain parts have been stagnating or even changed for the worse.

Introduce Style Guidelines

In a usual student project, code has to be written, needs to meet its functional requirements, must pass its demonstration and eventually is forgotten. As a result, the use of a consistent code style standard is often neglected.

In long running agile projects however, developers must be able to read each others code. Hence, introducing meaningful code style guidelines is essential, especially since the students preferences tend to vary largely, based e.g. on their favorite programming languages recommendations [69, p. 5].

A code style has been composed for the project at hand, making use of existing best practices (captured in Appendix A.7) and was integrated in the developers IDE. Before committing any piece of code, each developer had to use the existing auto-format tool, shipped with almost any IDE. This made compliance to the given standard easy, although ultimately verifying it was part of the code review process.

4.3.4 Reviews

This section covers all aspects of how a project tailored review process was designed and implemented.

Rationale

Reviews are an important part of quality assurance. Not only do they enhance quality and prevent errors, but also help spreading knowledge on a technical, domain specific as well as process oriented level [69, p. 5].

Especially in a long running student project, where distribution of knowledge is considered more important then progress, a sound review process cannot be neglected.

Addressed Research Questions

The research questions covered in this chapter are:

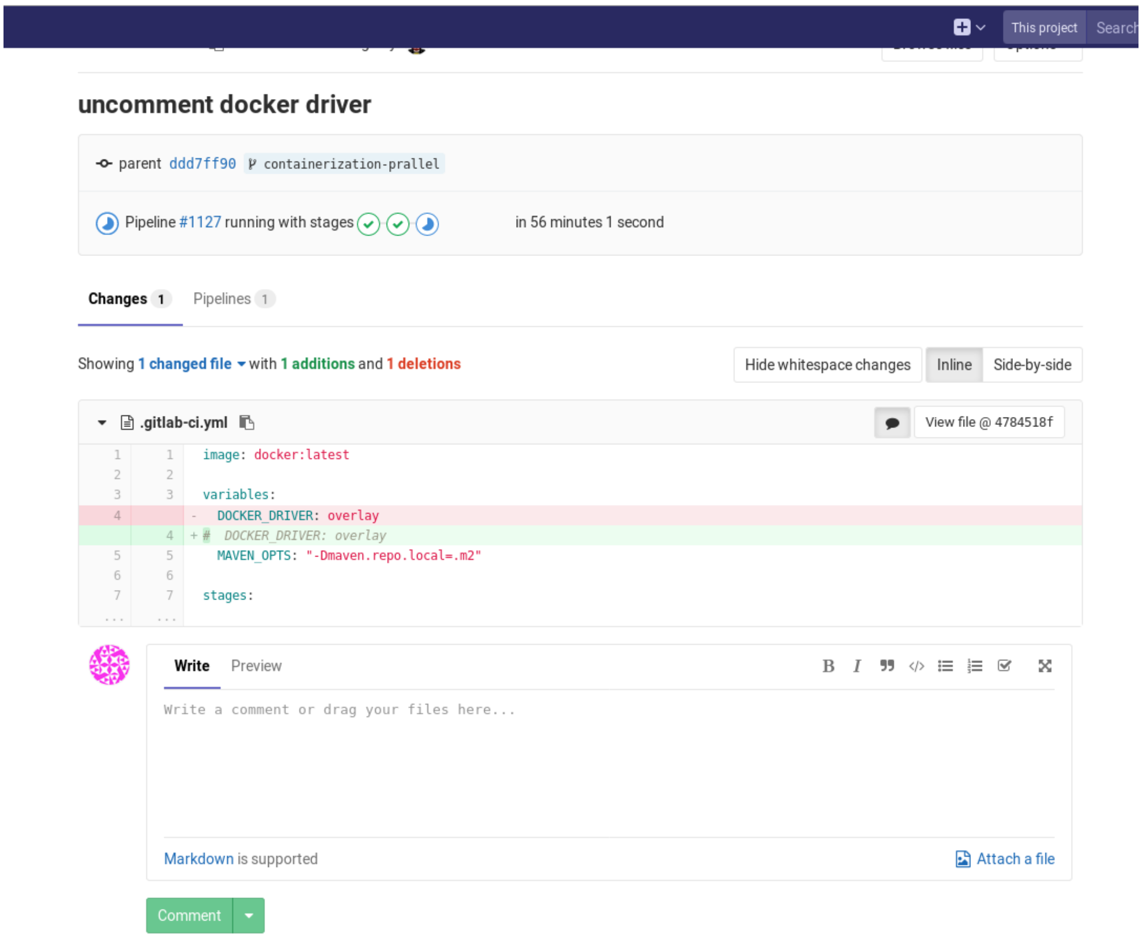
- Establishing a code review process (Description in Chapter 3.3 RQ 1)
- Unit testing (Description in Chapter 3.3 RQ 1): Reviews also highlight code that has not been tested properly

Code Reviews

Baseline of the desired review process is, that every line of code must be checked by a second pair of eyes, before being accepted in the code base. This applies to *feature branches* as well as to *sprint branches* and the *master* branch:

- **Reviews on feature branches**:

  Reviews of code, that is about to be submitted into a feature branch are carried out by fellow developers. In the best case, the reviewer is currently working on the same feature or both are having a pair programming session. On a feature branch, it suffices if all tests are passing and a fellow developer sees the commit as fit for being merged. The SCM in use provided a simple way of comparing and discussing changes 4.16.



uncomment docker driver

parent ddd7ff90 containerization-prallel

Pipeline #1127 running with stages in 56 minutes 1 second

Showing 1 changed file with 1 additions and 1 deletions

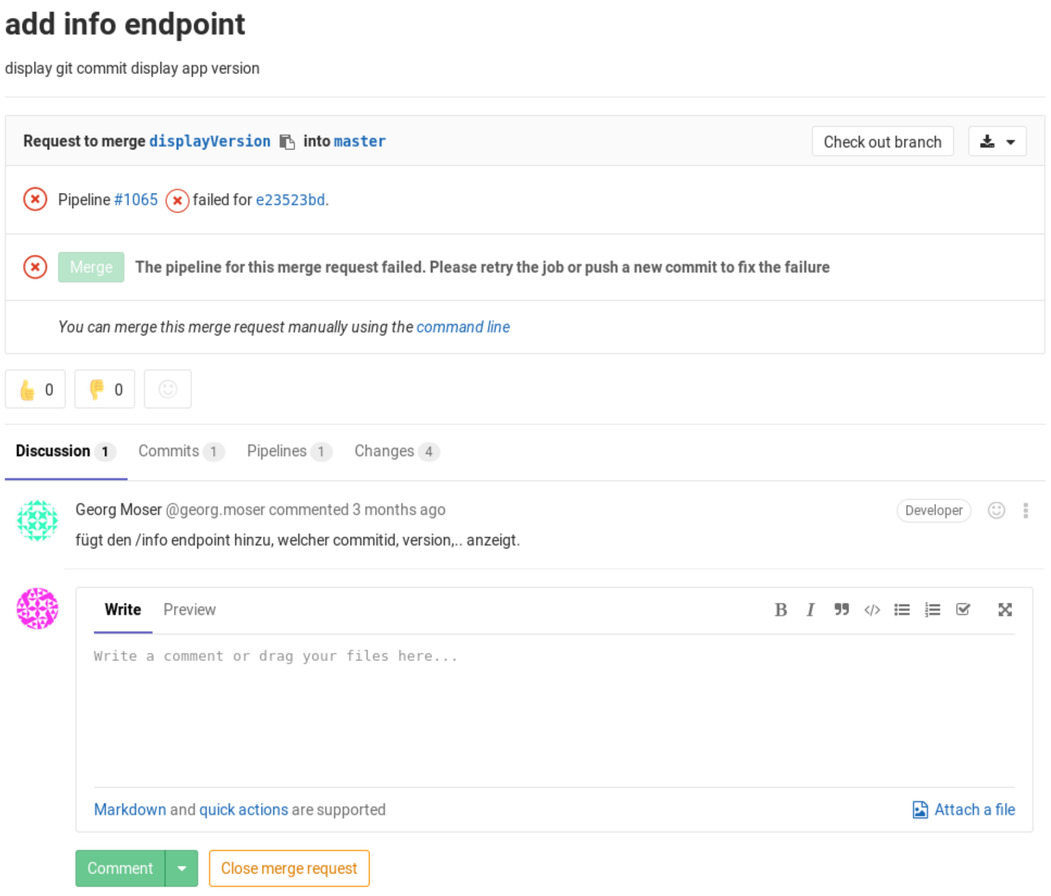
.gitlab-ci.yml View file @ 4784518f

```
  image: docker:latest

  variables:
-   DOCKER_DRIVER: overlay
+ #  DOCKER_DRIVER: overlay
    MAVEN_OPTS: "-Dmaven.repo.local=.m2"

  stages:
```

Figure 4.16: Review on a feature branch

- **Reviews on sprint branches**: Merge requests must be approved by 2 developers.
- **Reviews on master branch**:

  On the master branch however a stricter review process is applied: As outlined in 4.3.2, the complete test suite is executed and must of course pass. Additionally, the changes must be approved by 2 developers as well as scanned by an assistant in order to be merged. 4.17


add info endpoint

display git commit display app version

Request to merge displayVersion into master

Pipeline #1065 failed for e23523bd.

The pipeline for this merge request failed. Please retry the job or push a new commit to fix the failure

You can merge this merge request manually using the command line

Georg Moser @georg.moser commented 3 months ago

fügt den /info endpoint hinzu, welcher commitid, version,.. anzeigt.

Write a comment or drag your files here...

Markdown and quick actions are supported

Figure 4.17: Merge request for the master branch

A checklist for code reviews was provided, in order to help developers focus on the important parts of a code review A.6.

4.3.5 Testing

The following section describes how the effort of testing has been reduced.

Rationale

Testing software in a certain state/configuration must be easy. Multiple problems in the observed case resulted from developers/testers neglecting or not being able to test the software in the desired configuration.

Addressed Research Questions

Software testing is mandatory to SQA. Research questions concerning this section are:

- Testing in a production like environment (Description in Chapter 3.3 RQ 4): Gitlab is used for building images for each major version of the software.

Testing each build

Testing of a software is best done in the exact same configuration as it will run in production. Often during development different dependencies are used in order to speed up the application, making debugging easier or get getting of unnecessary configuration.

This is also the case for the Mineralbay application. In production mode:

- JavaScript is bundled, uglified and minified and HTML is minified too.
- the software uses a persistent PostgreSQL database, whereas in development an ephemeral H2 database is used.

As a result, some errors only occur when the application is run in production mode. Developers and testers however were forced to establish a local production like environment, i.e. maintain a local PostgreSQL database, as well as making sure the necessary test data is available and making a production build of the software.

This turned out to cause too much effort and consequently was never done, thus allowing to slip severe errors into production.

As a result, it can be concluded, that manual testing itself must be as simple as possible in order to be done properly and consistently. High initial and setup efforts might result in reduced testing or even total neglect.

Efficient testing

The goal of conveniently and swiftly establishing a local test setup, is to offer a routine which manages this task in one click.

In order to do so in the Mineralbay project a container image was generated for every revision. Projects members were then able to run the application wrapped inside this container on almost any given hardware, without having to install numerous dependencies.

The PostgreSQL database can also be run locally inside a container.

For achieving this goal the OpenSource container platform engine docker [11] was used. The orchestration was done using docker-compose, as depicted in 4.18

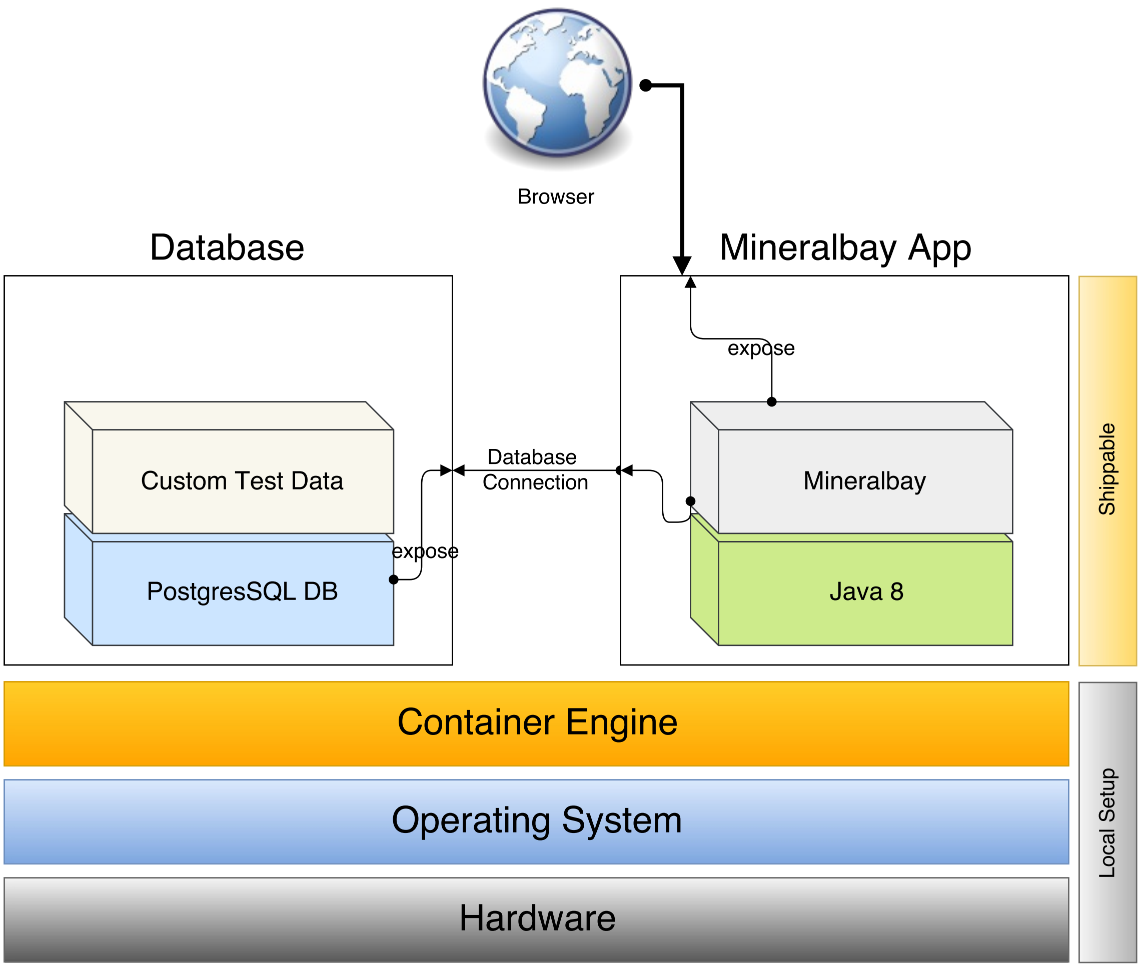


Figure 4.18: Docker compose

This way a production like local setup, ready to be tested, can be established in a convenient way.

[11] docker.io

4.3.6 Versioning

The following section describes how a versioning scheme for the Mineralbay software was elaborated and included in the development process.

Rationale

In the current development process it is not specified under which circumstances the version number is increased. This can become a problem, because usually features and bugs are linked to a specific version. However since there is no versioning, developers have no clue which version/build of the software is currently deployed on the servers. As a result they don't know if e.g. a bug is already fixed in on this certain instance (problematic for e.g. testing) or not.

Addressed Research Questions

The research questions covered by this chapter are:

- Establishing a versioning process (Description in Chapter 3.3 RQ 2)

Semantic Versioning

The version-number will be composed based on the semantic versioning [12] approach.

A version-number consists of multiple parts separated by a dot:

- *MAJOR version*: Raised if the introduced changes make the Advanced Programming Interface (API) incompatible to previous versions.
- *MINOR version*: Raised if new functionality is added in a backwards-compatible manner
- *PATCH version*: Raised if backwards-compatible fixes are released.
- *Additional Labels*: Give further insight in the version:
  - *SNAPSHOT*: Indicates a version which is still in development
  - *ALPHA*: Indicates that the overall software is still under construction. It is not stable, might crash or not be feature-complete.
  - *RC*: Release Candidate

Applying this scheme to the present development process the author proposes the following versioning strategy:

- *MAJOR version*: Raised once the software is ready for go-live
- *MINOR version*: Raised after completion of each sprint, thus replicating the sprint number.
- *PATCH version*: Raised if an already released version is patched.

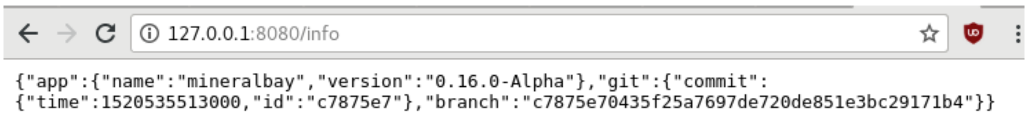
[12] http://semver.org/

Process Versioning

Each time a sprint branch is forked from the master, the version number is increased and the SNAPSHOT label is added. Upon merging the branch bag, the SNAPSHOT label is removed again.

Display Version

The version of each deployed artifact is easily determined, since a special endpoint "/info" provides information about the build.



```
{"app":{"name":"mineralbay","version":"0.16.0-Alpha"},"git":{"commit":
{"time":1520535513000,"id":"c7875e7"},"branch":"c7875e70435f25a7697de720de851e3bc29171b4"}}
```

Figure 4.19: Screenshot of the /info endpoint

4.3.7 Gamification

This section describes how the author combined elements of gamification with software quality assurance, implementing rankings on the number of quality related tasks performed.

Rationale

There are many tasks in the process of QA which are considered boring and usually do not reward the developer in any way. The effort put into them might not seem worth it. This is exactly the place were mechanisms of gamification can be exploited. Utilizing elements of gamification can make developers feel rewarded for improving software quality, thus increasing motivation.

Addressed Research Questions

The research questions covered by this chapter are:

- Use of gamification for QA tasks (description in chapter 3.3 RQ 5):
- Executing tests before pusing to the SCM (description in chapter 3.3 RQ 3):

Description

As described in chapter 2.8 mechanics of gamification (i.e. "applying game-design thinking to non-game applications to make them more fun and engaging" [80, p. 744].) are well suited for the field of software engineering.

The author of this thesis tried to apply the mechanisms of gamification in the context of SQA.

These methods have already been applied to motivate users to do tasks that otherwise seem quite unrewarding. One of the most popular examples is the question-and-answer site StackOverflow [13]. Answering questions asked by other users is rewarded with points and badges. This motivated people to participate and boost their ranking [25].

Similar methods can be applied on QA tasks.

Considering the process at hand the author proposes the gamification of the following elements:

- *Code Issues*: Fixing bad smells, bugs, etc. should be rewarded.
- *Builds*: Building working software should be rewarded, committing erroneous code (i.e. non-compiling or test failures) will have a negative impact on the users scores.
- *Tests*: Creating tests and increasing coverage should be rewarded.

Used Elements of Gamification

The implementation will make use of the following mechanics of gamification:

- *Points*: Points will be rewarded for "positive" (i.e. improving quality) actions and deducted for "negative" (i.e. jeopardizing quality) ones.

[13] https://stackoverflow.com/

- *Badges*: Badges will be given to users upon completing a certain achievement.
- *Achievements*: Achievements are completed if users complete a set of predefined actions, usually involving exceptional effort. These will be rewarded with additional points.
- *Leader Boards*: These Boards will display a ranking of the users, based on the points they earned.

Defined Actions

A list of possible

Several quality related tasks were listed and evaluated as candidates for gamification. These actions were then linked to a number of points rewarded for completing them. Achievements were defined by completing special tasks or a series of actions.

Finally a list of these elements was compiled in tables 4.1, 4.2, 4.3:

Action	Description	Points
Fix Blocker Bug	Fixing a Blocker Bug reported by SonarQube	8
Fix Critical Bug	Fixing a Critical Bug reported by SonarQube	5
Fix Major Bug	Fixing a Major Bug reported by SonarQube	3
Fix Minor Bug	Fixing a Minor Bug reported by SonarQube	2
Fix Info Bug	Fixing a Info Bug reported by SonarQube	2

Table 4.1: List of Bugfix Actions

Action	Description	Points
Successful Build	Triggering a Build on the CI Server that succeeds	1
Failed Build	Triggering a Build on the CI Server that fails	-2

Table 4.2: List of CI Actions

Achievement	Description	Points
Early Bird	Fixing an issue that is open for longer than X	10
Unit Tester	Fixing UT coverage for X legacy classes	10
Quality Contributer	Triggering 10 successful builds in a row	5

Table 4.3: List of Achievements

The work done and effort put in these tasks by the developers will be displayed on a ranking board. Based on the listed elements the following boards might be suitable:

- *Issue Statistics*: A board of the users fixing the most issues on the Static Code Analysis server
- *Build Statistics*: A board of the users running the most successful builds
- *Test Statistics*: A board of the users submitting the most test cases.

In order to do so, suitable data sources had to be found:

- *Static Code Analysis*: Tracks bugs, issues, bad smells in the form of tickets. A developer can grab a ticket and resolve it.
- *Build Server*: Holds information about the test results of commits.
- *SCM Server*: Holds statistical information about commits.

The monitoring and calculation of the results will be done by a software created in the context of this thesis, which is described in Appendix A.5.

4.4 Data Collection

The following section describes how the necessary data was collected, describing units of analysis and the procedure of data collection techniques.

4.4.1 Experiment Schedule

Description and Goal

The author of this thesis proposed a modified development process, which needed to be evaluated in order to answer the prepared research questions.

An experiment was conducted, where the participants were asked to develop some minor features in the context of the Mineralbay task, following a development process, that implemented all proposed measures.

The Participants

The experiment was attended by 6 students of the TU (depicted in Table 4.20). In advance to the experiment, the participants were asked to answer a small demographical questionnaire and self asses their skills regarding technologies (results displayed in figures 4.21 and 4.22) and test skills (Figures 4.23 and 4.24) .

Id	**Program Name**	**Project Member since (Month)**	**Study Type**	**Enrolled since (Semester)**	**Employed**	**Employed since (Years)**	**Programming Experience (Years)**
P1	Software Engineering	24	Master	WS2017	Yes	2	2
P2	Wirtschaftsinformatik	17	Bachelor	WS2013	Yes	1	2
P3	Software Engineering	24	Master	WS2017	Yes	2.5	3
P4	Software Engineering	17	Master	WS2010	Yes	1	0
P5	Software Engineering	12	Master	WS2010	Yes	8	20
P6	Software Engineering	6	Bachelor	SS2008	Yes	5	13

Figure 4.20: List of Participants

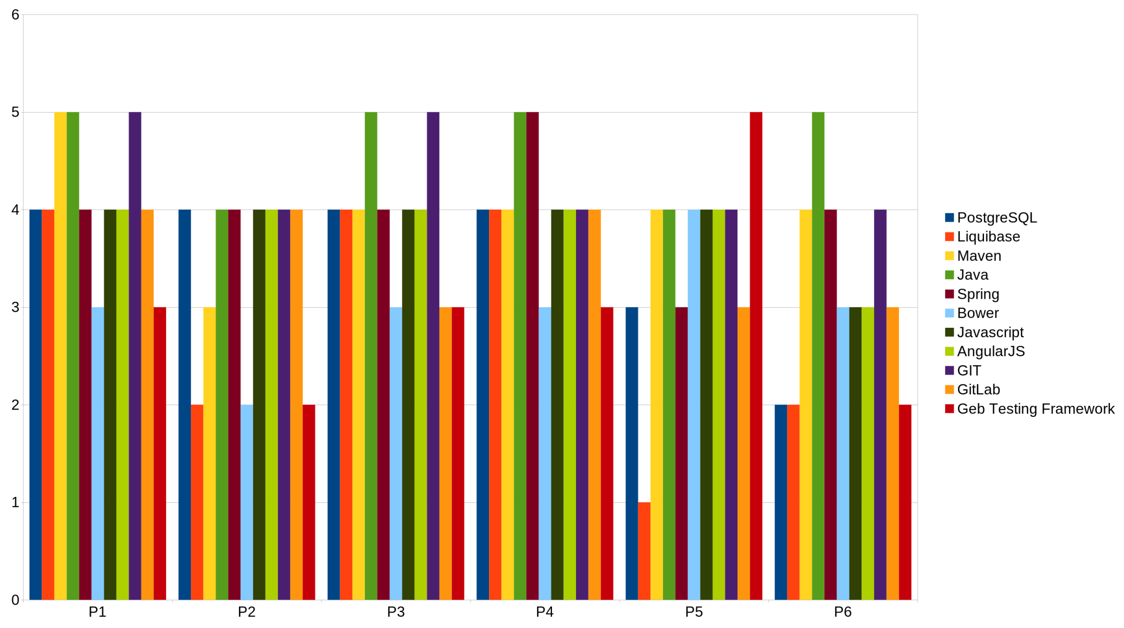


Figure 4.21: Participant Tech-Skills Self Assessment

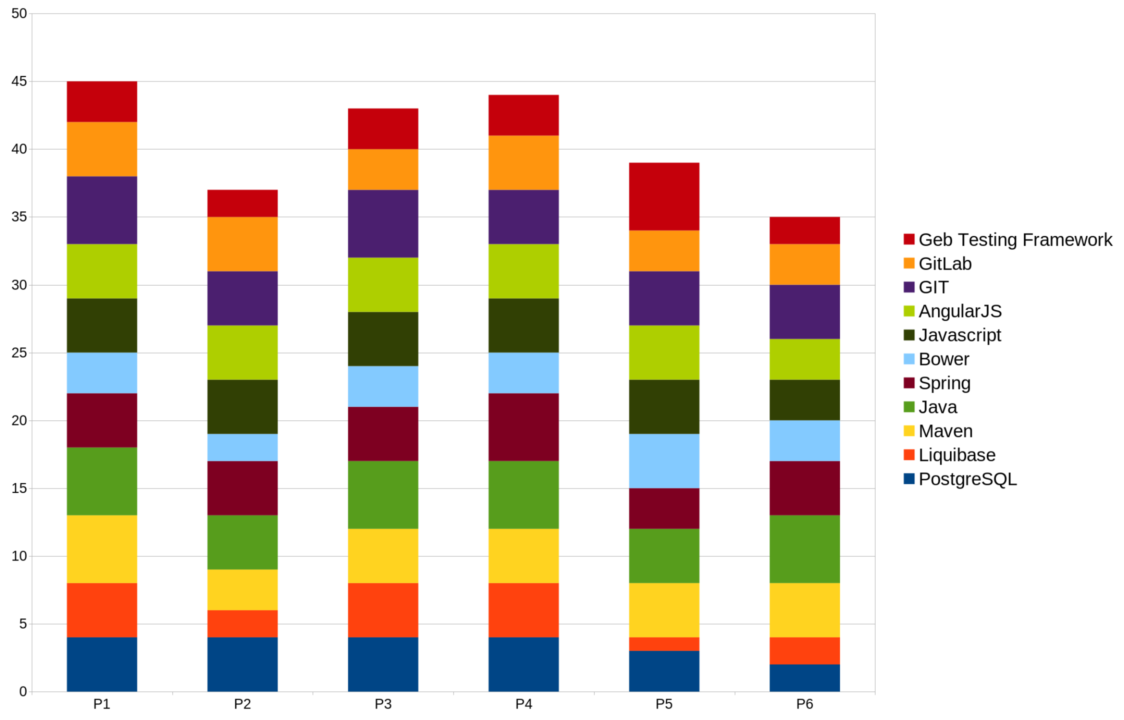


Figure 4.22: Participant Tech-Skills Self Assessment (Stacked)

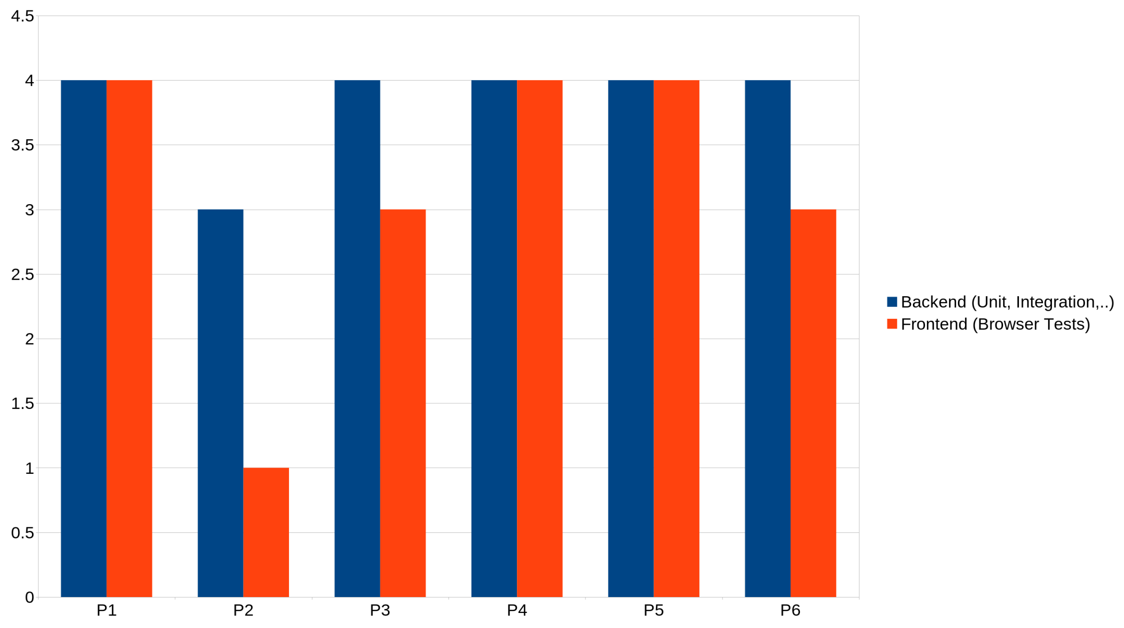


Figure 4.23: Participant Tech-Testing-Skills Self Assessment

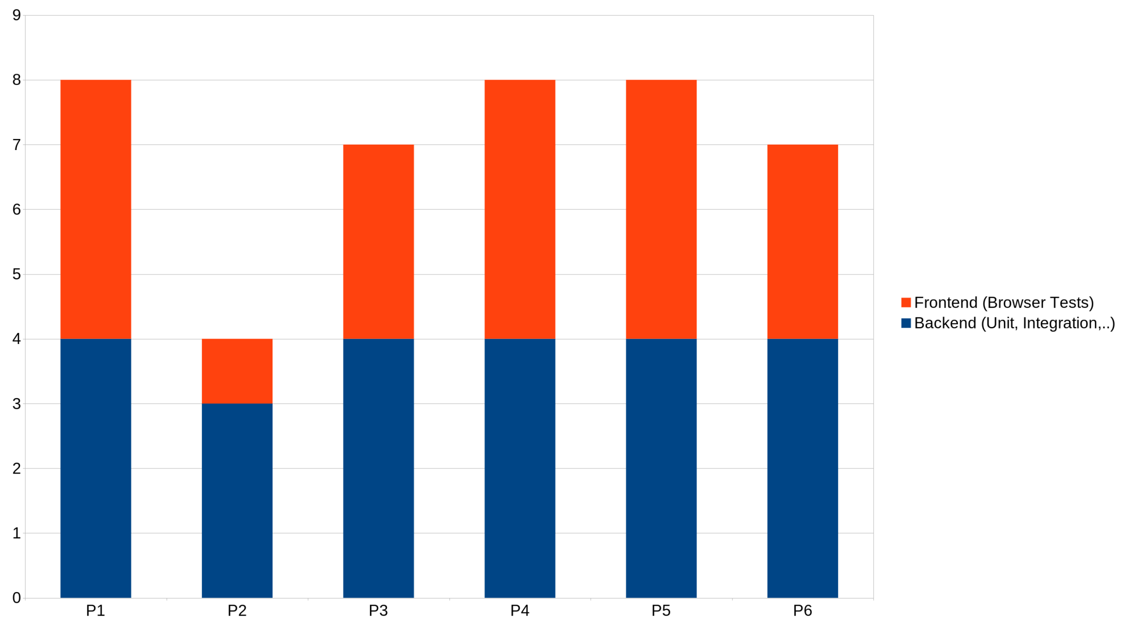


Figure 4.24: Participant Tech-Testing-Skills Self Assessment (Stacked)

Introduction

At the beginning of the lab the participants were informed about the purpose of the laboratory, being the gathering of information for the authors masters thesis. They then were asked again if this was ok for them, as well as if all of them would agree to be interviewed after the lab. It was also confirmed, that all information will only be used aiding the thesis.

The conductor (i.e. the author of this thesis), then described his proposed development process in a short presentation.

A simple task was then implemented by the conductor, demonstrating the procedure.

Feature Description

In the following, the tasks that would be implemented, were introduced. Simple and small tasks were picked in order to be able for the participants to solve them in the given amount of time. Furthermore, since the focus of the experiment was on the process, the amount of written code was not of priority:

- A product can be persisted to the database without providing all necessary parameters.
- A back-button is missing on the grainsize distribution page.
- On the production plant detail page, only the first name of the plant manager is displayed in a dropdown.
- On the production plant detail page, the operator fields aren't properly persisted.

The tasks were then presented to the participants, showing the current state of the application and where changes need to be made, respectively.

Team building

The teams for pair programming were selected based on their skills concerning frontend and backend development, regarding the given technology stack.

The goal was to build teams of complementary knowledge, i.e a developer skillful at frontend developing, but lacking backend skills would be joined by an experienced backend developer, which again stated deficits in frontend development.

This was achieved by letting the participants do a short self assessment prior to the team building.

The conductor of the experiment than assembled the teams according to their skills.

Implementation

After the teams got familiar with their development setup, the actual implementation of the given tasks could be tackled.

4.4.2 Experiment Data Collection

Interviews

After completing the experiment, the participants were asked to take part in an interview covering their previous experiences.

The author chose to conduct the interview in a semi structured way (described in chapter 2.2.3), because of the fitting characteristics of this type of interview.

The schedule of the interview is listed in the appendix A.4.

Questionnaire

The author also prepared a questionnaire which he asked the attendees to complete as soon as possible. Fortunately, all results arrived within a week.

The entire questionnaire is attached in the appendix A.3.

Participant Observation during Lab

Since the author was present during the whole implementation phase of the experiment, he could gather notes about the participants behavior.

5 Results

The following section displays the collected data in an objective way, broken down by the measures that have been tested for their impact on quality during the experiment. It provides the basis for the succeeding chapter which will analyze the data presented in this chapter and link it to each individual research question.

5.1 Code Reviews

Description

As described in chapter 4.3.4, the author introduced a comprehensive review process in the Mineralbay development lifecycle.

Results

During the laboratory, the participants could experience the process first hand and were asked to state their thoughts about it during the subsequent interview session.

It is noteworthy that none of the participants did feel like the review process does take too much time.

Reviews during development and on a feature basis were generally thought to be "very important" (P4, P6, P3), by some even as "necessary" (P3).

Sprint reviews on the other hand were only received with doubts: "unnecessary, since the code has already been reviewed" (P6, P2). The conductor had to clarify that these reviews were part of a sprint - DoD and will focus on different aspects than regular code reviews. The author further illustrated the benefits by reminding the developers how many of them did forget to set the status in their tickets, etc. during the laboratory.

The impact of reviews on knowledge transfer was evaluated in a questionnaire and revealed that participants are strongly benefiting from reviews in this aspect (as shown by figures 5.1 and 5.2) :

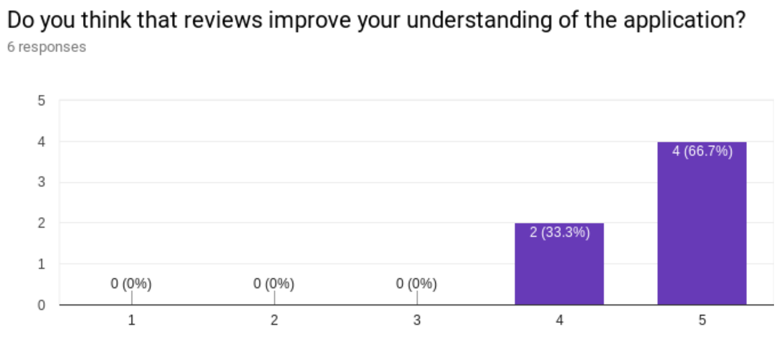


Figure 5.1: Code Reviews increasing application understanding

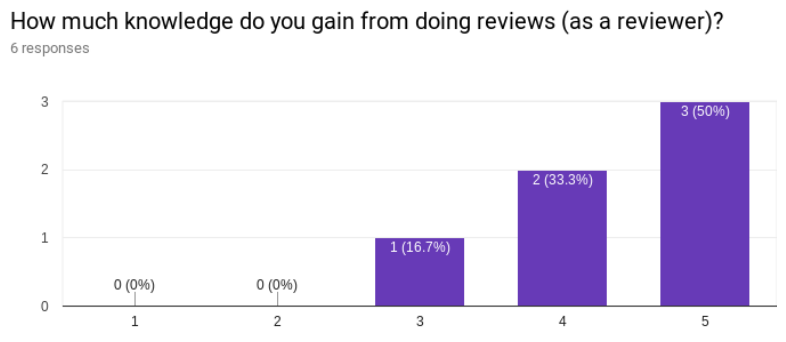


Figure 5.2: Reviews - Knowledge Transfer

Code reviews and their implications on code quality was also surveyed and provided promising results:

Whilst feature reviews were perceived as code quality enhancing factor by all participating developers (Figure 5.3), the acceptance concerning sprint reviews was generally mixed (Figure 5.4).

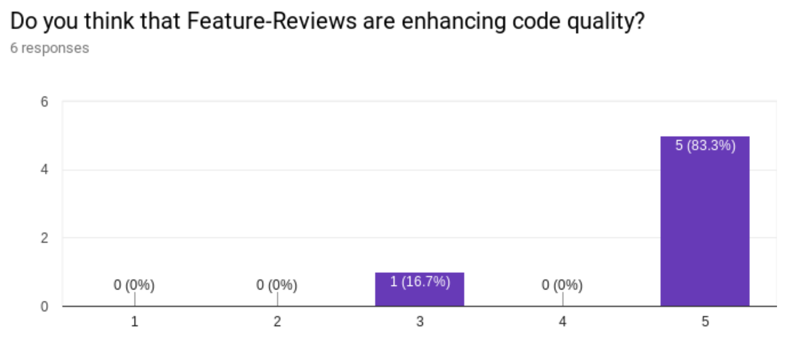


Figure 5.3: Feature Reviews - Code Quality

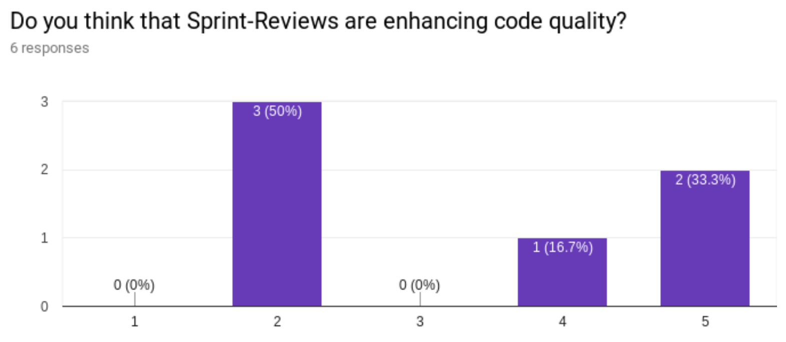


Figure 5.4: Sprint Reviews - Code Quality

5.2 Checklists

Description

Checklists were elaborated in the context of this thesis, purposing to aid code reviews and thus improving their positive impact on code quality.

Results

The idea of checklists was generally accepted well:

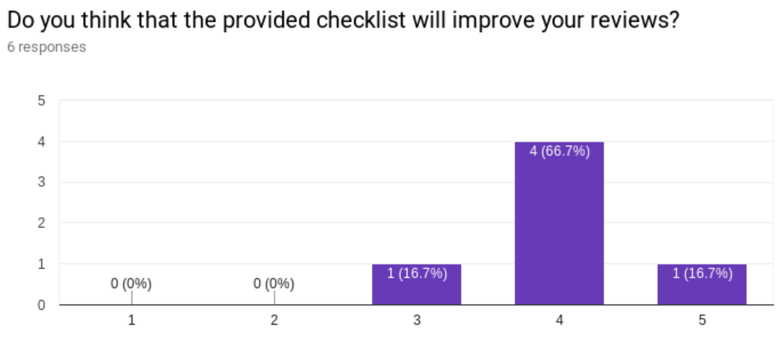


Figure 5.5: Code Review Checklists Acceptance

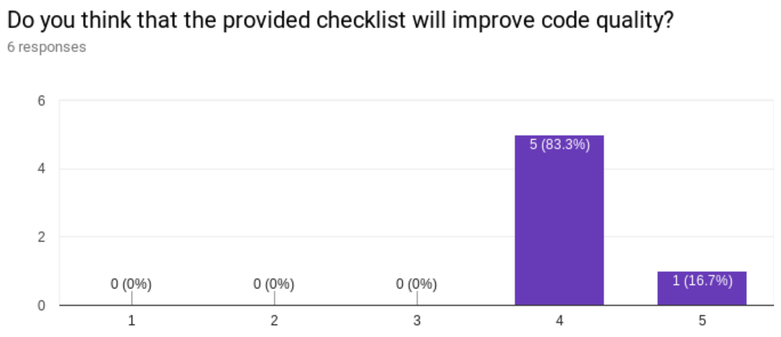


Figure 5.6: Code Review Checklists Code Quality Impact

During the interviews students stated that it might be hard to find a balance between covering all necessary items of a code review and not going too much into detail, in which case the checklist most likely will be ignored (P6, P5). Furthermore a checklist should be kept “compact” and “easily accessible” during the review process (P3, P1).

A student stated that some tasks being part of a proper code review are "easily forgotten and therefore placed well in a checklist" (P6).

Generally the provided checklist was considered “useful” by the participants.

5.3 Self assessment for team building

Description

The author evaluated the idea of building teams for pair programming by assigning developers with complementary weaknesses to a team.

Results

The interview session revealed generally mixed convictions concerning this idea:

While some interviewees stated that "new members will benefit from this approach", others were the opinion that it is "more important to place new members of the team with experienced ones" (P1, P3). In the succeeding questionnaire, the feedback regarding the question if the developers did benefit from their selected partner, the responses were diversified too (Figure 5.7). In terms of learning experience, results were similar (Figure 5.8), but 83.3 % of the participants stated that their selected partners did not slow them down at all.

Furthermore during the interview, all participants agreed that "shuffling partners is also very important for knowledge exchange" (P3).

The general consensus concerning self assessment was that implementation in a meaningful way is "very difficult in software engineering"(P6) and only "works for a given technology stack"(P3, P2).

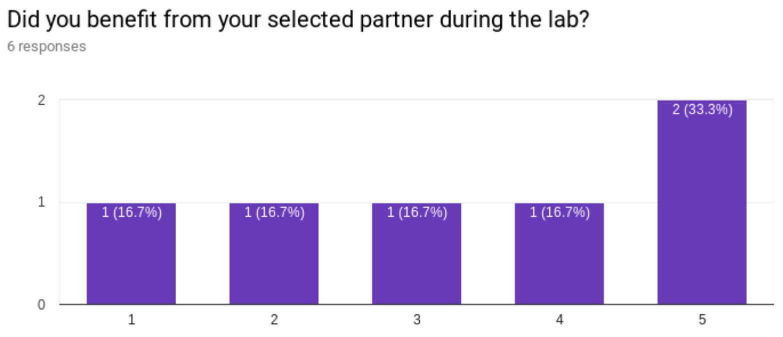


Figure 5.7: Self Assessment: Benefits from partner

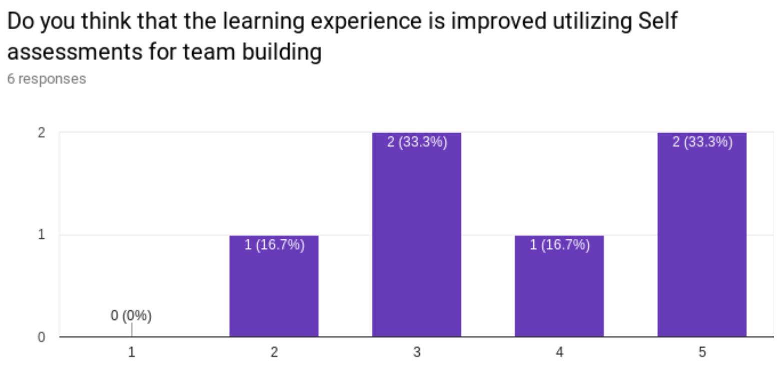


Figure 5.8: Self Assessment: Learning experience

5.4 Versioning

Description

A project tailored versioning scheme and process was proposed as part of this thesis and evaluated during the laboratory.

Results

During the interview session all participants stated that they understood the versioning scheme. They also agreed that versioning is necessary to "relate features and bugs to a software version" (P4). These statements were confirmed in the post laboratory questionnaire, which showed that general understatement is given (Figure 5.10) and given the proposed versioning schema, linking features to a specific software version should not be a problem (Figure 5.9):

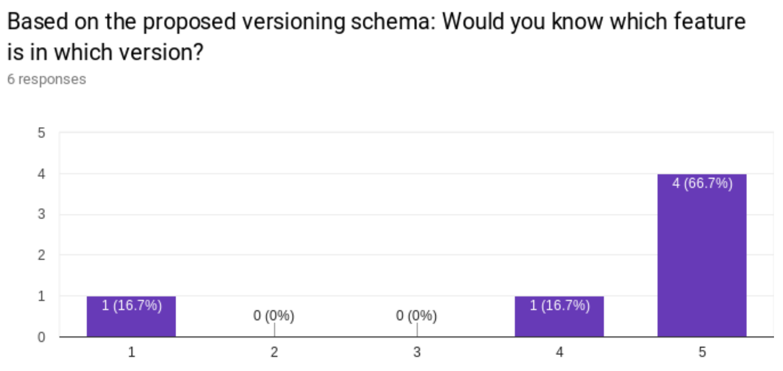


Figure 5.9: Feature to Release Version Affiliation

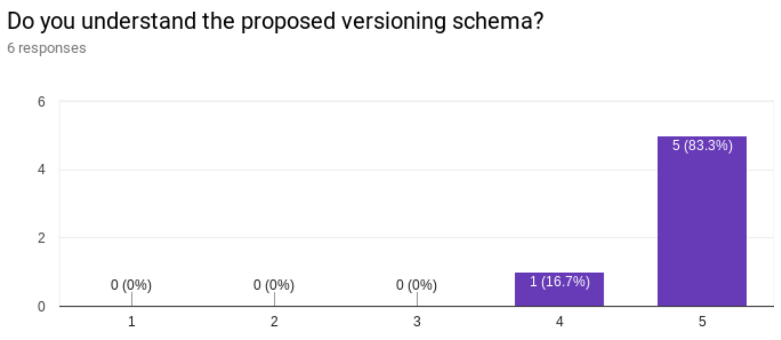


Figure 5.10: Understanding of Versioning Schema

5.5 Build Process

Description

As part of this thesis the build setup of the Mineralbay project was migrated and reworked using Gitlab.

Results

The changes were appreciated by the developers and described as "great improvement" (P2) during the interview session. Furthermore the participants found "the immediate feedback from the pipeline if the tests succeed" (P2) together with "build artifacts, test reports and screenshots of failed tests" "very helpful" (P1).

After the author explained the basics of the setup to the interviewees, most of them felt confident to adapt steps on their own, if the setup is "well documented" (P1, P5, P2). The concerns of one developer could be discarded when he was told that the pipeline itself was versioned itself and could be tested on a dedicated branch (P6).

Developers showed to be generally satisfied with the performance of the build pipeline, as surveyed (Figure 5.13). Similar to the interviews they assured their understanding of the setup (Figure 5.11) and all of the participants thought the pipelines to be important for quality assurance.

The confidence in the pipelines of preventing errors from slipping into production was mostly given too (Figure 5.12).

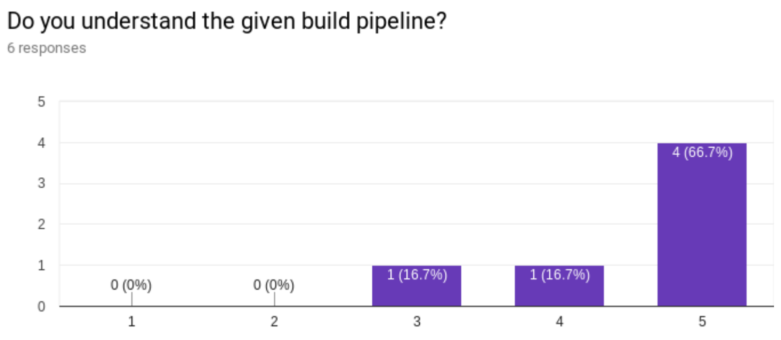


Figure 5.11: Understanding of Build Pipeline

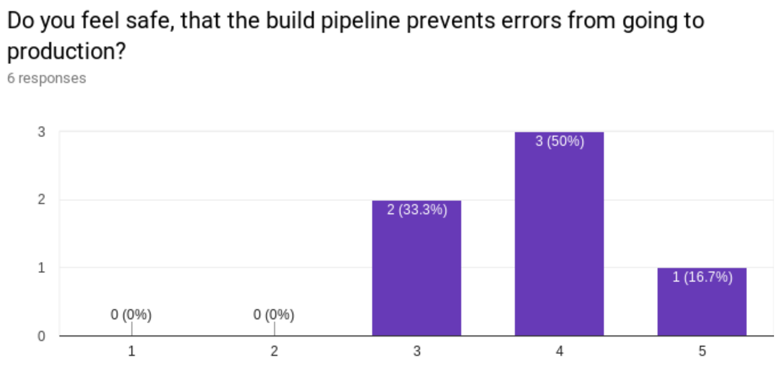


Figure 5.12: Build Pipeline - Error Prevention

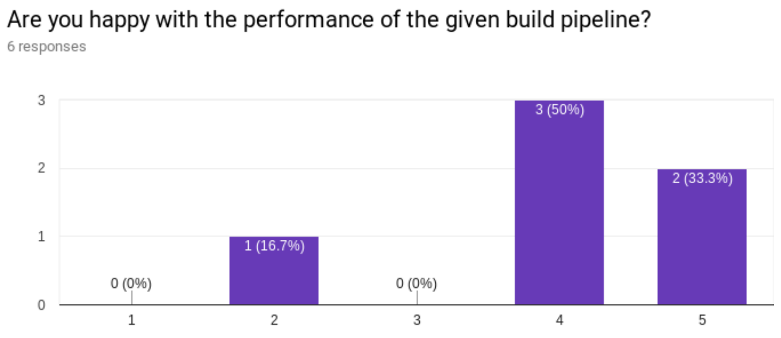


Figure 5.13: Build Pipeline Performance Satisfaction

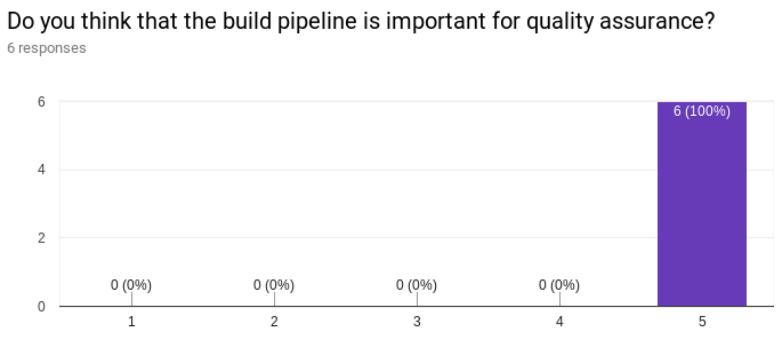


Figure 5.14: Build Pipeline - QA Contribution

5.6 Branching Model

Description

Since the original branching approach did not prove satisfying, a modified one (described in section 4.3.1) was evaluated in the laboratory.

Results

Some students had difficulties understanding why "features weren't merged in the master directly" (P3), others stated that merging sprint branches at the end of each sprint assures "that the features are not left behind" (P2).

In the subsequent questionnaire most students showed confidence according their understanding of the branching model (Figure 5.15) , but also doubts when being asked if features will be merged faster into the master branch (Figure 5.16).

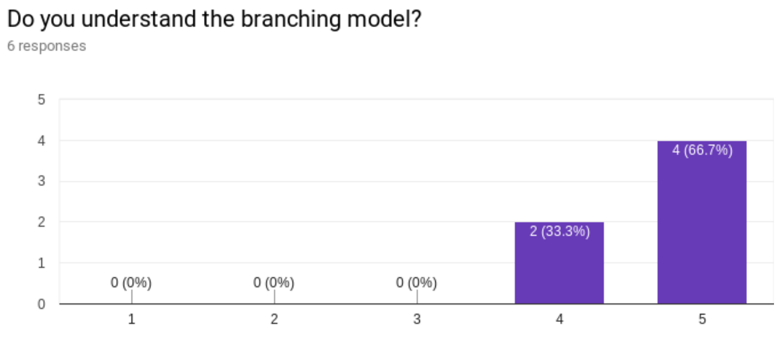


Figure 5.15: Branching Model Understanding

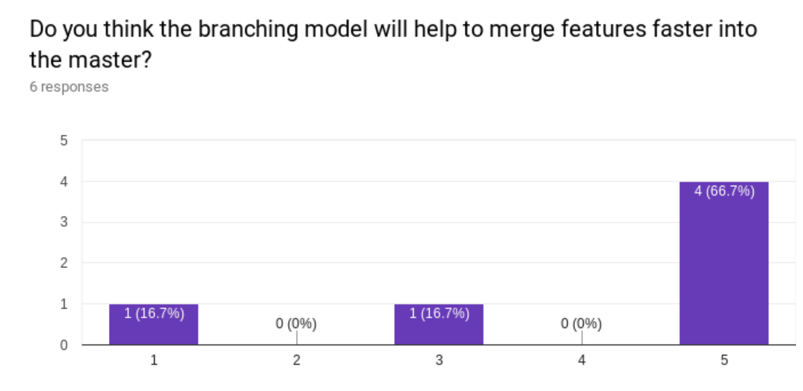


Figure 5.16: Branching Model Merge Speed

5.7 Test Environment

Description

Since deploying a specific version of the software locally did provide difficulties for the testers, an easier solution was designed throughout this thesis: Software containers are created for each major change in the software and made available to the developers. For testing purposes a container of e.g. a certain feature can easily deployed locally. During the laboratory, the participants were asked to give this approach a try and were questioned about their experience later on.

Results

The interview showed that all participants understood the workflow and its improvements and named use cases like "testing container of sprint branch before merging into master" (P1, P2). In order to adopt this process however it was asked that "documentation is provided" (P3).

The results of the questionnaire displayed a general understanding of the improvements regarding local deployment of the software (Figure 5.17) , as well as deploying a specific version locally (Figure 5.19), even in a production like setup (Figure 5.18).

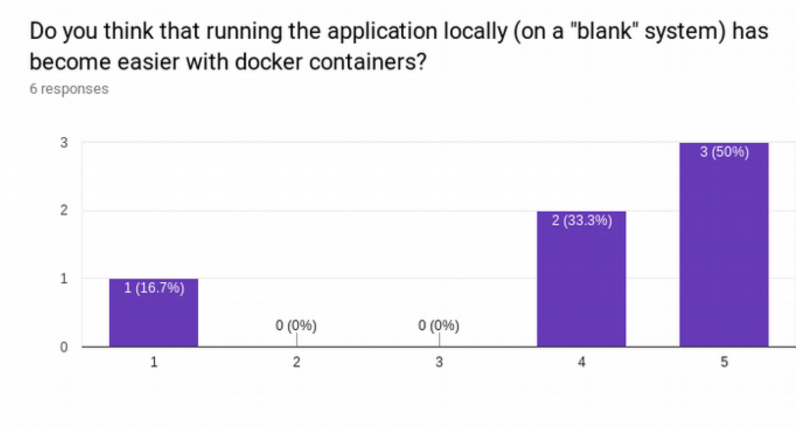


Figure 5.17: Test Environment: Difficulty of local setup with containers

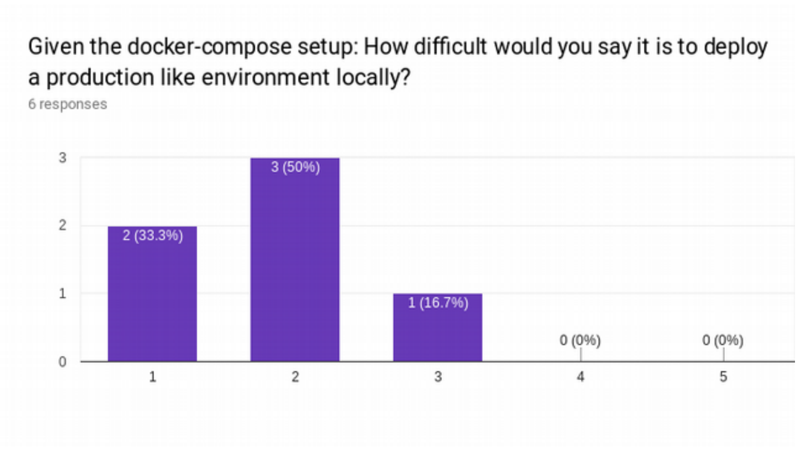


Figure 5.18: Test Environment: Difficulty of production like local setup

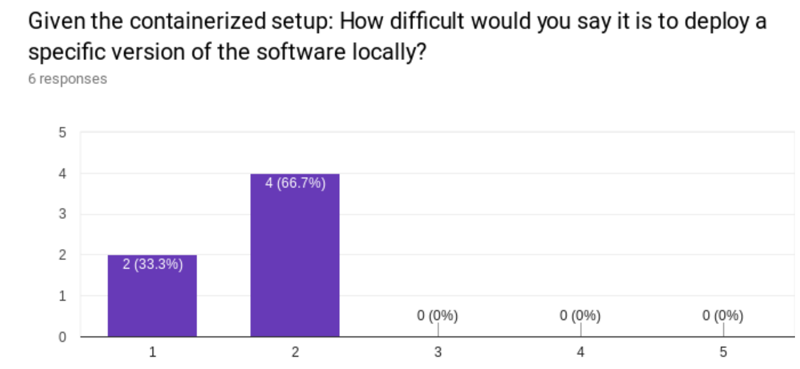


Figure 5.19: Test Environment: Difficulty of deploying a specific version

5.8 Code Style Guidelines

Description

Code style guidelines were introduced in the development setup and integrated in the IDE in order to establish a uniform code style.

Results

In the interview the feedback was of positive nature: The general consensus was that code style tools in the IDE would perform well for formatting the code" (P3) and linting tools were "a good idea, but will cause lots of work in the beginning" (P3) (editors note: Since the tool marks all deviations from its guidelines at once, which might be a lot). The "immediate feedback" (P6) of code linting and code style software was praised and the developers already started thinking about "refactoring code in the course of developing new features" (P3), improving all "files in scope of the current task" (P2, P3).

The positive reception reflected in the questionnaire where all participants thought code style checks in the IDE would improve code quality (Figure 5.21) and almost all agreed that they would enhance code readability (Figure 5.22).

Furthermore all of the polled developers stated that they found badly formatted code annoying (Figure 5.20).

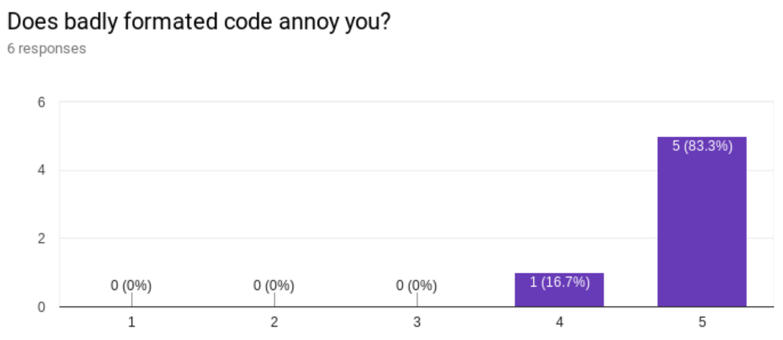


Figure 5.20: Developers Opinion on badly formatted code

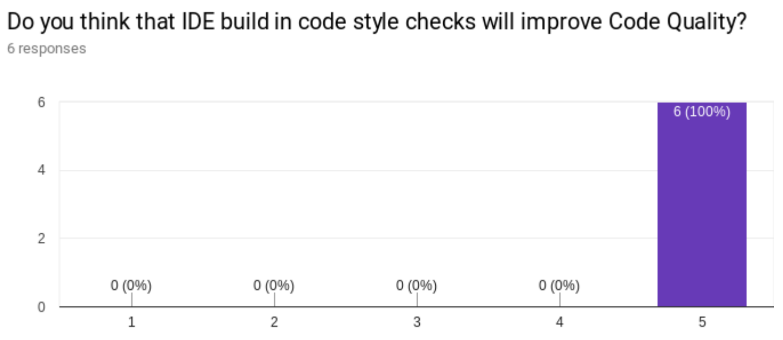


Figure 5.21: Code Style IDE Tools: Impact on Code Quality

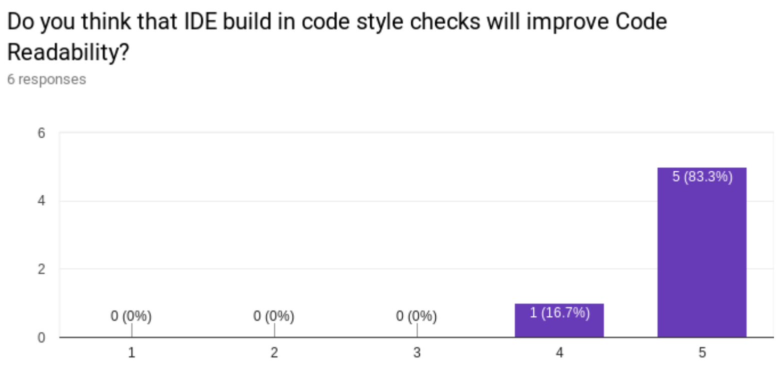


Figure 5.22: Code Style IDE Tools: Impact on Code Readability

5.9 Static Code Analysis

Description

Static code analysis tools (i.e. SonarQube) were integrated in the development setup, aiming at improving code quality 4.3.3.

Results

During the interview session, the interviewees agreed that tools like SonarQube would aid code quality, at the condition that the "generated statistics must be easily accessible to everybody" (P4, P6).

A participant suggested introducing "bugfixing and quality improving sprints" (P4) while others were the opinion that "fixing small issues" were best done "along other tasks"(P2, P6).

The proposed goal of improving overall tool-calculated quality in each sprint was considered unrealistic by some developers, who thought it would be better to strive for longer distances between improvements (P3). This position was also reflected in the outcome of the questionnaire (Figure 5.24) .

Results from the survey showed that all of the respondents agreed that the provided code analysis setup will lead to better code quality (Figure 5.23) . The majority also thought that they might learn something from the analysis results too (Figure 5.26) .

Several developers saw the comparison of statistics before and after merging their code as a motivating factor (Figure 5.25).

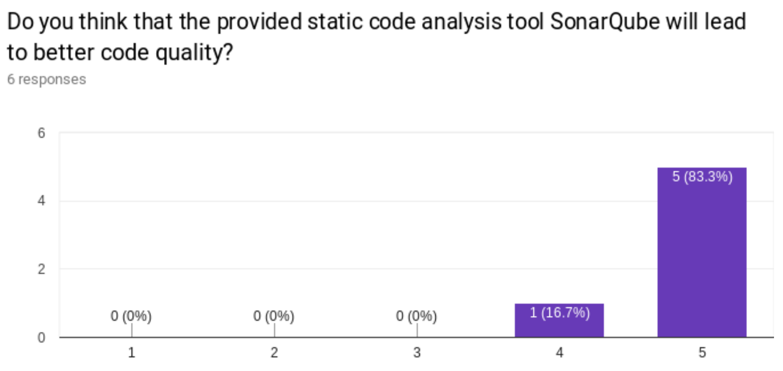


Figure 5.23: Static Code Analysis: Impact on Code Quality

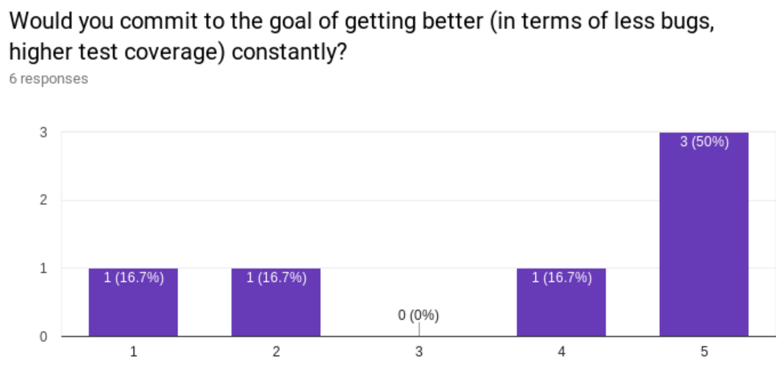


Figure 5.24: Static Code Analysis: Constant Quality Improvement

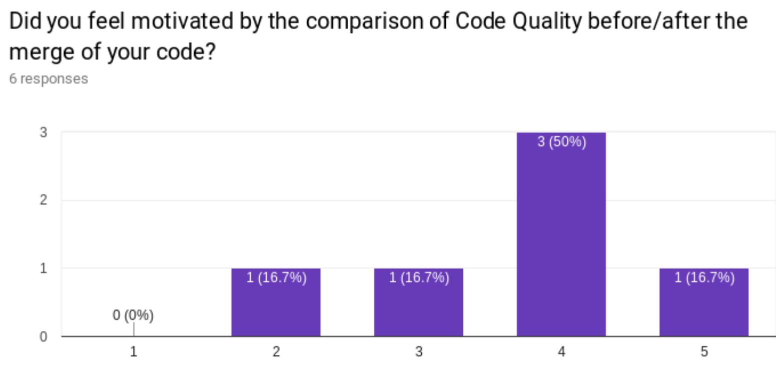


Figure 5.25: Static Code Analysis: Motivation

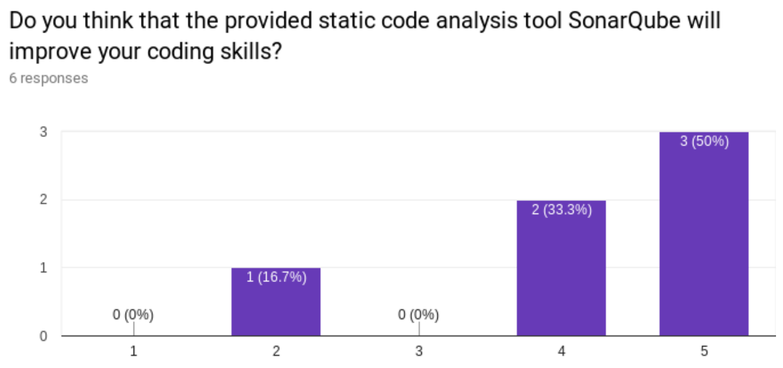


Figure 5.26: Static Code Analysis: Learning Experience

5.10 Code Quality Game

Description

The author tried to combine typical quality assurance tasks with elements of gamification. He hypothesized that thus, tasks likely considered boring by the developers, would be performed more willingly and frequently. Such tasks included:

- Fixing of Software Bugs
- Closing software quality related tickets
- Submitting working code

Again the hypotheses was evaluated in a laboratory.

Results

The idea of awarding points for successful builds was not considered practical by the majority of the developers, as shown by the interview and questionnaire. Objections were that most developers "would not care about failed builds" (P4) and that assigning points in a fair way was hard to achieve (P3), since one could randomly trigger builds in order to collect points. Those concerns also showed in the questionnaire (Figure 5.27)

The gamification of code quality related tickets however received positive feedback and were described as "highly motivating" (P4, P1). The results of the questionnaire also showed that most of the developers thought the combination of gamification and QA tasks was a motivating one (Figure 5.29). Furthermore the code quality games approach to gamify bug and quality related tickets was seen as motivating too (Figure 5.28).

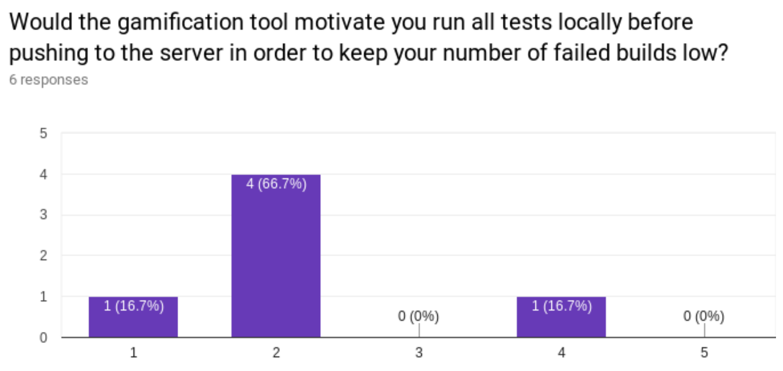


Figure 5.27: Code Quality Game: Local testing motivation impact

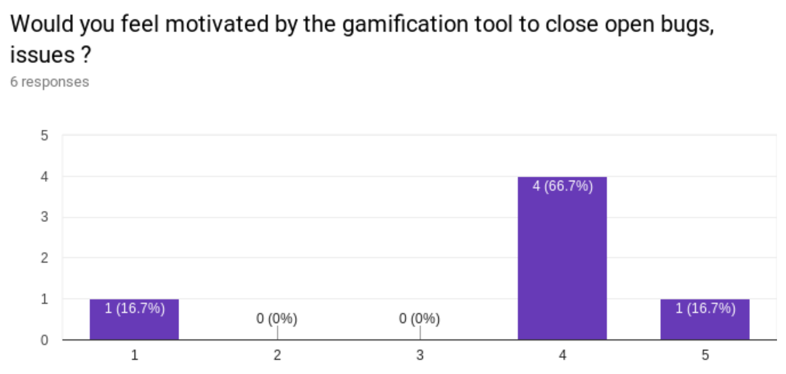


Figure 5.28: Code Quality Game: Motivation

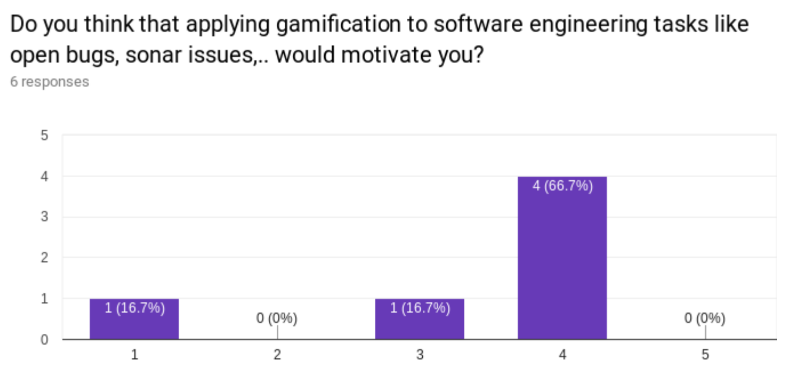


Figure 5.29: Code Quality Game: Overall QA Task - Gamification Motivation

5.11 Acceptance Tests

Description

The author suggested that providing acceptance criteria alongside the feature description will improve understanding of the given task, as well as ensure the developers, that they did not forget any part in their implementation.

Results

After implementing a feature, handed over to them with provided acceptance criteria, the developers were asked if in their opinion this approach did provide any benefits. A participant replied that he would benefit the most in "complex features" (P3), while another stated that the provided acceptance criteria made him feel sure that he "didn't miss anything" (P1).

The questionnaire revealed that all developers thought that acceptance criteria provided will help improving their understanding of the task (Figure 5.30) and again help them feel confident that they covered all aspects of the feature (Figure 5.31)

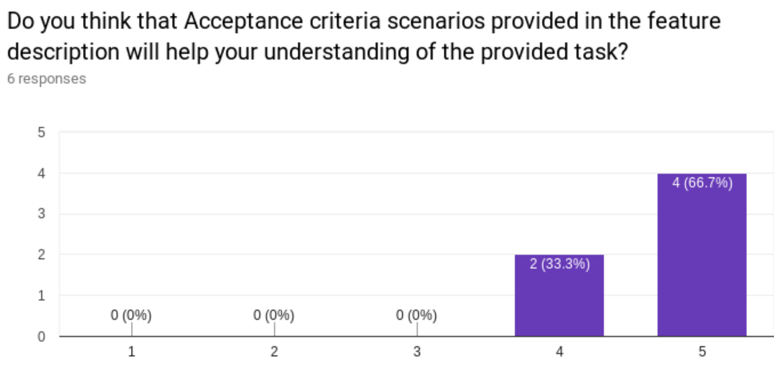


Figure 5.30: Acceptance Test - Feature Understanding

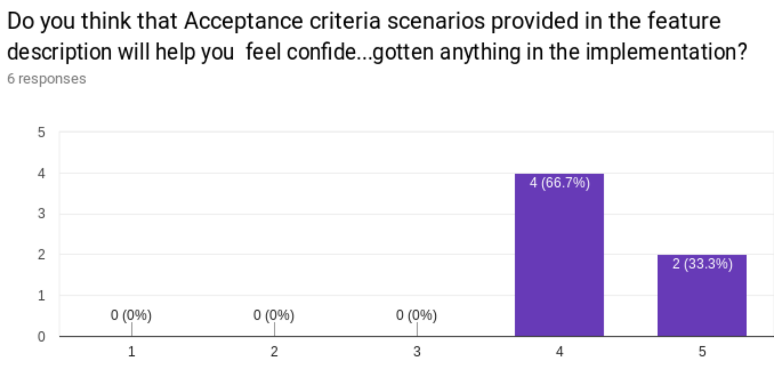


Figure 5.31: Acceptance Test - Feature Completeness

6 Discussion

This chapter discusses how and to what extent the stated research questions could be answered, using the collected data. Each research question will be discussed individually following the structuring provided in Section 3.3.

6.1 Introduction and Overview

The case study, forming the heart of this thesis, features the Mineralbay project as underlying case.

The author of this thesis had been part of the Mineralbay team for over a year, as a software developer. During this time, a careful analysis of the development process and all its weaknesses concerning SQA was performed. Furthermore the co-developers were surveyed and interviewed about their thoughts concerning the process and its potential flaws. A variety of possible enhancements was elaborated and subsequently expressed in a list of research questions.

In the next step, the author implemented all these changes and continued evaluating them in a laboratory setup.

In the following each research question will be addressed according to Section 3.3 and brought together with results of data collection and analysis.

6.2 Improvements on Process Level

The first research question asked which modifications were applied to the Mineralbay development process and whether or to which extend these modifications contributed to code quality.

6.2.1 Reviews

A sub-question focused on the impact of an adapted code review process on code quality: Reviews were performed on a feature basis, as well on a sprint basis.

The developers were advised to review each commit on its own, a task which usually was not a problem, since most of the code was written in pair programming.

Mandatory feature reviews took place, each time a feature branch was merged, aiming at spreading knowledge, as well as improving code quality. A checklist highlighting the most important tasks of a review was provided, aiding the reviewers.

A second review step was executed when a sprint branch was merged into the master (i.e. production ready branch). In contrast to feature reviews, these did not focus on low level code specifics. Procedures necessary for code base integration and process related tasks (like checking documentation, verifying correct software version number, etc.) were subject of sprint reviews.

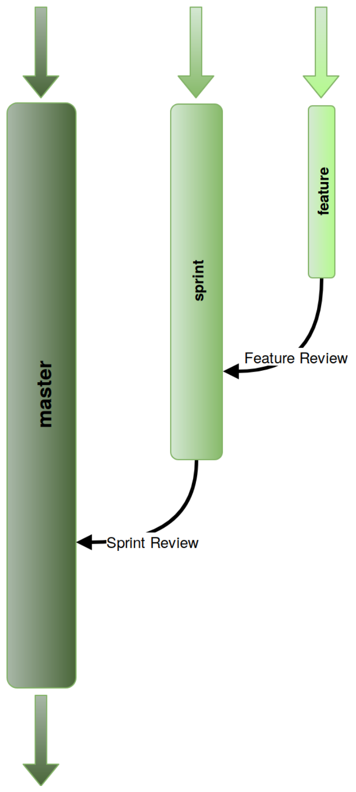


Figure 6.1: Code Review Process Illustrated

Analyzing the results it can be concluded that the importance of code reviews is indeed known to developers. They regard reviews as quality improving tasks, which are not perceived as an unnecessary burden. The developer's understanding of the application is improved by doing reviews, as well as their knowledge about developing software is widened.

Performing reviews before merging features, is important for code quality.

Reviews at the end of a sprint received mixed feedback. A less formal approach, like defining a DoD for Sprints, which hold all tasks that were featured in sprint reviews could be thought of. This DoD must be checked (e.g. by a developer responsible for the sprint) before concluding a sprint.

Furthermore assembling a simple checklist to guide the developers through the review process seems promising. The participating students felt supported by these checklists. One has to be careful though to find a balance between covering all important aspects of a review and keeping the checklists efficient. Additionally it is advised to keep checklists easily accessible through the review process: e.g. provide a link, or textual representation in the code review software.

6.2.2 Versioning

The author claimed that a profound versioning scheme was essential for meaningful quality assurance practices.

Prior to the authors changes this was not the case: developers did not know which version of the software was deployed at the development server, or even at production. They could not tell which features were already in production, nor relate them to a specific version.

A versioning scheme roughly based on sprint iterations was introduced and evaluated. Simple by nature, it was considered comprehensible by the students and did help relating features to software versions.

Versioning of the software must be part of the development process, changes carried out at well defined points and reviewed, in order to prevent neglect.

6.2.3 Self Assessments for Team Building

The setup of development teams was subject of another sub-question. The author experimented with skill-self-assessments for team building. The basic idea was that developers would rate their own skills in certain areas and then the ones with "complementary" knowledge should team up, in order to maximize knowledge transfer.

The desired approach did prove difficult since

- quantifying developer skills is hard (although possible since restricted to the given technology stack)
- assessing their own skills is difficult for developers

Evaluation in an experiment did show that most students prefer building teams according to intuition, e.g. placing a novice on the side of a senior.

Furthermore the importance of rotation of team-members was emphasized, a finding that is also represented in the results of Katrin Amtmanns thesis on pair programming [3, p. 86]. Amtmann also recommends that pairs should include at least one expert (senior project member). This will lead to better code quality [3, p. 86]. The rather complex domain of the Mineralbay project also requires the developers to be familiar with e.g. given terms, which means that in addition to technical skills, domain knowledge is also of importance and has to be transferred.

6.2.4 Test Software in a production like Environment

Testing software in a production like environment and configuration is essential for quality assurance. The author elaborated techniques using containerization, to allow for an easy deployment of a production like environment on the developers local workstations.

The setup was implemented by the author and afterwards evaluated in an experiment featuring the developers of Mineralbay. It can be concluded that containerization simplifies local deployment for developing purposes, as well as software testing. A carefully designed build process, producing images of each major version (e.g. feature, sprint, etc.) of the software facilitates testing substantially: Artifacts, ready for testing can be easily deployed and tested.

6.2.5 Gamification as tool for improving software quality

As part of this thesis the author evaluated if gamification techniques could be utilized for software quality assurance tasks.

In particular a software platform was developed which displayed rankings of:

- the number of bugs a developer had fixed

- the number of times a developer had broken the build (by contributing erroneous code)

Results showed that the bugfixing ranking has a highly motivating effect on developers. Gamification can be used to animate developers to fix software issues like bugs (even those that have been known for a long time, but were not considered worth fixing) or quality issues.

Gamification as a measure to prevent developers from submitting "bad" code, could not provide promising results in the way it was implemented. The reward system was not balanced and sophisticated well enough. If an approach like this is taken, the scenario must fit and more work has to be done, balancing the system.

In the current setup however the results did not have a measurable positive impact on motivating the developers not to break the build.

6.3 Improvements on Coding Level

The second research question asked if further quality enhancing methods could be included in the process of the developer writing code at his workstation.

6.3.1 Successful Local Build and Test Run

Development was slowed down, because students did not execute the test suites before pushing their code to the SCM, thus triggering many unnecessary builds. Since a build took approximately 40 minutes and only one build could be executed at the same time, this caused a serious bottleneck for development.

The author planed to utilize gamification to reward developers for successful builds and impose penalties for failed once. A global ranking should motivate them to trigger builds only after being confident that they will pass.

However the evaluation of this approach did prove less promising, since students did not feel motivated by the ranking approach.

The author then rebuilt the whole CI process of the project using state of the art tools (i.e. GitLab and GitLabCI). The resulting setup did allow for many parallel builds which moreover did execute much faster than in the previous setup. As a direct result the recorded issue ceased to exist and developers were very pleased with the build setup.

Consequently, resources spent in improving project infrastructure, aimed at removing bottlenecks and easing development are well invested. Existing tools and infrastructure must not hinder development in any way, since this frustrates developers and ultimately harms development.

6.3.2 Common Code Styles

Each developer usually acquires his own coding style, depending on education and preferred coding language. Developers are not indifferent to, but usually annoyed by badly formatted code in their projects. They are aware of the fact that well and uniformly formatted code is enhancing code readability, maintainability as well as quality.

Selecting an appropriate code style guideline for a software project is therefore especially important. In any case this is best done at the beginning of a project, because if neglected code styles will diverge, reducing readability and possibly harming team climate.

IDE plugins can be configured to support a given code style. These tools provide immediate feedback if the developers code diverges from the standard. Not only do such tools aid developers adhering to the given code styles by visually highlighting violations, but also allow for instant fixes using auto code-formatting.

Computer assistance does not end at simple code format. Style guidelines often cover more than simple style rules. Naming conventions, bad smells, etc. can be detected and visualized by linter tools, executed as external tools alongside every build or as IDE plugins on every code change. Again the setup is easily done, but provides developers with instant feedback in case of infringements.

It is advised to include, or enable code style checking tools in every developers IDE, as early as possible, in order to avoid code style divergences and help developers getting a feel for how code should be written. Such tools provide great assistance to developers in terms of ultimately increasing code quality, readability and maintainability.

6.3.3 Static Code Analysis

Static Code Analysis can be used to check the code basis for known issues like bugs, bad smells, security issues, etc. Including such tools into the software development process and providing access to the developers, for them to see the generated reports and statistics will aid overall quality. Doing so early in the course of the project (best at the beginning) will avoid having to deal with a big accumulation of issues later on.

Revealing statistics to the team about their improvements before and after a sprint can also have a motivating effect. Furthermore solving issues raised by the static code analysis tool will further improve the technical knowledge of developers, as they have to figure out why their code has led to an issue.

Goals in terms of committing to periodical code quality improvements can be formulated by the team. The author suggested a "not getting worse" strategy: After each sprint the overall calculated quality metrics must be equal or better than before the sprint. However a dedicated strategy has to be came up with for each project.

6.4 Improvements on Documentation Level

The third research question asked if in the given scenario, software quality could be enhanced by improving documentation and sharing knowledge.

6.4.1 Acceptance Tests written and fulfilled

Providing high quality, easily comprehensible requirements are a demanding task on its own. Considering the given case, the requirements sometimes failed at transferring the customers needs to the developers. Since developers were designing and implementing the BDD tests, these tests did only represent their view on requirements, which might differ from the customers.

Acceptance criteria as part of the requirements assures that BDD tests cover the customers needs exactly. Handing them over to the developers also provides multiple benefits during implementation:

- *Better understanding of the given task*: The compressed, summarized form of acceptance criteria provides a different view on required functionality, offering superior documentation alongside user stories.
- *Highlight the extent of features requested*: Since processes are described in very high detail, acceptance criteria gives confidence that all facets of the feature have been covered and implemented.

6.5 Discussion and Conclusions

6.5.1 Improvements for Student Software Projects

A significant output of this thesis are the results for upcoming student projects to draw knowledge from and a blueprint for development processes.

The following section will prepare the findings of this thesis as suggestions and improvements of the implementation of QA in student software projects.

- *Code Reviews*: Code reviews are most important for improving software quality and transferring technical as well as domain knowledge. It is strongly advised to implement a sound code review process at the beginning of the project to maximize its benefits. Furthermore checklists should be compiled which cover all aspects of a review. These should be kept easily accessible during the review process, supporting developers.
- *Management Reviews*: It is beneficial to perform reviews on documentation and process compliance as well, since tasks like increasing version numbers, setting ticket states, etc. are easily forgotten. Such process tied reviews could take place when completing a sprint.
- *DoD*: Compile a realistic, easily accessible DoD. An overshooting DoD, featuring unrealistic or extremely tiresome tasks, will most likely be ignored by the developers.
- *Development Infrastructure*: Tools like SCM, issue tracker, CI- and Continuous Delivery (CD)-Server have to be picked carefully. It is advised to setup an infrastructure that can be entirely managed by students. This forces students to get used to the complexity of DevOps as well as prevents tasks that can only be performed by system administrators from being a bottleneck.
- *Versioning*: Establishing a consistent versioning scheme at the beginning of the project is important. Otherwise versioning will never be done in a comprehensible way and thus nullifying all benefits of versioning.
- *Code Style*: The team must commit itself to a code style to prevent different "flavors" of code from growing inside the project. This code style should be supported by the IDE, but also be part of the review process.
- *Static Code Analysis*: Introducing static code analysis tools early in the development process will prevent potential issues from growing unnoticed. Overall quality can be kept on a high level without the need of performing dedicated "bugfixing" periods.
- *Team building*: Teams should consist of at least one experienced team member. While pair programming is perfectly fitted for knowledge transfer and tackling tricky tasks, it does not have to be mandatory, since there are tasks that do not benefit.
- *Testing of Deployables*: Testing of a certain version must be kept as simple as possible.
- *Acceptance criteria*: Requirements should be prepared with acceptance criteria. This verifies that the given task is easily comprehensible by developers.

6.6 Limitations of Results

One of the limitations of the presented approach is that due to the projects general development, the results of the experiments could not be transported over to the working project setup and thus be validated in a larger context. This was amended by having various rounds of interviews with different project members on qualitative aspects of both the existing setup and the proposed setup. For general transportability of the findings expert interviews were used to provide a classification of the setup its self as a typical student project.

The gained results are derived from a project regarded as a typical student project concerning its characteristics, via case study research. Provided insights will not apply to projects that differ in any of the stated properties. Moreover, generalization of results in the given environment is difficult, since each case setup is individual in design.

Although the development process is described as SCRUM like, it can only be considered a remote deviation from SCRUM, since it lacks important key characteristics like dailies, fixed sprint length, etc.

6.7 Related Work

Fortunato et al. evaluated and analyzed relevant studies on quality assurance practices in agile software development [31].

Their key findings can be cross referenced with this studies results:

- *"techniques such as coding standards, refactoring, creating unit tests, review code, pair programming and use of continuous integration tools are important factors for achieving the quality assurance"*: All these techniques were introduced in the course of this study and could be linked to a positive effect on code quality.
- *"Software testing proved as the factor, most likely to be incorrectly performed"*: During this study most developers stated that they were unhappy with the way they are testing their own code.
- *"The use of good programming practices is regarded as one of the pillars of quality assurance"*: Implementing and pushing towards good programming practices was part of this thesis approach towards improving software quality.

In [2] Ahtee et al. recognized that the four major risks in student projects are tools and skills to use the tools (61% of projects), technological problems (53%), scheduling problems (61%) and working or studying too many other courses during the project (45%). The same risks/issues could be observed during the course of this thesis too by the author.

Su Serene Hang and Scharff Christelle stated that motivation is the key factor of success [109]. The author of this study shares this opinion and thus took special care that non of the introduced practices will reduce motivation in any way. On the contrary, gamification techniques were adopted for the purpose of improving motivation.

Tiisetso Khalane Maureen Tanner were analyzing the concerns of stakeholders new to SCRUM in [55], stating that there is not enough guidance on how to implement SCRUM, i.e. how to implement software quality assurance. Their conclusion was that due to this lack of tangible guidance in Scrum, a development team has to come up with innovations which may include adopting practices from other methodologies. Furthermore the study proposes to view SCRUM as a framework of "empty buckets" which need to be filled with situation specific SQA practices and processes. Ming et al. arrived at a similar conclusion, stating that "agile methods do have practices that have QA abilities, some of them are inside the development phase and some others can be separated out as supporting practices" [44]. Furthermore, "developers also need to know how to revise or tailor their agile methods in order to attain the level of quality they require." [44] The results of this research inspired the author to do exactly that, by tailoring the software development process, by fitting adequate QA gates and measures.

The importance of testing is highlighted by the results of Doherty et al. by stating that "Acceptance, Integration and Performance (AIP) testing captures the largest quantity of defects at 26.14%" [23].

Janus et al. added continuous measurement and continuous improvement as subsequent activities to CI and established metric-based quality-gates for an agile quality assurance [51]. During the course of this study, the author implemented and successfully evaluated a similar approach, continuously measuring quality and thus striving for improvement.

Félix García et al. applied gamification techniques to parts of the software development cycle, by designing a framework for gamification in software engineering. A central gamification engine helped integrating existing tools, leading to the gamification of the areas of project management, requirements management, and testing. [33]

Another approach was taken by Gasca-Hurtado et al. who used gamification for defect tracking [34].

7 Conclusion and Future Work

This chapter concludes the results of the conducted case study, stating its goals and purposes as well as presenting its outcome. Since the scope of this thesis is limited, further research topics for future work will also be presented.

7.1 Conclusion

SQA in student projects is not just a topic, only sparsely covered by research, but also presents an important objective for educational institutions. Special challenges have to be addressed, because these projects differ from industry projects in various ways.

Software projects in typical Software Engineering (SE) courses typically have a very reduced scope and do not focus on the development of high quality software, but learning the practices and principles of a certain aspect in SE. Furthermore these projects usually do not provide the infrastructural frame necessary for implementing a sophisticated QA process.

The present thesis explored how a typical student project could be enriched with a sophisticated QA process. For the purpose of doing so a case study was conducted that examined the student project Mineralbay at the INSO of the TU. This project represents a typical student project, developed by students of the TU, each being different in terms of skill and educational level as well as practical experience. Development of the software was done using a tailored version of the iterative SCRUM paradigm.

The goal of the study was to find and evaluate a feasible set of QA practices and apply them to the process. Evaluation stretched for over a year and included the author actively participating in development as well as collecting data from the process and fellow developers. After identifying applicable practices and dividing them based on their level of application (inspired by [20] i.e. process, development or documentation) the author implemented these practices in the underlying setup. Subsequently their impact on software quality as well as development itself was measured. This was done in a controlled experiment, with data being collected by semi structured interviews, surveys as well as participant observation.

A set of suggestions on how QA could be implemented in a typical student project formed the outcome of the study together with a composition of how each implemented element performed on its own.

Results showed that the wide spread topic of QA can be integrated into a student software project of adequate scale and characteristics well.

The foundation of a solid source code basis is build by a sophisticated SCM tool, which supports developers during their daily business (e.g. code merges, branch management, etc.), making it as easy as possible. Introducing a new, state of the art SCM, offering a simple and powerful interface simplified processes like code reviews or code merges significantly.

Another aspect of QA is, that the code which is kept in the SCM does compile and functions according to its specification. This can be assured by providing a meaningful set of tests, but also by implementing a CI process that "guards" the code base from erroneous code being submitted. A CI pipeline was implemented as part of this thesis, which increased the developers satisfaction in terms of performance, maintainability as well as their confidence that changes which have passed the pipeline are free from serious faults and do not impact the core functionality of the software in a negative way.

Furthermore it was established that a solid review process is essential for improving quality and also supports knowledge transfer. Such a review process was implemented as part of this study, leading to an improvement in developer satisfaction concerning the process and greater confidence in its quality enhancing capabilities. Checklists were used to support developers during the process of reviewing, which helped them focus on its important parts as well as improving review quality.

The use of containers for application shipping significantly simplified local deployment of the software. In the process of QA this proved especially helpful when testing a specific version of the software, since deploying from a container tag is easier than from a source code checksum.

Directly connected to containerization and tagging is software versioning. Neglect of doing so lead to the developers not knowing which feature was related to which software version and further what exact version of the software was currently deployed. Establishing a versioning process helped assuring that features could be related to versions.

Uniform code style is essential for readability, maintainability and also related to working atmosphere. A code style guideline helped improving those factors, while IDE support assured that compliance did only require minimal effort.

Constant static analysis of the code base, using a dedicated server was introduced. This helped developers improving code quality, as well as their coding skills. Additionally, the comparison of code quality statistics (before and after a specific point in development like e.g. a sprint) could be used as a motivating factor.

Based on data aggregated by code analysis as well as build status information (deliverer by the CI tool) a gamification approach was evaluated with the objective of making QA related tasks more rewarding. Points and badges were awarded for each quality-increasing activity and the scores displayed on a board. Results showed that gamification can be used in the context of QA and did motivate developers in the evaluated setup.

Acceptance tests should not be designed by developers, but be part of the requirements. This strengthens the developers understanding of the given task and helps them feeling confident that they did not overlook any functionality.

Team building is important for knowledge transfer, but also code quality. Teams for pair programming should contain at least one experienced developer to maximize those factors. Occasional rotation is beneficial for knowledge transfer.

7.2 Future Work

The subject of gamification in the context of QA was only briefly touched by this study. It was established that elements of gamification can be utilized to motivate students into tackling otherwise tiresome QA related tasks.

However this subject offers many possibilities and current research on this topic is scarce. Further work could build on the given tools and ideas and evaluate the setup in more detail. Additionally more parts of the QA cycle could be combined with gamification.

Moreover this thesis could only highlight bits and parts of QA in a student software project. Further research could be done in terms of describing a general development process, including the findings of this study.

Due to the development of the thesis underlying case, transferring all of the experiment's results to the development process at hand was not possible. Future work could implement all evaluated measures in a student project of similar setup for the purpose of collecting quantitative data.

Bibliography

References

[1] Adeel Ahmed et al. „Agile software development: Impact on productivity and quality“. In: *Management of Innovation and Technology (ICMIT), 2010 IEEE International Conference on*. IEEE. 2010, pp. 287–291.

[2] Tero Ahtee and Timo Poranen. „Risks in students’ software projects“. In: *Software Engineering Education and Training, 2009. CSEET’09. 22nd Conference on*. IEEE. 2009, pp. 154–157.

[3] Katrin Amtmann. „Analysing the Effects of Pair Programming on Knowledge Transfer between Students in Academic Software Development“. MA thesis. TU Wien, 2016.

[4] Carina Andersson and Per Runeson. „A spiral process model for case studies on software quality monitoring—method and metrics“. In: *Software Process: Improvement and Practice* 12.2 (2007), pp. 125–140.

[5] Ben-Zion Barta, S. L. Hung, and K. R. Cox. *Software Engineering Education: Proceedings of the IFIP WG3.4/SEARCC (SRIG on Education and Training) Working Conference, Hong Kong, 28 September - 2 October, 1993*. IFIP Transactions A: Computer Science and Technology. Elsevier Science, 2013. ISBN: 9781483293257.

[6] Victor R Basili. „The role of controlled experiments in software engineering research“. In: *Empirical Software Engineering Issues. Critical Assessment and Future Directions*. Springer, 2007, pp. 33–37.

[7] Pamela Baxter and Susan Jack. „Qualitative case study methodology: Study design and implementation for novice researchers“. In: *The qualitative report* 13.4 (2008), pp. 544–559.

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

[9] Izak Benbasat, David K. Goldstein, and Melissa Mead. „The Case Research Strategy in Studies of Information Systems“. In: *MIS Q.* 11.3 (Sept. 1987), pp. 369–386. ISSN: 0276-7783. DOI: 10.2307/248684.

[10] Mark Benney and Everett C Hughes. „Of sociology and the interview: Editorial preface“. In: *American journal of sociology* 62.2 (1956), pp. 137–142.

[11] Lars Bratthall and Magne Jørgensen. „Can you trust a single data source exploratory software engineering case study?“ In: *Empirical Software Engineering* 7.1 (2002), pp. 9–26.

[12] John Paul Campbell, Richard L Daft, and Charles L Hulin. *What to study: Generating and developing research questions*. Vol. 32. Sage Beverly Hills, CA, 1982.

[13] Michael Christie et al. „Implementation of realism in case study research methodology“. In:

[14] Reidar Conradi and Alf Inge Wang. *Empirical Methods and Studies in Software Engineering: Experiences from ESERNET*. Lecture Notes in Computer Science. Springer, 2003. ISBN: 9783540406723.

[15] David Coppit and Jennifer M. Haddox-Schatz. „Large team projects in software engineering courses“. In: 37 (Feb. 2005), pp. 137–141.

[16] Juliet Corbin and Anselm Strauss. *Basics of qualitative research 3e*. 2008.

[17] John .W. Creswell. *Research Design: Qualitative, Quantitative, and Mixed Methods Approaches*. SAGE Publications, 2013. ISBN: 9781483321479.

[18] Philip B. Crosby. *Quality is Free: The Art of Making Quality Certain*. Mentor book. McGraw-Hill, 1979. ISBN: 9780070145122.

[19] Steve Dale. „Gamification: Making work fun, or making fun of work?“ In: *Business Information Review* 31.2 (2014), pp. 82–90.

[20] Galin Daniel. *Software Quality Assurance: From Theory to Implementation*. Pearson Education India, 2004.

[21] Norman K Denzin. *The research act: A theoretical introduction to sociological methods*. Transaction publishers, 1973.

[22] Sebastian Deterding et al. „From game design elements to gamefulness: defining gamification“. In: *Proceedings of the 15th international academic MindTrek conference: Envisioning future media environments*. ACM. 2011, pp. 9–15.

[23] B. Doherty et al. „Defect Analysis in Large Scale Agile Development: Quality in the Agile Factory Model“. In: *2016 Joint Conference of the International Workshop on Software Measurement and the International Conference on Software Process and Product Measurement (IWSM-MENSURA)*. 2016, pp. 180–180. DOI: 10.1109/IWSM-Mensura.2016.034.

[24] Peter Drucker. *Innovation and Entrepreneurship*. Routledge, 2012. ISBN: 9781136017612.

[25] Daniel J Dubois and Giordano Tamburrelli. „Understanding gamification mechanisms for software development“. In: *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*. ACM. 2013, pp. 659–662.

[26] Joseph Dyro. *Clinical Engineering Handbook*. Academic Press series in biomedical engineering. Elsevier Academic Press, 2004. ISBN: 9780122265709.

[27] Martin Entacher et al. „Wiederverwertung von Tunnelausbruchmaterial–Abfallrecht im Berg-und Tunnelbau“. In: *BHM Berg-und Hüttenmännische Monatshefte* 156.10 (2011), pp. 379–383.

[28] Hartmut Erben and Robert Galler. „Tunnel spoil–New technologies on the way from waste to raw material/Tunnelausbruch–Neue Technologien für den Weg vom Abfall zum Rohstoff“. In: *Geomechanics and Tunnelling* 7.5 (2014), pp. 402–410.

[29] Hartmut Erben, Robert Galler, and Thomas Grechenig. „MineralBay – the portal for raw materials and projects from subsurface construction / MineralBay – das Portal für mineralische Rohstoffe und Projekte aus dem Untertagebau“. In: *Geomechanics and Tunnelling* 8.4 (2015), pp. 321–332. ISSN: 1865-7389.

[30] Bent Flyvbjerg. „Five misunderstandings about case-study research“. In: *Qualitative inquiry* 12.2 (2006), pp. 219–245.

[31] Carlos Alberto Fortunato et al. „Quality Assurance in Agile Software Development: A Systematic Review“. In: *Brazilian Workshop on Agile Methods*. Springer. 2016, pp. 142–148.

[32] Andrés Francisco Aparicio et al. „Analysis and application of gamification“. In: Oct. 2012. DOI: 10.1145/2379636.2379653.

[33] Félix García et al. „A framework for gamification in software engineering". In: *Journal of Systems and Software* 132 (2017), pp. 21 –40. ISSN: 0164-1212. DOI: https://doi.org/10.1016/j.jss.2017.06.021.

[34] Gloria Piedad Gasca-Hurtado et al. „Gamification Proposal for Defect Tracking in Software Development Process". In: *Systems, Software and Services Process Improvement*. Ed. by Christian Kreiner et al. Cham: Springer International Publishing, 2016, pp. 212–224. ISBN: 978-3-319-44817-6.

[35] Reinhard Gerndt, Ina Schiering, and Jens Lüssem. „Elements of Scrum in a Students Robotics Project – A Case Study". In: 8 (Jan. 2014), pp. 37–45.

[36] Alan Gillies. *Software Quality: Theory and Management*. 2011. ISBN: 9781446753989.

[37] Barney G Glaser and Anselm L Strauss. *The discovery of grounded theory: Strategies for qualitative research*. Transaction publishers, 2009.

[38] Thomas Grechenig. *Softwaretechnik: Mit Fallbeispielen aus realen Entwicklungsprojekten*. IT Informatik. Pearson Studium, 2010. ISBN: 9783868940077.

[39] Brian Henderson-Sellers. *Object-oriented Metrics: Measures of Complexity*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1996. ISBN: 0-13-239872-9.

[40] Anthony Hill et al. „Non-response bias in a lifestyle survey". In: *Journal of Public Health* 19.2 (1997), pp. 203–207.

[41] John W. Horch. *Practical Guide to Software Quality Management*. Artech House computing library. Artech House, 2003. ISBN: 9781580536042.

[42] Martin Host and Per Runeson. „Checklists for software engineering case study research". In: *Empirical Software Engineering and Measurement, 2007. ESEM 2007. First International Symposium on*. IEEE. 2007, pp. 479–481.

[43] Siw Elisabeth Hove and Bente Anda. „Experiences from conducting semi-structured interviews in empirical software engineering research". In: *Software metrics, 2005. 11th ieee international symposium*. IEEE. 2005, 10–pp.

[44] Ming Huo et al. „Software quality and agile methods". In: *Computer Software and Applications Conference, 2004. COMPSAC 2004. Proceedings of the 28th Annual International*. IEEE. 2004, pp. 520–525.

[45] Kai Huotari and Juho Hamari. „Defining gamification: a service marketing perspective". In: *Proceeding of the 16th International Academic MindTrek Conference*. ACM. 2012, pp. 17–22.

[46] „IEEE Standard for a Software Quality Metrics Methodology". In: *IEEE Std 1061-1998* (1998), pp. i–. DOI: 10.1109/IEEESTD.1998.243394.

[47] „IEEE Standard for Software Quality Assurance Plans". In: *IEEE 730-1989* (1989), pp. 1–12. DOI: 10.1109/IEEESTD.1989.82361.

[48] „IEEE Standard for Software Quality Assurance Processes". In: *IEEE Std 730-2014 (Revision of IEEE Std 730-2002)* (2014), pp. 1–138. DOI: 10.1109/IEEESTD.2014.6835311.

[49] „IEEE Standard Glossary of Software Engineering Terminology". In: *IEEE Std 610.12-1990* (1990), pp. 1–84. DOI: 10.1109/IEEESTD.1990.101064.

[50] Professor Carolyn S Ridenour Isadore Newman Carolyn R. Benz. *Qualitative-quantitative Research Methodology: Exploring the Interactive Continuum*. Southern Illinois University Press, 1998. ISBN: 9780809321506.

[51] André Janus et al. „The 3C Approach for Agile Quality Assurance“. In: *Proceedings of the 3rd International Workshop on Emerging Trends in Software Metrics*. WETSoM '12. Zurich, Switzerland: IEEE Press, 2012, pp. 9–13.

[52] Joseph M Juran et al. *Juran on planning for quality*. Free Press New York, 1988.

[53] Stephen H. Kan. *Metrics and Models in Software Quality Engineering*. Addison-Wesley, 2003. ISBN: 9780201729153.

[54] Stephen Kemmis and Robin McTaggart. *Participatory action research: Communicative action and the public sphere.* Sage Publications Ltd, 2005.

[55] T. Khalane and M. Tanner. „Software quality assurance in Scrum: The need for concrete guidance on SQA strategies in meeting user expectations“. In: *Adaptive Science and Technology (ICAST), 2013 International Conference on*. 2013, pp. 1–6.

[56] Barbara Kitchenham and Shari Lawrence Pfleeger. „Software quality: the elusive target [special issues section]“. In: *IEEE software* 13.1 (1996), pp. 12–21.

[57] Barbara Kitchenham, Lesley Pickard, and Shari Lawrence Pfleeger. „Case Studies for Method and Tool Evaluation“. In: *IEEE Softw.* 12.4 (July 1995), pp. 52–62. ISSN: 0740-7459. DOI: 10.1109/52.391832.

[58] Barbara A. Kitchenham. „Software Quality Assurance“. In: *Microprocess. Microsyst.* 13.6 (July 1989), pp. 373–381. ISSN: 0141-9331. DOI: 10.1016/0141-9331(89)90045-8.

[59] Jyrki Kontio, Johanna Bragge, and Laura Lehtola. „The focus group method as an empirical tool in software engineering“. In: *Guide to advanced empirical software engineering*. Springer, 2008, pp. 93–116.

[60] Jyrki Kontio, Laura Lehtola, and Johanna Bragge. „Using the focus group method in software engineering: obtaining practitioner and user experiences“. In: *Empirical Software Engineering, 2004. ISESE'04. Proceedings. 2004 International Symposium on*. IEEE. 2004, pp. 271–280.

[61] Martin Kunz, Reiner R Dumke, and Niko Zenker. „Software metrics for agile software development“. In: *Software Engineering, 2008. ASWEC 2008. 19th Australian Conference on*. IEEE. 2008, pp. 673–678.

[62] Stephen D Lapan et al. *Foundations for research: Methods of inquiry in education and the social sciences*. Routledge, 2003.

[63] Craig Larman. *Agile and iterative development: a manager's guide*. Addison-Wesley Professional, 2004.

[64] Timothy C. Lethbridge, Susan Elliott Sim, and Janice Singer. „Studying Software Engineers: Data Collection Techniques for Software Field Studies“. In: *Empirical Software Engineering* 10.3 (2005), pp. 311–341. DOI: 10.1007/s10664-005-1290-x.

[65] Timothy C Lethbridge, Susan Elliott Sim, and Janice Singer. „Studying software engineers: Data collection techniques for software field studies“. In: *Empirical software engineering* 10.3 (2005), pp. 311–341.

[66] Yvonna S Lincoln and Egon G Guba. *Naturalistic inquiry*. Vol. 75. Sage, 1985.

[67] K Louise Barriball and Alison While. „Collecting Data using a semi-structured interview: a discussion paper“. In: *Journal of advanced nursing* 19.2 (1994), pp. 328–335.

[68] L.S. Luton. *Qualitative Research Approaches for Public Administration*. Taylor & Francis, 2015. ISBN: 9781317461562.

[69] Andrew Marrington, James M Hogan, and Richard Thomas. „Quality assurance in a student-based agile software engineering process“. In: *Software Engineering Conference, 2005. Proceedings. 2005 Australian*. IEEE. 2005, pp. 324–331.

[70] John McManus. *Risk Management in Software Development Projects*. Computer weekly professional series. Elsevier Butterworth-Heinemann, 2004. ISBN: 9780750658676.

[71] K Mirnalini and Venkata R Raya. „Agile-A software development approach for quality software“. In: *Educational and Information Technology (ICEIT), 2010 International Conference on*. Vol. 1. IEEE. 2010, pp. V1–242.

[72] Ernest Mnkandla and Barry Dwolatzky. „Defining agile software quality assurance“. In: *Software Engineering Advances, International Conference on*. IEEE. 2006, pp. 36–36.

[73] Yasuhiro Monden. *Toyota production system: an integrated approach to just-in-time*. CRC Press, 2011.

[74] Michal Musial. „Introduction to Empirical Experiments in Software Engineering“. In: Citeseer, 2010.

[75] Shari Lawrence Pfleeger Norman E. Fenton. *Software Metrics: A Rigorous and Practical Approach, Third Edition*. Chapman & Hall/CRC Innovations in Software Engineering and Software Development Series. CRC Press, 2014. ISBN: 9781439838235.

[76] Charles L Owen. „Product integrity by design“. In: ().

[77] Gerard O'Regan. *Introduction to Software Quality*. Undergraduate Topics in Computer Science. Springer International Publishing, 2014. ISBN: 9783319061061.

[78] Christopher J Pannucci and Edwin G Wilkins. „Identifying and avoiding bias in research“. In: *Plastic and reconstructive surgery* 126.2 (2010), p. 619.

[79] Oscar Pedreira et al. „Gamification in software engineering–A systematic mapping“. In: *Information and Software Technology* 57 (2015), pp. 157–168.

[80] Pedro Pereira et al. „A review of gamification for health-related contexts“. In: *International conference of design, user experience, and usability*. Springer. 2014, pp. 742–753.

[81] Dewayne E. Perry, Adam A. Porter, and Lawrence G. Votta. „Empirical Studies of Software Engineering: A Roadmap“. In: *Proceedings of the Conference on The Future of Software Engineering*. ICSE '00. Limerick, Ireland: ACM, 2000, pp. 345–355. ISBN: 1-58113-253-0. DOI: 10.1145/336512.336586.

[82] Dewayne E Perry, Susan Elliott Sim, and Steve M Easterbrook. „Case studies for software engineers“. In: *Software Engineering, 2004. ICSE 2004. Proceedings. 26th International Conference on*. IEEE. 2004, pp. 736–738.

[83] Shari Lawrence Pfleeger. „Design and Analysis in Software Engineering: The Language of Case Studies and Formal Experiments“. In: *SIGSOFT Softw. Eng. Notes* 19.4 (Oct. 1994), pp. 16–20. ISSN: 0163-5948. DOI: 10.1145/190679.190680.

[84] Mary Poppendieck. „Lean software development“. In: *Companion to the proceedings of the 29th International Conference on Software Engineering*. IEEE Computer Society. 2007, pp. 165–166.

[85] Mary Poppendieck and Tom Poppendieck. *Implementing lean software development: From concept to cash*. Pearson Education, 2007.

[86] Mary Poppendieck and Tom Poppendieck. *Lean Software Development: An Agile Toolkit: An Agile Toolkit*. Addison-Wesley, 2003.

[87] Republik Österreich. *Bundesgesetz vom 7. Juni 1989 zur Finanzierung und Durchführung der Altlastensanierung (Altlastensanierungsgesetz).* https://www.ris.bka.gv.at/. 1989.

[88] Republik Österreich. *Bundesgesetz über mineralische Rohstoffe.* https://www.ris.bka.gv.at/. 1999.

[89] Republik Österreich. *Verordnung des Bundesministers für Land- und Forstwirtschaft, Umwelt und Wasserwirtschaft über Deponien.* https://www.ris.bka.gv.at/. 2008.

[90] C. Robson and K. McCartan. *Real World Research.* Wiley, 2016. ISBN: 9781118745236.

[91] Jennifer Rowley. „Using case studies in research“. In: *Management research news* 25.1 (2002), pp. 16–27.

[92] K.S. Rubin. *Essential Scrum: A Practical Guide to the Most Popular Agile Process.* Addison-Wesley Signature Series (Cohn). Pearson Education, 2012. ISBN: 9780321700377.

[93] Per Runeson and Martin Höst. „Guidelines for conducting and reporting case study research in software engineering“. In: *Empirical Software Engineering* 14.2 (2008), p. 131. ISSN: 1573-7616. DOI: 10.1007/s10664-008-9102-8.

[94] Per Runeson et al. *Case Study Research in Software Engineering: Guidelines and Examples.* 1st. Wiley Publishing, 2012. ISBN: 1118104358, 9781118104354.

[95] Maria Sagheer, Tehreem Zafar, and Mehreen Sirshar. „A Framework For Software Quality Assurance Using Agile Methodology“. In: *International Journal of Scientific and Technology Research* 4.02 (2015), pp. 44–50.

[96] Alexander Schatten et al. *Best Practice Software-Engineering: Eine praxiserprobte Zusammenstellung von komponentenorientierten Konzepten, Methoden und Werkzeugen.* Spektrum Akademischer Verlag, 2010. ISBN: 9783827424877.

[97] Wilbur Schramm. „Notes on Case Studies of Instructional Media Projects.“ In: (1971).

[98] Wolfgang Schramm, Christopher Draeger, and Thomas Grechenig. „Issues and mitigation strategies when using agile industrial software development processes in student software engineering projects“. In: *AFRICON, 2011.* IEEE. 2011, pp. 1–4.

[99] Ken Schwaber. „Scrum Development Process. OOPSLA'95 Workshop on Business Object Design and Implementation“. In: *Austin, USA* (1995).

[100] Ken Schwaber and Mike Beedle. *Agile software development with Scrum.* Vol. 1. Prentice Hall Upper Saddle River, 2002.

[101] Ken Schwaber and Jeff Sutherland. *The scrum guide - the definitive guide to scrum: The rules of the game.* Ed. by scrum.org. 2013. URL: https://www.scrumguides.org/docs/scrumguide/v1/scrum-guide-us.pdf.

[102] Carolyn B. Seaman. „Qualitative methods in empirical studies of software engineering“. In: *IEEE Transactions on software engineering* 25.4 (1999), pp. 557–572.

[103] Todd Sedano, Paul Ralph, and Cécile Péraire. „Software development waste“. In: *Proceedings of the 2017 International Conference on Software Engineering, ICSE.* Vol. 17. 2017, p. 98.

[104] Mary Shaw and James Tomayko. *Models for Undergraduate Project Courses in Software Engineering.* Tech. rep. CMU/SEI-91-TR-010. Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 1991.

[105] Forrest Shull, Janice Singer, and Dag I. K. Sjberg. *Guide to Advanced Empirical Software Engineering*. 1st. Springer Publishing Company, Incorporated, 2010. ISBN: 1849967121, 9781849967129.

[106] Janice Singer and Norman G. Vinson. „Ethical issues in empirical studies of software engineering“. In: *IEEE Transactions on Software Engineering* 28.12 (2002), pp. 1171–1180.

[107] Narinder Pal Singh and Rachna Soni. „Agile software: Ensuring quality assurance and processes“. In: *High Performance Architecture and Grid Computing* (2011), pp. 640–648.

[108] Dag I. K. Sjoberg, Tore Dyba, and Magne Jorgensen. „The Future of Empirical Methods in Software Engineering Research“. In: *Future of Software Engineering, 2007. FOSE '07*. 2007, pp. 358–378. DOI: 10.1109/FOSE.2007.30.

[109] Serene Hang Su and Christelle Scharff. „Know Yourself and Beyond: A students' global software development project experience with Agile Methodology“. In: *Computer Science and Education (ICCSE), 2010 5th International Conference on*. IEEE. 2010, pp. 412–417.

[110] Hirotaka Takeuchi and Ikujiro Nonaka. „The new new product development game“. In: *Harvard Business Review* (1986).

[111] Steven J Taylor, Robert Bogdan, and Marjorie DeVault. *Introduction to qualitative research methods: A guidebook and resource*. John Wiley & Sons, 2015.

[112] The Standish Group. *Chaos Report*. 2015.

[113] Jeff Tian. *Software Quality Engineering: Testing, Quality Assurance, and Quantifiable Improvement*. Wiley - IEEE. Wiley, 2005. ISBN: 9780471722335.

[114] June M Verner et al. „Guidelines for industrially-based multiple case studies in software engineering“. In: *Research Challenges in Information Science, 2009. RCIS 2009. Third International Conference on*. IEEE. 2009, pp. 313–324.

[115] VersionOne. „State of Agile Development Survey Results“. In: 2015.

[116] K. Werbach and D. Hunter. *For the Win: How Game Thinking Can Revolutionize Your Business*. Wharton Digital Press, 2012. ISBN: 9781613630235.

[117] Claes Wohlin et al. *Experimentation in Software Engineering*. Springer Publishing Company, Incorporated, 2012. ISBN: 3642290434, 9783642290435.

[118] R.K. Yin. *Case Study Research: Design and Methods*. Applied Social Research Methods. SAGE Publications, 2009. ISBN: 9781412960991.

[119] G. Zichermann and C. Cunningham. *Gamification by Design: Implementing Game Mechanics in Web and Mobile Apps*. O'Reilly Series. O'Reilly Media, 2011. ISBN: 9781449397678.

A Appendix

A.1 Initial Questionnaire

Initial Questionnaire

This questionnaire is part of my master thesis. All submissions are anonymous and will only be used in the context of my thesis.

Demographic Questions

Please answer some questions about yourself!

1. **What is your gender**

   Mark only one oval.

   - Female
   - Male
   - Prefer not to say

2. **What ist the Name of your study programme?**

   Mark only one oval.

   - Medieninformatik und Visual Computing
   - Medizinische Informatik
   - Software Engineering
   - Technische Informatik
   - Logic and Computation
   - Visual Computing
   - Medieninformatik
   - Other: ______________________________

3. **How many month have you been part of the project?**

   Mark only one oval.

   - I'm new!
   - Other: ______________________________

4. **Are you currently enrolled as a Bachelors or Masters student?**

   Mark only one oval.

   - Bachelor
   - Master

5. **Since when are you enrolled in your current studies? (Year and term: e.g. WS2011)**

   ____

6. **Are you participating as a full-time student, or are you employed next to your studies?**

   Mark only one oval.

   - Full-time student
   - employed

7. **(If employed) How many years have you been working in your current job?**

____

8. **(If employed) How many years of programming experience did you gather so far?**

____

9. **Did you ever work in a software engineering team with more than 5 people?**
Mark only one oval.

- ⬭ No
- ⬭ Yes

10. **(If experience in teams >= 5) How big was your team?**

____

11. **(If experience in teams >= 5) Where did you work in a software engineering team with more than people?**
Check all that apply.

- ☐ University
- ☐ Work
- ☐ Other: ______________________________

12. **Did you ever use SCRUM in software development so far?**
Check all that apply.

- ☐ No
- ☐ Yes (University)
- ☐ Yes (Work)
- ☐ Other: ______________________________

Thoughts about your Project

The following section contains questions that will ask you to rate parts of your project

13. **How would you rate the overall code quality of the project?**
Mark only one oval per row.

	Miserable	Not satisfying, needs to be improved	Average	OK, but needs polishing	I think its fine
Backend	⬭	⬭	⬭	⬭	⬭
Frontend	⬭	⬭	⬭	⬭	⬭

14. **How would you rate the overall test quality of the project?**
Mark only one oval per row.

	We have tests?	Poor	Average	OK, but needs polishing	Well tested
Backend	⬭	⬭	⬭	⬭	⬭
Frontend	⬭	⬭	⬭	⬭	⬭

15. **Which step do you think is causing the most defects?**
Mark only one oval.

- ◯ Requirements & Specification
- ◯ Coding
- ◯ Review
- ◯ QA / Testing
- ◯ Other: ______________________

16. **Do you think that all features are "done" when they are merged in the master?**
Mark only one oval.

	1	2	3	4	5	
Do not agree	◯	◯	◯	◯	◯	Strongly agree

17. **Are you sure about which feature is in which version of the software?**
Mark only one oval.

	1	2	3	4	5	
I have no clue	◯	◯	◯	◯	◯	Yes, I know exactly

18. **How happy are you with the current build process?**
Mark only one oval.

	1	2	3	4	5	
Totally unhappy	◯	◯	◯	◯	◯	It is perfect!

Review Process

This section contains some questions about the current review process

19. **How happy are you with the current review process?**
Mark only one oval.

	1	2	3	4	5	
Totally unhappy	◯	◯	◯	◯	◯	Yes, its perfect

20. **Do you think that Pair Programming is enhancing code quality?**
Mark only one oval.

	1	2	3	4	5	
Not at all	◯	◯	◯	◯	◯	Absolutely

21. **How much additional effort do you think Pair Programming costs?**
Mark only one oval.

	1	2	3	4	5	
No time at all	◯	◯	◯	◯	◯	A lot of time

22. **Do you think that Peer-Code-Reviews are enhancing code quality?**
Mark only one oval.

	1	2	3	4	5	
Not at all	⬭	⬭	⬭	⬭	⬭	Absolutely

23. **Do you think that Pre-Master-Merge Reviews are enhancing code quality?**
Mark only one oval.

	1	2	3	4	5	
Not at all	⬭	⬭	⬭	⬭	⬭	Absolutely

24. **How much time do Pre-Master-Merge Reviews cost in your opinion?**
Mark only one oval.

	1	2	3	4	5	
Very little	⬭	⬭	⬭	⬭	⬭	Very much

Self assesment

The following section will ask question about how you rate your own work/skills

25. **How would you rate your own skills, concerning these technologies?**
Mark only one oval per row.

	Never heard of it	I know what it is and does	I tried it and know a bit about it	I get things done with it	I am very good at it
PostgreSQL	⬭	⬭	⬭	⬭	⬭
Liquibase	⬭	⬭	⬭	⬭	⬭
Maven	⬭	⬭	⬭	⬭	⬭
Java	⬭	⬭	⬭	⬭	⬭
Spring	⬭	⬭	⬭	⬭	⬭
Bower	⬭	⬭	⬭	⬭	⬭
Javascript	⬭	⬭	⬭	⬭	⬭
AngularJS	⬭	⬭	⬭	⬭	⬭
GIT	⬭	⬭	⬭	⬭	⬭
GitLab	⬭	⬭	⬭	⬭	⬭
Geb Testing Framework	⬭	⬭	⬭	⬭	⬭

26. **How would you rate your own testing skills?**
Mark only one oval per row.

	Never done it	I copy and paste it	I know the basics	I get things done with it	I am very good at it
Backend (Unit, Integration,..)	⬭	⬭	⬭	⬭	⬭
Frontend (Browser Tests)	⬭	⬭	⬭	⬭	⬭

27. **Are you happy with the way you test your own code?**
Mark only one oval per row.

	I'm not happy	-	Its ok, but I could do better	-	I'm happy
Backend	◯	◯	◯	◯	◯
Frontend	◯	◯	◯	◯	◯

28. **Do you run all tests locally before pushing into the SCM?**
Mark only one oval.

- ◯ No
- ◯ Yes, but sometimes when im in a hurry I ignore them
- ◯ Always
- ◯ Other: ____________________

29. **Do you know that there is a Definition of Done (DOD)?**
Link to DOD:
Mark only one oval.

- ◯ Yes
- ◯ No

30. **Do you understand all items of the DOD?**
If not please state which:
Mark only one oval.

- ◯ Yes
- ◯ Other: ____________________

31. **Do you check the DOD before pushing your features?**
Mark only one oval.

- ◯ Yes
- ◯ No

32. **Do you ignore parts of the DOD before pushing your features?**
If so please state why:
Mark only one oval.

- ◯ No
- ◯ Other: ____________________

33. **Do you think that QA would work better if a production like environment was be available for testing?**
Mark only one oval.

	1	2	3	4	5	
Not at all	◯	◯	◯	◯	◯	Yes a lot

34. **What do you think about written down patterns describing how reoccurring tasks are done?**

Mark only one oval.

- Would help me a lot
- Would help me a bit
- Don't Care
- Other: ____________________

35. **Do you think that Continuous Code Analysis would improve code quality**

Mark only one oval.

	1	2	3	4	5	
Not at all						Yes, a lot

A.2 Initial Interview Schedule

A.2.1 Introduction

- The interview session starts with the author introducing himself to the attendees and then stating the purpose of the interview, i.e. to contribute to this thesis.
- Afterwards it is emphasized that the results will only be used in an anonymized way.
- The respondents are then asked if it is ok to record the session.
- It is made clear, that interrupting is fine at any give time.

A.2.2 Starter Questions

General questions are asked, to "break the ice" and make the interviewees familiar with the situation:

- How many active developers are there at the moment?
- Is that enough / or would you need support?
- Are you doing a thesis or a "praktikum"?

A.2.3 Main Questions

The main body of questions is asked:

- Concerning the code: Where do you see the biggest deficits?
- Do you think that unnecessary errors are introduced during coding? Do you see any problems?
- How do you know which feature is in which version of the software? Do you know which version is deployed dev/prod? How is versioning done?
- What are the problems with the current build process?
- What are the problems with the current review process?
- What is the biggest bottleneck in the development process? What are the impediments? What is annoying you?
- What problems do you see when you think of testing? How is testing done?
- Do you see any problems with our DoD?
- Do you think that knowledge exchange is a problem?

A.3 Post Experiment Questionnaire

Post Lab Questionnaire

This questionnaire is part of my master thesis. All submissions are anonymous and will only be used in the context of my thesis.

Demographic Questions

Please answer some questions about yourself!

1. **What is your gender**

   Mark only one oval.

   - ( ) Female
   - ( ) Male
   - ( ) Prefer not to say

2. **What ist the Name of your study programme?**

   Mark only one oval.

   - ( ) Medieninformatik und Visual Computing
   - ( ) Medizinische Informatik
   - ( ) Software Engineering
   - ( ) Technische Informatik
   - ( ) Logic and Computation
   - ( ) Visual Computing
   - ( ) Medieninformatik
   - ( ) Other: ______________________

3. **How many month have you been part of the project?**

   Please state only the numerical value (e.g. "5")

   ____

4. **Are you currently enrolled as a Bachelors or Masters student?**

   Mark only one oval.

   - ( ) Bachelor
   - ( ) Master

5. **Since when are you enrolled in your current studies?**

   (Year and term: e.g. "WS2011" or "SS2014")

   ____

6. **Are you participating as a full-time student, or are you employed next to your studies?**

   Mark only one oval.

   - ( ) Full-time student
   - ( ) employed

7. **(If employed) How many years have you been working in your current job?**
   Please state only the numerical value (e.g. "5")

   ____

8. **(If employed) How many years of programming experience did you gather so far?**
   Please state only the numerical value (e.g. "5")

   ____

9. **Did you ever work in a software engineering team with more than 5 people?**
   Mark only one oval.

   ◯ No
   ◯ Yes

10. **(If experience in teams >= 5) How big was your team?**
    Please state only the numerical value (e.g. "5")

    ____

11. **(If experience in teams >= 5) Where did you work in a software engineering team with more than people?**
    Check all that apply.

    ☐ University
    ☐ Work
    ☐ Other: ______________________________

12. **Did you ever use SCRUM in software development so far?**
    Check all that apply.

    ☐ No
    ☐ Yes (University)
    ☐ Yes (Work)
    ☐ Other: ______________________________

Review Process - Checklist

The following section will ask question about the proposed review process

13. **Do you think that the provided checklist will improve your reviews?**
    Mark only one oval.

	1	2	3	4	5	
Not at all	◯	◯	◯	◯	◯	Totally

14. **Do you think that the provided checklist will improve code quality?**
Mark only one oval.

	1	2	3	4	5	
Not at all	◯	◯	◯	◯	◯	Totally

15. **Did the provided checklist help you focus on the important bits of a review?**
Mark only one oval.

	1	2	3	4	5	
Not at all	◯	◯	◯	◯	◯	Totally

Review Process - Reviews

The following section will ask question about the proposed review process

16. **How much knowledge do you gain from doing reviews (as a reviewer)?**
Mark only one oval.

	1	2	3	4	5	
Nothing at all	◯	◯	◯	◯	◯	Quite a lot

17. **Do you think that reviews improve your understanding of the application?**
Mark only one oval.

	1	2	3	4	5	
Not at all	◯	◯	◯	◯	◯	Quite a lot

18. **Do you think that Feature-Reviews are enhancing code quality?**
feature reviews: Reviews from a merge request from a feature branch to the sprint branch
Mark only one oval.

	1	2	3	4	5	
Not at all	◯	◯	◯	◯	◯	Quite a lot

19. **Do you think that Sprint-Reviews are enhancing code quality?**
sprint reviews: Reviews from a merge request from a sprint branch to the master branch
Mark only one oval.

	1	2	3	4	5	
Not at all	◯	◯	◯	◯	◯	Quite a lot

Self assessments

The following section will ask question about the "self assessments" used for team building during the lab

20. **Do you think that the learning experience is improved utilizing Self assessments for team building**

Self assessment team building: Selecting partners based on their project experience/skills

Mark only one oval.

	1	2	3	4	5	
Not at all	⬭	⬭	⬭	⬭	⬭	Quite a lot

21. **Did you benefit from your selected partner during the lab?**

Mark only one oval.

	1	2	3	4	5	
Not at all	⬭	⬭	⬭	⬭	⬭	Quite a lot

22. **Did your partner slow you down?**

Mark only one oval.

	1	2	3	4	5	
Not at all	⬭	⬭	⬭	⬭	⬭	Quite a lot

Versioning

23. **Do you understand the proposed versioning schema?**

Versioning based on the sprint number

Mark only one oval.

	1	2	3	4	5	
Not at all	⬭	⬭	⬭	⬭	⬭	Totally

24. **Based on the proposed versioning schema: Would you know which feature is in which version?**

Mark only one oval.

	1	2	3	4	5	
Not at all	⬭	⬭	⬭	⬭	⬭	Totally

Build Process

25. **Do you understand the given build pipeline?**

Mark only one oval.

	1	2	3	4	5	
Not at all	⬭	⬭	⬭	⬭	⬭	totally

26. **Do you think that the build pipeline is important for quality assurance?**
Mark only one oval.

	1	2	3	4	5	
Not at all	◯	◯	◯	◯	◯	totally

27. **Do you feel safe, that the build pipeline prevents errors from going to production?**
Mark only one oval.

	1	2	3	4	5	
Not at all	◯	◯	◯	◯	◯	totally

28. **Are you happy with the performance of the given build pipeline?**
Mark only one oval.

	1	2	3	4	5	
Not at all	◯	◯	◯	◯	◯	totally

Branching Model

29. **Do you understand the branching model?**
Mark only one oval.

	1	2	3	4	5	
Not at all	◯	◯	◯	◯	◯	totally

30. **Do you think the branching model will help to merge features faster into the master?**
Mark only one oval.

	1	2	3	4	5	
Not at all	◯	◯	◯	◯	◯	totally

Test Environment

31. **Given the containerized setup: How difficult would you say it is to deploy a specific version of the software locally?**
An Image is build for each version and stored in the registry
Mark only one oval.

	1	2	3	4	5	
Very Easy	◯	◯	◯	◯	◯	Very Difficult

32. **Given the docker-compose setup: How difficult would you say it is to deploy a production like environment locally?**
Mark only one oval.

	1	2	3	4	5	
Very Easy	◯	◯	◯	◯	◯	Very Difficult

33. **Do you think that running the application locally (on a "blank" system) has become easier with docker containers?**
"blank" system: Project has not been set up for development,.. Just the source code available
Mark only one oval.

	1	2	3	4	5	
Not At All	◯	◯	◯	◯	◯	Totally

Code Style Guidelines

34. **Does badly formated code annoy you?**
Mark only one oval.

	1	2	3	4	5	
Not At All	◯	◯	◯	◯	◯	Totally

35. **Do you think that IDE build in code style checks will improve Code Quality?**
IDE build in code style checks: Tools like the JS Linter presented in the lab
Mark only one oval.

	1	2	3	4	5	
Not At All	◯	◯	◯	◯	◯	Totally

36. **Do you think that IDE build in code style checks will improve Code Readability?**
Mark only one oval.

	1	2	3	4	5	
Not At All	◯	◯	◯	◯	◯	Totally

Static Code Analysis

37. **Do you think that the provided static code analysis tool SonarQube will lead to better code quality?**
Mark only one oval.

	1	2	3	4	5	
Not At All	◯	◯	◯	◯	◯	Totally

38. **Do you think that the provided static code analysis tool SonarQube will improve your coding skills?**
Note: Sonar provides background information for each "Bug" it recognizes
Mark only one oval.

	1	2	3	4	5	
Not At All	◯	◯	◯	◯	◯	Totally

39. **Did you feel motivated by the comparison of Code Quality before/after the merge of your code?**
Mark only one oval.

	1	2	3	4	5	
Not At All	◯	◯	◯	◯	◯	Totally

40. **Would you commit to the goal of getting better (in terms of less bugs, higher test coverage) constantly?**
Mark only one oval.

	1	2	3	4	5	
Not At All	◯	◯	◯	◯	◯	Totally

Code Quality Game

41. **Do you think that applying gamification to software engineering tasks like open bugs, sonar issues,.. would motivate you?**
Mark only one oval.

	1	2	3	4	5	
Not At All	◯	◯	◯	◯	◯	Totally

42. **Would you feel motivated by the gamification tool to close open bugs, issues ?**
Mark only one oval.

	1	2	3	4	5	
Not At All	◯	◯	◯	◯	◯	Totally

43. **Would the gamification tool motivate you run all tests locally before pushing to the server in order to keep your number of failed builds low?**
Mark only one oval.

	1	2	3	4	5	
Not At All	◯	◯	◯	◯	◯	Totally

Acceptance Tests

44. **Do you think that Acceptance criteria scenarios provided in the feature description will help your understanding of the provided task?**

Acceptance criteria scenarios: Files that describe the features in a "when"->"then" form (https://en.wikipedia.org/wiki/Behavior-driven_development#Tooling_examples)

Mark only one oval.

	1	2	3	4	5	
Not At All	⬭	⬭	⬭	⬭	⬭	Totally

45. **Do you think that Acceptance criteria scenarios provided in the feature description will help you feel confident, that you haven't forgotten anything in the implementation?**

Mark only one oval.

	1	2	3	4	5	
Not At All	⬭	⬭	⬭	⬭	⬭	Totally

A.4 Post Experiment Interview Schedule

A.4.1 Introduction

- The conductor introduces himself and thanks the interviewees for their participation
- He informs the participants about the goal of the interview and how the data will be used.
- It is assured that all data will only be used in an anonymized way.
- Before starting the session he asks if everybody is ok with the interview being recorded.

A.4.2 Reviews

A.4.2.1 What do you think of the implemented review process?

- Is it too much overhead?
- Feasible?
- What would you change?

A.4.2.2 What do you think of the provided checklist for code reviews?

- Did you use it?
- Would you keep using it?
- Is it going too much into detail / too little?

A.4.2.3 Do you gain knowledge while doing reviews?

- Do you think they are important for spreading knowledge?

A.4.3 Team building via self assessment

A.4.3.1 What do you think of the team building process via self assessment?

- Is the combination of backend pro + frontend pro a good one?
- Do you think it will improve knowledge transfer?
- Would you prefer simple senior + junior teams?

A.4.4 Versioning

A.4.4.1 Do you understand the proposed versioning process?

- Is the composition of the version number understandable?
- Do you know when the version is raised?
- Do you think it is too complex?

A.4.5 Adapted Build Process

A.4.5.1 Do you understand the implemented build pipelines?

- How does it assist QA?
- Which tests are executed and when?
- Do you think it is fast enough?
- Improvement to the previous setup?

A.4.5.2 Would you feel fit to change the pipeline?

- If there is an error?
- If there is a version update?

A.4.6 Branching Model

A.4.6.1 What do you think about the introduced Branching Model?

- Do you see how quality gates are implemented?
- Do you think it is too complex?
- Do you think that it will assist rapid merging?

A.4.7 Test Environment

A.4.7.1 Would you know how to deploy a specific version of the software locally?

- Easier than before?
- Impact on manual testing?

A.4.7.2 Would you know how to deploy a production like setup locally?

- Easier than before?
- Would you do it consequently
  - Why? Why not?

A.4.8 Static Code Analysis

A.4.8.1 Do you think the goal of "getting better" each sprint is feasible?

- Why, Why not?

A.4.9 Style Guidelines

A.4.9.1 Did you find the IDE tools for static code analysis helpful?

- Help you?
- Learn something?

A.4.10 Code Quality Game

A.4.10.1 Do you think that Gamification in QA will motivate you?

- Why, Why not?
- Would you use it?

A.4.10.2 Did the reward based system motivate you to run all test locally before pushing?

- Why, Why not?

A.4.10.3 Did the bug fixing ranking motivate you to fix code quality issues?

- Why, Why not?

A.4.11 BDDs

A.4.11.1 Did the provided acceptance critieria help you with the given task?

- Feel safer that everything has been done?
- Help your understanding of the task?

A.5 Code Quality Boosting Game

The Soruce Code of the "Code Quality Boosting Game" is hosted on Github [1]. It was inspired by an open source software, which has not been maintained for a long time [2] The software uses data present in most development setups to generate statistics about the "quality boosting rate" of each developer. This way a gamification approach on improving software quality is taken.

A.5.1 Software Description

The software itself is based on the Java Framework SpringBoot and can be run locally or on a server.

An overall high level visual description of the architecture is given in A.1.

The software uses *Adapters* to communicate with servers present in the development infrastructure.

At the moment there are two adapters implemented:

- SonarQube: Fetches information about static code analysis (i.e. bugs, bad smells, etc.)
- Gitlab CI: Fetches information about builds (i.e. successful, failed, tests executed, etc.)

The adapters are then used to fetch data for each user known to the system. At the moment the user information is collected from the SCM too.

The raw data is processed by *Statistics Services*, one for each adapter. These services are fed with configuration about:

- *Actions*: Actions that are defined by a certain behavior witnessed in the data processed. Each action is mapped to a number of points (negative or positive)
- *Achievements*: Achievements usually are a composition of actions and rewarded with badges (which are mapped to a number of points).

The processing of Achievements is outsourced to separate services. After calculating the statistics for a certain user, the information is stored in his internal profile.

The application provides a web server, hosting a small webapp, displaying the results A.2.

[1] github.com
[2] https://github.com/mechero/code-quality-game Accessed September 2017

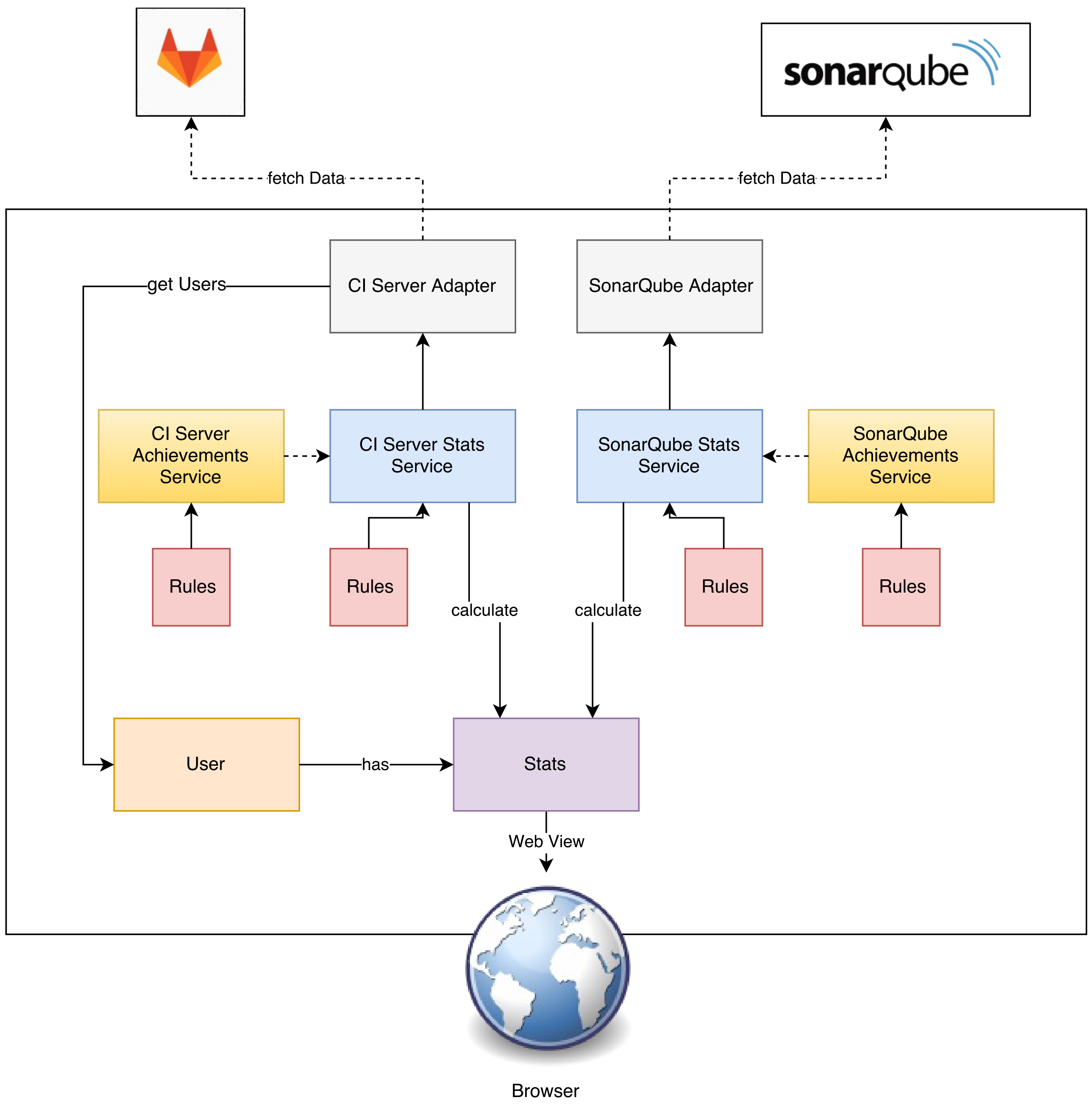
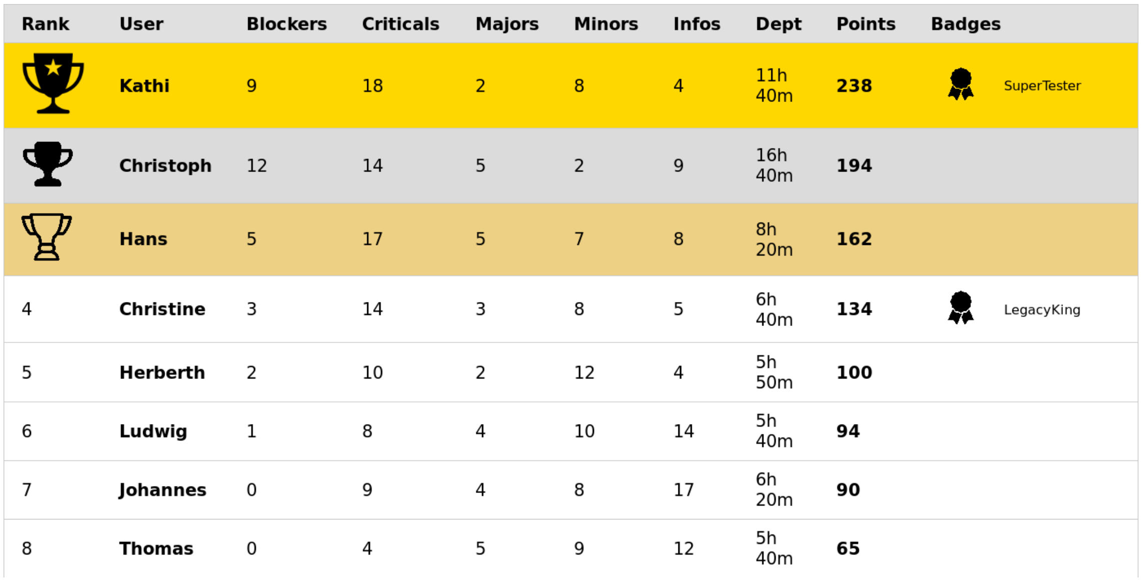


Figure A.1: Software High Level Diagram



Rank	User	Blockers	Criticals	Majors	Minors	Infos	Dept	Points	Badges
🏆	**Kathi**	9	18	2	8	4	11h 40m	**238**	SuperTester
🏆	**Christoph**	12	14	5	2	9	16h 40m	**194**	
🏆	**Hans**	5	17	5	7	8	8h 20m	**162**	
4	**Christine**	3	14	3	8	5	6h 40m	**134**	LegacyKing
5	**Herberth**	2	10	2	12	4	5h 50m	**100**	
6	**Ludwig**	1	8	4	10	14	5h 40m	**94**	
7	**Johannes**	0	9	4	8	17	6h 20m	**90**	
8	**Thomas**	0	4	5	9	12	5h 40m	**65**	

Figure A.2: Screenshot of Leader Board

A.6 Code Review Checklist

A.6.1 Simple Code Review Checklist

In the following a simple Code Review Checklist is proposed, build upon the authors experience as a software developer and past Mineralbay project member:

A.6.1.1 Process based

Some tasks have to be carried out for each commit in order to comply to the process and prevent errors:

- Has the branch been brought up to date with its source?
- Have all tests been run?
- Has the ticket status been set?

A.6.1.2 Code based

The main focus of the review lies on the submitted code:

- Is the code easy to read/understand. Do you understand all parts of the code? E.g:
  - Is there any commented out code?
  - Is the method naming expressive enough?
  - Is there too much code in a class, function,.. ?
- Does the code do what it is supposed to do, does it meet the functional requirements?
- Could the code introduce any unwanted side effects?
- Exception Handling?
- Is the code defensive enough (e.g. does not throw NullPointerExceptions)?
- Is the code tested well enough?
- Is the code documented well enough?
- Does the commit introduce duplicate or redundant code (has something already been implemented somewhere else)?
- Does the code comply to the style guidelines? Auto formated?
- Are there any undocumented/suspicious TODOs left?

A.7 Code Style Standard

A.7.1 JavaScript / AngularJS

The frontend part of the application is build using AngularJS and Javascript. The author chose to adopt John Papa's Angular 1 Style Guide [3] as reference for the AngularJS project. The guide is based on Papa's "development experience" with the framework, aiming at providing "guidance on building Angular applications by showing the conventions" and why they have been chosen.

However the author is in the opinion, that just pointing at a styleguide is not enough, because although the guide's rules might be accepted, understood and considered meaningful by the developers, it will soon be forgotten ans there is no one to remind them.

Therefore the tool ESlint [4] was utilized to point out noncompliances with the given guide. The IDE in use (IntelliJ IDEA) comes with a plugin, that highlights such noncompliances, making them visible to the developers. Furthermore most of the issues can be fixed automatically with the build in auto-formatting tool.

Fortunately the GitHub user Emmanuel DEMEY has gone through the trouble of creating and maintaining an ESlint implementation of John Papa's styleguide [5].

A.7.2 Java

For the Java part of the software, the styleguide build in the IDE was used, since it is based on the original SUN guideline [6] and familiar to most Java developers.

Again issues can be easily repaired, utilizing the build in auto-formatting tool, thus easing compliance.

Checking adherence is also a part of the review process (see **??**).

[3] https://github.com/johnpapa/angular-styleguide/tree/master/a1
[4] https://eslint.org/
[5] https://github.com/Gillespie59/eslint-plugin-angular
[6] http://www.oracle.com/technetwork/java/codeconventions-150003.pdf

A.8 Expert Interview Schedule

A.8.1 Introduction

- The conductor introduces himself and thanks the interviewees for their participation
- He informs the participants about the goal of the interview and how the data will be used.
- It is assured that all data will only be used in an anonymized way.
- Before starting the session he asks if everybody is ok with the interview being recorded.

A.8.2 General questions about the interviewee

A.8.2.1 How long have you been overseeing Student Software Projects??

A.8.3 Defining a Student Software Project

Please answer the qustions accordint to your experience:

A.8.3.1 How big are the teams in typical student Software Project?

A.8.3.2 Which software methodologies are usually followed?

- If so: Which?
  - SCRUM?
  - Kanban?

A.8.3.3 How is communication handled by the developers?

- Online?
- Physical Meetings?

A.8.3.4 Which meetings usually take place?

- Daily SCRUM?
- Weekly?
- Why are they important?

A.8.3.5 Who is the customer in such projects?

- Fictional?
- A company?
- A university institution?

A.8.3.6 Who does the requirements engineering?

- A student?
- In advance prepared requirements?
- Are there acceptance criteria defined in the requirements?
- Are you under the impression, that students understand the users needs?

A.8.3.7 Are code reviews performed by the students?

- When?
- On which parts of the code?

A.8.3.8 Do participants establish and follow a certain versioning scheme?

- How do they look like?

A.8.3.9 Is pair programming done?

- How often?

A.8.3.10 Are developers writing tests?

- How much?
- Is it a requirement?

A.8.3.11 Are there any goals concerning code coverage?

- How high?

A.8.3.12 Do students follow a certain code style guideline?

- Which?
- IDE defaults?

A.8.3.13 Are you under the feeling that students care about code quality?

- Are they satisfied if things "just work"?
- Where do they put their focus?

A.8.4 Student Software Project Specifics

A.8.4.1 Is lack of time usually an issue?

- Because of deadlines?
- Bad time management of students?

A.8.4.2 Where do you see the focus on student projects?

- Learning experience?
- Delivering a working product?

A.8.4.3 How much time do students spent on the project per week?

- Why?

A.8.4.4 Why do students usually take part in such projects?

- What is their driving power?
- What is motivating them?

- Fun?
- Gaining knowledge?
- To get a certificate?

A.8.4.5 How would you rate the skill distribution of students in a group?

- Same educational level (master student, bachelor student)
- Similar knowledge?
- Different knowledge levels?

A.8.5 Introducing Mineralbay

Mineralbay is a project of the INSO in cooperation with the Montan Uni Leoben. The outcome should be a online platform where tunnel drivers can sell the tunnel spoil directly (e.g. to production plants, etc.), instead of hiring a disposal company.

- The software is developed by students of the TU Wien:
  - Bachelor Thesis
  - Master Thesis
  - Practical course
- The process is SCRUM based with sprints varying in length:

- One Week and more
- A weekly jour fix is held:
  - Sprint commitment done
  - Tasks are assigned
  - Teams build
  - Retros
- Communication is mostly online.
- Up to 8 developers
- Ticketing system is used for issue tracking (Redmine)
- Students do requirements engineering

A.8.5.1 Keeping in mind the given description: Would you say that Mineralbay is a typical student project?

- In which ways is it and in which not?