Lorenz Kofler, BSc

Code Refactoring and Code Complexity Analysis in Android Applications

MASTER’S THESIS

to achieve the university degree of

Diplom-Ingenieur

Master's degree programme:

Software Engineering and Management

submitted to

Graz University of Technology

Supervisor

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

Institute of Software Technology

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

Graz, December 2020

AFFIDAVIT

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

Signiert von:	Lorenz Kofler
Datum:	10.12.2020 15:19:05

Dieses mit einer qualifizierten elektronischen Signatur versehene Dokument hat gemäß Art. 25 Abs. 2 der Verordnung (EU) Nr 910/2014 vom 23. Juli 2014 ("eIDAS-VO") die gleiche Rechtswirkung wie ein handschriftlich unterschriebenes Dokument.

Dieses Dokument ist digital signiert!

Prüfinformation:
Informationen zur Prüfung der elektronischen Signatur finden Sie unter:
www.handy-signatur.at

www.a-trust.at

TRUST
einfach sicher

Date, Signature

Abstract

Nowadays, many companies are concerned about the high maintenance costs of their software. Complex software is more difficult to maintain. Therefore, software quality is related to maintenance costs. As a result, by reducing the complexity, the maintenance effort can be reduced and thus the costs. Code refactoring and clean coding are well-known methods to reduce the code complexity. Therefore, the aim of this master's thesis is to discuss such approaches and to find ways to evaluate the complexity of code.

Hence, these methods are first discussed theoretically und then applied in the implementation part. The discussed methods are applied in an Android application of a soccer platform to easier understand complex parts, to facilitate adding new features and to improve the code quality. To show to what extent the complexity was reduced, software complexity metrics are calculated and compared to the subjective perception of the software complexity, which was evaluated by a survey.

The results of the evaluation show that by applying the described methods and techniques to improve the understandability of code the complexity was reduced. This is also confirmed by the combination of the subjective and quantitative analysis.

The results of this master's thesis can help any software engineering team, especially those programming Java in Android Studio, to identify complex code parts, refactor these parts in the proposed ways, and write clean code to enhance the quality and understandability, reduce the complexity, and reduce the maintenance costs in the end.

Kurzfassung

Viele Unternehmen müssen sich heutzutage mit hohen Wartungskosten ihrer Software auseinandersetzen. Da komplexe Software auch schwieriger zu warten ist, hängen die Wartungskosten auch von der Qualität der Software ab. Durch gezieltes Reduzieren der Komplexität kann man den Wartungsaufwand reduzieren und somit auch die Kosten. Bekannte Methoden, um die Komplexität von Software zu verringern, sind Code Refactoring oder Clean Coding Praktiken. Daher ist das Ziel dieser Masterarbeit, solche Methoden zu analysieren und Wege zu finden, um die Codekomplexität evaluieren zu können.

Im Zuge dieser Arbeit werden diese Methoden theoretisch behandelt und schließlich im Implementierungsteil praktisch angewendet. Dafür werden gezielt in der Android-Applikation einer Fußballplattform die besprochenen Techniken angewendet, um schließlich komplexe Teile leichter zu verstehen, neue Features leichter einbauen zu können und die Qualität des Codes aufzuwerten. Um aufzuzeigen, inwiefern sich die Komplexität verringert hat, werden quantitativ erworbene Metriken über Softwarekomplexität mit der subjektiven Wahrnehmung der Softwarekomplexität, welche durch einen Fragebogen erhoben wurde, in Beziehung gesetzt.

Durch die Evaluierung wird aufgezeigt, dass sich die Anwendung der theoretisch behandelten Techniken und Praktiken, um den Code verständlicher zu machen, in einer Verringerung der Komplexität widerspiegelt. Dies wird auch durch die Übereinstimmung der quantitativen Analyse mit jener der subjektiven Evaluierung sichtbar.

Die Ergebnisse dieser Masterarbeit können jedem Softwareentwicklungsteam helfen, vor allem jenen, die Java in Android Studio programmieren, komplexe Codestellen zu identifizieren, diese nach den vorgeschlagenen Methoden und Techniken zu refactoren und schönen Code zu schreiben, um die Codequalität und dessen Verständlichkeit zu erhöhen, die Komplexität zu verringern und die Wartungskosten am Ende zu minimieren.

Contents

List of Equations

List of Figures

List of Listings

List of Tables

1 Introduction

"[…] software systems almost always degrade into a mess." (Feathers, 2004, p. xiii) claims the author in the foreword of his book "Working Effectively with Legacy Code". The problem of such a mess addressed by Michael C. Feathers is that the software development is slowed down by it as the readers of the code are not able to work with it effectively. According to Robert C. Martin (2008) the proportion of reading and writing source code is 10:1. So, to accelerate the development of software, the understandability of code should be improved. However, Nazir, Khan, and Mustafa (2010) describe that software systems become increasingly complex from which suffers the quality including maintainability and understandability. Hence, by actively increasing the software quality, the maintenance effort can be reduced. To achieve this, code refactoring techniques can be used, and clean coding principles can be followed (Fowler, 2018). To measure the complexity of code, there are various complexity metrics which can quantify the degree of complexity for software systems or for individual parts (Kafura & Reddy, 1987). However, as the understandability of code is related to subjective estimation, it is also important taking the developer's perception into account when evaluating the software complexity. Several studies were conducted not only on the quantitative evaluation of code complexity and its reduction by code refactoring (Kataoka, Imai, Andou, & Fukaya, 2002; Leitch & Stroulia, 2003; Ratzinger, Fischer, & Gall, 2005; Moser, Sillitti, Abrahamsson, & Succi, 2006), but also on the empirical evaluation of its improvement (Wang, 2009; Kim, Zimmermann, & Nagappan, 2012). Others have combined empirical data with quantitative measures to conduct a subjective evaluation (Kafura & Reddy, 1987).

The next paragraphs will provide the goals, objectives, and an overview of this master's thesis.

1.1 Goals and Objectives

The purpose of this master's thesis is to evaluate the improvement of code complexity by using well-known code refactoring techniques and following clean coding guidelines. The evaluation combines quantitative measurable software complexity metrics with subjective estimations of software developers collected by a survey. Therefore, we collect and describe theoretical aspects of code refactoring, clean coding, design principles and patterns, and the handling of code that someone else has written. The implementation part of this master's thesis is conducted in a medium-sized software system. This platform provides soccer fans content of their favourite clubs and leagues, including a live ticker of soccer matches, or news. It consists in the front end of an Angular web application, an iOS application, and an Android application. For this thesis, we improve the Android application of the platform and develop in Java[1] using the Integrated Development Environment Android Studio[2]. While implementing the defined features, we used the refactoring techniques, patterns, and guidelines collected in the literature review. Moreover, we describe known problems of the application and removed them, like unused resources, or exchange them, like raster graphics by vector graphics. For evaluating the change in code complexity by following the described patterns, we use a plugin for Android Studio and take further six snapshots of the past two years with an interval of four months to be able to compare the change in complexity over time. For the subjective evaluation, we use a community-based evaluation technique by sending out an online survey consisting of a questionnaire which tries to evaluate the improvement of the understandability of the code in a subjective way by asking some questions about code complexity and decisions made during the course of the implementation of this master's thesis. Afterwards, we combine both evaluations to form a subjective evaluation technique described by Kafura and Reddy (1987).

1.2 Thesis Overview

As described above, theoretical aspects needed for the implementation of the defined features are collected and described in the chapter Literature Review. It includes background knowledge of code refactoring including motivation for refactoring, conducting refactoring along with other developers working on the same repository, testing, and a set of selected refactorings, which are also described for usage in Android Studio. Moreover, design principles, a special object-oriented pattern, and clean coding principles are presented. The second chapter ends with suggestions on how to work with complex code that someone else has written. The third chapter shows the application of the theoretical aspects of the literature review. First, we describe the developed naming convention and coding standard for the Android project, then we show applied code refactoring during the implementation of some features including the development of them. Not only the results but also the requirements and the previous status of each of the selected features are shown. After that sub-section, the removal of unused code and resources in Android Studio is described and the results are shown. Finally, the exchange of raster graphics with vector graphics is presented at the end of the third chapter.

After the description of the implementation, the fourth chapter, Evaluation, explains code complexity analysis in general and its application on the project. After this, the explanation of the survey is presented along with its evaluation. Afterward, both results are combined and

[1] https://www.java.com/ [accessed on 17 November 2020]
[2] https://developer.android.com/studio/index.html [accessed on 17 November 2020]

discussed at the end of chapter four. The thesis ends with chapter five, Conclusion, including a description of lessons learned, the limitations that needed to be faced, and chapter Future Work concludes this master's thesis in the end.

2 Literature Review

The aim of this master's thesis is to find ways to reduce code complexity. On the one hand, suggested ways, which can be found in literature can be applied to enhance code quality which should not only reduce code complexity regarding quantitative measures but also regarding its readability. Therefore, code refactoring methods, techniques, and testing are discussed within the next section. As the understandability of code within the chosen project shall further be simplified and optimized for humans to read, some clean coding issues are discussed within the next section. As the implementation must be done in legacy code, which is code that is written by someone else, some best practices for working with legacy code are described.

2.1 Code Refactoring

Martin Fowler (2018) has published in 1999 a book about code refactoring and explains, that the term refactoring can be used as a verb or a noun, but both definitions highlight that the structure of the software is changed and that there is no functional change in the observable behaviour. He further mentions that this is all done by applying a series of small refactoring steps and the result should always improve the understandability and reduce the effort to modify the code. In cases where it is too difficult to refactor, Fowler (2018) suggests rewriting the code instead, as refactoring should always give a benefit.

Reported by Griswold and Opdyke (2015), refactoring was originally invented at the end of the 1980s by two graduate students in computer science independently, namely by Bill Opdyke in Illinois and by Bill Griswold in Washington. They also mention Kent Beck and Ward Cunningham, who mentions Fowler (2018) also for their work with Smalltalk in the 1980s highlighting the importance of refactoring. Griswold and Opdyke (2015) connect Martin Fowler with Bill Opdyke, as he contributes to writing the first version of the refactoring book in 1999. Kent Beck, on the other side, is said to be the inventor of extreme programming in the same year, publishing a new development approach with refactoring as a fundamental part of it.

Within the next few sections, some refactoring issues are presented and explained.

2.1.1 Introduction to Refactoring

Refactoring is done because of various purposes. According to Fowler (2018), not only the design and understandability are improved, but it also helps to fix and to find bugs faster, it lets the developer program faster and accelerates the overall development time. He explains this phenomenon by the "DesignStaminaHypothesis", which is graphically displayed in Figure 1. On his website (Fowler, 2017) he explains that neglecting design helps to progress faster in software development at first, but at the blue design payoff line this approach will slow down the progress in the end. Otherwise, if developers choose taking the effort in good design, the progress will be lower initially, but above the payoff line it will not slow down. Fowler highlights, though, that this hypothesis is not objectively proofed.

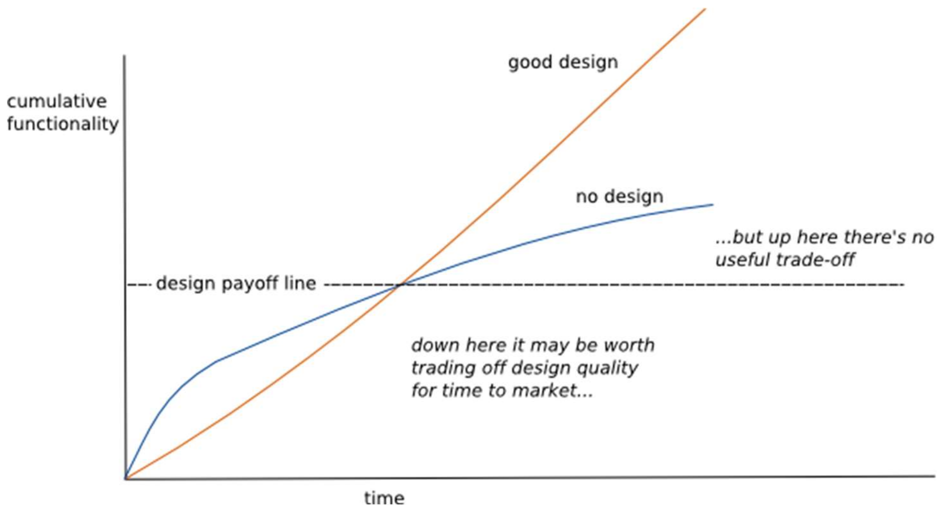


Figure 1: The DesignStaminaHypothesis graphically explained. Above the payoff line good design is worth. (Fowler, DesignStaminaHypothesis, 2007)

However, for that reason code refactoring is important as having a good design in the beginning is often difficult and not possible (Fowler, 2018).

A problem in the daily life of many software developers that Fowler (2018) raises is the frequent lack of understanding of managers. He explains that managers often claim that refactoring does not add valuable features, and therefore, Fowler suggests not telling managers about refactoring as they may not have the technical awareness to see its advantages for faster development in the end.

Martin Fowler (2018) advises that it should be refactored even if the performance gets worse. Fowler explains further, that even if it really gets much worse, then revising it after the rewrite process is much easier. Djoudi and Jalby (2018) or Xie, Wolf, and Lekatsas (2003) referencing Hennessy and Patterson (2007) who calls it the *Principle of Locality*, speak about the 90-10 or 90/10 rule meaning that a program executes only 10% of code in 90% of total time. This means that the focus should lie on the readability and understandability and not on performance when refactoring. Otherwise, a lot of time is wasted as performance optimizations should be focused mainly on that 10% of code, which can be found by using a profiler monitoring the execution for time and space consumption and not on the whole codebase (Fowler, 2018). Furthermore, any bad influence on the compilation time can be ignored, because according to Fowler (2018) a few more compile cycles are ignorable compared to the extra hours a programmer has to invest to understand poorly written code.

Design patterns are important ways to apply documented best practices to different known software problems (Kuchana, 2004). In refactoring, though, Fowler (2018) advises not to think, design and build patterns for possible future needs, but only for the current needs in an optimized way. In case of changing requirements, refactoring should be used to adapt current patterns.

2.1.2 Reasons for Refactoring

Besides the technical reasons like reducing complexity or changing the software design by identifying and removing antipatterns and bad code smells (Bavota, De Lucia, Marcus, & Oliveto, 2014; Du Bois, Demeyer, & Verelst, 2004), Bavota, De Lucia, Di Penta, Oliveto, and Palomba (2015), and Wang (2009) explain subjective reasons for doing refactoring. In the former paper they say that the main pushing factor is low readability, and by performing refactoring maintainability and readability are improved (Bavota, De Lucia, Di Penta, Oliveto, & Palomba, 2015). The latter paper by Wang (2009) presents an empirical model that shows the found refactoring motivations which they extracted from interviews with ten software developers. This model can be seen in Figure 2.

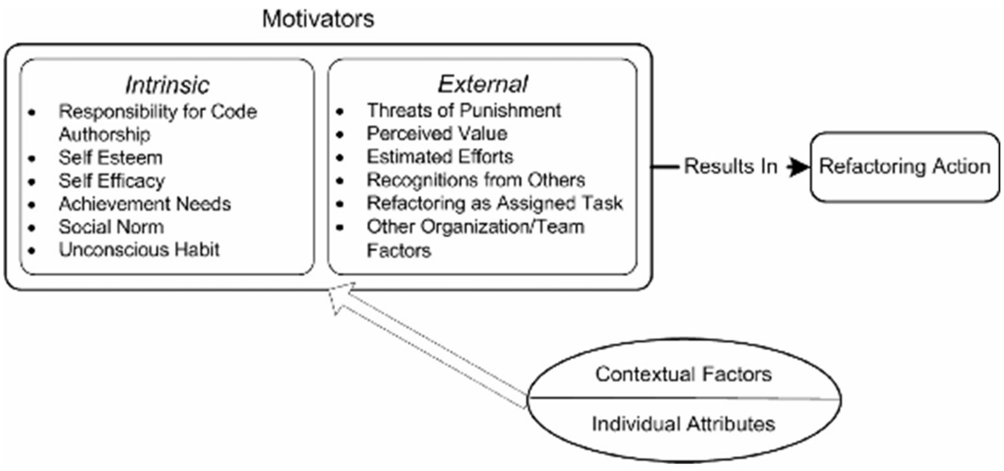


Figure 2: Extracted intrinsic and external refactoring motivators by Wang (2009), retrieved from IEEE Xplore (Wang, What motivate software engineers to refactor source code? evidences from professional developers - IEEE Conference Publication, 2009).

Wang (2009) differs between intrinsic and external motivators which push the developers to refactor. He explains that intrinsic motivators do not offer the affected person an obvious external reward, but the extrinsic ones do. Wang extracted six intrinsic motivators including that the developers want to have a high quality in their own code, they are motivated by their own self-efficacy and self-esteem as they believe in their potential and they want to improve their skills through refactoring. The social norm suggests that developers tend to follow development practices as they do not want to be isolated. With the last item, Wang wants to highlight the habit of developers doing practicing refactoring in the daily work.

Wang 2009 counts external punishment by managers to improve quality, the perception that refactoring leads to an obvious additional value like reusability, the feared additional effort without refactoring, and good reputation for their technical ability as the six inferred external motivators. According to Wang, it is often the case that developers get refactoring as their own task to do. Moreover, he highlights other factors, that vary from organization to organization like competing developers for the best code to achieve some award.

Besides, other factors influencing the motivation for refactoring are highlighted like the usage and assessments of refactoring tools, the developer's personality, or other environmental factors (Wang, 2009).

In contrast to this, Fowler (2018) describes the reasons for refactoring as an action to a specific situation. He explains that in cases where new features need to be added, preparatory refactoring can be used to facilitate its implementation, and that it is the best time for refactoring. Fowler (2018) further mentions another reason can be that this makes it easier to understand the code. He refers to Ward Cunningham who explains this action of refactoring by moving the understanding of the code into the written program itself. This situation of not understanding the code is responded to with so-called comprehension refactoring. As result, as Fowler (2018) states, the code is more understandable not only to the directly affected developer but also to the following programmers.

Besides the preparatory refactoring and comprehension refactoring, Fowler (2018) mentions litter-pickup refactoring meaning changing the structure of code motivated by the intrinsic desire of changing malformed code even if it must not be done immediately. These three types are summarized by Fowler as opportunistic refactorings, which help not only for the current situation but also for future work on the affected code. Fowler says that refactoring should be part of the daily work of a developer, planned sessions should be conducted rarely. He highlights that refactorings should only be conducted in code that is needed somehow in the future or in the current situation. Finally, Fowler speaks about refactoring in code reviews. Such code reviews, especially peer code reviews, can aid in gaining substantial knowledge when conducted in daily work (Spohrer, Kude, Schmidt, & Heinzl, 2013). Spohrer, Kude, Schmidt, and

Heinzl (2013) highlight here the combination of peer code reviews and peer programming as an efficient way to transfer knowledge between team members, where refactoring takes also place in a more regular habit, according to Fowler (2018).

To summarize it, code refactoring should always have a good reason to be conducted, like speeding up development by making it easier to add features or fixing bugs, but not only due to clean up code or other moral reasons (Fowler, 2018).

2.1.3 Code Refactoring Along the Development Process

Fowler (2018) explains that refactoring can fail and therefore it is always important to go by small steps. He suggests a version control tool. For example, GitLab[3], which allows to commit small changes. He justifies this by saying that the advantage of committing in small steps is that it is possible to go back to the previous working step in case something went wrong. Fowler writes that as these commits should be very small, it is important to squash these small commits to a significantly bigger commit, which can be then pushed to the remote repository. Moreover, Fowler (2018) sees advantages in committing workable code in small steps that the refactoring can always be stopped without breaking the code base even if the refactoring may not be finished. He concludes by saying that correct refactoring does not break code and does not provoke debugging if it is done correctly.

For documentation and maintenance reasons, the changes made should be described within commit messages. The local commit messages for the small commits can be neglected, but the squashed commit message should include a meaningful and informative description of the changes. (Linares-Vásquez, Cortés-Coy, Aponte, & Poshyvanyk, 2015)

To not mess the development process, it is important to create a new branch which means diverging from the most recent status quo of the product. As a result, it is possible to do refactoring without disturbing or changing the mainline. (Chacon & Straub, 2014)

However, as Meyer (2014) calls for continuous integration such feature branches should be kept short and be merged into the mainline each day. The problem is that refactoring can easily affect many parts of the codebase, and then merging conflicts can occur (Fowler, 2018). When doing extreme programming, which is presented by Kent Beck with support by Cynthia Andres (2004), they suggest using such temporary branches only for few hours as a single code base

[3] https://about.gitlab.com/ [accessed on 17 November 2020]

is desired. As result, they claim that the code base stays healthy and promises faster development.

Chacon and Straub (2014) show two ways to integrate changes into the mainline, namely merging and rebasing. The following three examples are based on the idea presented by Chacon and Straub (2014). In Figure 3 a simple, but typical commit history is shown. “Commit F” has pushed to a separate feature branch, and on the main line there was also pushed a commit called “Commit M”.

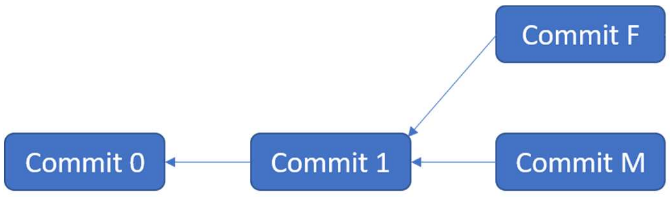


Figure 3: In a feature branch which branches from the main line, a commit "Commit F" was made. On the main line a commit "Commit M" was made. By rebasing these commits can be combined to one.

By merging these two commits, Chacon and Straub (2014) explain that the original two commits, “F” and “M”, stay, and another merging commit “Commit E” is added as can be seen in Figure 4.

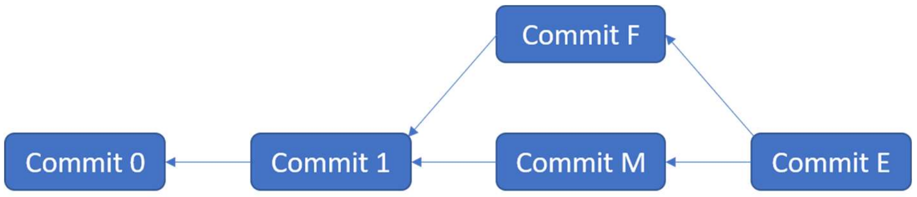


Figure 4: Merging the feature branch into the main line results in a new commit "Commit E". The problem here is the diverged work history.

In contrast to this, Chacon and Straub (2014) show that in case of rebasing the feature branch on top of the commit in the mainline, an additional commit “Commit E*” is also added, but it is in line with the mainline meaning that it builds up only on the commit pushed on the mainline. Chacon and Straub highlight that the snapshots of commits “Commit E” and “Commit E*” are equal, though. The result of rebasing is shown in Figure 5.

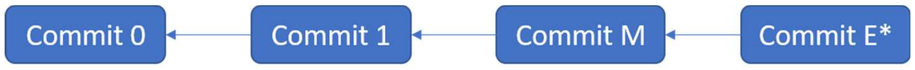


Figure 5: The rebasing process results in a much cleaner commit history but the snapshot of the merging (“Commit E”) and rebasing results (“Commit E”) are the same.*

To conclude, Chacon and Straub (2014) explain that by rebasing a feature branch frequently onto the mainline, the new commits can be merged cleanly in. As a result, the merge into the master branch can be applied without conflicts.

2.1.4 Testing

Since clean refactoring should not change the observable behaviour, as explained above (Fowler, 2018), testing the code is crucial during refactoring. However, as Feathers (2004) describes, in case there are no tests available, it is also important to preserve the behaviour and to introduce tests. In addition to this, above all for code that is unknown, the so-called legacy code which is described further in 2.3 below, Feathers presents an approach to understanding the code without documentation and without knowing requirements. This approach is declared to be even better as legacy systems mostly tend to behave differently as defined in any requirements or documentation. He calls such new tests in legacy code characterization tests as they should characterize the system and its behaviour if done correctly and excessively. In case it is not known what value is returned by a specific function, it is possible to make tests fail first, and then use the value for the value-check. Feathers also warns that it is important to let bugs in the system alive during refactoring and fix them later, as the goal is to understand the system's behaviour and not changing the code. Changing the code without tests could lead to new errors, but by this approach refactoring and bug fixing can be done without harm (Fowler 2018).

Prominent examples of tests are unit tests or integration tests. Brar and Kaur (2015) describe the differences between these two approaches. While unit testing focuses on the correct functionality of small units, integration tests connect such components to test the interaction between the individual units. They further say that regarding functionality testing, the focus should lie on unit testing, which is a so-called white box testing approach. Farcic and Garcia (2015) explain that white box testing uses knowledge about the software to define tests. They say that it requires a good internal understanding of the system.

Feathers (2004) also affirms that unit tests should be preferred in contrast to large tests, as large tests do not localize bugs in that detail, they run longer, and small changes can affect many parts of the software, so it can lead to many tests fail. However, if choosing unit tests, problems can be localized more easily, and the test runs do not take that long. He explains that good unit tests should run fast, meaning that tests which take more than 100 milliseconds are slow unit tests. Additionally, they should not talk to the database, they should not use the network or file system, and they are not dependent on the environment like a configuration file to be executed.

One famous software development process, where testing is a major part, is test-driven development (TDD). It is used in short development cycles and is based on the idea to write first tests and then implement the code (Farcic & Garcia, 2015). The connection to refactoring is explained aptly by Beck and Andres (2004) who explain that test-first programming solves a rhythm problem as developers always know what to do next as the cycle is clearly defined. They say that first tests are written and fail, then the implementation should make the tests work, but in case of failing they refactor until the tests succeed. This rhythm is repeated for the whole development process. The approach of testing first before coding is also used in extreme programming (Beck & Andres, 2004). Here, regression testing is used to ensure safe code refactoring (Cheon, 2014). Basu (2015) states that software changes, as they arise during refactoring, can make failing tests in completely other parts of the code, highlighting the importance to check such side effects. Therefore, regression testing is needed to recheck all defined tests for their success. Basu explains that this approach helps to find changes in the behaviour of software but needs a great number of tests and is time-consuming, though.

As for Java, there is a famous open-source framework available invented by Kent Beck and Erich Gamma called JUnit, which allows to write and run tests easily and is built upon the xUnit architecture (Clark, 2006). According to Clark (2006) the highlights of JUnit includes assertions to check expected values, test fixtures to let tests be repeatable, and test runners which are responsible for the test runs. The written unit tests shall aid in finding breaks or defects. Cheon says that this framework "encourages a close integration of testing with development by allowing a test suite to be built incrementally" (Cheon, 2014). At the time of creation of this master's thesis, JUnit 5 with version 5.7.0 is the most recent release[4].

To summarize the previous two sections, for efficient and clean refactoring not only testing or even self-testing code is important, but also the above-mentioned continuous integration (Fowler, 2018). Fowler says that testing helps preventing the introduction of new bugs if done correctly and continuous integration prevents from having more work because of merge conflicts or other problems related to too different version histories. Fowler further explains that

[4] https://github.com/junit-team/junit5/ [accessed on 3 December 2020]

these two approaches with refactoring are the fundamentals of extreme programming. Feathers (2004) also affirms that high code coverage is needed to catch any errors during changing code.

As the focus of this master's thesis is using automated refactorings of Android Studio, it is not that important having that many tests or any tests as such refactorings can be trusted (Fowler, 2018). For refactorings that are not automated, this is not the case, though. In the next section, it is explained how automated code refactoring works in Android Studio.

2.1.5 Automated Code Refactoring

Since Android Studio is founded by JetBrains who are claimed for the automated refactoring tools in their IntelliJ IDEA[5] for developing in Java, automated refactorings in such IDEs are completely safe, and therefore, testing is not needed for preserving the observable behaviour in general (Fowler, 2018).

JetBrains (2020) explain, that their IDEs, like the used Android Studio, parse files in two steps. The structural representation of the program is created as an AST, or abstract syntax tree. The semantical enhancement is added to the AST by the program structure interface (PSI). JetBrains explains that the nodes of the abstract syntax representation of the program map directly to the corresponding text passages. As a result, any changes in the AST are reflected in the corresponding document like insertions, deletions, or reorderings. With this cooperation of the AST and the PSI, common refactorings, like renaming, usage findings, or TODO findings, to name some features, are possible.

2.1.6 Selected Set of Refactorings

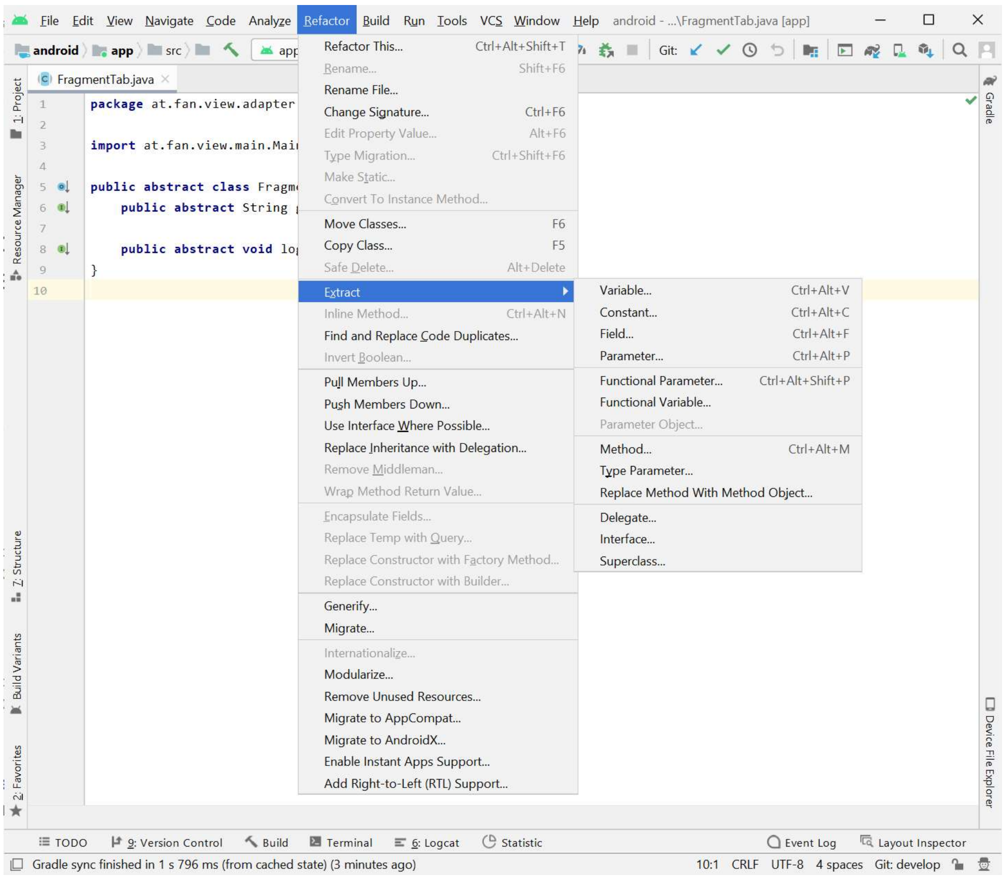
Fowler (2018) explains several refactorings in his book. Here, a set of them is selected and presented shortly with the corresponding automated usage in Android Studio. In the following, the term function also refers to methods for simplicity.

Fowler (2018) firstly presents *Extract Function*, *Inline Function*, *Extract Variable*, and *Inline Variable* which extract or inline code from or into functions or variables. Reasons for extracting functions can be for example long functions, which are functions that consist of more than six lines according to Fowler. Inlining functions or variables are useful in cases where the content of them is as clear as the chosen name itself. Extracting a variable provides the possibility of clarifying complex statements. Moreover, Fowler highlights the advantage of debugging as variables can be analysed more easily than their code representations. To read the extracted

[5] https://www.jetbrains.com/idea/ [accessed on 3 December 2020]

functions in a natural flow, Martin (2008) suggests placing the extracted function directly beneath the function. If this is not possible, the caller should be in any case above the extracted function.

In Figure 6, the different possibilities for refactorings in Android Studio are shown. When it comes to selecting a variable, “Inline Method…” is changed to an activated “Inline Field…” menu item. The extractions of methods, functions, variables, or constants are visible in the screenshot. Android Studio offers keyboard shortcuts for many of the refactorings as the default setting.


File Edit View Navigate Code Analyze Refactor Build Run Tools VCS Window Help android - ...\FragmentTab.java [app]
android app src app
FragmentTab.java
Refactor This... Ctrl+Alt+Shift+T
Rename... Shift+F6
Rename File...
Change Signature... Ctrl+F6
Edit Property Value... Alt+F6
Type Migration... Ctrl+Shift+F6
Make Static...
Convert To Instance Method...
Move Classes... F6
Copy Class... F5
Safe Delete... Alt+Delete
Extract
Inline Method... Ctrl+Alt+N
Find and Replace Code Duplicates...
Invert Boolean...
Pull Members Up...
Push Members Down...
Use Interface Where Possible...
Replace Inheritance with Delegation...
Remove Middleman...
Wrap Method Return Value...
Encapsulate Fields...
Replace Temp with Query...
Replace Constructor with Factory Method...
Replace Constructor with Builder...
Generify...
Migrate...
Internationalize...
Modularize...
Remove Unused Resources...
Migrate to AppCompat...
Migrate to AndroidX...
Enable Instant Apps Support...
Add Right-to-Left (RTL) Support...
Variable... Ctrl+Alt+V
Constant... Ctrl+Alt+C
Field... Ctrl+Alt+F
Parameter... Ctrl+Alt+P
Functional Parameter... Ctrl+Alt+Shift+P
Functional Variable...
Parameter Object...
Method... Ctrl+Alt+M
Type Parameter...
Replace Method With Method Object...
Delegate...
Interface...
Superclass...
1: Project
Resource Manager
7: Structure
Build Variants
2: Favorites
Gradle
Device File Explorer
TODO 9: Version Control Build Terminal 6: Logcat Statistic Event Log Layout Inspector
10:1 CRLF UTF-8 4 spaces Git: develop

```
package at.fan.view.adapter

import at.fan.view.main.Mai

public abstract class Fragm
    public abstract String 

    public abstract void lo
}
```

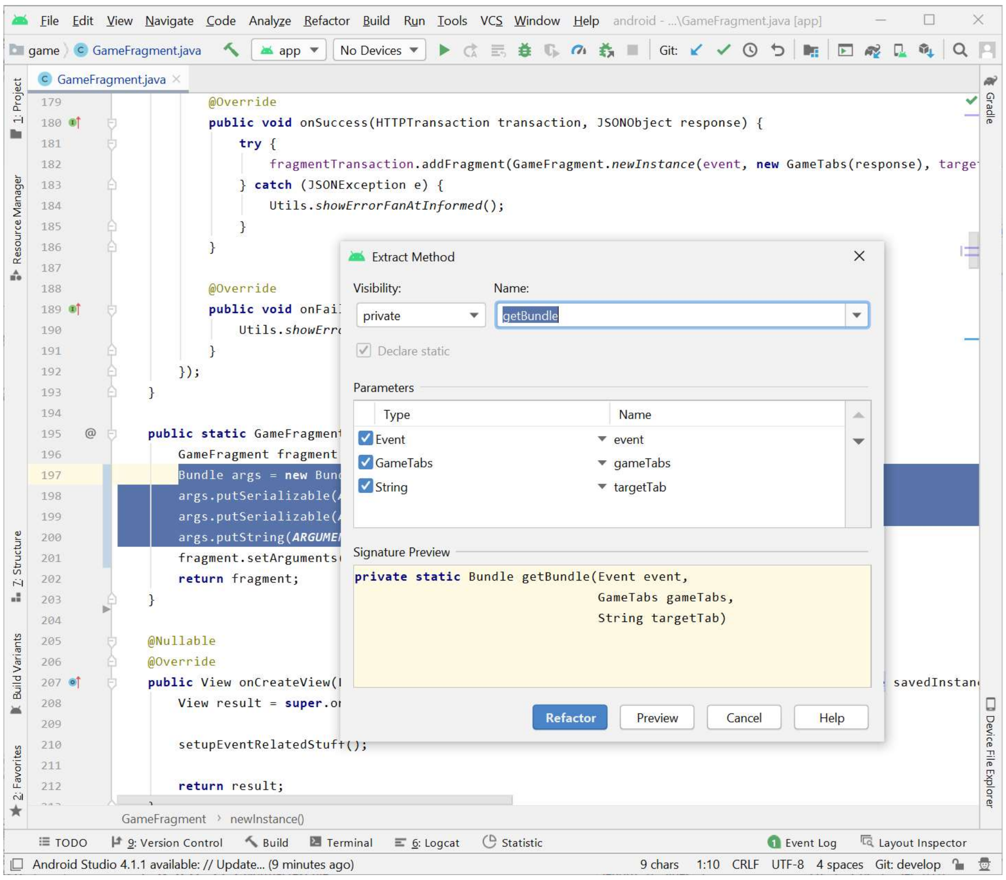
Gradle sync finished in 1 s 796 ms (from cached state) (3 minutes ago)

Figure 6: Android Studio dedicates refactoring an own menu item.

In Figure 7 an example for extracting a method can be seen. For multiple refactorings, Android Studio proposes suggestions for names of the affected entities. Moreover, the signature can be changed directly in the opened window. When it comes to checking the changes, a preview can be shown.

After these composing refactoring methods (Refactoring.Guru, 2020), Fowler (2018) summarizes the act of renaming a function, adding, or removing parameters, which is available by the menu item “Change Signature…” in Android Studio as can be seen above, by the term *Change*

Function Declaration. Renaming any named entities in Android Studio is available by the "Rename" commands. Fowler suggests a way to find good names by commenting on the purpose or functionality of the function and then using this comment for creating a new name.



```
            @Override
            public void onSuccess(HTTPTransaction transaction, JSONObject response) {
                try {
                    fragmentTransaction.addFragment(GameFragment.newInstance(event, new GameTabs(response), targe
                } catch (JSONException e) {
                    Utils.showErrorFanAtInformed();
                }
            }

            @Override
            public void onFai
                Utils.showErr
            }
        });
    }

    public static GameFragmen
        GameFragment fragment
        Bundle args = new Bun
        args.putSerializable(
        args.putSerializable(
        args.putString(ARGUME
        fragment.setArguments
        return fragment;
    }

    @Nullable
    @Override
    public View onCreateView(                                              savedInstan
        View result = super.o

        setupEventRelatedStuff();

        return result;
```

Extract Method

Visibility: private

Name: getBundle

Declare static

Parameters

Type	Name
✓ Event	event
✓ GameTabs	gameTabs
✓ String	targetTab

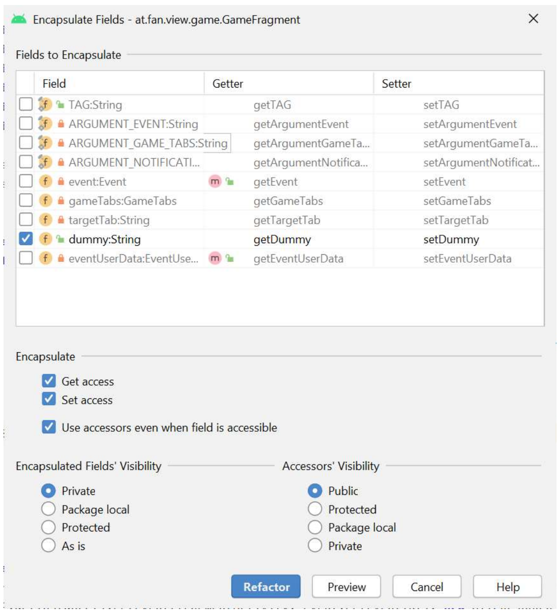
Signature Preview

```
private static Bundle getBundle(Event event,
                                GameTabs gameTabs,
                                String targetTab)
```

Figure 7: Android Studio provides needed parameters with suggested names and proposes a name for the extracted function. By "Preview" the intended change can be previewed.

The next two refactoring methods Fowler (2018) describes are related to a variable or field, namely *Encapsulate Variable*, making variables only accessed by functions, and *Rename Variable*. According to Fowler, encapsulating a variable is more important for mutable data, as immutable data is not needed to be validated or be prevented to be copied, but for mutable data this holds. In Figure 8 an example of encapsulating fields is shown. The variable dummy is public and hence, encapsulating here yields a private field with getter and setter which are automatically preselected.

Fowler (2018) justifies using the presented *Introduce Parameter Object*, which is available under the same name for selected parameters in Android Studio, by saying that grouping parameters improves consistency, reduces the number of arguments. Additionally, it expresses the relationship between the selected parameters. Android Studio automatically suggests creating a new class for the selected data items, which can be seen in Figure 9. The result is a complete class with a constructor and getter for the individual fields.


Encapsulate Fields - at.fan.view.game.GameFragment

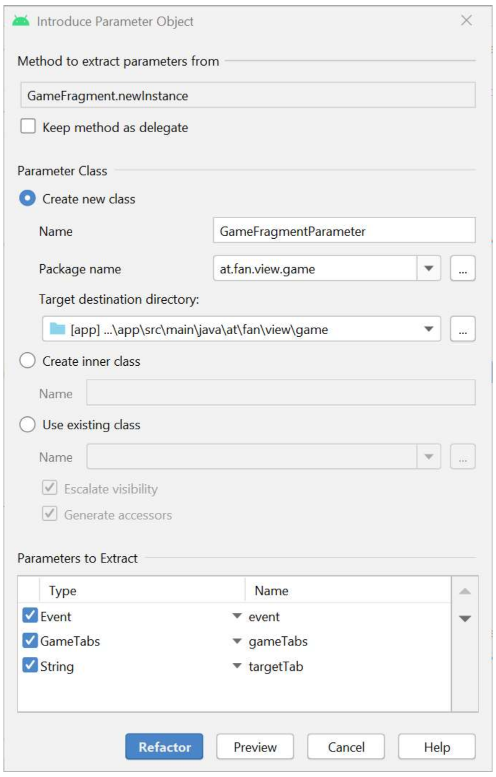
Fields to Encapsulate

	Field	Getter	Setter
☐	TAG:String	getTAG	setTAG
☐	ARGUMENT_EVENT:String	getArgumentEvent	setArgumentEvent
☐	ARGUMENT_GAME_TABS:String	getArgumentGameTa...	setArgumentGameTa...
☐	ARGUMENT_NOTIFICATI...	getArgumentNotifica...	setArgumentNotificat...
☐	event:Event	getEvent	setEvent
☐	gameTabs:GameTabs	getGameTabs	setGameTabs
☐	targetTab:String	getTargetTab	setTargetTab
✓	dummy:String	getDummy	setDummy
☐	eventUserData:EventUse...	getEventUserData	setEventUserData

Encapsulate
- [✓] Get access
- [✓] Set access
- [✓] Use accessors even when field is accessible

Encapsulated Fields' Visibility: [✓ Private] Package local Protected As is

Accessors' Visibility: [✓ Public] Protected Package local Private


Introduce Parameter Object

Method to extract parameters from: GameFragment.newInstance

☐ Keep method as delegate

Parameter Class

[✓ Create new class]
Name: GameFragmentParameter
Package name: at.fan.view.game
Target destination directory: [app] ...\app\src\main\java\at\fan\view\game

Create inner class
Name:

Use existing class
Name:
✓ Escalate visibility
✓ Generate accessors

Parameters to Extract

	Type	Name
✓	Event	event
✓	GameTabs	gameTabs
✓	String	targetTab

Figure 8: Encapsulate fields provides the functionality to encapsulate mutable fields and adds getter and setter for them instead.

Figure 9: Android Studio suggests creating a new class for the selected parameters.

Fowler (2018) explains much more refactoring techniques not discussed in this literature review like moving entities or others. One of them should conclude the selected set of refactoring methods, namely *Remove Dead Code*. This method is needed in case dead code occurs and is not needed anymore (Fowler, 2018). Fowler suggests that it is better to remove the unused code because if it is needed in future version control can help. Android Studio aids in discovering such code smells by highlighting unused or unreachable code. This is further discussed below in 3.3.

2.2 Clean Code

According to Martin (2008), author of the book "Clean Code", clean code is code that other people can easily change, so the readability should be given, tests should be included, and it should be kept simple without many dependencies, following the Single Responsibility Principle (SRP). According to Martin (2017) this principle is part of the SOLID principles which serve as design guidelines to divide data into classes, like functions or data structures. He explains that the overall goals are flexibility to changes and understandability. Two of them are taken further into account. The first of them is the mentioned SRP which says that there should be only one reason for each component to be changed, meaning among others that there is no duplication and the Open-Closed Principle OCP, which means that changing a system should only be possible by adding new code to it instead of adapting it.

As Kerievsky (2005) describes in the foreword of his book “Refactoring to Patterns”, patterns always transform programs. This can be visualized by comparing a code before, and after the use of the pattern. Because of this, patterns are strongly connected to refactoring. The most relevant pattern for this thesis is explained in the next section.

2.2.1 Abstract Class Factory

As Gamma, Helm, Johnson, and Vlissides (1994) discuss, there are several advantages of using an abstract factory. On the one hand, it completely isolates concrete implementations. That means, that the concrete classes are implemented independently from the affected developers, so they can only manipulate the instances by the methods defined by the abstract interfaces. Moreover, Gamma et al. say that the class names are not visible to them either and the created products are only dependent on the passed arguments which are well-known to the developers. On the other hand, abstract factories enable easy exchanging of the concrete classes as there is only one point in code where the concrete implementation of the factory appears, namely only on the instantiation. Additionally, consistency among concrete implementations is implicitly forced as it uses only one product family whose concrete products implement the same behaviour defined by the abstract product. Nevertheless, there is also a disadvantage according to Gamma et al. In case of another kind of product, all subclasses of the abstract factory and the abstract factory itself must be extended. However, this is not a problem in case there are no sub classes for the abstract factory.

2.2.2 Naming Conventions

As Kent Beck (1997) suggests, instance variables should be named according to the role they play within the program. Collections should be in plural form; all others should be singular. Consistency is key. The type is also important to tell what the variable can do and how it is used. But because of readability, variables should be named short and simple. So, in some cases where it is not clear what type a variable has, it is also possible to include its type. Robert C. Martin (2008) however, suggests to not do type encodings as objects in Java are strongly typed and therefore it is not necessary to indicate of which type a variable is. Moreover, clean code means smaller functions and classes and therefore the context will clearly tell of which type a variable or a class member is. According to Martin, class members should not use prefixes like "m_", "m" or "_". Modern editors can highlight members and even show their types in the highlighted information. In Android Studio, for example, holding "Strg" and hovering a variable typically shows its first usage, which means its declaration or even initialization. Much more important is the choice of a meaningful name (Ritchie, 2007). It should reveal its intent, it should not encode wrong information in names, like any reserved words, it should be descriptive, like using "source" and "destination" instead of using "a" and "b", it should be pronounceable and searchable, so no abbreviations and one character names should be used, and, as mentioned, encodings to show the type should not be used (Martin, 2008). Martin suggests further that class names should consist of nouns and should not encode general umbrella terms like "Data" or "Manager". For methods, however, Martin suggests using verbs singularly or in combination with nouns. Getter and Setter should be consistent with the corresponding value and should be prefixed with "get" and "set", and for Booleans, it should be used "is" according to JavaBeans'[6] accessors of variables (Sun Microsystems Inc., 1997).

Fowler (2018) states that renaming is not complicated in modern IDEs as automated refactoring tools are often integrated to apply such refactorings using one click. Therefore, he further explains that any renaming is worth during development to clearly communicate what the code is doing. Fowler (2018) highlights the importance of choosing the right name, though. He states that it is always important that the reader knows exactly what a variable is for, or what happens within an extracted function for example. Even if it takes sometimes multiple tries for choosing the correct name, Fowler (2018) says that it is always important for extracted functions because it only makes sense to use this refactoring technique if the chosen function name clearly explains what the extracted function does in its body without the need to look into it.

In the next section, a selection of unclean code examples is presented.

2.2.3 Code Smells

Fowler (2018) and Kent Beck present a list of 24 bad smells in code in his book, like *Mutable Data*, *Temporary Field*, or *Feature Envy*, and they refer to parts in code that are not bugs or errors but "smell badly" because of the uncleanness. However, within this explanation, the focus lies on selected aspects and is structured differently.

[6] https://www.oracle.com/java/technologies/ [accessed on 15 November 2020]

To start with, one of the worst code smells is duplicated code. According to Fowler (2018) code should say exactly what it does but only once, so any redundant code should be removed to improve the design. Hunt and Thomas (2000) say that redundant code makes it impossible to develop software reliably. In contrast to this, they suggest following the principle DRY instead, which they derive from "Don't Repeat Yourself" (Hunt & Thomas, 2000). They say that only then it is possible to increase understandability and maintainability because knowledge must be implemented in only one single place. Reasons for duplicate code can be various. Hunt and Thomas describe four of them, *Imposed*, *Inadvertent*, *Impatient*, and *Interdeveloper Duplication*. When duplication is imposed, the developers feel to be forced to use redundant code. In case the developers do not actively understand that they duplicate code, Hunt and Thomas call that inadvertent. Software engineers often are lazy and therefore they copy code leading to *Impatient Duplication*. They use the last type for explaining the cases where multiple developers duplicate code. The problem Fowler (2018) describes is that when there is something to change within the copied code, the change needs to be done in each of the copied code sections. Hence, duplicate code is also the source of potential inconsistency (Hunt & Thomas, 2000).

Next, functions are discussed. According to Martin (2008), they should be as small as possible. He says that each block within a function should only include one line of code, for example, a function call and that the indent level should not be greater than two for better readability and understandability. As a result, he says that the name of the function can then be easily chosen. Moreover, Martin suggests that functions should only do one thing, exactly that thing which is described by the function's name. Fowler (2018) states that the maximum number of lines in a function is six. He advises further not commenting any code but using a function instead, even for a single line of code. With the help of *Extract Method* and by using a descriptive name, documentary value is added implicitly, and the methods are shortened. To prevent from having the code smell *Long Parameter List*, Fowler suggests for example the above-described refactoring technique *Introduce Parameter Object* (Fowler, 2018). Martin (2008) suggests only three types of functions when it comes to the number of arguments, namely niladic, monadic, and dyadic, meaning holding zero, one, or two arguments. Triadic and polyadic functions should never be used.

As described above, functions should be as small as possible. According to Robert C. Martin (2008), the indent should be two at maximum and blocks should never hold more than one single line. Therefore, local variables should not be introduced that often. Nevertheless, if a variable is used whose value is returned, Fowler (2018) suggests calling this variable "result" for indicating its purpose. Furthermore, Fowler describes that other locally scoped variables should be used as little as possible as these temporary variables complicate further extractions. He suggests using *Inline Variable* as described above to solve such inconveniences.

Another selected code smell is the use of repeated switch statements. Martin (2008) suggests solving this problem of having more than one switch statements with the same cases by using the pattern abstract class factory explained above. Martin states that by instead introducing these polymorphic objects, which implement the needed functionality in their inherited classes, the visibility of the implementation is completely hidden from the developers. This opinion is also expressed by Gamma et al. (1994).

Regarding comments, Fowler (2018) says that there is no need for documenting any functionality. In such cases, methods should be better extracted and named as the intended comment as already described above. Martin (2008) confirms this but explains some exceptions for using comments, this includes, for example, legal comments to explain the copyright holder for example, cases where developers want to warn for some consequences, or TODO comments. Out-commented code or any separation lines for structuring parts of the code should not be used.

2.2.4 Error Handling

According to the definition by Oracle (2020), exceptions disrupt the standard execution of a program. They explain further that this event creates an exception object containing important information and passing or throwing it to the runtime system. Oracle explains that the runtime system then searches for a method in the call stack handling the exception. In case there is no such exception handler defined, the program exits unsuccessfully. In Java, try/catch blocks are used to handle such situations.

To be able to write clean, robust code, it is important to separate the handling of errors from the main logic (Martin, 2008). The solution is explained in the next few paragraphs.

Martin (2008) suggests that the exception handler should only handle exceptions as functions should always do just one thing. He says further, when having a try/catch block, each of the blocks should call a function implementing the corresponding functionality. Additionally, he warns against using other techniques than exceptions for handling errors, like error codes, as this leads to maintenance overhead.

Moreover, Martin (2008) presents another problem with the above-mentioned approach. Not using exceptions forces the user to check for the error immediately after the call. On contrary, when using exceptions, the handling can be ignored and does not affect the cleanness of the corresponding code. Furthermore, Martin explains that exceptions should never be checked directly except for working in critical libraries. This means that the function should not throw that error explicitly as otherwise, the exception must be checked immediately. This implies that third-party libraries, which throw exceptions should be explicitly wrapped.

Special handling requires the usage of null. Even if null is common in Java, Martin (2008) explicitly warns to not return and pass it due to possible null pointer exceptions. Using exceptions in case of an error or unsuccess is the cleaner choice.

In the next sub-chapter, legacy code is described and issues about it are discussed.

2.3 Legacy Code

Michael C. Feathers published in 2004 a book in which he presents some suggestions for working effectively with legacy code. In the foreword, he tells that "software systems almost always degrade into a mess" (Feathers, 2004, p. xiii). He further states that the change of requirements is not the single explanation for this degradation as requirements do change and software designs should be built for change. While Feathers calls rotten programs or code without tests legacy code, Eli Lopian (2018) speaks about the difficulty of defining it for non-programmers. Moreover, she highlights the fact that each codebase is legacy code, as it has a writing history. Using bad code as a synonym is also not meaningful enough but summarizes all ideas and implications in thinking "of legacy code as code that developers are afraid to change" (Lopian, 2018).

The next section explains how to work with such type of code.

2.3.1 Working with Legacy Code

The importance is being able to work with such legacy code that someone else has written, or saying it more strictly by referencing to Feathers' foreword in the book, "reversing the entropy" (Feathers, 2004, p. xiii). Yu, Wang, Mylopoulos, Liaskos, Lapouchnian, and do Prado Leite (2005) suggested to include refactoring at the beginning of a reverse engineering process of legacy code. They directly use the *Extract Method* as explained in 2.1.6 above as a refactoring technique while using any comments made in the given codebase. The result of this step where semi-automatically refactorings of IDEs are used is a refactored source code which can be better analysed for structural problems. They say that all comments are removed by it.

However, this approach is dependent on comments, so as they are not complete, Yu et al. explain the need for further steps to be taken to recover stakeholder goals which they aimed for.

An important aspect in working with legacy code is that the code should never get worse even if there is no time (Feathers, 2004). Feathers (2004) gives some advice on what to do at a minimum when working with legacy code. By using the so-called *Sprout Method*, any new code should not be written inline, but be extracted with all necessary parameters and an optional return value. To get this code part under test, TDD should be used. The same holds for *Sprout Class* with a constructor and internal functionality. Feathers explains, however, that the problem here is that there is added always at least one line in the original function. When using *Wrap Method*, or *Wrap Class*, respectively, this is not the case as the original function is wrapped by another function, which then calls the original function.

According to Feathers (2004), the software is changed because of new features, bugs, design improvements, or optimization of time or memory. Feathers explains that the standard way of doing such changes is what he calls "edit and pray" – trying around after changing to check for any issues. The better solution is "cover and modify" – modifying the code securely by covering it with tests.

The next paragraphs are dedicated to testing in legacy code and describes its importance, and how to introduce them in case they are missing.

2.3.2 Testing Legacy Code

Besides the general consideration of the importance of testing during refactoring, as explained above, Feathers (2004) emphasizes that legacy code is often affected by high dependency. The problem is that classes that depend on other classes that are hard to test, testing this dependent class is also hard. So, the proposed solution by Feathers is breaking these dependencies to make changes easier. But here arises the problem which Feathers calls The Legacy Code Dilemma, which he explains by saying "when we change code, we should have tests in place. To put tests in place, we often have to change code." (Feathers, 2004, p. 15). Feather solves this problem by an algorithm that consists of finding points to change and points to test, then break potential dependencies, test the affected code parts, and finally make the changes including refactoring. Doing this effectively improves the code coverage in testing in the end.

There are two ways of breaking dependencies, on the one hand, sensing is used to get access to values which are needed for our tests, and, on the other hand, separation to make it possible testing code which was not possible before (Feathers, 2004). The solution presented for checking the values is so-called fake objects or mock objects. According to Mackinnon, Freeman, and Craig (2001) mock objects are more complex as they substitute functionality for various reasons, like the emulation or instrumentation of foreign code. As a result, they explain that mock objects can be used for testing. They say further that it is important that the code of mock objects should be as simple as possible and only aid to let unit tests run, as for example for sensing reasons. Feathers (2004) explains that not only mock objects but also fake objects need to implement the identified method to test. Therefore, both implement an interface that the original implementation also implements. While the new real object only calls the function of the original object, the test object executes simpler functionality. Feathers explains that fake objects implement further functionality to store values and give these values back to be checked within the unit test. So, the desired value could be fetched. Mock objects on the other side, are enhanced by directly asserting the desired values internally, as Feathers describes.

Feathers (2004) speaks about introducing seams, places where behaviour can be introduced without changing the original code when it comes to separation. Examples are the usage of subclassing, where a new class that is only used for testing overrides the original method without changing it in place. Feathers calls such a seam an *object seam*. Moreover, *link seams* can be used to change the call of a function by manipulating for example the Java class path.

As already described above, for legacy code, it is important to write characterization tests to understand the code and change it without changing the actual behaviour while refactoring (Feathers, 2015). By using the proposed *Method Use Rule* by Feathers, the testing coverage can be increased. It tells that developers are only allowed to use a method in case a test exists for it.

To conclude, using these principles, guidelines, and techniques help to make the code more understandable, less complex, and more maintainable.

3 Implementation Details

For applying the techniques and patterns discussed in the previous section the already mention medium-sized software project was used. The project is a good example for illustrating one of the main problems, why refactoring is needed, which Fowler (2018) discusses in his book. He notes that programmers often follow short-term goals ignoring the architecture or clean coding. In the case of this project, this is true as the main goal was always adding features. This can be seen in the timeline of the code complexity evaluation and also in 4.1.2.

As already mentioned, Java can be tested by unit tests using JUnit. In our case, we used version 4.12[7]. However, as described above, automated refactorings in IDEs like the used Android Studio are completely safe. Therefore, testing is not that important when using the provided refactoring tools. Furthermore, as already mentioned, testing is not the focus of this master's thesis, and therefore not explained in great detail. However, we followed our implementation guidelines regarding legacy code and TDD when possible.

In the next few sections, we present the implemented naming convention and coding standard, the applied code refactoring in Android Studio including the presentation of some implementation examples, the removal of unused resources, and in the end, the exchange of raster graphics with vector graphics.

3.1 Naming Convention and Coding Standard

As already mentioned above, in a naming convention, it is important to be consistent. Regarding the coding standard, fixed settings should be described to have a consistent code layout throughout all developers. The defined solutions are explained below. Moreover, we show how to apply a coding style in Android Studio, and where we show our naming conventions in the project, so that all developers follow the same rules.

3.1.1 Defined Naming Convention

In the developed convention four practices of writing names are distinguished: mixed case with lowercase first letter (also called "lowerCamelCase"), mixed case with the first letter capitalized (also called "UpperCamelCase"), lower case with underscores separating words and upper case with underscores separating words.

Like Oracle suggests for Java (Oracle, 1999), we write classes and interfaces in "UpperCamelCase". Methods and variables or class members are written in "lowerCamelCase". For

[7] https://github.com/junit-team/junit4/blob/HEAD/doc/ReleaseNotes4.12.md [accessed on 25 November 2020]

names of constants, we capitalize all letters and separate words by underscores. View identifiers (IDs) of Extensible Markup Language (XML)[8] files are written in lowercase letters with underscores as separator between words.

The defined naming convention for member variables is based on the suggestions, for instance variables of Kent Beck (1997), which are described in 2.2.2. Generally, collections like lists or hash maps are named in the plural form and describe their usage purpose. Other variable types like numbers or singular objects are named in the singular form.

Regarding the involvement of the type, the defined convention distinguishes between Views[9] and all other types. View names include their type as a prefix in a shortened form. For example, a TextView[10], which shows the title is called "tvTitle" in the Java file and "tv_title" in the corresponding XML file. So, we decided to use a Hungarian notation for view related variables to program faster, which means using a lowercase prefix for describing the type of usage of a variable (Legowski, 1996).

Apart from that, we only force the developers within the team to use descriptive names and be consistent in naming any entities.

For reasons of simplifications, we use the terminus view for any graphical user interface element in the further course of this work.

3.1.2 Drawable Resources

These types of resources are collected within the "drawable" folder of the project. For reasons of simplification and grouping, we differ between five different types. Depending on the type, which is derivated by the usage of the drawable and the file type of the resource, we use a specific prefix. Icons are prefixed by "ic_", names of pictures or images start with "im_", vector drawable icons from Google's Material Design Icons[11], which are converted to XMLs are prefixed by "mt_", and shapes which are drawable resources that are not icons themselves but can enhance the design of different views are identified by the prefix "sh_". Such files were created solely by writing XML code. As the project is in development, some resources need to be updated in future to fit the corporate identity. To know which files need to be adapted in the

[8] https://www.w3.org/XML/ [accessed on 15 November 2020]
[9] https://developer.android.com/reference/android/view/View [accessed on 15 November 2020]
[10] https://developer.android.com/reference/android/widget/TextView [accessed on 15 November 2020]
[11] https://material.io/resources/icons [accessed on 15 November 2020]

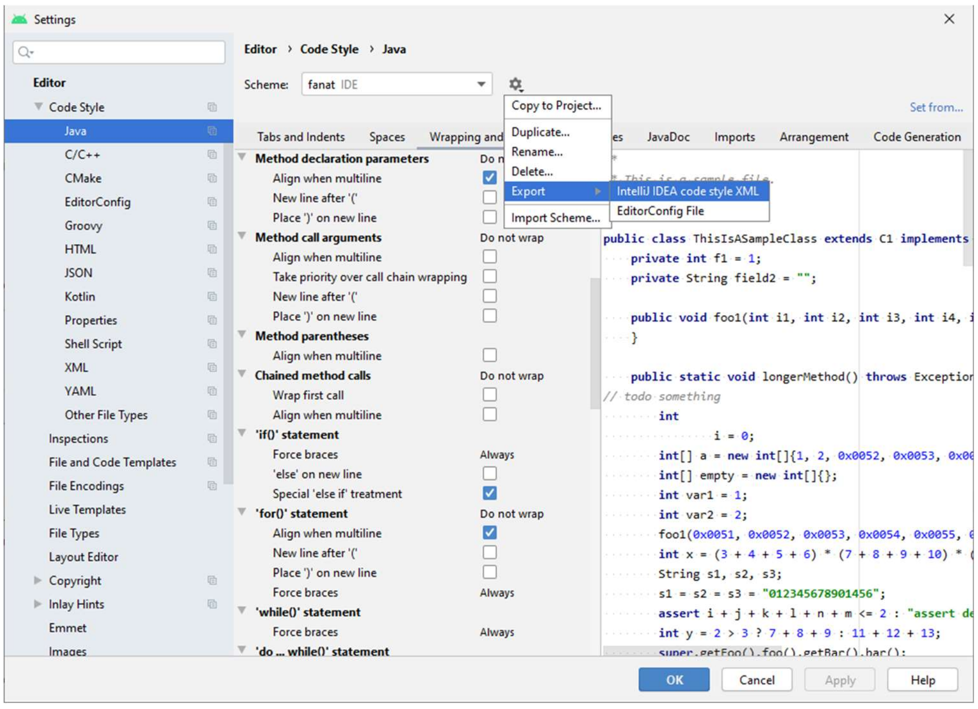
future, “td_” is used at the beginning of such raster graphic resource files. A summary of the defined naming convention for drawable files is displayed in Table 1.

Prefix	Derivation	Description	File Type
_	-	The users currently do not see these drawables prefixed by “_” but they are referenced in the code.	no restriction
ic_	"icon"	These icons are well-defined vector graphics.	XML
im_	"image"	These files are images or pictures and base on raster graphics.	JPEG, PNG
mt_	"material"	Google Material Design Icons get prefixed by "mt_" after conversion from SVG to XML.	XML
sh_	"shape"	Shapes are drawable resources to enhance the design of views. These files are created purely by XML.	XML
td_	"todo"	The prefix "td_" identifies drawables that are in use but still have to be adapted to the new design and need to be replaced in future.	JPEG, PNG

Table 1: The defined naming convention for drawable resources.

3.1.3 Coding Standard in Android Studio

In Android Studio, the coding standard or coding style can be adapted in the settings. Under “Editor” the settings for different file types as Java or XML can be adapted in “Code Style”. The defined settings can then be exported to an XML file, which can also be imported by the other team members. In Figure 10 it can be seen how the defined selections can be exported. For this master’s thesis, the first selection, “IntelliJ IDEA code style XML”, was used. The generated XML file content can be found in the Appendix.



Scheme: fanat IDE

Method declaration parameters: Do n
- Align when multiline ☑
- New line after '(' ☐
- Place ')' on new line ☐

Method call arguments: Do not wrap
- Align when multiline ☐
- Take priority over call chain wrapping ☐
- New line after '(' ☐
- Place ')' on new line ☐

Method parentheses
- Align when multiline ☐

Chained method calls: Do not wrap
- Wrap first call ☐
- Align when multiline ☐

'if()' statement
- Force braces: Always
- 'else' on new line ☐
- Special 'else if' treatment ☑

'for()' statement: Do not wrap
- Align when multiline ☑
- New line after '(' ☐
- Place ')' on new line ☐
- Force braces: Always

'while()' statement
- Force braces: Always

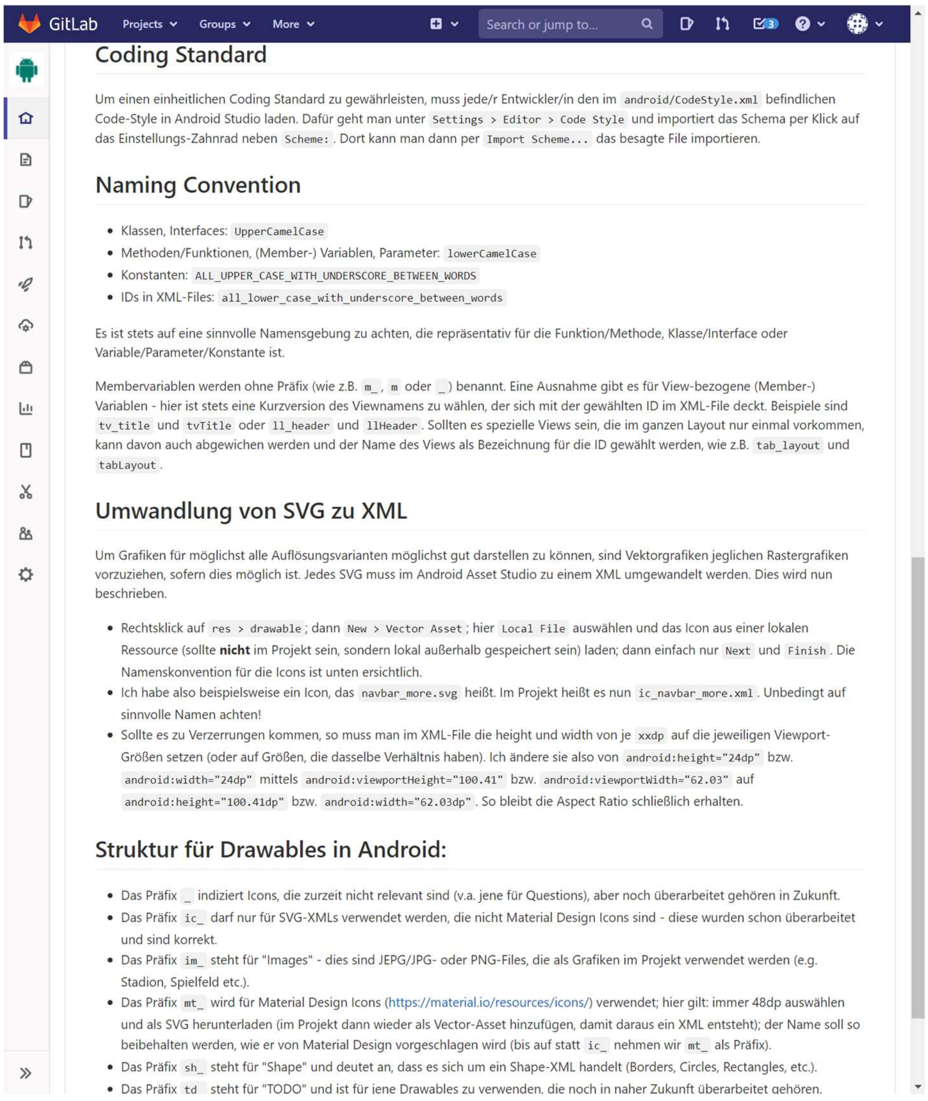
'do ... while()' statement

```
* This is a sample file.
public class ThisIsASampleClass extends C1 implements
    private int f1 = 1;
    private String field2 = "";

    public void foo1(int i1, int i2, int i3, int i4, i
    }

    public static void longerMethod() throws Exceptio
// todo something
        int
                i = 0;
        int[] a = new int[]{1, 2, 0x0052, 0x0053, 0x00
        int[] empty = new int[]{};
        int var1 = 1;
        int var2 = 2;
        foo1(0x0051, 0x0052, 0x0053, 0x0054, 0x0055, 0
        int x = (3 + 4 + 5 + 6) * (7 + 8 + 9 + 10) * (
        String s1, s2, s3;
        s1 = s2 = s3 = "012345678901456";
        assert i + j + k + l + n + m <= 2 : "assert de
        int y = 2 > 3 ? 7 + 8 + 9 : 11 + 12 + 13;
        super.getFoo().foo().getBar().bar();
```

Figure 10: Exporting code style settings in Android Studio.

3.1.4 Implementation for Acceptance of Team Members

The implemented coding standard and the naming convention is part of the README.md. With the help of this, these two important things are automatically shown in GitLab's project overview as displayed in Figure 11. Such team rules are important to be followed due to consistency reasons (Martin, 2008). Hence, placing it at a central point is crucial for the adoption of the new rules. Moreover, we instructed the participating developers to discuss the conventions and to adapt it accordingly.



Coding Standard

Um einen einheitlichen Coding Standard zu gewährleisten, muss jede/r Entwickler/in den im `android/CodeStyle.xml` befindlichen Code-Style in Android Studio laden. Dafür geht man unter `Settings > Editor > Code Style` und importiert das Schema per Klick auf das Einstellungs-Zahnrad neben `Scheme:`. Dort kann man dann per `Import Scheme...` das besagte File importieren.

Naming Convention

- Klassen, Interfaces: `UpperCamelCase`
- Methoden/Funktionen, (Member-) Variablen, Parameter: `lowerCamelCase`
- Konstanten: `ALL_UPPER_CASE_WITH_UNDERSCORE_BETWEEN_WORDS`
- IDs in XML-Files: `all_lower_case_with_underscore_between_words`

Es ist stets auf eine sinnvolle Namensgebung zu achten, die repräsentativ für die Funktion/Methode, Klasse/Interface oder Variable/Parameter/Konstante ist.

Membervariablen werden ohne Präfix (wie z.B. `m_`, `m` oder `_`) benannt. Eine Ausnahme gibt es für View-bezogene (Member-) Variablen - hier ist stets eine Kurzversion des Viewnamens zu wählen, der sich mit der gewählten ID im XML-File deckt. Beispiele sind `tv_title` und `tvTitle` oder `ll_header` und `llHeader`. Sollten es spezielle Views sein, die im ganzen Layout nur einmal vorkommen, kann davon auch abgewichen werden und der Name des Views als Bezeichnung für die ID gewählt werden, wie z.B. `tab_layout` und `tabLayout`.

Umwandlung von SVG zu XML

Um Grafiken für möglichst alle Auflösungsvarianten möglichst gut darstellen zu können, sind Vektorgrafiken jeglichen Rastergrafiken vorzuziehen, sofern dies möglich ist. Jedes SVG muss im Android Asset Studio zu einem XML umgewandelt werden. Dies wird nun beschrieben.

- Rechtsklick auf `res > drawable`; dann `New > Vector Asset`; hier `Local File` auswählen und das Icon aus einer lokalen Ressource (sollte **nicht** im Projekt sein, sondern lokal außerhalb gespeichert sein) laden; dann einfach nur `Next` und `Finish`. Die Namenskonvention für die Icons ist unten ersichtlich.
- Ich habe also beispielsweise ein Icon, das `navbar_more.svg` heißt. Im Projekt heißt es nun `ic_navbar_more.xml`. Unbedingt auf sinnvolle Namen achten!
- Sollte es zu Verzerrungen kommen, so muss man im XML-File die height und width von je `xxdp` auf die jeweiligen Viewport-Größen setzen (oder auf Größen, die dasselbe Verhältnis haben). Ich ändere sie also von `android:height="24dp"` bzw. `android:width="24dp"` mittels `android:viewportHeight="100.41"` bzw. `android:viewportWidth="62.03"` auf `android:height="100.41dp"` bzw. `android:width="62.03dp"`. So bleibt die Aspect Ratio schließlich erhalten.

Struktur für Drawables in Android:

- Das Präfix `_` indiziert Icons, die zurzeit nicht relevant sind (v.a. jene für Questions), aber noch überarbeitet gehören in Zukunft.
- Das Präfix `ic_` darf nur für SVG-XMLs verwendet werden, die nicht Material Design Icons sind - diese wurden schon überarbeitet und sind korrekt.
- Das Präfix `im_` steht für "Images" - dies sind JEPG/JPG- oder PNG-Files, die als Grafiken im Projekt verwendet werden (e.g. Stadion, Spielfeld etc.).
- Das Präfix `mt_` wird für Material Design Icons (https://material.io/resources/icons/) verwendet; hier gilt: immer 48dp auswählen und als SVG herunterladen (im Projekt dann wieder als Vector-Asset hinzufügen, damit daraus ein XML entsteht); der Name soll so beibehalten werden, wie er von Material Design vorgeschlagen wird (bis auf statt `ic_` nehmen wir `mt_` als Präfix).
- Das Präfix `sh_` steht für "Shape" und deutet an, dass es sich um ein Shape-XML handelt (Borders, Circles, Rectangles, etc.).
- Das Präfix `td` steht für "TODO" und ist für jene Drawables zu verwenden, die noch in naher Zukunft überarbeitet gehören.

Figure 11: The implemented coding standard and naming convention is part of the project's README.md.

Even if there are still loads of badly coded parts or files within the project which do not meet the conventions, we urge the team members not to refactor the code until they use the code. As described in 2.1 above, it is best practice to do refactoring only in places where something needs to be done, like understanding something, as described for comprehension refactoring, or when somewhere a feature needs to be added, as described for preparatory refactoring. As

Martin Fowler (2018) cites a camping adage in his book, "… always leave the campsite cleaner than when you found it." (Fowler, 2018, p. 52), refactoring is only needed in case a file is touched. Therefore, forced code refactoring, or change of names in files that are not currently needed, is not needed and waste of time in most cases.

3.2 Applied Code Refactoring in Android Studio

Within the project, we identified some complex parts which need urgent a reduction of complexity. As described in 2.1.2, there are several different motivations to do code refactoring. In this section, some implementations of these two types are described and shown. Besides the presented examples, we refactored and implemented other parts of the application also, like the notification listener and enhanced notification settings, but they are not explained in detail here.

While implementing and refactoring the code, we tried to follow the theoretically discussed clean code principles, like SRP, or OCP, abstract class factory, short methods, and much more, and the code refactoring techniques, as described above, to code as clean as possible and to increase the readability while reducing the complexity.

In order not to display real data, we have used fictitious names for all figures and placeholder images for all logos. The presented listings were taken from the repository of the project, whereby the copyright owner has approved the usage. Any listings that were not written in the course of this master's thesis are marked with copyright[12].

3.2.1 Comprehension Refactoring

As we know from the definition, comprehension refactoring is used in cases where parts of code are too complex to understand (Fowler, 2018). Therefore, the aim of this refactoring type is to make the source code easier to understand. An implementation of the mentioned project is described in the next section. To further simplify termini, any small written views refer to their according to view representation, like fragment always refers to Fragment[13].

Dynamic Tabbed Layout and Collapsible Header

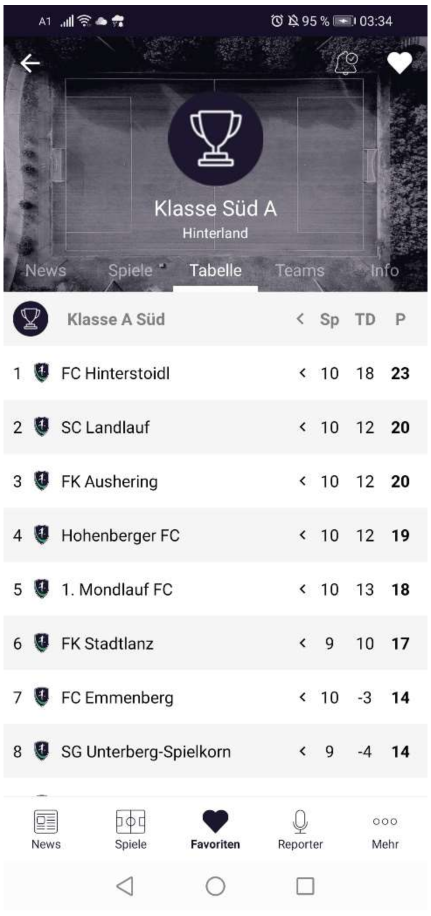
The app includes several views were tabbed layouts are used. That means, that the fragment that holds the view includes within itself several other fragments that can be changed by swiping to the left or the right or clicking the tab view titles, respectively. The challenge is now to make these views dynamic and find a way to load different amounts of tabs. Another problem

[12] © by SFA Sport GmbH
[13] https://developer.android.com/reference/android/app/Fragment [accessed on 17 November 2020]

that we need to solve is that the upper views need to be collapsed in case the user loads more content and navigates down because otherwise there is too less space for the content on the mobile screen. Both problems need to be solved within the project. The codebase of these views is so complicated, though, that it is very hard to reach the goals in a pleasant and fast way without introducing new bugs. Therefore, we use comprehension refactoring to understand the architecture of these views and to understand how to cope with the dynamic not only in the header but also in the tab views. Moreover, a goal here is to remove the bad smell of *Interdeveloper Duplication* as described above in 2.2.3, which Hunt and Thomas (2000) use for redundant code, that was introduced by multiple developers. In this case, it means that the developers tended to copy files and methods simply instead of using inheritance. So, this goal can be reached by using polymorphism.

Status Quo

In Figure 12 a typical screen is presented. It shows the profile page for a Styrian soccer league with its five content tabs for news, events, standings, clubs, and information about the league. The problem at this development status is, though, that it takes much effort to change the code in a way that the fixed-sized header can be collapsed and that the static tab views can be removed and added dynamically. Not only the many different implementations for all the views, namely the game view for presenting events, the club profile page, and the league profile page, but also the understanding of how the implementations work and how to make it possible to change the code in a safe way without introducing new bugs to the legacy code is difficult.



	Klasse A Süd	<	Sp	TD	P
1	FC Hinterstoidl	<	10	18	**23**
2	SC Landlauf	<	10	12	**20**
3	FK Aushering	<	10	12	**20**
4	Hohenberger FC	<	10	12	**19**
5	1. Mondlauf FC	<	10	13	**18**
6	FK Stadtlanz	<	9	10	**17**
7	FC Emmenberg	<	10	-3	**14**
8	SG Unterberg-Spielkorn	<	9	-4	**14**

Figure 12: A typical view with a fixed sized header and tab views below.

Before these dynamic tabs, a placeholder was shown to tell the user that there is no content available or not available yet. Sometimes these messages were completely inadequate. Furthermore, regarding usability, such messages are very distracting and not user-friendly.

Regarding the dynamic tabs, the developers tried to implement it before doing comprehension refactoring as can be seen in Listing 1. This implementation did work to show a dynamic number of views in the game fragment but as can be seen, it is very complicated and not very easy to understand. Many principles are OCP, or SRP are ignored. Moreover, in case there is a change in logic, it is very difficult to find out what to do next. This code example belongs to the game pager adapter of type Adapter[14] which is responsible for handling the different views in the tabbed event fragment. Not only the correct fragment instance needs to be returned, but also the correct name for the tab title and the correct number of views needs to be returned for the given settings and the passed index or position. So, this code is repeated two times leading to code duplication. The public static variable "staticEvent" should also be removed.

[14] https://developer.android.com/reference/android/widget/Adapter [accessed on 17 November 2020]

```
@Override
public Fragment getItem(int index) {
    Constants.Phase phase = GameFragment.staticEvent.getPhase();
    boolean videoEnabled = GameFragment.staticEvent.getTickerURL() != null;
    boolean lineupEnabled = GameFragment.staticEvent.homeHasLineup() ||
            GameFragment.staticEvent.awayHasLineup();
    if (index == 0) {
        return new GameOverviewFragment();
    }
    if (index == 1 && !isPhaseUpcomingCancel(phase)) {
        return new TimelineFragment();
    }
    if (videoEnabled && ((index == 1 && isPhaseUpcomingCancel(phase)) ||
            (index == 2 && isPregameStartPhase(phase)))) {
        return new VideosFragment();
    }
    if (isGamePhase(phase) && GameFragment.showOverview && ((index == 3 &&
            videoEnabled) || (index == 2 && !videoEnabled))) {
        return new OverviewFragment();
    }
    if ((isGamePhase(phase) && GameFragment.showOverview &&
            isGamePhase(phase) && lineupEnabled &&
            ((index == 4 && videoEnabled) || (index == 3 && !videoEnabled))) ||
            (isPregamePhase(phase) && lineupEnabled &&
                    ((index == 3 && videoEnabled) ||
                            (index == 2 && !videoEnabled)))) {
        return new LineupGameFragment();
    }
    return new LineupGameFragment();
}
```

Listing 1: An example of how complex fragments were returned before. © by SFA Sport GmbH

For the collapsible header not only the tangled XML file needs to be understood, which can be seen partly and shortened in Listing 2, not following any vertical order for neighboured views, but also the complex code within the game fragment which represents the container for the header and the tabbed view. These problems apply not only to the event view but also to the league and club profile pages.

```
<FrameLayout android:layout_width="match_parent">
    <ImageView android:id="@+id/background_image" />
    <View android:id="@+id/background_image_overlay" />
    <LinearLayout android:id="@+id/header">
        <RelativeLayout>
            <TextView android:id="@+id/event_header_time"
                android:layout_above="@+id/event_header_score" />
            <RelativeLayout android:id="@+id/sponsor_click_listener_holder"
                android:layout_below="@id/event_header_score">
                <TextView android:id="@+id/sponsored_by"
                    android:layout_above="@+id/event_header_sponsor" />
                <TextView android:id="@+id/event_header_sponsor" />
            </RelativeLayout>
            <FrameLayout
                android:layout_above="@+id/dummy"
                android:layout_toRightOf="@+id/event_header_score">
                <ImageView android:id="@+id/event_logo_away_team_black" />
                <ImageView android:id="@+id/event_logo_away_team"></ImageView>
            </FrameLayout>
            <TextView android:id="@+id/event_header_team_away_name"
                android:layout_toRightOf="@+id/event_header_score" />
            <LinearLayout android:id="@+id/dummy"
                android:layout_above="@+id/sponsor_click_listener_holder" />
            <FrameLayout android:id="@+id/logos_home"
                android:layout_above="@id/dummy"
                android:layout_toLeftOf="@+id/event_header_score">
                <ImageView android:id="@+id/event_logo_home_team_black" />
                <ImageView android:id="@+id/event_logo_home_team"/>
            </FrameLayout>
            <TextView android:id="@+id/event_header_team_home_name"
                android:layout_below="@+id/logos_home"
                android:layout_toLeftOf="@+id/event_header_score" />
            <TextView android:id="@+id/event_header_score" />
        </RelativeLayout>
        <android.support.design.widget.TabLayout>
        </android.support.design.widget.TabLayout>
    </LinearLayout>
</FrameLayout>
```

Listing 2: An excerpt of the XML layout for the header of the game fragment in the event view where only the IDs and the references to them were kept for this listing. © by SFA Sport GmbH

Requirements

When viewing a lot of information on a smartphone, it is always important to use the available space on the screen as much as possible. That means for this problem statement, that the user should see the most important and revealing information in the beginning when entering a screen, namely the event view, the club profile page, or the profile page for the league, but when the user wants to see more specialized content like news, standings, or games, the screen should be filled with that information while the header shrinks and should only display the very most important information. To reach this goal, a general solution for all three views should be found and it should meet all design proposals explained below.

Additionally, there should never be displayed any placeholder for empty screens, but these empty screens should not be visible in these cases, instead. To put it differently, the goal is to hide and show all defined views according to given settings. For instance, for future events it does not make sense to display an empty live ticker view, an empty timeline of the events within the game, or the not yet published line-ups. To be more specific, the moment of defining the settings for the tabbed view is before creating the responsible game pager adapter.

In the next few paragraphs, the graphical requirements are presented.

Generally, the developers tend to first draw mock-ups for the graphical interfaces on a black-board or on paper. Then, after some refinements, they draw the graphical user interfaces within Figma[15], which helps to create designs collaboratively online. The used version within the project is totally free. In addition to the easy and fast creation of mock-ups, this browser supported

[15] https://www.figma.com/design/ [accessed on 18 November 2020]

platform provides a possibility to test the design and to click through the different screens, comments can be made and be discussed within the platform (Figma, 2020).

To begin with, in Figure 13 the new header can be seen. In this section the focus is on the mechanism of collapsing the header. The design itself is shown and discussed below in the examples of preparatory refactoring for the new profile views of the clubs and leagues and the new event view for games, respectively.

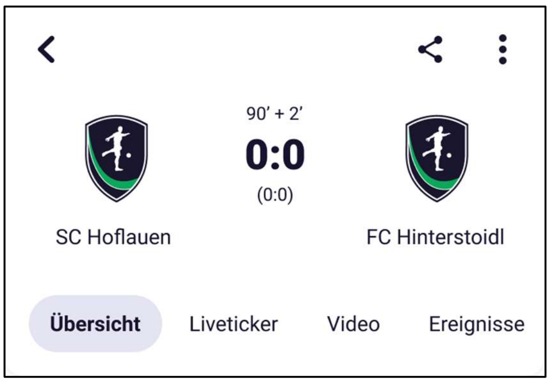


Figure 13: The new designed full header for a game with detailed content being displayed.

In fact, the focus should lie on the structure of this layout to understand its components and to choose the most important characteristics for refactoring. Generally, the view can be split up into three main parts, namely the menu bar, the main bar, and the tab bar.

The menu bar is predefined by Android with a fixed size. The main bar consists here of a centred score and minute view with variable sized content, two fixed-sized images to the left and to the right, and strings centred to the bottom of each image of variable size. On the bottom of the header, the dynamic number of tab titles can be found. In short, the height of the header should be of a fixed size, but when scrolling down on the tab views, the header should shrink in a way that only the most important information can be seen within the header. More about these individual details can be found below in section 3.2.2.

Results

After applying comprehension refactoring not only to the files associated with the event but also to the files for the profile pages of the clubs and leagues, it turned out that the main logic of the code for the header should be in the class for the container of all the tabbed fragments.

These container classes are responsible for the data, which is needed for handling the TabLayout[16] and the PagerAdapter[17].

All necessary data that is needed to load the views of the different tabs, like data for the opened event in the game view or the instance of the clicked club in its profile page, need to be known and loaded before creating a new instance of the container fragment. This data gets loaded within the static *open*-function or is passed by argument. If this function succeeds, a new instance for the container fragment class will be created.

Within the responsible *newInstance*-function, the constructor of the container class will be called. Then, arguments for the new fragment are created. These arguments remain available even if the fragment gets destroyed or newly created, except the arguments get overwritten or deleted explicitly[18]. The data type of these arguments is Bundle[19], which maps from a string to values which are from type Parcelable[20]. Besides methods for primitive data types as int or boolean, Bundle provides also methods for objects of type Serializable[21], which is an interface without methods or fields but marks the implementing class and all subclasses as being able to be serialized and deserialized. An example would be the class representing an event within the project. To prevent creating a new instance for this class without calling *open* before, we set the constructor for the container class and the static *newInstance*-function to private. This functionality for creating a new instance of the container fragment class is listed in Listing 3.

```
public static void open(...) {
    onSuccess() {
        SomeFragment.newInstance(...);
    }
}
private static SomeFragment newInstance(...) {
    SomeFragment result = new SomeFragment();
    result.setArguments(createArguments(...));
    return result;
}
private Bundle createArguments(...) {
    Bundle result = new Bundle();
    result.putSerializable(SOME_CONSTANT, ...);
    return result;
}
private SomeFragment () {}
```

Listing 3: Defined functionality for each container fragment class which is needed for creating a new instance safely.

[16] https://developer.android.com/reference/com/google/android/material/tabs/TabLayout [accessed on 18 November 2020]
[17] https://developer.android.com/reference/androidx/viewpager/widget/PagerAdapter [accessed on 18 November 2020]
[18] https://developer.android.com/reference/android/app/Fragment [accessed on 18 November 2020]
[19] https://developer.android.com/reference/android/os/Bundle [accessed on 18 November 2020]
[20] https://developer.android.com/reference/android/os/Parcelable [accessed on 19 November 2020]
[21] https://developer.android.com/reference/java/io/Serializable [accessed on 19 November 2020]

In the overridden *onCreateView*-method of the already mentioned Fragment class, the layout for the container class gets inflated, the Bundle arguments get parsed, the tabbed layout gets setup, and the menu items get updated and not created, as they exist for other fragments also.

Within the *setupTabbedLayout*-method, the pager adapter is set up at first as it populates the pages for the corresponding ViewPager[22], which represents the layout manager for the tab view pages. After the creation of the view pager, the tab layout gets set up in the end, which is also connected to the view pager. This layout is needed, among other things, for listening on tab selection changes or handling the tab titles including their design.

```
@Override
public View onCreateView(...) {
    inflateFragmentView(inflater, container);
    parseArguments();
    ...
    setupTabbedLayout();
    updateMenuItems();
    ...
}

private void setupTabbedLayout() {
    setupPagerAdapter();
    setupViewPager();
    setupTabLayout();
}

protected void setupPagerAdapter() {
    SomePagerAdapter.setupViewPager(viewPager);
    pagerAdapter = new SomePagerAdapter(...);
}
```

Listing 4: Steps taken to create the tabbed layout within the container fragment class.

One crucial thing, which we understood by the comprehension refactoring, is the passing of the reference to the view pager layout to the pager adapter, which stores this reference statically. The problem is that the pager adapter needs the containing view, which is the view pager in this case, in its *instantiateItem*-method. As the tabbed view is built dynamically, the functionality is changed so that this requirement is satisfied. Such a pager adapter needs to provide the functionality shown in Listing 5. As the page title is needed before *instantiateItem* would be called, this method calls it implicitly before. In case the desired fragment is not null, it can be returned from the list without doubt. With this technique, each fragment is prevented to be initialized two times. We found this phenomenon during comprehension refactoring. We revealed that the fragments were constructed two times and therefore, we found a lot of bugs and could remove them afterwards.

[22] https://developer.android.com/reference/androidx/viewpager/widget/ViewPager [accessed on 27 November 2020]

```java
protected FragmentTab getFragmentTab(int position) {
    return fragments.get(position);
}

@NonNull
@Override
public Fragment getItem(int position) {
    if (getFragmentTab(position) != null) {
        return getFragmentTab(position);
    }

    return makeFragmentTab(position);
}

@Override
public String getPageTitle(int position) {
    if (getFragmentTab(position) != null) {
        return getFragmentTab(position).getPageTitle();
    }

    return ((FragmentTab) instantiateItem(viewPager, position)).getPageTitle();
}


@Override
public Object instantiateItem(ViewGroup container, int position) {
    if (getFragmentTab(position) != null) {
        return getFragmentTab(position);
    }

    Fragment createdFragment = super.instantiateItem(container, position);

    fragments.set(position, (FragmentTab) createdFragment);

    return createdFragment;
}
```

Listing 5: The functionality which a pager adapter needs to implement to satisfy the dynamically created tabbed view.

One thing to mention is the method call of *makeFragmentTab*. It is a method of an interface *FragmentTabFactory,* which defines beside of this the method *getPageTitle*. This interface defines some functionality that must be implemented by all pager adapters that want to satisfy the dynamic functionality for their tab views. The interface can be seen in Listing 6.

```java
public interface FragmentTabFactory {
    FragmentTab makeFragmentTab(int position);
    String getPageTitle(int position);
}
```

Listing 6: The FragmentTabFactory defines the method for creating new FragmentTabs and getting their title for the tab layout.

As described in 2.2.1 above, this design pattern follows the object creation design pattern abstract factory as the interface provides the possibility to create objects without knowing the specific classes, whereby the interface *FragmentTabFactory* represents the abstract factory as it declares the interface to create tab fragments (Gamma et al., 1994). The *TabPagerAdapter*, which we explain in detail below, is a concrete factory as it implements the defined methods of the abstract factory. The factory creates objects of type *FragmentTab* which is abstract and defines two methods, namely *getPageTitle* and *logTabSelected*. These two methods are crucial for any tab view. The corresponding interface can be seen in Listing 7. Concrete tab fragments represent any tab views that are used within the project. During the creation of this master's thesis, we implemented in 15 classes the fragment tab interface.

As described in 2.2.1 above, the big advantage of this abstract factory is that the clients only use the interfaces of the abstract factory and the abstract product classes (Gamma et al., 1994).

```
public abstract class FragmentTab extends MainActivityFragment {
    public abstract String getPageTitle();
    public abstract void logTabSelected();
}
```

Listing 7: The abstract class which defines an interface for creating concrete objects by the factory.

"Duplication may be the root of all evil in software." (Martin, 2008, p. 48), Robert C. Martin aptly writes in his book. Therefore, a base class is created for eliminating the redundant code of all tabbed fragment pager adapters. All the functionality shown in listing 5 is implemented within this abstract class called *TabPagerAdapter* which extends FragmentPagerAdapter[23], and implements the above presented *FragmentTabFactory*. At the time of this master's thesis, three pager adapters extended this abstract class, namely those for the club profile page, those for the league profile page, and those for the game or event page.

To handle the tab dynamics, a backend endpoint was added to receive information about what tabs to show for the fragments. This information is passed to the *PagerAdapter* by a special class. The concrete implementation for the game view is partly shown in Listing 8.

[23] https://developer.android.com/reference/androidx/fragment/app/FragmentPagerAdapter [accessed on 18 November 2020]

```java
public FragmentTab getGameFragmentTabAtPosition(int position) {
    int index = 0;
    if (overview.isActivated()) {
        if (index == position) {
            return OverviewFragment.newInstance(…);
        } else {
            index += 1;
        }
    }
    if (liveticker.isActivated()) {
        if (index == position) {
            return LivetickerFragment.newInstance(…);
        } else {
            index += 1;
        }
    }
    if (videos.isActivated()) {
        if (index == position) {
            return VideosFragment.newInstance(…);
        } else {
            index += 1;
        }
    }
    if (timeline.isActivated()) {
        if (index == position) {
            return TimelineFragment.newInstance(…);
        }
    }
    return LineupFragment.newInstance(…);
}
```

Listing 8: An example for handling dynamic tab views according to the tab settings for the event view.

The second problem we need to face is the collapsing header. After the comprehension refactoring, it was clear that the layout for all container layouts needed to be adapted. Therefore, the layout outlined in Listing 9 is chosen.

```
<androidx.coordinatorlayout.widget.CoordinatorLayout>

    <com.google.android.material.appbar.AppBarLayout>

        <com.google.android.material.appbar.CollapsingToolbarLayout
            app:layout_scrollFlags="scroll|exitUntilCollapsed|snap"
            app:contentScrim="@color/white">

            <androidx.appcompat.widget.Toolbar
                app:layout_collapseMode="pin" />

            <LinearLayout>

                <LinearLayout
                    android:id="@+id/ll_header">
                      ...
                </LinearLayout>

                <LinearLayout
                    android:id="@+id/ll_header_collapsed">
                      ...
                </LinearLayout>

                <com.google.android.material.tabs.TabLayout />

            </LinearLayout>

        </com.google.android.material.appbar.CollapsingToolbarLayout>

    </com.google.android.material.appbar.AppBarLayout>

    <androidx.viewpager.widget.ViewPager
        app:layout_behavior="@string/appbar_scrolling_view_behavior" />

</androidx.coordinatorlayout.widget.CoordinatorLayout>
```

Listing 9: A simplified draft for the new collapsing layout with the most important components and properties.

The outermost container is a CoordinatorLayout[24] which can handle the desired collapsing header. The AppBarLayout[25] is dependent on the outer CoordinatorLayout and has the CollapsingToolbarLayout[26] as child, which is of fixed size. By *contentScrim* the colour for the collapsed header can be defined, here white is chosen. Furthermore, that colour is shown when extending or reducing the size of the header by scrolling gestures. The *layout_scrollFlags* are set to "scroll", which is needed for any scrolling behaviour, to "exitUntilCollapsed" makes the header be collapsed completely before scrolling any other thing vertically, and "snap" makes the header never stuck between open and collapsed[27]. The Toolbar[28] element is of the same

[24] https://developer.android.com/reference/androidx/coordinatorlayout/widget/CoordinatorLayout [accessed on 17 November 2020]
[25] https://developer.android.com/reference/com/google/android/material/appbar/AppBarLayout [accessed on 17 November 2020]
[26] https://developer.android.com/reference/com/google/android/material/appbar/CollapsingToolbarLayout [accessed on 17 November 2020]
[27] https://developer.android.com/reference/com/google/android/material/appbar/AppBarLayout.LayoutParams [accessed on 17 November 2020]
[28] https://developer.android.com/reference/android/widget/Toolbar [accessed on 18 November 2020]

size as the common toolbar within the app, namely 45 density pixels (dp). The *layout_collapseMode* set to “pin” makes the view stuck in place on scroll until the header is collapsed completely[29]. Therefore, it serves as the white background for the collapsed header.

The ViewPager is the part of the layout where the views will be scrolled. Therefore, to indicate this to the AppBarLayout, its *layout_behavior* is set to the predefined string resource “appbar_scrolling_view_behavior”. As a result, the AppBarLayout knows when to scroll[30]. One important detail to mention is that the child tab layouts need to use NestedScrollView with the same *layout_behavior* instead of simple ScrollView as outermost layout to work correctly.

Not only the collapsed header and the full header with the detailed information is included within the nested LinearLayout[31], indicated in Listing 9 by their identifiers “ll_header” and “ll_header_collapsed”, but also the TabLayout which is connected with the ViewPager as discussed above. In Figure 14 the most important components for the new layout are sketched.

Regarding the Java part for this implementation, we decided to introduce a new abstract class for the handling of the collapsible header and the dynamic tab views due to redundancy and maintenance issues as all container of tab views need this implementation, these are the event page and the profile pages of leagues and clubs. This new class is called *TabFragment* and implements all functionality discussed at the beginning of this section for the dynamic tab views.

[29] https://developer.android.com/reference/com/google/android/material/appbar/CollapsingToolbarLayout.LayoutParams [accessed on 19 November 2020]
[30] https://developer.android.com/reference/com/google/android/material/appbar/AppBarLayout.ScrollingViewBehavior [accessed on 19 November 2020]
[31] https://developer.android.com/reference/android/widget/LinearLayout [accessed on 20 November 2020]

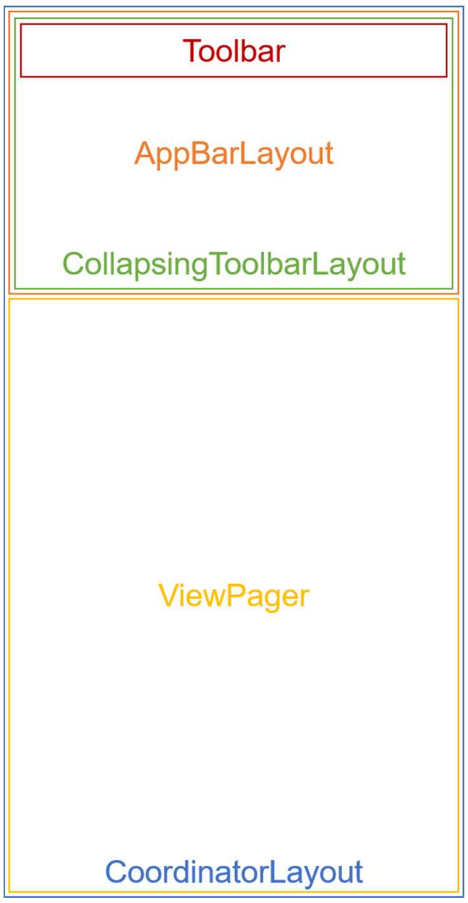


Figure 14: A sketch for the layout with a collapsing header and tabbed views.

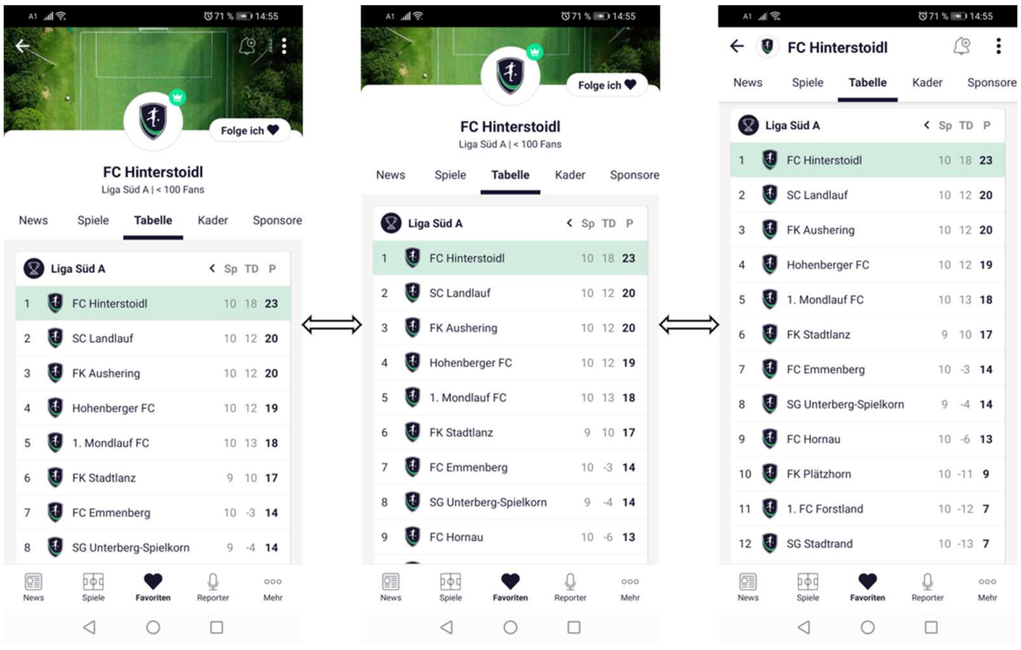
For this purpose, we include the setup of the AppBarLayout to the end of the setup of the tabbed layout shown in Listing 4. Within this setup, an *OnOffsetChangedListener*[32] is added to the AppBarLayout**.** This callback is invoked when the header gets scrolled. As can be seen in Listing 10, there are three different states for the header, namely, “OPEN”, “IN_SCROLL”, and “COLLAPSED”. The state gets toggled on each vertical offset change, whereby only on the extrema the state is not “IN_SCROLL”. These three states are important for the design. If the header is collapsed, the small layout is shown. That is only in case the smallest extremum is reached. Otherwise, the large layout with detailed information is visible. In case the state is “IN_SCROLL” the menu items get invisible to not interfere with the content of the header.

[32] https://developer.android.com/reference/com/google/android/material/appbar/AppBarLayout.OnOffsetChangedListener [accessed on 17 November 2020]

```
protected enum AppBarState {
    OPEN, IN_SCROLL, COLLAPSED
}

private void setupAppBarLayout() {
    appBarLayout.addOnOffsetChangedListener(new OnOffsetChangedListener() {
        @Override
        public void onOffsetChanged(AppBarLayout appBarLayout, int vertOffset) {
            toggleAppBarLayout(vertOffset, appBarLayout.getTotalScrollRange());
        }
    });
}

private void toggleAppBarLayout(int vertOffset, int totalScrollRange) {
    if (appBarState != getAppBarState(vertOffset, totalScrollRange)) {
        updateLayoutAndState();
    }
}

private AppBarState getAppBarState(int vertOffset, int totalScrollRange) {
    if (Math.abs(Math.abs(vertOffset) - totalScrollRange) == 0) {
        return AppBarState.COLLAPSED;
    }
    if (vertOffset == 0) {
        return AppBarState.OPEN;
    }
    return AppBarState.IN_SCROLL;
}
```

Listing 10: Needed functionality for a working collapsing header in pseudo code.

An example of all three states in action is shown in Figure 15. As can be seen, also the background picture is changed slightly on scroll. Here, in the profile page of a club, or also in those of a league, the colour of the menu items is different in “OPEN” and “COLLAPSED”. This is handled in the concrete implementations of the abstract class *TabFragment*.



FC Hinterstoidl
Liga Süd A | < 100 Fans

News Spiele **Tabelle** Kader Sponsore

Liga Süd A		Sp	TD	P
1	FC Hinterstoidl	10	18	**23**
2	SC Landlauf	10	12	**20**
3	FK Aushering	10	12	**20**
4	Hohenberger FC	10	12	**19**
5	1. Mondlauf FC	10	13	**18**
6	FK Stadtlanz	9	10	**17**
7	FC Emmenberg	10	-3	**14**
8	SG Unterberg-Spielkorn	9	-4	**14**

FC Hinterstoidl
Liga Süd A | < 100 Fans

News Spiele **Tabelle** Kader Sponsore

Liga Süd A		Sp	TD	P
1	FC Hinterstoidl	10	18	**23**
2	SC Landlauf	10	12	**20**
3	FK Aushering	10	12	**20**
4	Hohenberger FC	10	12	**19**
5	1. Mondlauf FC	10	13	**18**
6	FK Stadtlanz	9	10	**17**
7	FC Emmenberg	10	-3	**14**
8	SG Unterberg-Spielkorn	9	-4	**14**
9	FC Hornau	10	-6	**13**

FC Hinterstoidl

News Spiele **Tabelle** Kader Sponsore

Liga Süd A		Sp	TD	P
1	FC Hinterstoidl	10	18	**23**
2	SC Landlauf	10	12	**20**
3	FK Aushering	10	12	**20**
4	Hohenberger FC	10	12	**19**
5	1. Mondlauf FC	10	13	**18**
6	FK Stadtlanz	9	10	**17**
7	FC Emmenberg	10	-3	**14**
8	SG Unterberg-Spielkorn	9	-4	**14**
9	FC Hornau	10	-6	**13**
10	FK Plätzhorn	10	-11	**9**
11	1. FC Forstland	10	-12	**7**
12	SG Stadtrand	10	-13	**7**

Figure 15: The header states in the profile page of a club: "OPEN" (left), "IN_SCROLL" (middle), and "COLLAPSED" (right).

In summary, the conducted comprehension refactoring aided in understanding the mechanics of the container views and the handling of the header. Not only bugs could be found but also redundancy could be removed by creating some abstract super classes. With the help of abstract class factory some code and with it the responsibility for the dynamic tab views could be extracted. We found a general approach for dynamic headers and developed easily after doing the refactoring.

Some implemented examples for the next type of refactoring are shown in the next section.

3.2.2 Preparatory Refactoring

As already mentioned in chapter 2.1.2, preparatory refactoring is used when new features are added to an existing code base and the goal is to make it easier to add that feature. Kent Beck (2012) sums up the intention for such an approach aptly on Twitter[33], where he suggests that refactoring can help to make any change more easily (Beck [KentBeck], 2012). The next few examples include the most complex parts within the project before we introduced refactoring. In the next chapters, the applied preparatory refactoring is shown, and the individual changes are presented.

New League and Club Profile Page

One core aspect of the application is the profile pages of the leagues and clubs. In leagues the user can see the news, the game list of the league, the standings, the clubs within the league, and some general information about the league. The profile page of a team shows their news, their list of games, the standings, their squad, sponsors, and information about the club itself.

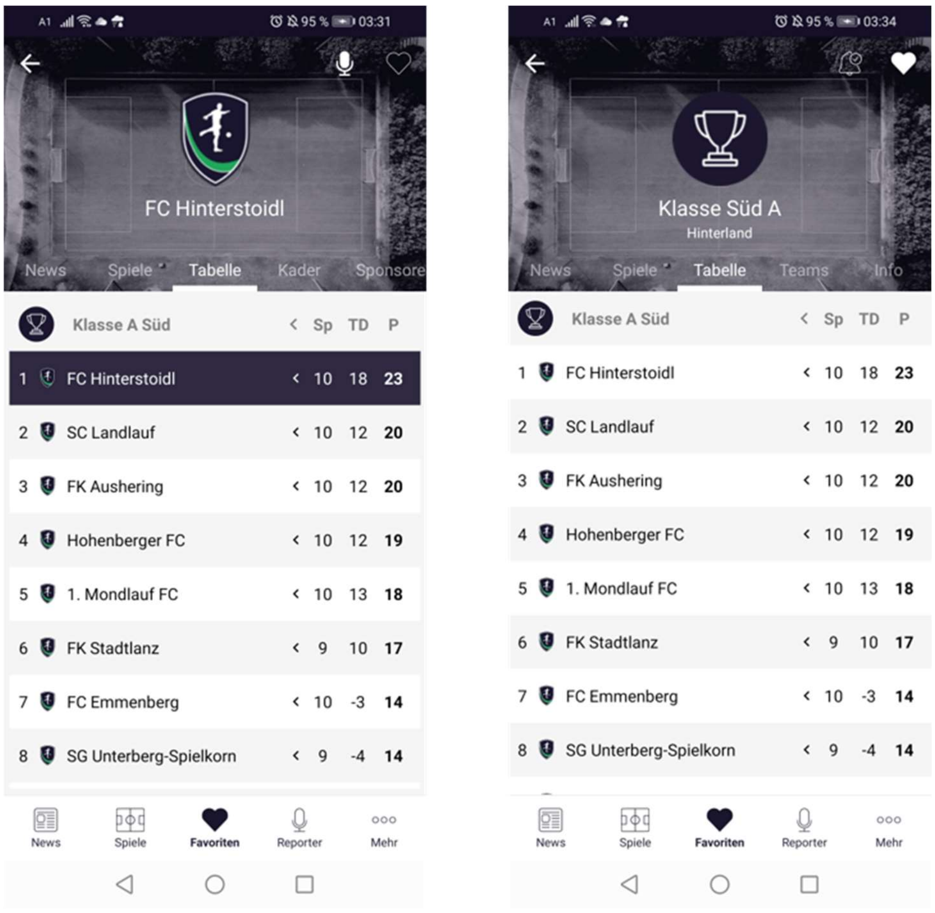
Generally, the profile pages did work but there were a lot of code smells within the legacy code. Not only redundant code but also hundreds of null pointers checks because of a not ordered creation of the different pages. Because of asynchronous calls to the backend, there were also some race conditions. Hence, it was very difficult to find the correct place to fix the bug. The developers did only look at the bugs registered by Firebase's Crashlytics[34] and added null pointer checks to the corresponding lines in the code. The code quality degraded that much, that even in code parts where it is not possible to have a null pointer for a specific variable a check was added. However, this approach was only used to combat the symptoms and not to combat the underlying problem. Henceforth, in the refactored code, null pointer checks are

[33] https://twitter.com/home [accessed on 27 November 2020]
[34] https://firebase.google.com/ [accessed on 30 November 2020]

only added in cases where it is necessary. In the next section, the situation before the refactoring sessions is shortly explained.

Status Quo

Not only the league page but also the club page work nearly as expected, except for some unwanted behaviour and bugs. However, as can be seen in Figure 16, the design is outdated. Moreover, as already mentioned above, the header for the profile pages is outdated as well. The requirements for the revision of the existing screens are explained in the next section.



	Klasse A Süd	<	Sp	TD	P
1	FC Hinterstoidl	<	10	18	23
2	SC Landlauf	<	10	12	20
3	FK Aushering	<	10	12	20
4	Hohenberger FC	<	10	12	19
5	1. Mondlauf FC	<	10	13	18
6	FK Stadtlanz	<	9	10	17
7	FC Emmenberg	<	10	-3	14
8	SG Unterberg-Spielkorn	<	9	-4	14

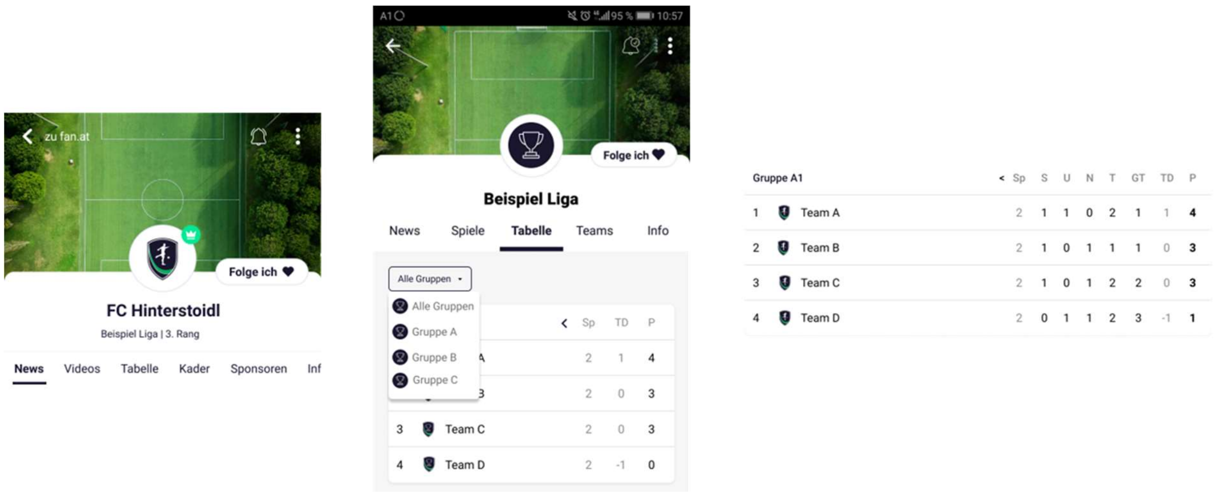
	Klasse A Süd	<	Sp	TD	P
1	FC Hinterstoidl	<	10	18	23
2	SC Landlauf	<	10	12	20
3	FK Aushering	<	10	12	20
4	Hohenberger FC	<	10	12	19
5	1. Mondlauf FC	<	10	13	18
6	FK Stadtlanz	<	9	10	17
7	FC Emmenberg	<	10	-3	14
8	SG Unterberg-Spielkorn	<	9	-4	14

Figure 16: The old-fashioned profile page of clubs and leagues.

Requirements

To begin with, during refactoring, not only the behaviour shall be preserved but afterwards also the bugs and null pointer checks shall be removed where possible. The header will be reworked completely. The background image is not required, the menu shall be adapted, and the header should be collapsible as described above. A design is not suggested and shall be found autonomously. In case the club has a registered reporter, a green crown shall indicate that fact as can be seen in the mock-up in Figure 17 on the left. Consequently, these clubs seem to be better supported. Regarding the back button the Android design shall be kept and “zu fan.at” is not needed. The tab “Videos” is not supported by the backend yet and can be ignored. However, thanks to the revision of the dynamic tab view, the insertion of another tab view is straightforward. The white follow button shall be coloured complementarily in the case of a missing subscription with the text “Folgen”. In the more menu, which is indicated by the three dots, a new design shall be found for the menu points. All other design elements and information like the rounded edges, the circle around the logo, or the place in the standings shall be implemented as exact as possible.

For the leagues, the standings shall provide the possibility to show groups and let them select individually by a dropdown. The selected value shall also include the logo of the league on the left side. The design of the standings shall be completely redesigned according to the mock-ups shown below. For the standings on the club profile page, the row of the club shall be coloured by “#D2ECDF”, which is a pale variant of the project green. The design for the news tab is explained below, the other graphical requirements are explained in the next section. All other tabs shall remain as they are for now.

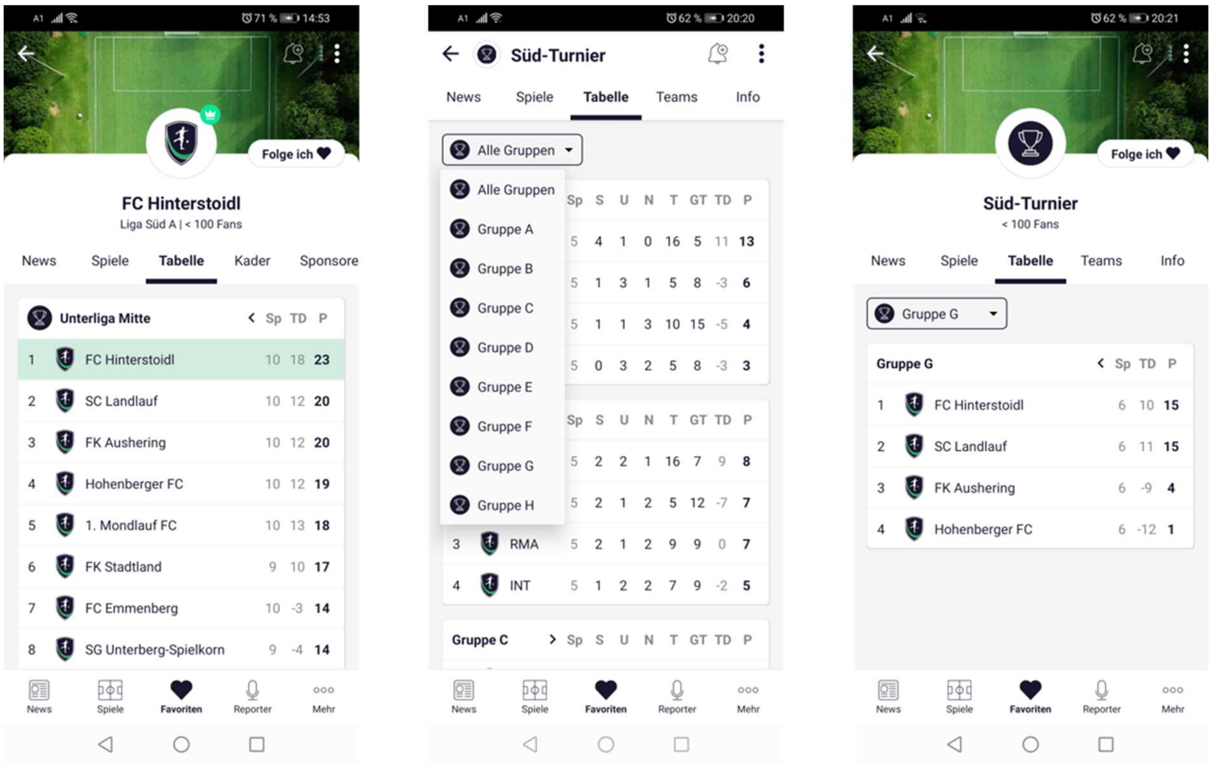


Gruppe A1		< Sp	S	U	N	T	GT	TD	P
1	Team A	2	1	1	0	2	1	1	4
2	Team B	2	1	0	1	1	1	0	3
3	Team C	2	1	0	1	2	2	0	3
4	Team D	2	0	1	1	2	3	-1	1

Figure 17: The mock-ups proposing the new header of the profile page and the new design for the standings. On the right side the expanded table for desktop views is shown, whereby the names shall be abbreviated in the mobile app.

Results

In Figure 18 the results are shown. In comparison to the presented mock-ups in Figure 17, the design is nearly the same. During refactoring and developing, though, we decided to show the followers instead of the rank in the club header. This information is also included within the league profile page. For the collapsed header we decided to make the background white, show the logo in the same grey bordered circle as in the open header, and show the full name of the entity, namely the league or the club. For providing the user the possibility to subscribe to the club or the team even if the header is collapsed, an outlined heart is added. As can be seen in Figure 18 below in the middle, we increased the padding for single entries within the dropdown as for broader fingers it is very difficult to click on the desired entry.



	Unterliga Mitte	Sp	TD	P
1	FC Hinterstoidl	10	18	23
2	SC Landlauf	10	12	20
3	FK Aushering	10	12	20
4	Hohenberger FC	10	12	19
5	1. Mondlauf FC	10	13	18
6	FK Stadtland	9	10	17
7	FC Emmenberg	10	-3	14
8	SG Unterberg-Spielkorn	9	-4	14

		Sp	S	U	N	T	GT	TD	P
		5	4	1	0	16	5	11	13
		5	1	3	1	5	8	-3	6
		5	1	1	3	10	15	-5	4
		5	0	3	2	5	8	-3	3

		Sp	S	U	N	T	GT	TD	P
		5	2	2	1	16	7	9	8
		5	2	1	2	5	12	-7	7
3	RMA	5	2	1	2	9	9	0	7
4	INT	5	1	2	2	7	9	-2	5

	Gruppe C	Sp	S	U	N	T	GT	TD	P

	Gruppe G	Sp	TD	P
1	FC Hinterstoidl	6	10	15
2	SC Landlauf	6	11	15
3	FK Aushering	6	-9	4
4	Hohenberger FC	6	-12	1

Figure 18: The finished profile pages of leagues and clubs.

As can be easily seen, the layouts of these two pages seem to be very similar. The only differences between them are the data objects of which the pages are filled, namely league or club objects, and the tab views. Nevertheless, they display very similar behaviour and design. Therefore, it was decided to use *Extract Superclass* refactoring method to unify the similarities of both profile pages together (Fowler, 2018). The new superclass for both fragments is called *EntityFragment*. Even if the name is not perfect yet, the team members of the project accepted the new name. After creating the new empty abstract superclass, which extends the above mentioned *TabFragment*, *LeagueFragment* and *TeamFragment* for the club extend the new superclass. For extracting similarities, *Pull Up Method*, *Pull Up Field* and *Extract Function* were used (Fowler, 2018).

In summary, the new *EntityFragment* prepared only 13 abstract methods that need to be implemented by the subclasses, like *getEntityLogoUrl* or *un-/followEntity*. The changes in complexity for entity related classes can be seen in the Appendix.

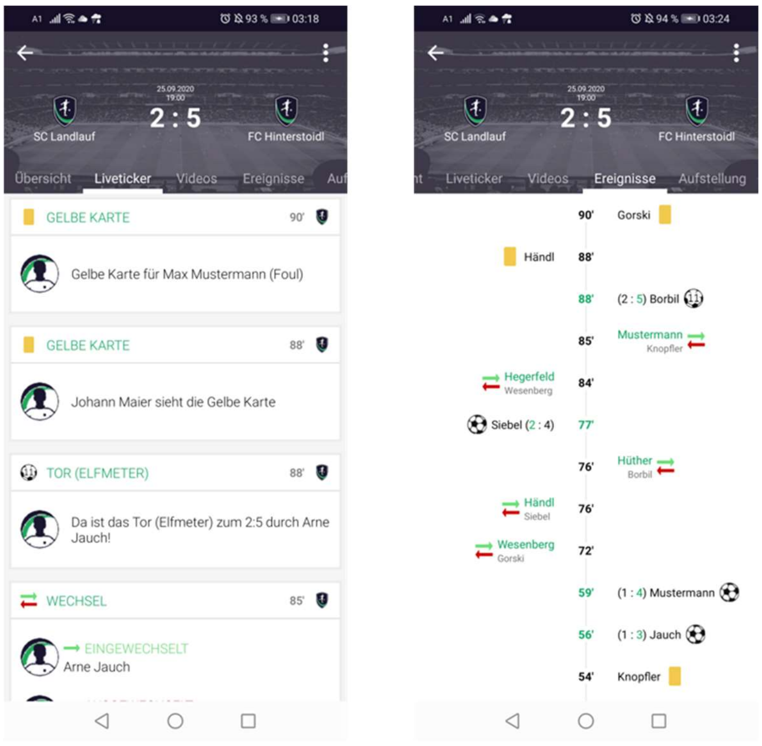
We combined the previously nearly ident but slightly different XML layout files for club and league to a single *fragment_entity.xml* file. By this combination, we could eliminate a whole file, and redundancy could be reduced enormously. Until that time, any single design change for one profile page had to be done exactly and without errors in the other profile page layout, also. This can be a source for an inconsistency, which is hard to find as already described above (Hunt & Thomas, 2000).

New Game Page

One of the most important and complex screens is the event page. Here, the most dynamics takes place. Not only web sockets are connected to inform about goals and time or event phases, but also the tabs get displayed according to specific settings. For the user, this page is very important as it is the core of the whole application – it informs the users about their favourite clubs at real-time. Because of this, the requirements often changed in past and the code got very complex and buggy. Therefore, the refactoring reduced its complexity, which can be seen in the Appendix.

Status Quo

As mentioned above, it was previously not possible to collapse the header. Thus, much space is wasted, and the behaviour is quite old-fashioned for modern applications. The header itself is not consistent with the modern design of the whole application. Moreover, the different tabs only show content in case a game has started, or it starts within the next few hours. As a result, there are only placeholders shown that say that no content is available yet. The live licker and the timeline screen need an update. Not only videos do not work but also the icons or the whole view holders are not appealing as can be seen in Figure 19. The new design is explained in the next section.


A1 93 % 03:18

25.09.2020 19:00

SC Landlauf 2 : 5 FC Hinterstoidl

Übersicht Liveticker Videos Ereignisse Auf

GELBE KARTE 90'

Gelbe Karte für Max Mustermann (Foul)

GELBE KARTE 88'

Johann Maier sieht die Gelbe Karte

TOR (ELFMETER) 88'

Da ist das Tor (Elfmeter) zum 2:5 durch Arne Jauch!

WECHSEL 85'

EINGEWECHSELT
Arne Jauch

A1 94 % 03:24

25.09.2020 19:00

SC Landlauf 2 : 5 FC Hinterstoidl

Liveticker Videos Ereignisse Aufstellung

	90'	Gorski
Händl	88'	
	88'	(2 : 5) Borbil
	85'	Mustermann Knopfler
Hegerfeld Wesenberg	84'	
Siebel (2 : 4)	77'	
	76'	Hüther Borbil
Händl Siebel	76'	
Wesenberg Gorski	72'	
	59'	(1 : 4) Mustermann
	56'	(1 : 3) Jauch
	54'	Knopfler

Figure 19: The current screens of the live ticker and the timeline screen for an event. The design should be adapted to a new design.

Requirements

As already shown, the new design of the header is quite simple, without a background image and can be collapsed on need as discussed for Figure 13. The tab views shall be added only in case content is available, as also described in 3.2.1. For the live ticker view, the icons shall be updated and the whole design should follow the design rules that are shown in Figure 20. The icons are newly designed and shall be displayed larger, like the logo for the clubs. The minute in which the action took place shall be centred and more apparent to the user. Goals shall be highlighted in a green colour. This should work not only for simple goals but also for penalties or freekicks. Furthermore, when clicking on real photos of soccer players, the photo should be shown in full screen. In case a video post is sent to the frontend, it shall be shown and played in full-screen mode. For the timeline screen the same logic regarding the colours should be used as for the live ticker, that means that only goals should be highlighted, and the design shall be adapted as indicated in the mock-up. The final design that was accepted can be seen in the next section.

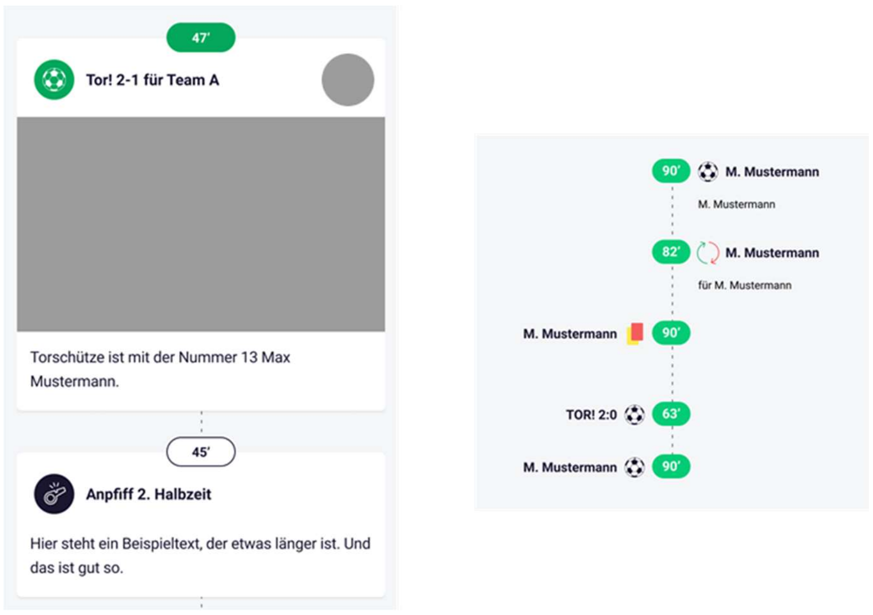


Figure 20: The new design rules that the live ticker and the timeline tab should follow.

Results

The refactoring took quite long but as can be seen in the Evaluation, it was worth doing it. Not only the complexity was reduced, but also the consistency was improved by a lot. Historically, a mix of different names has formed for different tab views. The container tab for handling the dynamic tabs and the collapsible header was named *GameFragment*, the "Übersicht" tab was named *GameOverviewFragment*, the "Liveticker" tab was named *TimelineFragment*, the "Ereignisse" tab was named *OverviewFragment* and the "Aufstellung" tab was named *LineupGameFragment*. To reach consistency, we refactored not only all the class and layout files, but also the corresponding constants or variables accordingly. For this purpose, we used the built-in functionality of Android Studio to refactor all the names safely as described in 2.1.6.

One important change was moving the responsibility of handling the web sockets to the right place. Within the legacy code, the live ticker fragment was responsible for connecting to the web sockets and handling the messages. Thus, in case the live ticker fragment is not loaded before the game due to no content, the header could not go live on kick-off. Moreover, the live ticker had to inform the overview fragment because of different changes within the view. All this functionality was centred to the container fragment, that is the *GameFragment*, and by the middleman *GamePagerAdapter*, which handles the different tab fragments, the messages for new posts or the kick-off got forwarded to the *LivetickerFragment* and the *OverviewFragment*. So, it is much clearer now and bugs are easier to find.

One example for very complex code is shown in Listing 11. Within this too long method it is difficult to find out, which club should be automatically selected in which situation.

```
private String getDefaultTeamChoice() {
    boolean followsHomeTeam =
            TeamSubscriptions.getInstance().containsKey(event.getHomeTeamId());
    boolean followsAwayTeam =
            TeamSubscriptions.getInstance().containsKey(event.getAwayTeamId());
    boolean followsBothTeams = followsHomeTeam && followsAwayTeam;
    boolean followsNone = !followsHomeTeam && !followsAwayTeam;
    boolean followsBothOrNone = followsBothTeams || followsNone;
    boolean followsSingleTeam = !followsBothOrNone;
    boolean isHomeTeamreporter =
            TeamSubscriptions.getInstance().isTeamReporterOf(
                    event.getHomeTeamId());
    boolean isAwayTeamreporter =
            TeamSubscriptions.getInstance().isTeamReporterOf(
                    event.getAwayTeamId());
    boolean isTeamReporterOfSingleTeam =
            (isHomeTeamreporter && !isAwayTeamreporter) ||
                    (!isHomeTeamreporter && isAwayTeamreporter);
    boolean isTeamReporterForBothTeams = isHomeTeamreporter && isAwayTeamreporter;
    Event.EventFanModStatus homeStatus = event.getFanModStatus(
            event.getHomeFanModData());
    Event.EventFanModStatus awayStatus = event.getFanModStatus(
            event.getAwayFanModData());
    boolean isHomeReporter = homeStatus ==
            Event.EventFanModStatus.STATUS_CONFIRMED ||
            homeStatus == Event.EventFanModStatus.STATUS_SELECTED ||
            homeStatus == Event.EventFanModStatus.STATUS_REGISTERED ||
            event.isHomeTeamReporter();
    boolean isAwayReporter = awayStatus ==
            Event.EventFanModStatus.STATUS_CONFIRMED ||
            awayStatus == Event.EventFanModStatus.STATUS_SELECTED ||
            awayStatus == Event.EventFanModStatus.STATUS_REGISTERED ||
            event.isAwayTeamReporter();
    boolean hasHomeReporter =
            (event.getHomeFanModData().getFanModId() != null &&
                    event.getHomeFanModData().getFanModId().length() > 0) ||
                    event.getHomeModeratorNames().size() > 0;
    boolean hasAwayReporter =
            (event.getAwayFanModData().getFanModId() != null &&
            event.getAwayFanModData().getFanModId().length() > 0) ||
            event.getAwayModeratorNames().size() > 0;
    boolean bothTeamsHaveReporters = hasHomeReporter && hasAwayReporter;
    boolean noTeamHasReporters = !hasHomeReporter && !hasAwayReporter;

    // past & running
    if (event.isPast() || event.isRunning()) {
        if (followsSingleTeam || isTeamReporterOfSingleTeam) {
            if (isTeamReporterOfSingleTeam) {
                return isHomeTeamreporter ? event.getHomeTeamId() :
                        event.getAwayTeamId();
            } else {
                return followsHomeTeam ? event.getHomeTeamId() :
                        event.getAwayTeamId();
            }
        } else if (isHomeReporter || isAwayReporter) {
            return isHomeReporter ? event.getHomeTeamId() : event.getAwayTeamId();
        } else if (followsBothOrNone || isTeamReporterForBothTeams) {
            if (bothTeamsHaveReporters || noTeamHasReporters) {
                return event.getHomeTeamId();
            } else {
                return hasHomeReporter ? event.getHomeTeamId() :
                        event.getAwayTeamId();
            }
        }
    }
    // future
    else {
        if (followsSingleTeam || isTeamReporterOfSingleTeam) {
            if (isTeamReporterOfSingleTeam) {
                return isHomeTeamreporter ? event.getHomeTeamId() :
                        event.getAwayTeamId();
            } else {
                return followsHomeTeam ? event.getHomeTeamId() :
                        event.getAwayTeamId();
            }
        } else if (isHomeReporter || isAwayReporter) {
            return isHomeReporter ? event.getHomeTeamId() :
                    event.getAwayTeamId();
        } else if (followsBothTeams || isTeamReporterForBothTeams) {
            return null;
        } else if (followsNone) {
            return event.getHomeTeamId();
        }
    }

    return null;
}
```

Listing 11: An example for a very complex method in which it is very hard to understand its outcome. © by SFA Sport GmbH

To understand the outcome, comprehension refactoring is the right way. The outcome by using well-known refactoring practices like *Extract Method* or *Inline Variable* is shown in Listing 12 (Fowler, 2018). When trying to understand the different outcomes, someone only needs to read the functions and understands what is going on. Of course, the user has to navigate to sub-methods in case to clarify the purpose for a specific return of a function, but by well-named methods, this is not necessary in most cases. Moreover, as can be seen above in Listing 11, the outcome depends always on the event. Therefore, *Move Method* to the class for representing an event was used to increase the modularity (Fowler, 2018). It is to say that the method is still a little bit too long, but as Robert C. Martin (2008) suggests, each block within this method is only one line long.

```
public String getDefaultTeamChoice() {
    if (isTeamReporterOfSingleTeam()) {
        return isHomeTeamReporter() ? getHomeTeamId() : getAwayTeamId();
    }
    if (followsSingleTeam()) {
        return followsHomeTeam() ? getHomeTeamId() : getAwayTeamId();
    }
    if (isHomeReporter()) {
        return getHomeTeamId();
    }
    if (isAwayReporter()) {
        return getAwayTeamId();
    }
    if (isPast() || isRunning()) {
        if (bothTeamsHaveReporters() || noTeamHasReporters() ||
                hasHomeReporter()) {
            return getHomeTeamId();
        }
        return getAwayTeamId();
    }
    else if (followsBothTeams() || isTeamReporterForBothTeams()) {
        return null;
    }
    return getHomeTeamId();
}
```

Listing 12: The refactored choice of a club depending on different settings. It is by far easier to understand which team club is chosen in which situation.

One problem that we needed to understand is when which minute or date is set, and when which score is shown. Listing 13 summarizes the old solution. As can be seen, the settings are dependent on the different phases of the event. Not only in the well-named function *setHeader-ByPhase* but also within *setHeaderScoreTime* it can be seen that the score is dependent on the set phase. The method call *isRunning* just summarizes many phases and thus, it is also dependent on the phases.

```
private void setHeaderScoreTime(Event event) {
    int homeScore = event.getHomeScore();
    int awayScore = event.getAwayScore();

    if (homeScore == -1 || awayScore == -1) {
        if (event.isRunning()) {
            scores.setVisibility(View.INVISIBLE);
        } else {
            scores.setText("- : -");
        }
    } else if (event.getPhase() == Phase.HT_RUNNING) {
        if (event.getEventScores() != null &&
                event.getEventScores().getFirstHalfEnd() != null &&
                event.getEventScores().getFirstHalfEnd().HOME != -1 &&
                event.getEventScores().getFirstHalfEnd().AWAY != -1) {
            String scoreTxt = event.getEventScores().getFirstHalfEnd().HOME +
                    " : " + event.getEventScores().getFirstHalfEnd().AWAY;
            scores.setText(scoreTxt);
        } else {
            String scoreTxt = homeScore + " : " + awayScore;
            scores.setText(scoreTxt);
        }
    } else {
        String scoreTxt = homeScore + " : " + awayScore;
        scores.setText(scoreTxt);
    }

    time.setTextSize(TypedValue.COMPLEX_UNIT_SP, 12);
    setHeaderByPhase(event.getPhase());
}

private void setHeaderByPhase(Phase phase) {
    switch (phase) {
        case UPCOMING:
        case PREGAME:
        case ENDED:
            setStartDate();
            break;
        case ABORTED:
            setStartDate();
            scores.setTextSize(TypedValue.COMPLEX_UNIT_SP, 24);
            scores.setText(getString(R.string.game_aborted));
```

```java
            break;
        case CANCELLED:
            setStartDate();
            scores.setTextSize(TypedValue.COMPLEX_UNIT_SP, 24);
            scores.setText(getString(R.string.game_cancelled));
            break;
        case POSTGAME:
            time.setText(getString(R.string.postgame));
            break;
        case FIRST_BREAK:
        case HT_RUNNING:
            time.setText(getString(R.string.first_break));
            break;
        case SECOND_BREAK:
            time.setText(getString(R.string.second_break));
            break;
        case THIRD_BREAK:
            time.setText(getString(R.string.third_break));
            break;
        case PENALTY:
            time.setText(getString(R.string.penalty));
            break;
    }
}
```

Listing 13: This method sets the score and time within the header but is too complicated to understand its functionality. Moreover, if phases change or need to be added, this is very time-consuming. © by SFA Sport GmbH

The goal is now to understand for which phase which score, and time is set. In a broader sense, the conditional setting of the score depending on the phase is like the other function shown in the above listing a switch-case of the current phase. Therefore, we have two nearly identical switch-cases and should handle it correctly as discussed above.

As suggested by Gamma et al. (1994), we create an abstract class factory for each phase. The implementations of the new abstract class *GamePhase* need to implement their *setTimeText* and *setScoreText* accordingly. With this introduction of an abstract class factory, it is by far easier to understand when which one's score, and time is set, and it is easier to add or adapt a phase.

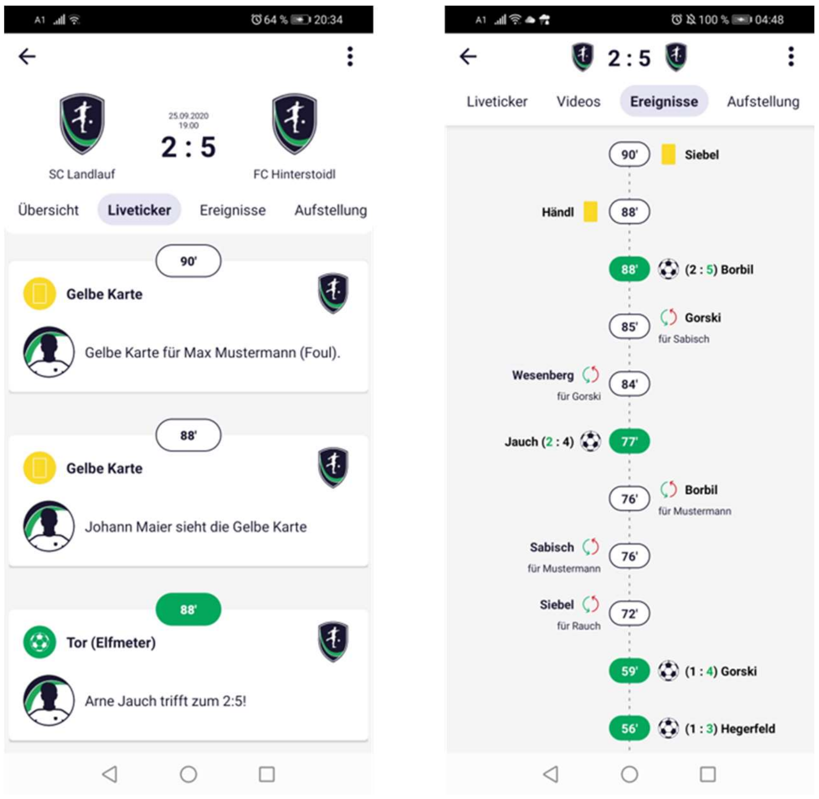
```java
private void setHeaderScoreTime() {
    GamePhase gamePhase = makeGamePhase(event.getPhase(), null);
    gamePhase.setScoreText();
    gamePhase.setTimeText();
}

@Override
public GamePhase makeGamePhase(Phase phase, String minute) {
    switch (phase) {
        case UPCOMING:
        case PREGAME_START:
        case PREGAME:
        case ENDED:
            return new GamePhaseNotCurrent(...);
        case ABORTED:
            return new GamePhaseAborted(...);
        case CANCELLED:
            return new GamePhaseCancelled(...);
        case FIRST_BREAK:
```

```
        case HT_RUNNING:
            return new GamePhaseFirstBreak(...);
        case SECOND_BREAK:
            return new GamePhaseSecondBreak(...);
        case THIRD_BREAK:
            return new GamePhaseThirdBreak(...);
        case PENALTY:
            return new GamePhasePenalty(...);
        case POSTGAME:
            return new GamePhasePostGame(...);
        case RUNNING:
        case FIRST_HALF:
        case SECOND_HALF:
        case FIRST_OVERTIME:
        case SECOND_OVERTIME:
            return new GamePhaseRunning(...);
    }
    return new GamePhaseNotCurrent(...);
}

interface GamePhaseFactory {
    GamePhase makeGamePhase(Phase phase, String minute);
}

abstract class GamePhase {
    ...

    public abstract void setTimeText();
    public abstract void setScoreText();
    ...
}
```

Listing 14: According to the phase a new object is created which implements its setting of time and score accordingly.

Finally, the new design of the header, the live ticker and the timeline fragment are shown in Figure 21. By the new collapsible header and the new design decisions, the content can be shown much clearer to the user and is much more user-friendly than before. Moreover, we added new functionality like the possibility of showing videos or viewing the photos of players in the full screen by doing preparatory refactoring beforehand.

Another important revision is that view of the list of games shown in the next section.



90'
Gelbe Karte
Gelbe Karte für Max Mustermann (Foul).

88'
Gelbe Karte
Johann Maier sieht die Gelbe Karte

88'
Tor (Elfmeter)
Arne Jauch trifft zum 2:5!

90' Siebel
Händl 88'
88' (2 : 5) Borbil
85' Gorski für Sabisch
Wesenberg für Gorski 84'
Jauch (2 : 4) 77'
76' Borbil für Mustermann
Sabisch für Mustermann 76'
Siebel für Rauch 72'
59' (1 : 4) Gorski
56' (1 : 3) Hegerfeld

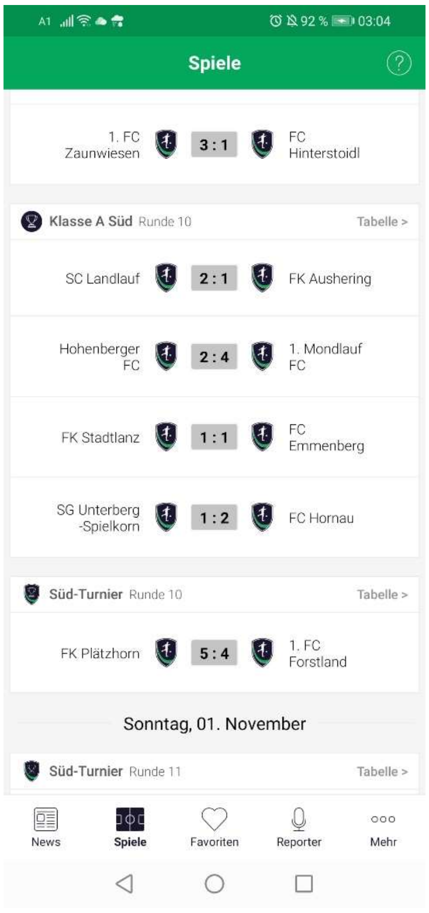
Figure 21: The new design of the fragment showing an event. Not only the design is aligned to the new standard of the application, but also the data is displayed by far clearer and more user-friendly.

New Games Page

The application does not support only leagues and clubs where men play but also supports youth clubs and clubs where women play. Therefore, it is very important to get the attention of the user for these games. Thus, the idea is to add another screen to the games view, namely a separate list for all events that can be filtered accordingly.

Status Quo

Currently, only games of subscribed leagues and teams are shown within the games tab as shown in Figure 22. As a result, if a user wants to see all the games, which are played on a specific date or even today, it is not possible yet. The only solution is to go through all the leagues or clubs. This is unfeasible. Therefore, a possible way to provide this is presented in the next section.


A1 03:04 92 %

Spiele

1. FC Zaunwiesen 3 : 1 FC Hinterstoidl

Klasse A Süd Runde 10 Tabelle >

SC Landlauf 2 : 1 FK Aushering

Hohenberger FC 2 : 4 1. Mondlauf FC

FK Stadtlanz 1 : 1 FC Emmenberg

SG Unterberg -Spielkorn 1 : 2 FC Hornau

Süd-Turnier Runde 10 Tabelle >

FK Plätzhorn 5 : 4 1. FC Forstland

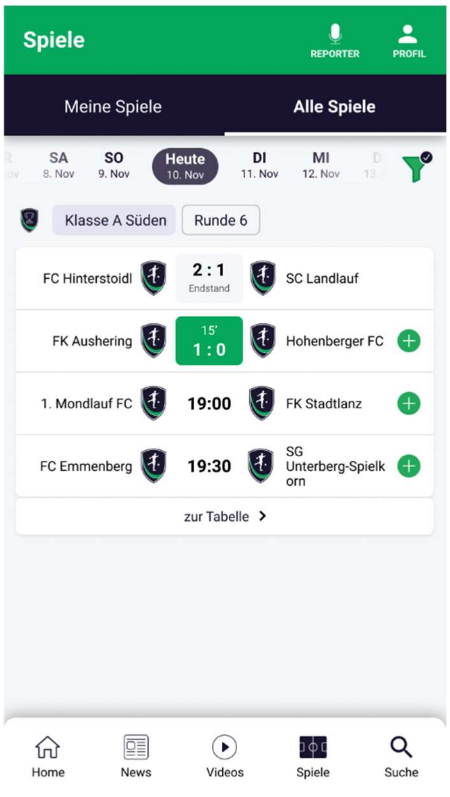
Sonntag, 01. November

Süd-Turnier Runde 11 Tabelle >

Figure 22: In the games tab only events of the subscribed clubs and leagues are shown.

Requirements

The games view shall be changed to a tabbed view, whereby the existing view shall be titled "Meine Spiele". A second tab shall be added with the title "Alle Spiele". Within this new *FragmentTab* a horizontally scrollable date picker shall be implemented which is initially set to "Heute" which means that only events get displayed that are played on that specific day. To the right six more days shall be displayed, to the left only three past dates shall be supported. By clicking on a new date, the filter is set accordingly, the games are refreshed, and the horizontal date picker is updated. By clicking on the filter symbol visible in Figure 23, the user has the possibility to filter for international leagues or for leagues of the Austrian states. Moreover, it shall be possible to filter for men, women, or youth leagues. Additionally, the filter should include a selection of Austrian states. Per default, all leagues are displayed. If a filter is set, the filter symbol is changed to that visible in the screen below. Within the filter screen, the users should be able to search for clubs and leagues to give them the possibility subscribing to the entities.



Klasse A Süden | Runde 6

FC Hinterstoidl 2 : 1 Endstand SC Landlauf
FK Aushering 15' 1 : 0 Hohenberger FC
1. Mondlauf FC 19:00 FK Stadtlanz
FC Emmenberg 19:30 SG Unterberg-Spielkorn

zur Tabelle

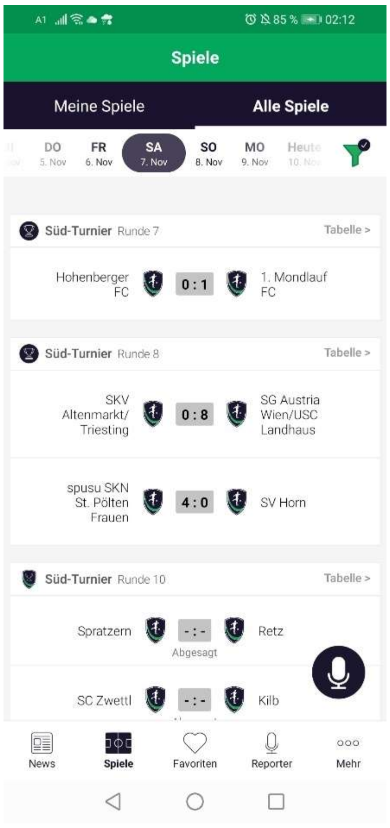
Figure 23: In the new tab that shows all games, the user can filter the results.

Results

If comparing Figure 24 with Figure 23, it can be seen that we implemented the design in great detail. The most difficult part was implementing the horizontal date picker. As there is not available any library that supports the needed requirements, we implemented it completely new. A horizontal scroll view without scrollbar holds items which show their given date and implement a click listener for opening the filter mask. The horizontal scroll view is covered by a linear layout which holds three equally weighted views. The inner view is transparent, the outer ones consist of gradient backgrounds that start on the inner side with transparent colour and end on the outer side with white.

The functionality for searching was already implemented but needed a great adoption as it was very complex. For supporting the new filter "all", we refactored the existing code base first. Furthermore, as the views exist for adding clubs and leagues, the view holder needed an update for setting the filter instead of clicking through the navigation for clubs and leagues.

To summarize, by smart refactoring the new functionality could be added quite easily while improving the code base for later adoptions.


Spiele

Meine Spiele | Alle Spiele

DO 5. Nov | FR 6. Nov | SA 7. Nov | SO 8. Nov | MO 9. Nov | Heute 10. Nov

Süd-Turnier Runde 7 Tabelle >

Hohenberger FC 0 : 1 1. Mondlauf FC

Süd-Turnier Runde 8 Tabelle >

SKV Altenmarkt/ Triesting 0 : 8 SG Austria Wien/USC Landhaus

spusu SKN St. Pölten Frauen 4 : 0 SV Horn

Süd-Turnier Runde 10 Tabelle >

Spratzern - : - Retz
Abgesagt

SC Zwettl - : - Kilb

News | Spiele | Favoriten | Reporter | Mehr

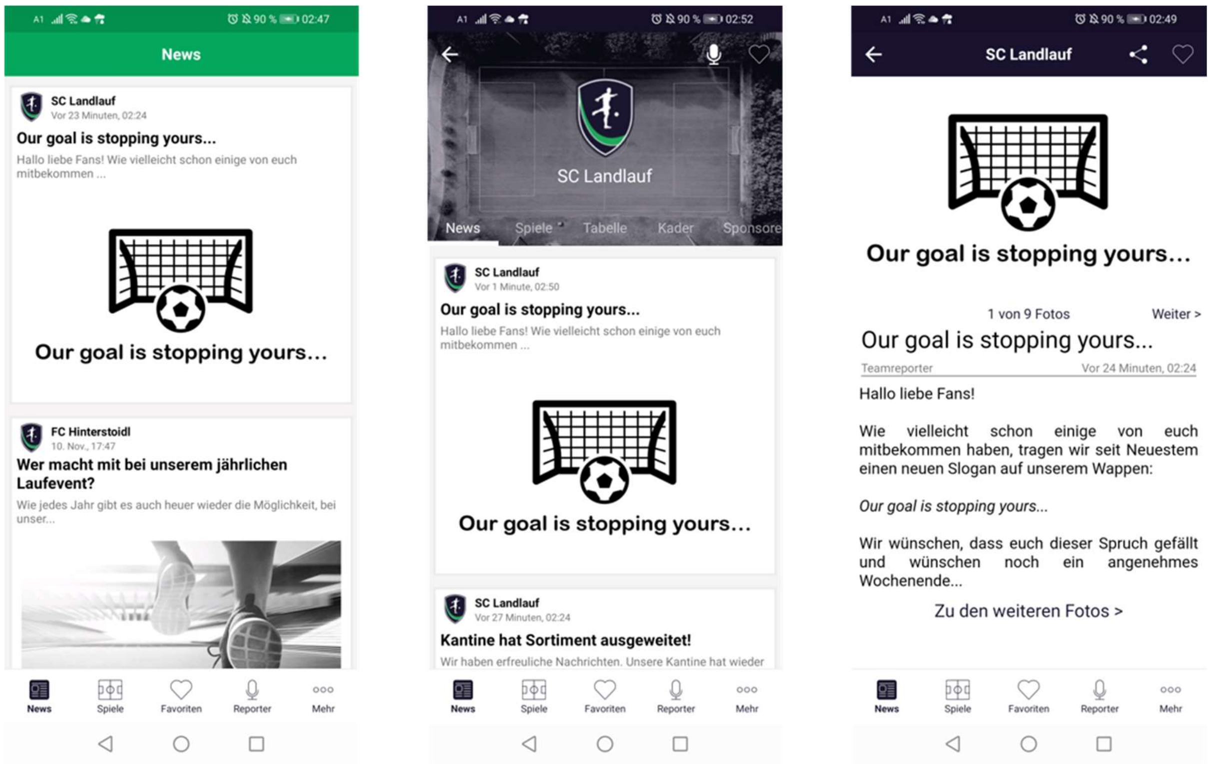
Figure 24: The users have now the possibility to filter games. In the example shown an filter was already set.

New News

The application is not only presenting events or profile pages of clubs and leagues but also supports news. This feature is very important for the platform as the user can be informed of new news by push notifications that lead on click to the application. Therefore, it is very important to make it possible to also show news of other teams and leagues and also of other reporters like those of famous newspapers or other platforms.

Status Quo

Currently, the news overview consists only of news of subscribed clubs and leagues. Therefore, users only see content here in case they are subscribed to clubs or leagues. This approach is not very user-friendly as they should also see content in the beginning, to get informed about clubs and leagues and to see the possibilities the application provides. It is the same problem as already described for the games in the previous section. The user has no possibility to see all news of all clubs and leagues but would have to go through all entities singularly. Moreover, the application does not support other types of news and does not show much meta information. Additionally, it does not provide the possibility to show videos.


News

SC Landlauf
Vor 23 Minuten, 02:24
Our goal is stopping yours...
Hallo liebe Fans! Wie vielleicht schon einige von euch mitbekommen ...

Our goal is stopping yours...

FC Hinterstoidl
10. Nov., 17:47
Wer macht mit bei unserem jährlichen Laufevent?
Wie jedes Jahr gibt es auch heuer wieder die Möglichkeit, bei unser...

SC Landlauf

News Spiele Tabelle Kader Sponsore

SC Landlauf
Vor 1 Minute, 02:50
Our goal is stopping yours...
Hallo liebe Fans! Wie vielleicht schon einige von euch mitbekommen ...

Our goal is stopping yours...

SC Landlauf
Vor 27 Minuten, 02:24
Kantine hat Sortiment ausgeweitet!
Wir haben erfreuliche Nachrichten. Unsere Kantine hat wieder

SC Landlauf

Our goal is stopping yours...

1 von 9 Fotos Weiter >
Our goal is stopping yours...
Teamreporter Vor 24 Minuten, 02:24
Hallo liebe Fans!

Wie vielleicht schon einige von euch mitbekommen haben, tragen wir seit Neuestem einen neuen Slogan auf unserem Wappen:

Our goal is stopping yours...

Wir wünschen, dass euch dieser Spruch gefällt und wünschen noch ein angenehmes Wochenende...

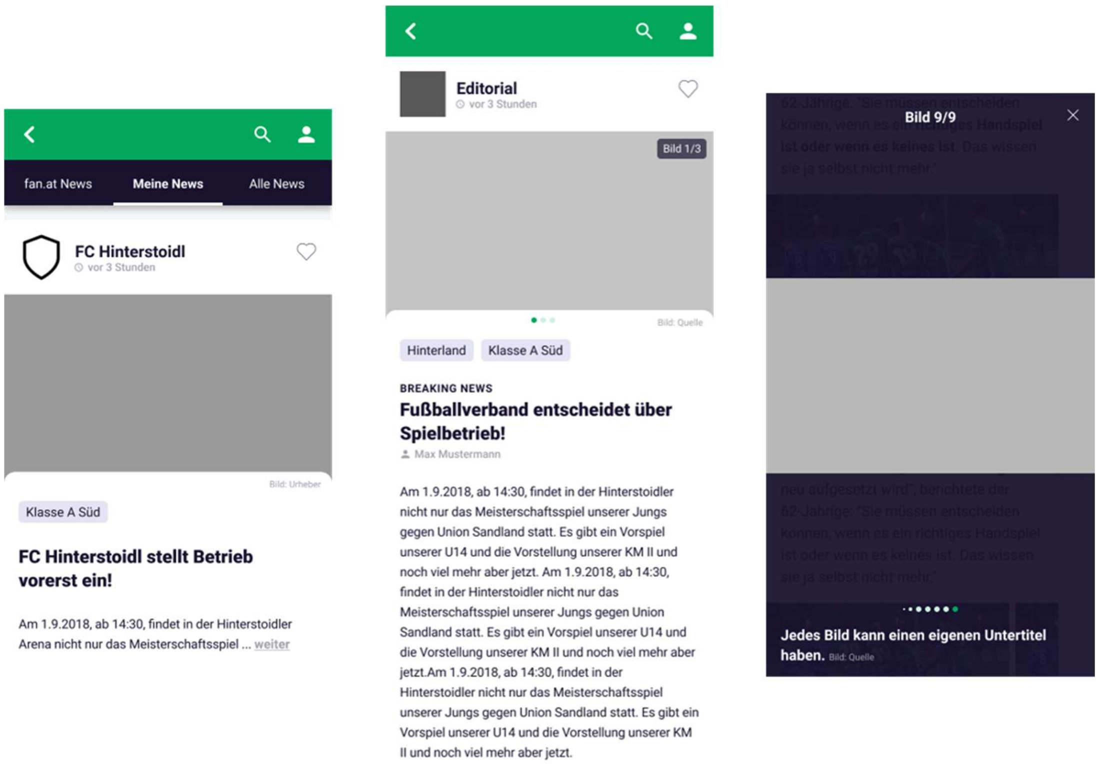
Zu den weiteren Fotos >

News Spiele Favoriten Reporter Mehr

Figure 25: The news view needs an update to fit to the new design. The pictures do not even have position indicators as commonly used nowadays.

Requirements

The new news is a feature that is very important for the success of the application. Not only providing new meta information like a subtitle or the shortened content but also photo indicators, tags to league or clubs or the possibility to show videos are requirements to building into the application. The design should be adapted as exactly as possible as presented in Figure 26. Besides these general requirements, new types of news shall be provided. The backend is able to send news of reporters of newspapers or other platforms. This means, that it is possible to receive news that is not created by reporters of leagues or clubs but is written by other reporters that are called internally "editorials". Moreover, the application itself can send news. These new variants of news should be shown in new tabs within the news view. In the middle tab news of subscribed leagues or clubs shall be shown. Additionally, there shall be another tab for showing all available news, and a third tab for showing editorial news and news created by the platform itself. Regarding the photos it should be possible to swipe between the photos not only in the full screen but also outside in the overview screen. Another feature is that there are social media embeddings like a posting from other platforms directly within the news article view.



FC Hinterstoidl stellt Betrieb vorerst ein!

Am 1.9.2018, ab 14:30, findet in der Hinterstoidler Arena nicht nur das Meisterschaftsspiel ... weiter

Fußballverband entscheidet über Spielbetrieb!

Am 1.9.2018, ab 14:30, findet in der Hinterstoidler nicht nur das Meisterschaftsspiel unserer Jungs gegen Union Sandland statt. Es gibt ein Vorspiel unserer U14 und die Vorstellung unserer KM II und noch viel mehr aber jetzt. Am 1.9.2018, ab 14:30, findet in der Hinterstoidler nicht nur das Meisterschaftsspiel unserer Jungs gegen Union Sandland statt. Es gibt ein Vorspiel unserer U14 und die Vorstellung unserer KM II und noch viel mehr aber jetzt.Am 1.9.2018, ab 14:30, findet in der Hinterstoidler nicht nur das Meisterschaftsspiel unserer Jungs gegen Union Sandland statt. Es gibt ein Vorspiel unserer U14 und die Vorstellung unserer KM II und noch viel mehr aber jetzt.

Jedes Bild kann einen eigenen Untertitel haben.

Figure 26: These mock-ups show the new views for any news. Not only the overview is changed but also the opened news article should be adapted accordingly and support new features.

Results

As shown in Figure 27, the design is implemented very similarly according to the mock-ups. For receiving new news, we needed to update some API routes, a PagerAdapter for the news needed to be created which holds the three *FragmentTabs*. The loading and updating of the news are implemented in an abstract superclass.

For the picture position indicators, an open-source library was used[35]. This library makes it possible to pass by the colours, the size of the dots and the amount at which the dots become smaller. In our case, this was set to five. As a result, five or fewer dots are of the same size. In case there are more pictures, there are differences within the sizes as can be seen below.

For showing a video a simple WebView[36] was used to load the given Uniform Resource Locator (URL)[37]. A problem occurred when more news articles loaded videos. In this case, the user could play more than one video at a time and it did not stop correctly. Therefore, a

[35] https://github.com/hrskrs/InstaDotView [accessed on 25 November 2020]
[36] https://developer.android.com/reference/android/webkit/WebView [accessed on 25 November 2020]
[37] https://tools.ietf.org/html/rfc1738 [accessed on 25 November 2020]

WebChromeClient[38] was extended to override the method *onHideCustomView*. As a result, the video stops and does not collide with other videos anymore.

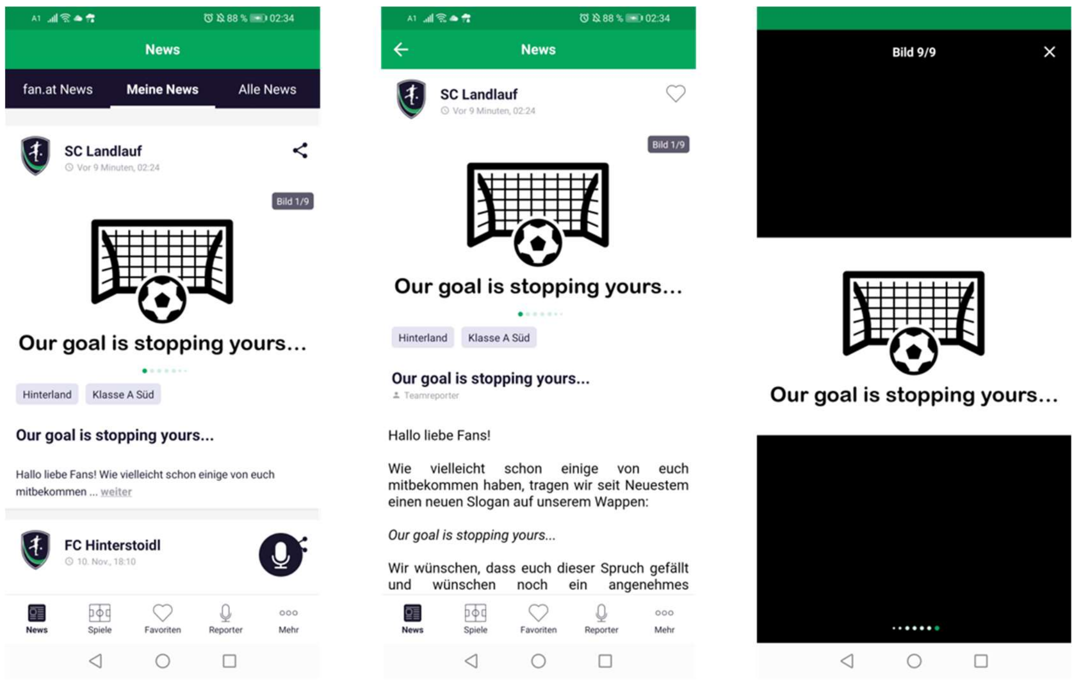
We also implemented the feature regarding showing external content like a posting from *Twitter*[39] or *Facebook*[40]. To show the code for displaying external content correctly within the application we used the settings shown in Listing 15.

```
webView.getSettings().setJavaScriptEnabled(true);
webView.getSettings().setRenderPriority(WebSettings.RenderPriority.HIGH);
webView.getSettings().setBuiltInZoomControls(false);
webView.getSettings().setLoadWithOverviewMode(true);
webView.getSettings().setLoadsImagesAutomatically(true);
webView.loadDataWithBaseURL(null, data, "text/html; charset=utf-8","utf-8", null);
```

Listing 15: For displaying external content correctly within the news article, the settings needed to be configured correctly and the data needed to be loaded with setting the encoding correctly.

For leading users to linked clubs or leagues the tags help. People are used to click on named entities to view more about that topic, and as there is some named entity shown in the tag, it should link to the corresponding endpoint (W3C, 2020).

All in all, the new news views are very important for gaining interest to the platform and application. As there is very much content available, the new news helps making the app more attractive to the user as displayed in Figure 27.



Our goal is stopping yours...

Hallo liebe Fans! Wie vielleicht schon einige von euch mitbekommen ... weiter

Hallo liebe Fans!

Wie vielleicht schon einige von euch mitbekommen haben, tragen wir seit Neuestem einen neuen Slogan auf unserem Wappen:

Our goal is stopping yours...

Wir wünschen, dass euch dieser Spruch gefällt und wünschen noch ein angenehmes

Figure 27: The new news views fit to the new design very well and offer much more features.

[38] https://developer.android.com/reference/android/webkit/WebChromeClient [accessed on 25 November 2020]
[39] https://about.twitter.com/ [accessed on 27 November 2020]
[40] https://www.facebook.com/pg/facebook/about/ [accessed on 27 November 2020]

3.3 Removing unused Code and Resources in Android Studio

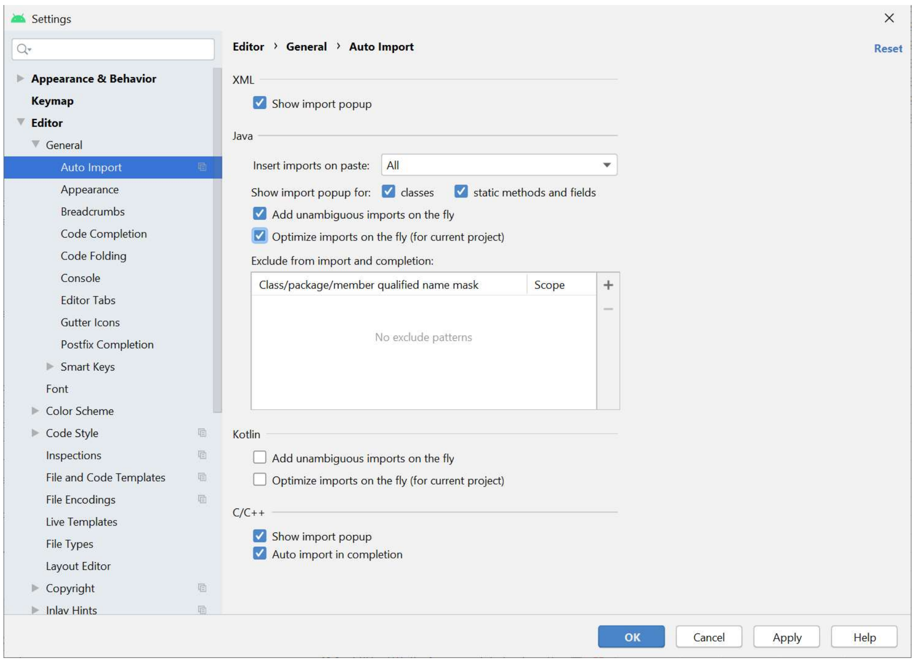
According to Romano, Vendome, Scanniello, and Poshyvanyk (2020) is the purpose of removing dead code to enhance software quality, as dead code makes the code harder to modify and to understand. They say further, that performing issues are less relevant, though. However, modern compilers remove this bad smell during the optimization phase for code compaction to reduce the size of code to compile as its removal does not influence the execution behaviour (Debray, Evans, Muth, & De Sutter, 2000). Nevertheless, the source code stays as it is without removing these pieces of dead code.

For resource identifiers, Android Studio follows a specific methodology. The Android Asset Packaging Toolt (AAPT) is used to compile and package the existing identifiers, and the outcome of this process is optimized for the usage in Android (Google LLC, 2020). The following explanation and usage description of assigning an identifier to a layout resource in Android is extracted from the documentation by Google LLC (2019). The *android:id*[41] attribute needs to be defined by using the syntax "+id/< name>" where "<name>" can be an arbitrary name, whereby it should follow the defined naming conventions for resource identifiers explained in 3.1 for this application. The AAPT tool then creates an integer corresponding to this alias. In case this identifier exists already, it will be the same integer as the existing one as the documentation describes (Google LLC, 2019).

3.3.1 Applying Removal of unused Code and Resources

We cleaned up the source files in five stages. Firstly, the source code within the "src" folder was simply reformatted by the command "Reformat Code". By this, the code gets reformatted in a way that it looks clean, which means that unnecessary spaces or blank lines get removed without changing the behaviour. Then, in the second stage, we optimize the imports which means that not only unused imports get removed but also that they are grouped logically and sorted automatically. To get rid of this bad smell during coding, the best way is to tick the setting "Optimize imports on thy fly (for the current project)" which can be seen in Figure 28.

[41] https://developer.android.com/guide/topics/resources/layout-resource [accessed on 8 December 2020]


Settings

Editor › General › Auto Import

Reset

XML

- [x] Show import popup

Java

Insert imports on paste: All

Show import popup for: [x] classes [x] static methods and fields

- [x] Add unambiguous imports on the fly
- [x] Optimize imports on the fly (for current project)

Exclude from import and completion:

Class/package/member qualified name mask	Scope
No exclude patterns	

Kotlin

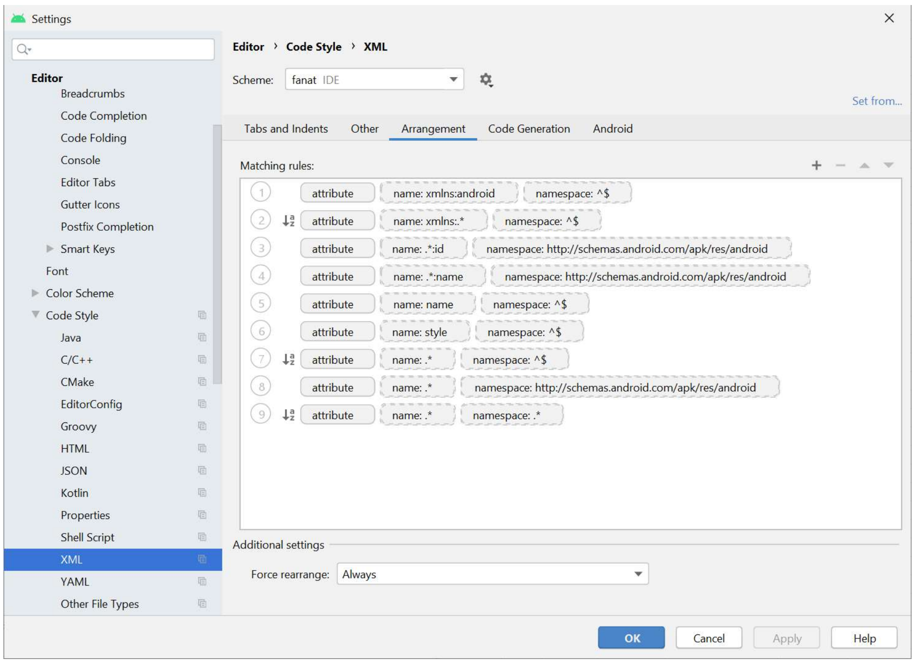
- [ ] Add unambiguous imports on the fly
- [ ] Optimize imports on the fly (for current project)

C/C++

- [x] Show import popup
- [x] Auto import in completion

Figure 28: To get rid of unused imports a setting exists which tells Android Studio to optimize imports automatically while coding.

We used stage three to rearrange the code. That means, that the source code within the Java files or XML files follows the rules defined in the code style settings for the project as described in 3.1.3. Moreover, the entries get rearranged as defined within the settings. The attribute of type "xmlns:android" namespace will always be arranged before any other "xmlns" attribute for the configuration shown in Figure 29. This arrangement helps comparing files or individual views faster.


Settings

Editor › Code Style › XML

Scheme: fanat IDE

Set from...

Tabs and Indents | Other | Arrangement | Code Generation | Android

Matching rules:

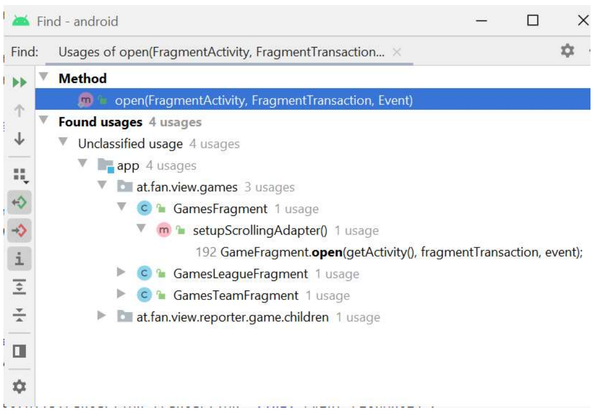
1		attribute	name: xmlns:android	namespace: ^$
2	↓a z	attribute	name: xmlns:.*	namespace: ^$
3		attribute	name: .*:id	namespace: http://schemas.android.com/apk/res/android
4		attribute	name: .*:name	namespace: http://schemas.android.com/apk/res/android
5		attribute	name: name	namespace: ^$
6		attribute	name: style	namespace: ^$
7	↓a z	attribute	name: .*	namespace: ^$
8		attribute	name: .*	namespace: http://schemas.android.com/apk/res/android
9	↓a z	attribute	name: .*	namespace: .*

Additional settings

Force rearrange: Always

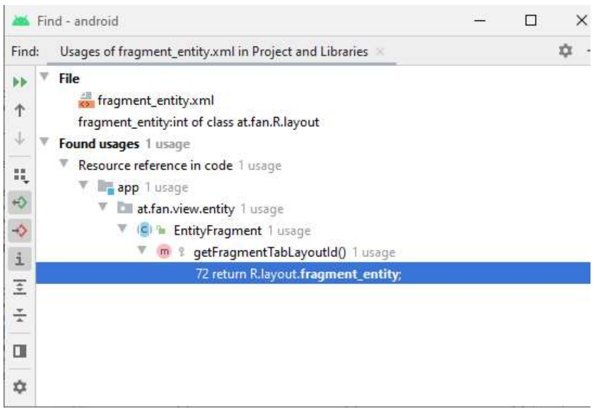
Figure 29: The configuration for the arrangement of XML attributes.

In the next stage, stage four, unused resources got removed. This does not include unused methods or variables but xml files or drawable files. By right-clicking, any referenceable item, namely files or identifiers within XML files, and selecting the option “Find Usages”, the usages of it are displayed as shown in Figure 30 for a method and in Figure 31 for an XML file.



```
Find - android
Find:   Usages of open(FragmentActivity, FragmentTransaction...
Method
    open(FragmentActivity, FragmentTransaction, Event)
Found usages 4 usages
  Unclassified usage 4 usages
    app 4 usages
      at.fan.view.games 3 usages
        GamesFragment 1 usage
          setupScrollingAdapter() 1 usage
            192 GameFragment.open(getActivity(), fragmentTransaction, event);
        GamesLeagueFragment 1 usage
        GamesTeamFragment 1 usage
      at.fan.view.reporter.game.children 1 usage
```

Figure 30: The outcome of finding the usages of the method "open".



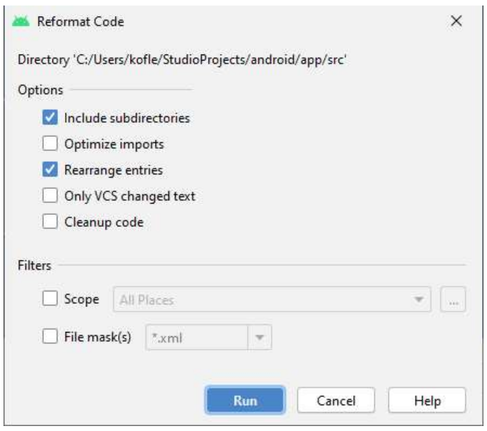
```
Find - android
Find:   Usages of fragment_entity.xml in Project and Libraries
File
    fragment_entity.xml
    fragment_entity:int of class at.fan.R.layout
Found usages 1 usage
  Resource reference in code 1 usage
    app 1 usage
      at.fan.view.entity 1 usage
        EntityFragment 1 usage
          getFragmentTabLayoutId() 1 usage
            72 return R.layout.fragment_entity;
```

Figure 31: This file is only referenced in one class.

Thus, in stage four unused files got removed but unused variables or methods do not get removed.

In the fifth stage, the last one, we used the same command as in the previous stage but indlucded unused resource identifiers also. By this, identifiers that are not referenced get removed also as they are not needed and therefore, these attributes are dead code and hence bad smells.

The first three stages can be executed by right-clicking the “src” folder and selecting “Reformat Code”. Within the popped-up window which is shown in Figure 32, the option “Include subdirectories” is always ticked and all places and file masks are scoped. In the first stage, no other option is selected. In the second stage, “Optimize imports” is ticked. In stage three, “Rearrange entries” gets ticked. For stages four and five, the “src” folder gets right-clicked again and in the “Refactor” selections “Remove Unused Resources” is clicked. The option for deleting identifier declarations gets only ticked for the fifth stage as shown in Figure 33.


Reformat Code

Directory 'C:/Users/kofle/StudioProjects/android/app/src'

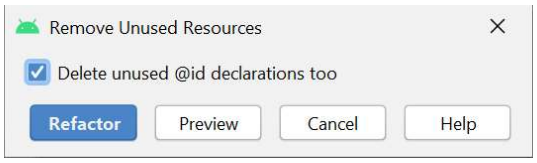
Options

- [✓] Include subdirectories
- [ ] Optimize imports
- [✓] Rearrange entries
- [ ] Only VCS changed text
- [ ] Cleanup code

Filters

- [ ] Scope: All Places
- [ ] File mask(s): *.xml

Figure 32: The options for rearranging code are ticked singularly for stages one to three.


Remove Unused Resources

- [✓] Delete unused @id declarations too

Figure 33: Unused identifiers can also be removed by ticking the option within the command for removing unused resources.

3.3.2 Results

A problem occurred after applying the removal of unused resources including unused “@id” declarations. Within the project, the developers of the legacy code used the method *getIdentifier* of the class Resources[42], which returns the corresponding identifier. As the used aliases are never referenced directly, the identifier attributes of those views were removed automatically. We detected this problem only during runtime, as the method returned “0” and thus could not find a resource. This caused the application to crash. Therefore, we exchanged all dynamically created identifiers to directly declared integer values as presented in Listing 16 and Listing 17.

```java
for (int i = 0; i < questionBullets.length; i++) {
    questionBullets[i] = fragmentView.findViewById(
            getResources().getIdentifier("question_circle_" + (i + 1),
                    "id", getContext().getPackageName()));
    ...
}
```

Listing 16: Getting a resource identifier by string leads to crashes in case the identifier does not exist. In that case the associating attributes were removed automatically as its created symbol was not referenced directly. © by SFA Sport GmbH

The advantage of using the IDs directly is not only the usage recognition but also that it is faster than searching for the corresponding identifier by name during runtime (Google LLC, 2020).

```java
int[] questionCircleIds = {R.id.question_circle_1, R.id.question_circle_2,
        R.id.question_circle_3, R.id.question_circle_4, R.id.question_circle_5,
        R.id.question_circle_6, R.id.question_circle_7, R.id.question_circle_8,
        R.id.question_circle_9, R.id.question_circle_10};
for (int i = 0; i < questionCircleIds.length; i++) {
    questionBullets[i] = fragmentView.findViewById(questionCircleIds[i]);
    ...
}
```

Listing 17: By declaring the integers directly via symbols their usage is recognized and the corresponding layout attributes are not removed automatically.

[42] https://developer.android.com/reference/android/content/res/Resources [accessed on 8 December 2020]

For the defined stages, the outcome of the clean-up process is summarized within the next section. For counting the lines and files within the project a plugin for Android Studio called "Statistic" is used[43]. To repeat the stages, in stage one the code was reformatted. In the second stage, we optimized the imports. We conducted the rearrangement of the code in stage three, and at stage four and five unused resources were automatically removed, whereby we removed the identifiers in the last stage. Stage 0 represents the data before any cleaning up. We used the code from the actual productive daily life of the developers. The stages are described shortly in Table 2.

In Table 3, the effects of the different stages on the changes in the version control can be found. For the version control, GitLab was used. Whenever a file was affected by the stages, the count in "Affected Files" is incremented. In the beginning, 938 files are in total within the "src" folder. After cleaning up there are 850 files left. If a line was added or removed, this is count in "Insertions" and "Deletions". The absolute number of changes in lines are presented in "Changes". In the beginning and after each stage an Android package (APK) is generated to see its changes. As this archive file contains all the content for installation of the application on Android-powered devices, it is interesting if these cleaning up stages influence their size (Google LLC, 2019). Both files contain all necessary files and resources, like the Android manifest XML file or the compiled source files, but the release APK is beyond that signed and optimized (Google LLC, 2020). The signature is created with the platform-specific certificate.

On Table 4 and Table 5, the changes in the different stages for Java and XML files are presented. There are the total count of Java classes and XML files, the total file sizes, and the number of lines in total, of source code and blank lines.

If we look at Figure 34, which displays the number of lines and the APK file sizes, we see that the trend of the APK size is almost the same as that of XML files. This means that the APK size correlates with the number of lines in the XML files, but not with that of the source code files. Therefore, we can assume that the APK size only depends on the resource files. But that means that there are also unused resource files in the APK. The fact that the changes in the source code do not result in any changes in the APK sizes can be explained by the compilation process and the optimization. The first stage only affects changes in the version control but does not change the number of lines. So, here is only formatting. Stages two and three only cause changes in the Java source code files and the fourth and fifth stages are only relevant for

[43] https://plugins.jetbrains.com/plugin/4509-statistic [accessed on 25 November 2020]

XML files, with no insertions but only deletions. In total, around 12.82% of XML lines and 0.86% of Java source code lines could be removed. An interesting fact is that the import optimization of stage two does only remove source code lines in Java but does not remove blank lines. On the other side, the rearrangement of code does mainly remove blank lines in Java source code files. As a result, the maximum amount of both differs in stages. So, both stages are important for a cleaner code. In stage four the count of XML files is reduced a lot including deletions of blank and source code lines. In stage five only identifier get deleted which results in no deletions of files or blank lines anymore. In brief, the first three stages result in cleaner code, the last two stages reduce the size of the APK and remove unnecessary resources.

Stage	Description
0	Status Quo
1	Reformat Code
2	Optimize Imports
3	Rearrange Code
4	Remove unused Resources
5	Remove unused Identifiers

Table 2: The description of the different stages in short.

	Changes in Version Control				APK	
Stage	Affected Files	Insertions	Deletions	Changes	Debug (kiB)	Release (kiB)
0	0	0	0	0	11,879.249	8,594.091
1	115	667	654	13	11,879.251	8,594.083
2	143	286	456	-170	11,879.154	8,593.931
3	379	6127	6600	-473	11,879.111	8,594.047
4	93	0	3646	-3646	11,440.339	8,235.610
5	64	0	169	-169	11,432.818	8,229.558

Table 3: The different stages have influenced the changes in the version control differently. The created APKs have different sizes in each stage.

	Java				
Stage	Count	Size (kB)	Total Lines	Source Code Lines	Blank Lines
0	410	3,047	76,953	59,970	13,579
1	410	3,047	76,949	59,967	13,578
2	410	3,040	76,764	59,784	13,577
3	410	3,039	76,291	59,762	13,104
4	410	3,039	76,291	59,762	13,104
5	410	3,039	76,291	59,762	13,104

Table 4: The changes of the different stages for Java files.

	XML				
Stage	Count	Size (kB)	Total Lines	Source Code Lines	Blank Lines
0	329	1,210	29,509	25,128	4,381
1	329	1,210	29,509	25,128	4,381
2	329	1,210	29,541	25,158	4,383
3	329	1,210	29,541	25,158	4,383
4	272	1,057	25,895	22,029	3,866
5	272	1,049	25,726	21,860	3,866

Table 5: The changes of the different stages for XML files.

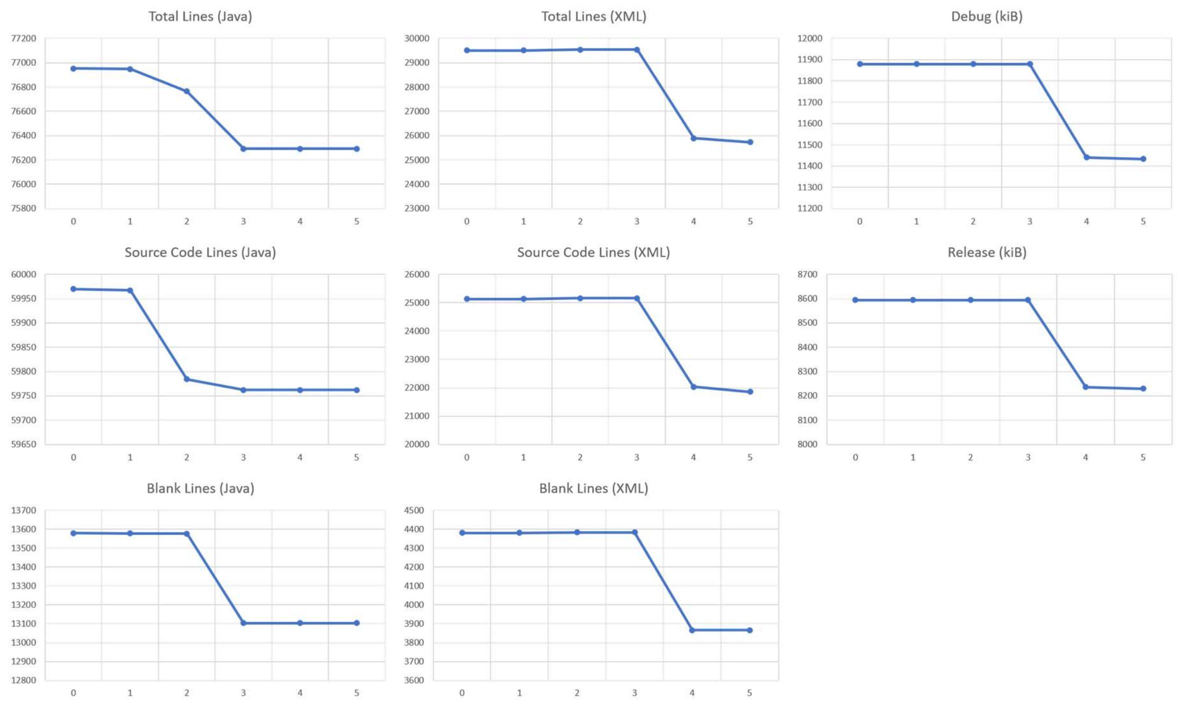


Figure 34: The five stages have different outcomes of the changes in line and APK file size.

3.4 Exchanging Raster Graphics with Vector Graphics in Android Studio

Images are important and needed resources for nearly any application. It is important to differ between file formats and choose the correct one in various situations. Nowadays, there exist two types of formats for graphics, namely raster graphics and vectors graphics (Sakshica & Gupta, 2015). According to A. Almutairi (2018), the most used file format for raster graphics is Portable Network Graphics (PNG). It has various features that other common file formats do not have. Almutairi says further, that regarding vector graphics, Scalable Vector Graphics (SVG) are the most used types, and that this open standard image format uses XML as basis.

Due to resolution and resizing issues the better format for icons is SVG (Google LLC, 2020). As stated, different screen densities can be supported without reducing the quality of the images. Moreover, updating one XML file in Android Studio is easier than maintaining different raster graphics for various screen resolutions. While these vector images are created by the composition of mathematical forms, in cases where complicated compositions and great detail is needed, raster or non-line digital images are used (Almutairi, 2018).

3.4.1 Status Quo

The local digital image resources which are used in Android Studio projects are commonly stored within the "drawable" folder. Before the cleaning up process explained in the previous section, 172 PNG files and 55 XML files were stored within the folder. During the five stages explained above, 21 PNG files and seven XML files were deleted due to not being used. The five JPEG files were not affected. These results are summarized in Table 6.

Graphics Type	File Ending	Before Clean-up	After Clean-up	Changes
Raster	.jpg	5	5	0
	.png	172	151	-21
Vector	.xml	55	48	-7

Table 6: The five stages for cleaning up the project sources folders had effects on the amount of used drawable files.

These resource files are completely inconsistent not only in the design but also in the naming as can be seen in Figure 35. All in all, the icons are completely differently designed in colour, structure, and size. Some icons were designed by the developers themselves like the blue left arrow even if a similar icon exists within the suggested and above-mentioned open Material Design Icons. Some of the icons are outdated and got an update by the designers of the platform to introduce consistency and align the icons to the corporate identity. Regarding the naming, it should be followed the defined naming convention explained in 3.1.2. The exchange of the resources is explained in the next section.

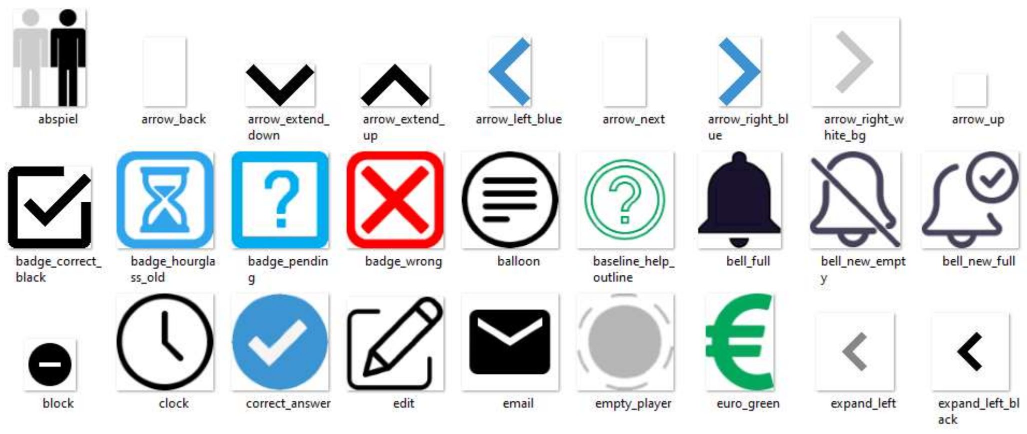


Figure 35: An excerpt of some PNG files within the drawable folder of the project. The design of the icons, like the size, the colours, the thickness of lines, are completely inconsistent and not in line with the corporate identity of the platform. The naming is inconsistent, also. © by SFA Sport GmbH

3.4.2 Applying Exchange of Graphics

We explored the whole drawable folder and for each file we chose what should happen with it. Accordingly, we made a separation between the different types explained in 3.1.2. In case an icon was redesigned by the designers of the project, we exchanged it by its SVG representation. To mark those icons which are not visible to the user, we need to explore their usages in code. For this purpose, context understanding and knowing the features of the application is a precondition. For icons where similar material representations exist, we downloaded the corresponding icon and converted it to XML. For icons that are in use but where no new icon are designed yet or where it does not exist a double in Google's Material Design Icons, we used "td_" as prefix to indicate that to the designers.

The process of converting an SVG to an XML representation is explained in Google's documentation for Android app developers in the section titled "Add multi-density vector graphics" (Google LLC, 2020). By the help of Vector Asset Studio, these conversions are executed. The Vector Asset Studio is started by right-clicking the drawable folder and selecting "New" and further "Vector Asset". The local SVG file is selected, the corresponding prefix and well-defined naming convection conform name is used and the conversion can be initiated finally. A precondition for using vector drawables is a minimum Application Programming Interface (API) level of 21. In the project, the minimum is set to 22 at time of this master's thesis.

3.4.3 Results

The process took a lot of time as each image resource needed to be explored and exchanged if necessary. The last commit for the first big exchanging process of graphics was made on 13 May 2020. At that point in time, we exchanged about 80 resource files by new SVG based representations as can be seen in Table 7 and Table 8. The total amount varies among others because of other changes to the source code during that period like introducing new icons or shapes. The remaining 70 PNG files consist of one new raster image representation of a field which is used as a background in the line-up view of a team. Previously, this was a JPEG file. 18 of them are not visible for the user at that point in time and do not need to be updated yet, and the remaining 51 files need a new design or vector representations in future. These findings are summarized in Table 9.

Graphics Type	File Ending	Before Exchange	After Exchange	Changes
Raster	.jpg	5	4	-1
	.png	151	70	-81
Vector	.xml	48	134	86

Table 7: The exchanging of graphics

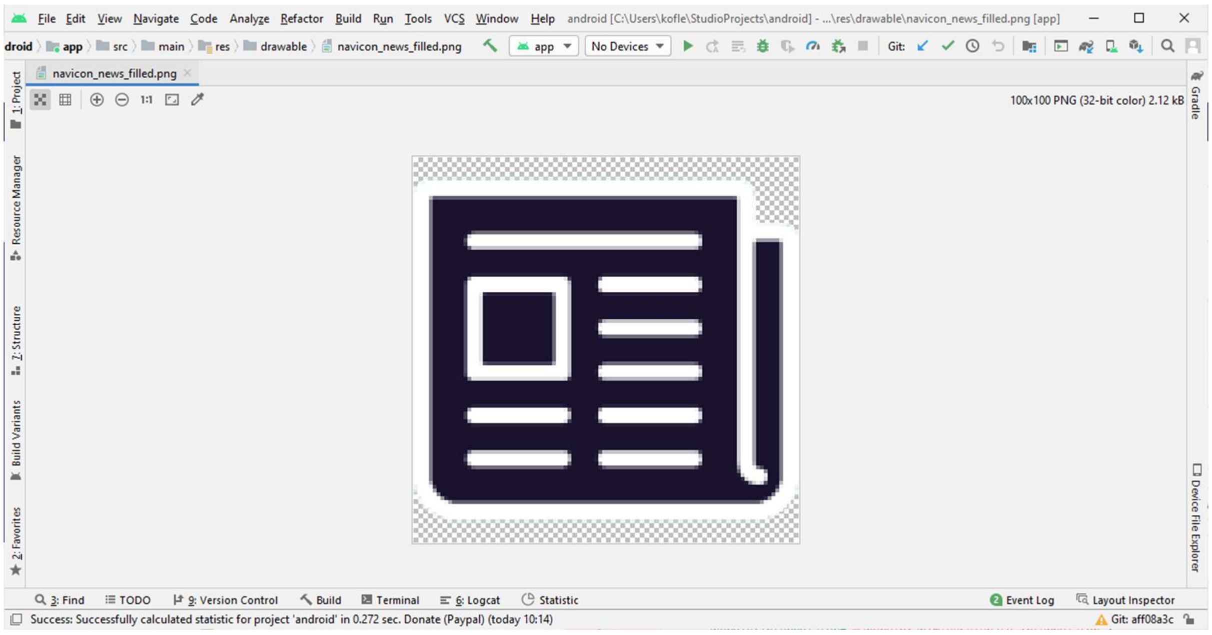
Prefix	After Exchange
_	23
ic_	79
im_	5
mt_	7
sh_	43
td_	51

Table 8: The drawable resource were grouped by their resource type with the usage of defined prefixes.

Prefix	PNG-Files
_	18
im_	1
td_	51
Total:	70

Table 9: The remaining PNG files consist of a complex field image, 18 resources that are not visible for the user and 51 resources that need to be exchanged in future.

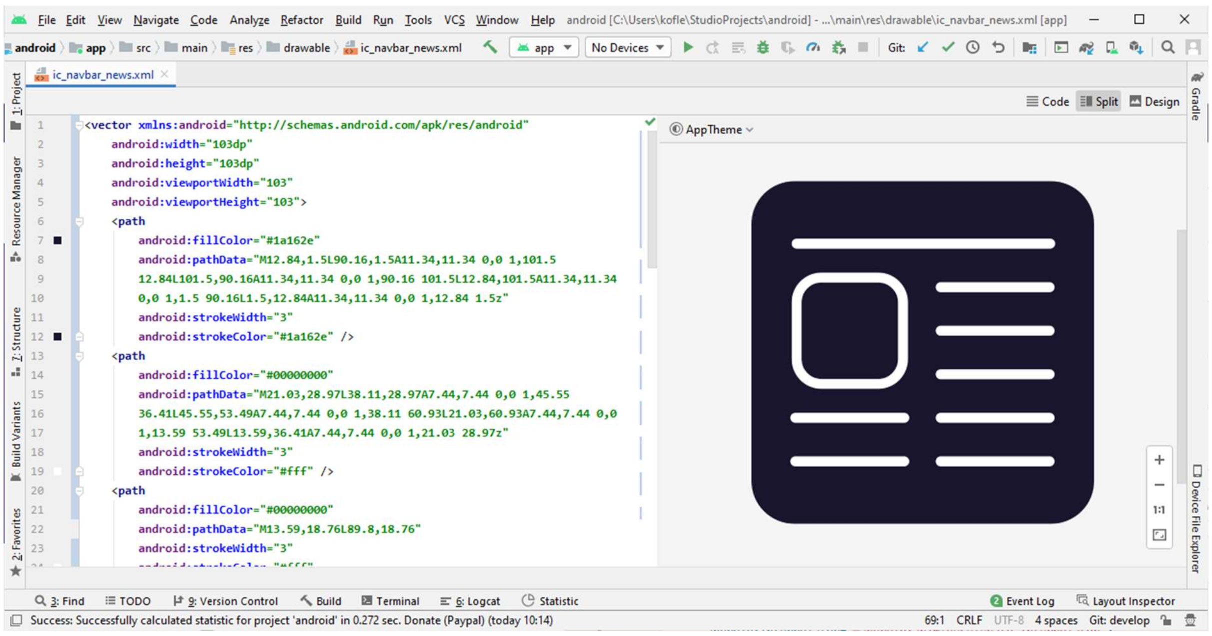
An example of exchanging a PNG file to a newly designed SVG file is shown in Figure 36 and Figure 37. As can be seen, the PNG representation of the navigation bar icon for news is pixeled, which means that is not scalable without loss of quality (Sakshica & Gupta, 2015).


File Edit View Navigate Code Analyze Refactor Build Run Tools VCS Window Help android [C:\Users\kofle\StudioProjects\android] - ...\res\drawable\navicon_news_filled.png [app]
droid app src main res drawable navicon_news_filled.png
app
No Devices
Git:
navicon_news_filled.png
100x100 PNG (32-bit color) 2.12 kB
1: Project
Resource Manager
7: Structure
Build Variants
2: Favorites
Gradle
Device File Explorer
3: Find
TODO
9: Version Control
Build
Terminal
6: Logcat
Statistic
Event Log
Layout Inspector
Git: aff08a3c

Success: Successfully calculated statistic for project 'android' in 0.272 sec. Donate (Paypal) (today 10:14)

Figure 36: The news icon in the navbar was previously stored in raster graphics format and is not optimized for each display resolution. Moreover, the design is not as clean as wanted. © by SFA Sport GmbH

In contrast, the news icon below seems to be very sharp. This is because of its data format of vector graphics. These files are scalable without loss of quality (Sakshica & Gupta, 2015).



```xml
<vector xmlns:android="http://schemas.android.com/apk/res/android"
    android:width="103dp"
    android:height="103dp"
    android:viewportWidth="103"
    android:viewportHeight="103">
    <path
        android:fillColor="#1a162e"
        android:pathData="M12.84,1.5L90.16,1.5A11.34,11.34 0,0 1,101.5
        12.84L101.5,90.16A11.34,11.34 0,0 1,90.16 101.5L12.84,101.5A11.34,11.34
        0,0 1,1.5 90.16L1.5,12.84A11.34,11.34 0,0 1,12.84 1.5z"
        android:strokeWidth="3"
        android:strokeColor="#1a162e" />
    <path
        android:fillColor="#00000000"
        android:pathData="M21.03,28.97L38.11,28.97A7.44,7.44 0,0 1,45.55
        36.41L45.55,53.49A7.44,7.44 0,0 1,38.11 60.93L21.03,60.93A7.44,7.44 0,0
        1,13.59 53.49L13.59,36.41A7.44,7.44 0,0 1,21.03 28.97z"
        android:strokeWidth="3"
        android:strokeColor="#fff" />
    <path
        android:fillColor="#00000000"
        android:pathData="M13.59,18.76L89.8,18.76"
        android:strokeWidth="3"
```

Success: Successfully calculated statistic for project 'android' in 0.272 sec. Donate (Paypal) (today 10:14)

File Edit View Navigate Code Analyze Refactor Build Run Tools VCS Window Help android [C:\Users\kofle\StudioProjects\android] - ...\main\res\drawable\ic_navbar_news.xml [app]
android app src main res drawable ic_navbar_news.xml
app
No Devices
ic_navbar_news.xml
Code Split Design
AppTheme
3: Find
TODO
9: Version Control
Build
Terminal
6: Logcat
Statistic
Event Log
Layout Inspector
69:1 CRLF UTF-8 4 spaces Git: develop

Figure 37: The new news icon is optimized for each screen resolution, the design is simple and fits to the corporate identity. To the left, an excerpt of the created XML code is shown.

At the end of the implementation of this master's thesis on 22 November 2020, we took another status for the drawable resources as shown in Table 10. Since we took the first status on 13 May 2020, 193 days have passed in which developers changed the resources possibly. During that time the situation has improved. 14 files in total were eliminated that should have been changed in future, which can be recognized by the changes of files with the prefixes "_" and "td_". 32 new vector graphics were added to the repository which consists of 26 self-created icons ("ic_"), and six new material icons ("mt_"). The changes in raster graphics images ("im_") and shapes ("sh_") are not that interesting for this thesis.

Prefix	13-May-20	22-Nov-20	Changes
_	23	21	-2
ic_	79	105	26
im_	5	6	1
mt_	7	13	6
sh_	43	56	13
td_	51	39	-12

Table 10: There are less files that need to be exchanged in future; these are files with prefixes "_" or "td_".

In Table 11 the two status are compared graphically. The outermost prefixes are decreased, which is positive for the project as minor work needs to be done for exchanging graphics. The number of files with prefixes "ic_" and "mt_" also increased which is positive for the introduction of vector graphics within the project. Moreover, their increase and the increase of the number of files with “im_” and “sh_” resources report of the increased features of the project which is also positive.

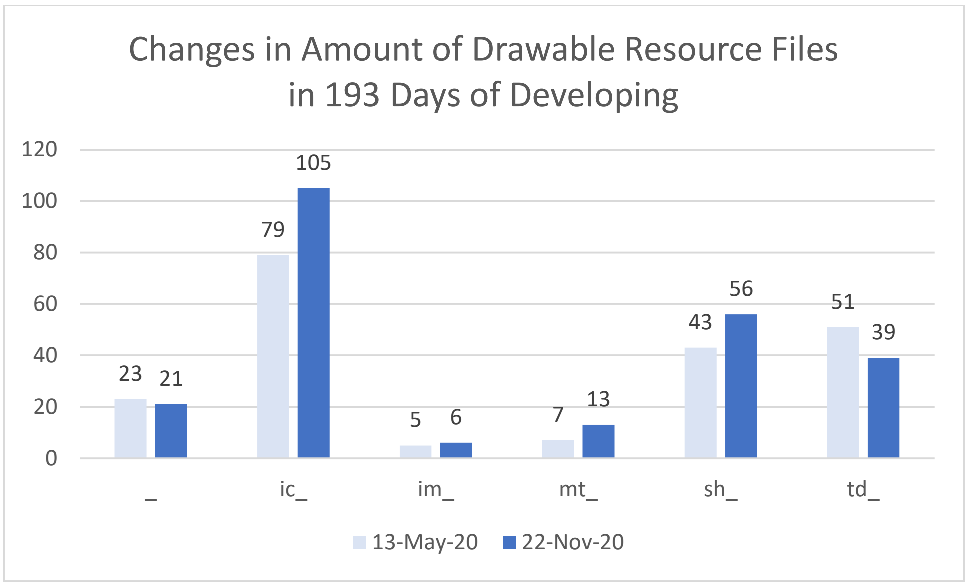


Table 11: The comparison between the two snapshots taken in April and November for each drawable file prefix.

All in all, exchanging raster graphics by vector graphics in Android Studio is very easy by the usage of Android’s Vector Asset Studio. By the help of prefixes, a grouping can be made to make for example clear which resources need to be refactored, or which resources are limited in resolution.

4 Evaluation

According to the IEEE standard (1990) complexity in the context of software engineering is defined as “the degree to which a system or component has a design or implementation that is difficult to understand and verify” (IEEE Standards Coordinating Committee, 1990, p. 18). Thus, the understandability of code is directly connected to its complexity (Kaur, Minhas, Mehan, & Kakkar, 2009). This means that code which is easy to understand is less complex. One of the goals of this master’s thesis is to refactor code to and write new clean code. To evaluate the improvement, methods need to be found, described, and applied.

Several studies were conducted on the quantitative improvement of code complexity by code refactoring similar to the examples described in the papers by Kataoka, Imai, Andou, and Fukaya (2002), Leitch, and Stroulia (2003), Moser, Sillitti, Abrahamsson, and Succi (2006), or Ratzinger, Fischer, and Gall (2005). Regarding the subjective measurable quality improvement by code refactoring there exist also some papers, like those by Kim, Zimmermann, and Nagappan (2012), or Wang (2009).

The methodology used for the evaluation here is an evaluation technique already used by Kafura and Reddy (1987). Therefore, on the one hand, we use software metrics, which are calculated automatically by a plugin in Android Studio, and, on the other hand, we relate them to data extracted by developed questionnaires. To be able to relate the outcomes together, we use real code examples extracted and adapted from the project in the survey.

Thus, in this section, selected code complexity analysis methods are discussed, available tools in Android Studio get applied, and finally, the improvements of the refactored code parts within this project are evaluated. Furthermore, the used questionnaire is evaluated. Finally, the results of the different evaluation methods and their relationship are discussed.

4.1 Quantitative Evaluation of Code Complexity by Software Metrics

To evaluate the quality of software, Kaur, Minhas, Mehan, and Kakkar (2009) suggest using complexity analysis, as by this evaluation, not only its maintainability but also its reliability can be quantified. For object-oriented languages, Kaur et al. say that there exist dynamic and static metrics. In this thesis, only static metrics are used as the focus is on readability, understandability, and maintainability. One important complexity method is discussed in detail, namely Mc Cabe’s Metric.

Such tools for analysing software are crucial to finding the most complex parts of a medium or large software system to find issues and enhance the quality at the end (Aguiar, Restivo, Correia, Ferreira, & Dias, 2019).

Within the next section, the used metrices are described.

4.1.1 Code Complexity Analysis

For evaluating the complexity of code at various points in time, we use a plugin for Android Studio. Beside of the used free and open-source tool, we explored another plugin during the review called CodeMR, which is neither open-source nor freely available[44]. Even if this tool shows interesting and valuable graphics and models about the architecture, dependencies, and code quality, it only offers a few complexity metrics (CodeMR, 2020). Therefore, the plugin explained below was chosen.

The selected plugin by Leijdekkers and Sixth and Red River Software (2020) is called MetricsReloaded and the code is freely available on GitHub[45]. The contributors of this software are D. Jemerov and B. Leijdekkers. Many papers have used this plugin already for calculating metrics (Molnar & Motogna, 2017; Saifan & Al-Rabadi, 2017).

Some of the possible complexity metrics are shown in Table 12 with the original description within the plugin. Leijdekkers has allowed the use of his description in this master's thesis personally by email[46].

[44] https://www.codemr.co.uk [accessed on 29 November 2020]
[45] https://github.com/BasLeijdekkers/MetricsReloaded [accessed on 27 November 2020]
[46] "*Of course you may use the descriptions of the metrics with a reference. I am happy you have found the plugin useful.*" (Leijdekkers, 3 December 2020, as email response for using the description within this master's thesis)

Abbr.	Name	Description
OCavg	Average operation complexity	Calculates the average cyclomatic complexity of the non-abstract methods in each class. Inherited methods are not counted for purposes of this metric.
WMC	Weighted method complexity	Calculates the total cyclomatic complexity of the methods in each class.
ev(G)	Essential cyclomatic complexity	Calculates the essential complexity of each non-abstract method. Essential complexity is a graph-theoretic measure of just how ill-structured a method's control flow is. Essential complexity ranges from 1 to v(G), the cyclomatic complexity of the method.
Iv(G)	Design complexity	Calculates the design complexity of a method. The design complexity is related to how interlinked a methods control flow is with calls to other methods. Design complexity ranges from 1 to v(G), the cyclomatic complexity of the method. Design complexity also represents the minimal number of tests necessary to exercise the integration of the method with the methods it calls.
v(G)	Cyclomatic complexity	Calculates the cyclomatic complexity of each non-abstract method. Cyclomatic complexity is a measure of the number of distinct execution paths through each method. This can also be considered as the minimal number of tests necessary to completely exercise a method's control flow. In practice, this is 1 + the number of if's, while's, for's, do's, switch cases, catches, conditional expressions, &&'s and ||'s in the method.
Cyclic	Number of cyclic dependencies	Calculates the number of classes or interfaces which each class directly or indirectly depends on, and which in turn directly or indirectly depend on it. Such cyclic dependencies may result in code which is difficult to understand and test.
Dcy	Number of dependencies	Calculates the number of classes or interfaces which each class directly depends on.
Dcy*	Number of transitive dependencies	Calculates the number of classes or interfaces which each class directly or indirectly depends on.
Dpt	Number of dependents	Calculates the number of classes or interfaces which directly depend on each class.
Dpt*	Number of transitive dependents	Calculates the number of classes or interfaces which directly or indirectly depend on each class.
PDcy	Number of package dependencies	Calculates the number of packages on which each package is directly dependent.
PDpt	Number of dependent packages	Calculates the number of packages which contain direct dependencies on each package.
Files	Number of files	Calculates the total number of files in each module.
LOC	Lines of code	Calculates the number of lines of code in each package. Comments are counted for purposes of this metric, but whitespace is not.
AHF	Attribute hiding factor	Calculates the degree of attribute (field) encapsulation in a project. Essentially, it gives the ratio of how many classes an average field is visible from, other than the defining class.

AIF	Attribute inheritance factor	Calculates the degree of attribute (field) inheritance in a project. Essentially, it gives the ratio of what percentage of the available fields on an average class are due to inheritance, rather than directly defined on the class.
CF	Coupling factor	Calculates the degree of coupling in a project as a whole. Essentially it reports what proportion of the classes in a project are used by (couple to) an average class in the project.
MHF	Method hiding factor	Calculates the degree of method encapsulation in a project. Essentially, it gives the ratio of how many classes an average method is visible from, other than the defining class.
MIF	Method inheritance factor	Calculates the degree of method inheritance in a project. Essentially, it gives the ratio of what percentage of the available methods on an average class are due to inheritance, rather than directly defined on the class. Methods inherited from library classes are not used in the calculation of this metric.
PF	Polymorphism factor	Calculates the degree of polymorphism in a project as a whole. Essentially it reports on the probability that a given method will be overridden in a subclass.

Table 12: Description of the calculated metrics used within this thesis. The content is the original description of the used metrics of the plugin provided by MetricsReloaded (Leijdekkers & Sixth and Red River Software, 2020).

To summarize the description by Leijdekkers et al. (2020), the first two metrics *OCavg* and *WMC* are complexity class metrics, *ev(G)*, *iv(G)*, and *v(G)* calculate their metrics on the method level. The next five metrics, *Cyclic*, *Dcy*, *Dcy**, *Dpt*, and *Dpt**, explore the dependencies for classes, *PDcy* and *PDpt* explore them on package level. According to Leijdekkers, *Files* and *LOC* count the number of files, this thesis only focuses on Java files, and lines in the chosen selection. The last six metrics belong to the metrics for object-oriented design (MOOD) which measure typical object-oriented patterns on project levels like encapsulation, represented by the Attribute Hiding Factor (*AHF*) and Method Hiding Factor (*MHF*), inheritance, measured by Attribute Inheritance Factor (*AIF*) and Method Inheritance Factor (*MIF*), coupling, shown by the Coupling Factor (*CF*), and polymorphism with the corresponding Polymorphism Factor (*PF*) (Harrison, Counsell, & Nithi, 1998).

According to Harrison, Counsell, and Nithi (1998), *AHF* and *MHF* show the quality of hiding information which means the degree of code visibility on system-level, therefore *AHF* should be optimally at 100%, while a higher *MHF* means less functionality. However, Harrison et al. explain that a low value may indicate poor abstraction. They say further, that *AIF* and *MIF* measure directly how many attributes and methods are inherited in relation to the total number of each, so they should not be too low, which is a sign for a lack of inheritance, and not too high, indicating extreme inheritance. Harrison et al. explain that *CF* tells in which degree the classes of a system are in relation to each other by message passing or a reference to each other and should be very low, but not too low as a value of zero would indicate only communicating by inheritance or too much redundant code. However, very high coupling factors decrease maintainability, reusability, and understandability. The last factor, *PF*, highlights the potential of introducing polymorphism, so low values should be targeted as mentioned by Harrison et al. However, as the goal of this master's thesis is reducing complexity and improving readability, these values are not that relevant.

The most relevant metrics for this master's thesis are those calculating the complexity of methods and classes. To start with, the cyclomatic complexity v(G) was developed and first described by Thomas J. McCabe in 1976 and relies on the control flow graph of methods or functions (McCabe, 1976). As he states in his paper, his formula is based on the cyclomatic number given by

$$v(G) = e - n + p$$

Equation 1: v(G) represents the cyclomatic number of graph G.

with e, n, and p as the number of edges, nodes, and connected components within graph G (McCabe, 1976). For the directed case, this formula is correct for strongly connected components, where each node can be reached from each other node, that means that for each pair of nodes u and v within the component there exists a path from u to v and vice versa (Nuutila & Soisalon-Soininen, 1994).

In other cases, where there are a beginning and an end for each component without a connection back to the initial node such as in typical programs, the cyclomatic complexity is given by

$$v(G) = e - n + 2p$$

Equation 2: The cyclomatic complexity of program graphs.

and can be further simplified to

$$v(G) = e - n + 2$$

Equation 3: The cyclomatic complexity for methods or subroutines can be simplified as p is always one.

if only one connected component exists as for single methods or functions (McCabe, 1976). McCabe shows further that for structured programs the calculation can be simplified to the number of predicates within the component plus one as shown in

$$v(G) = \pi + 1$$

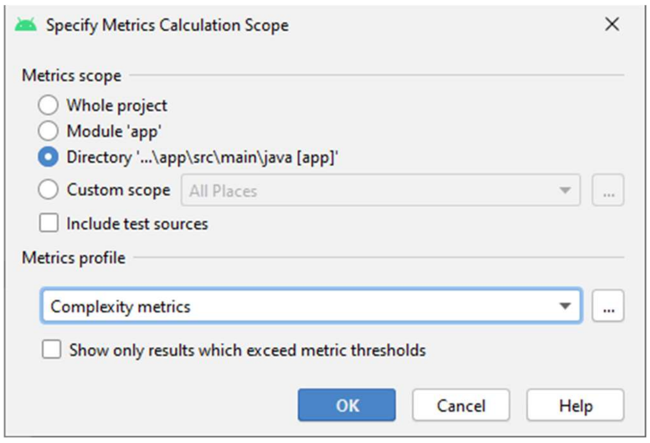
Equation 4: For a single-component program it is sufficient to count the predicates and add one to it to calculate its cyclomatic complexity.

with π representing the number of predicates within the program.

Equation 4 is used for the explanation shown in Table 12. As described by Leijdekkers et al. (2020), the other metrics *ev(G)* and *iv(G)* are connected to *v(G)* and can never be higher than that. The former shows how ill-structured the explored code itself is, while the latter represents the interlink to other methods called within the explored code. *WMC* and *OCavg* finally sum the cyclomatic complexities up. According to Leijdekkers et al., the first metric is the total sum and the second is the average of all methods within each class. They highlight the connection between testing effort and the number of paths within the class. Hence, the more paths there are within the control flow graph of the method, the more tests are needed to fully cover the method by tests.

4.1.2 Applied Code Complexity Analysis in Android Studio

MetricsReloaded can be installed in Android Studio as a plugin. For this master's thesis, version 1.9[47] was used. After installation, in the menu "Analyze" another option is added called "Calculate Metrics…". In Figure 38 the appearing window can be seen. Here, we selected the folder "java" and chose the option "Complexity metrics". Besides this, there are a few other options available, like "JUnit testing metrics" or "Javadoc coverage metrics". For this evaluation, the options "Complexity metrics", "Dependency metrics", "Lines of code metrics", "MOOD metrics", and "Number of files metrics" are applied. (Leijdekkers & Sixth and Red River Software, 2020)


Specify Metrics Calculation Scope

Metrics scope

Whole project [ ]
Module 'app' [ ]
[✓ Directory '...\app\src\main\java [app]']
Custom scope [ ] All Places
Include test sources [ ]

Metrics profile

Complexity metrics

Show only results which exceed metric thresholds [ ]

Figure 38: Calculating complexity metrics for folder "java".

To evaluate the improvement of the refactoring, we calculated the metrics at various points in time for the project. We evaluated it in the course of two years, taking a snapshot of the

[47] https://plugins.jetbrains.com/plugin/93-metricsreloaded [accessed on 30 November 2020]

complexity every four months. The dates of the seven evaluations are shown in Table 13. We copied each of the values manually into a Microsoft Excel[48] table and created graphs to show the changes over the two years.

Version	Date
1	22 November 2018
2	22 March 2019
3	19 July 2019
4	15 November 2019
5	24 March 2020
6	28 July 2020
7	24 November 2020

Table 13: Dates of the complexity snapshots of the projects each four months over a period of two years.

To justify these chosen dates, Figure 39 illustrates the most important dates regarding the change in code complexity. The used dates are chosen to see the improvement of the quantitative measurable quality in the two last snapshots of complexity, because within this period we removed, on the one hand, the unused resources as explained in 3.3, and, on the other hand, we started implementing features while taking into account the decisions we made regarding clean code and other principles as described above. During this time, not only preparatory refactoring was used excessively, but also comprehension refactoring to understand the legacy code better. However, it should not be forgotten that other developers continued to develop the application in the same way as before and that we still added features like the others.

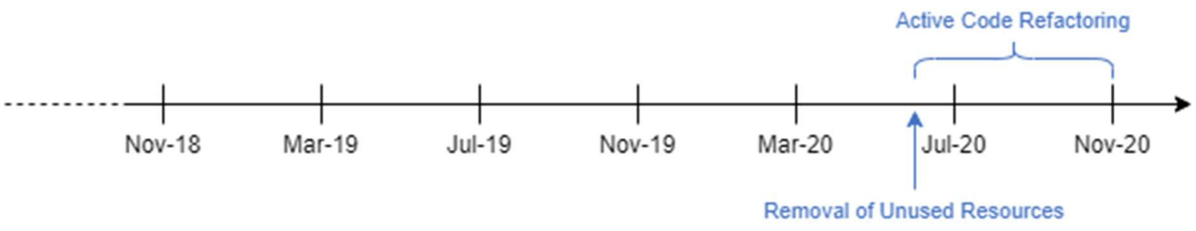
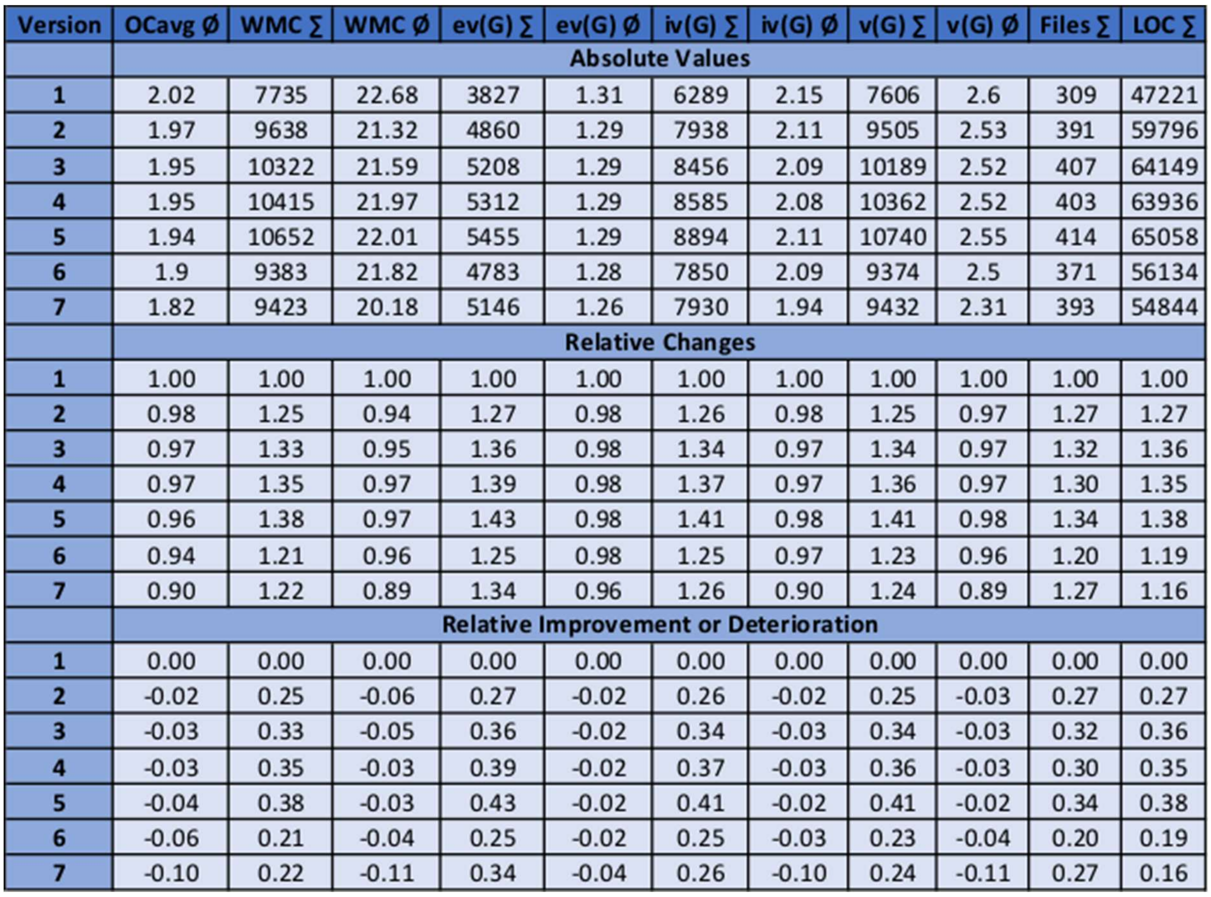


Figure 39: The used dates are chosen to see the improvement of the quantitative measurable complexity of code.

We decided to focus on three different aspects for our evaluation. These are first the package "adapter", holding any adapters within the project, secondly, "entity", which combines any code which is related to the profile pages of clubs and leagues, and finally the "game" package, containing code, which is related to the view showing an event. Moreover, to see the changes in the whole project within these two years, we examine the "java" package containing all Java code also.

[48] https://www.microsoft.com/en-us/microsoft-365/excel [accessed on 4 December 2020]

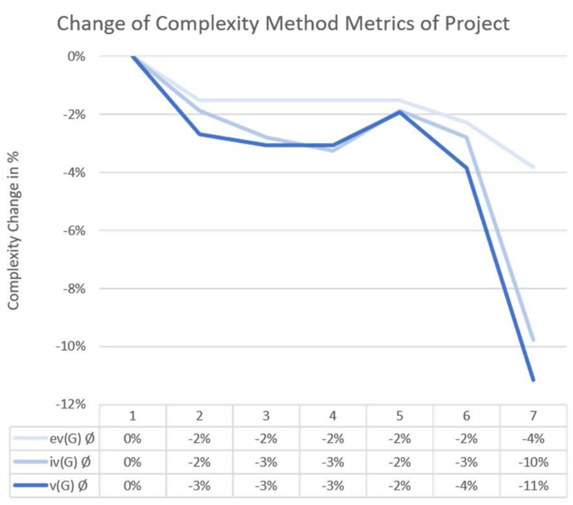
As the main focus for this evaluation lies on the complexity metrics, we will only present those metrics values including a number of files and lines of code. However, all data that was collected can be explored in the Appendix. Selected metrics values for the project can also be seen in Figure 40. To be able to compare the metrics in one diagram, we normalized each metrics by that value at version 1, meaning that we divided the metrics values for versions 2 to 7 by that value of version 1 as Kafura and Reddy suggest in their paper (1987).



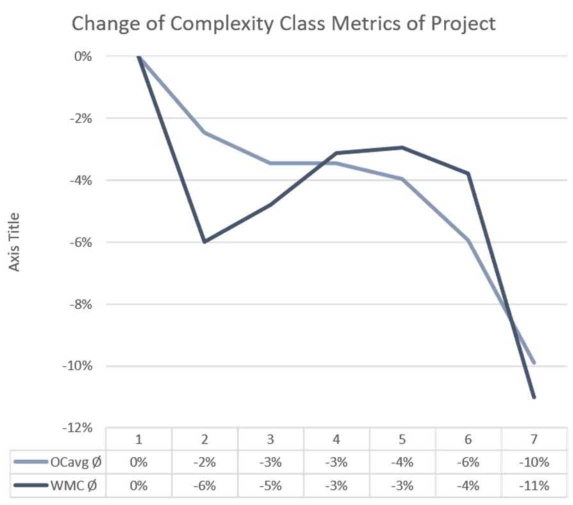
Version	OCavg Ø	WMC ∑	WMC Ø	ev(G) ∑	ev(G) Ø	iv(G) ∑	iv(G) Ø	v(G) ∑	v(G) Ø	Files ∑	LOC ∑
	Absolute Values										
1	2.02	7735	22.68	3827	1.31	6289	2.15	7606	2.6	309	47221
2	1.97	9638	21.32	4860	1.29	7938	2.11	9505	2.53	391	59796
3	1.95	10322	21.59	5208	1.29	8456	2.09	10189	2.52	407	64149
4	1.95	10415	21.97	5312	1.29	8585	2.08	10362	2.52	403	63936
5	1.94	10652	22.01	5455	1.29	8894	2.11	10740	2.55	414	65058
6	1.9	9383	21.82	4783	1.28	7850	2.09	9374	2.5	371	56134
7	1.82	9423	20.18	5146	1.26	7930	1.94	9432	2.31	393	54844
	Relative Changes										
1	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
2	0.98	1.25	0.94	1.27	0.98	1.26	0.98	1.25	0.97	1.27	1.27
3	0.97	1.33	0.95	1.36	0.98	1.34	0.97	1.34	0.97	1.32	1.36
4	0.97	1.35	0.97	1.39	0.98	1.37	0.97	1.36	0.97	1.30	1.35
5	0.96	1.38	0.97	1.43	0.98	1.41	0.98	1.41	0.98	1.34	1.38
6	0.94	1.21	0.96	1.25	0.98	1.25	0.97	1.23	0.96	1.20	1.19
7	0.90	1.22	0.89	1.34	0.96	1.26	0.90	1.24	0.89	1.27	1.16
	Relative Improvement or Deterioration										
1	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
2	-0.02	0.25	-0.06	0.27	-0.02	0.26	-0.02	0.25	-0.03	0.27	0.27
3	-0.03	0.33	-0.05	0.36	-0.02	0.34	-0.03	0.34	-0.03	0.32	0.36
4	-0.03	0.35	-0.03	0.39	-0.02	0.37	-0.03	0.36	-0.03	0.30	0.35
5	-0.04	0.38	-0.03	0.43	-0.02	0.41	-0.02	0.41	-0.02	0.34	0.38
6	-0.06	0.21	-0.04	0.25	-0.02	0.25	-0.03	0.23	-0.04	0.20	0.19
7	-0.10	0.22	-0.11	0.34	-0.04	0.26	-0.10	0.24	-0.11	0.27	0.16

Figure 40: Calculated metrics values for the whole project including relative changes and relative improvement or deterioration.

For evaluating the relative change of the overall complexity of the project, we decided to combine all averaged complexity values for method and class in one diagram each as shown in Figure 41. The absolute values of the complexity metrics are shown in Figure 42 in form of a diagram.



	1	2	3	4	5	6	7
ev(G) Ø	0%	-2%	-2%	-2%	-2%	-2%	-4%
iv(G) Ø	0%	-2%	-3%	-3%	-2%	-3%	-10%
v(G) Ø	0%	-3%	-3%	-3%	-2%	-4%	-11%



	1	2	3	4	5	6	7
OCavg Ø	0%	-2%	-3%	-3%	-4%	-6%	-10%
WMC Ø	0%	-6%	-5%	-3%	-3%	-4%	-11%

Figure 41: Relative change of complexity method and complexity class metrics of the project.

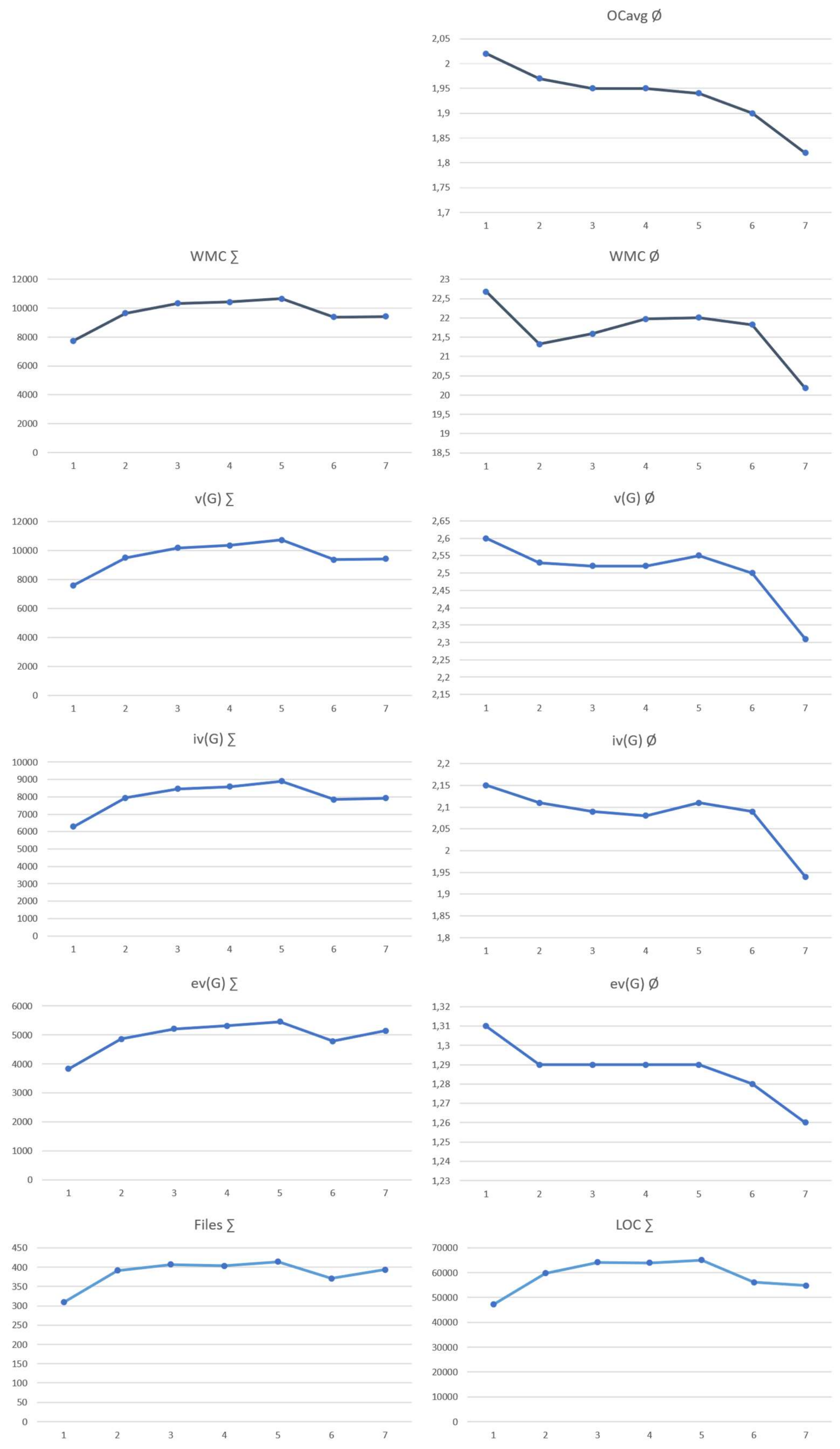


Figure 42: Absolute complexity class metrics, complexity method metrics, number of files, and lines of code of the project.

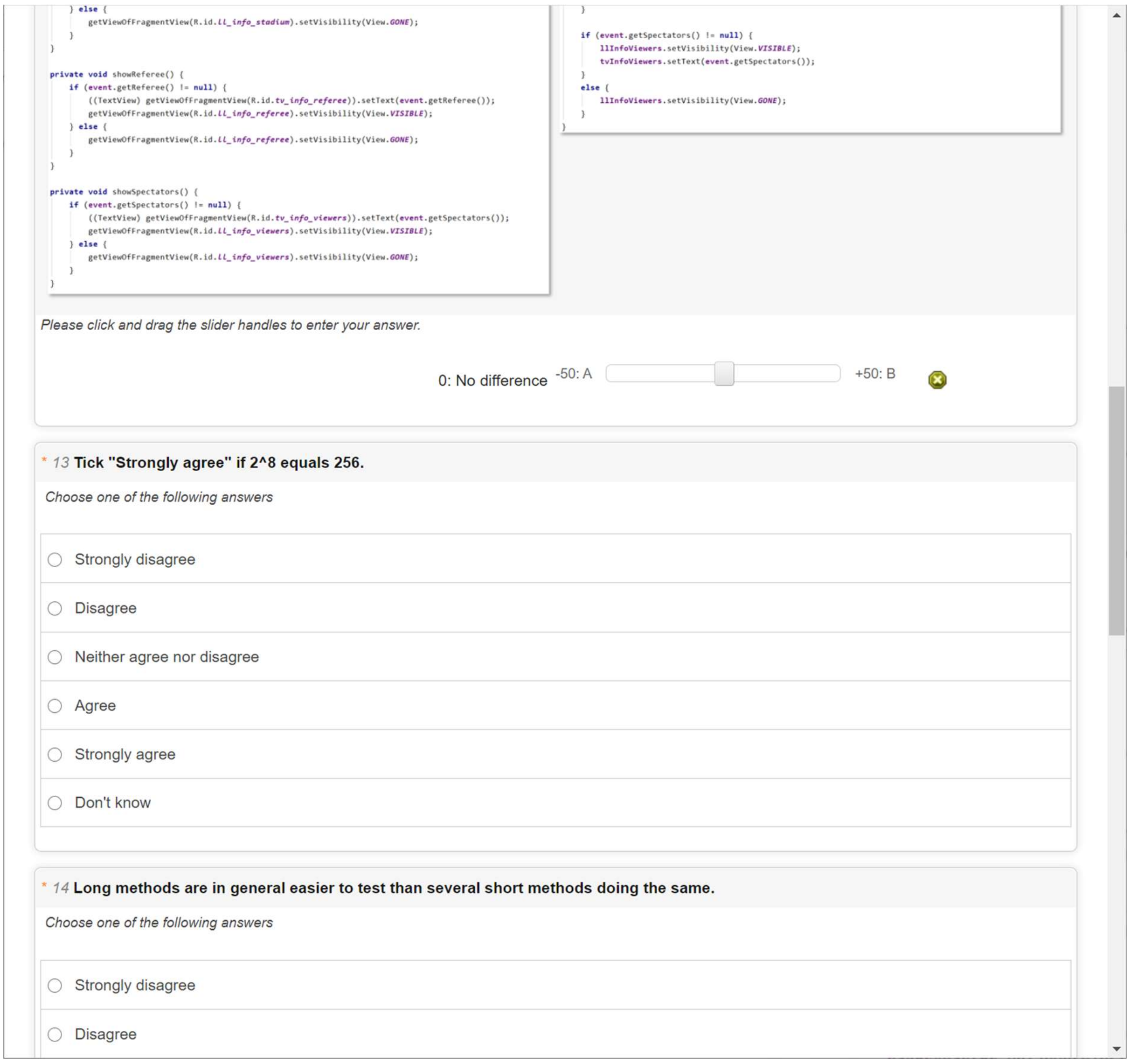
4.2 Community-based Evaluation of Code Complexity

For the subjective evaluation of complexity, we chose a community-based evaluation technique, which means that the participants need to have experience in programming. This precondition was asked before starting the questionnaire. We decided to use *LimeSurvey*[49], which can be freely used for students with an account for *KFU Online*[50]. This online survey tool enables creating questions of different types, like numerical or textual, and can be sent out online (Apdevries, 2020; Cdorin, 2018). The printed questionnaire can be seen in the Appendix and is described in the next section.

4.2.1 Survey Preparation and Execution

The questionnaire consists of two different question types. On the one hand, a five-point Likert scale was used to determine the preferences of the participants regarding the corresponding statement (Likert, 1932). The possible answers are "Strongly disagree", "Disagree", "Neither agree nor disagree", "Agree", "Strongly agree", and "Don't know". On the other hand, there are listed some code snippets or file name lists extracted and adapted from the real repository of the mentioned project, which needs to be rated with the help of a numerical slider which ranges from -50 to 50 and starts at zero. The shown code was asked to be used for the questionnaire. To evaluate improvements or confirm some design and refactoring decisions made during the implementation explained above, there are always compared situations without conducted refactorings, with the same situations in which the item was refactored. Indeed, the listings do not match the history of the repository exactly but are slightly adapted to match the desired state. This suffices for recognizing a trend for the various questions by this subjective evaluation. The listings to compare were arbitrarily set to left or right and marked with "A" and "B" accordingly. The user is given the information that the default value zero means "No difference" between both examples. Negative values, moving the slider to the left, means preferring the example marked with "A" with the extreme value of -50. Moving the slider to the right means preferring listing "B" with the extreme value of 50, respectively. There is also included a question that serves as an attention test to check for the trustworthiness of the participants. In case a participant does not tick "Strongly agree" for the statement "Tick 'Strongly agree' if 2^8 equals 256.", the corresponding questionnaire gets discarded. In Figure 43 an excerpt of the online questionnaire is presented. It shows a question with a slider, the question for checking the attention of the participant, and an excerpt of a standard Likert question.

[49] https://www.limesurvey.org/ [accessed on 4 December 2020]
[50] https://online.uni-graz.at/ [accessed on 4 December 2020]



```
    } else {
        getViewOfFragmentView(R.id.ll_info_stadium).setVisibility(View.GONE);
    }
}

private void showReferee() {
    if (event.getReferee() != null) {
        ((TextView) getViewOfFragmentView(R.id.tv_info_referee)).setText(event.getReferee());
        getViewOfFragmentView(R.id.ll_info_referee).setVisibility(View.VISIBLE);
    } else {
        getViewOfFragmentView(R.id.ll_info_referee).setVisibility(View.GONE);
    }
}

private void showSpectators() {
    if (event.getSpectators() != null) {
        ((TextView) getViewOfFragmentView(R.id.tv_info_viewers)).setText(event.getSpectators());
        getViewOfFragmentView(R.id.ll_info_viewers).setVisibility(View.VISIBLE);
    } else {
        getViewOfFragmentView(R.id.ll_info_viewers).setVisibility(View.GONE);
    }
}
```

```
    }

    if (event.getSpectators() != null) {
        llInfoViewers.setVisibility(View.VISIBLE);
        tvInfoViewers.setText(event.getSpectators());
    }
    else {
        llInfoViewers.setVisibility(View.GONE);
    }
}
```

Please click and drag the slider handles to enter your answer.

0: No difference   -50: A   +50: B

* 13 **Tick "Strongly agree" if 2^8 equals 256.**

Choose one of the following answers

- ○ Strongly disagree
- ○ Disagree
- ○ Neither agree nor disagree
- ○ Agree
- ○ Strongly agree
- ○ Don't know

* 14 **Long methods are in general easier to test than several short methods doing the same.**

Choose one of the following answers

- ○ Strongly disagree
- ○ Disagree

Figure 43: Screenshot of the survey giving a good general overview. It includes a question with a slider, the "x" button resets the value to zero, the question for testing the attention of the participant, and another question of type Likert.

The questionnaire consists of twenty different questions grouped by four pages of five questions each, whereby question number 13 serves as the attention test described above. All questions need to be filled out to continue, whereby the five numerical slider questions are preselected by the default value 0. The 14 Likert scale questions are not preselected. Some questions are grouped, some are not, and sometimes questions are negated consciously to increase the variation of answers. In Table 14 the most important aspects of the questionnaire are shown with a description of the intention of asking every single question.

No.	Code	Type	Intention of Question
1	A1	Likert	Importance of naming convention in software project
*2	A2	Likert	Importance of naming convention for files
*3	A3	Slider	Rating for implemented naming convention for files
4	A4	Slider	Rating for implemented naming convention for view related code
*5	A5	Likert	Importance of always prefixing class member names
*6	B1	Likert	Attitude about cleanliness of long methods
*7	B2	Likert	Attitude about extracting method instead of comment
*8	B3	Likert	Attitude about repeated switch statements and polymorphism
9	B4	Likert	Attitude about long methods and testing (code coverage)
10	B5	Likert	Attitude about statement that complex code requires always more effort
11	C1	Likert	Attitude about extracting method for simplifying long methods
12	C2	Slider	Rating for inlined view variables
13	C3	Likert	Attention test
*14	C4	Likert	Attitude about long methods and testing (more effort)
*15	C5	Slider	Rating for implemented polymorphism instead of switch statements
*16	D1	Slider	Rating for extracted methods for simplifying long methods
*17	D2	Likert	Attitude about code refactoring if it always pays off
18	D3	Likert	Attitude about code complexity and testing effort
19	D4	Likert	Attitude about link between maintenance costs and understandability
20	D5	Likert	Attitude about enforcing code refactoring to improve code quality

Table 14: A list of all 20 questions with corresponding code, type, and a short description about what the intention of the individual question is. An asterisk "" marks questions which are negated according to our opinion.*

The survey was sent out on 29 November 2020 and ended on 1 December 2020. The automatically created URL[51] was sent out targeted to programmers via social media channels and email. Participants were asked to send the survey to other software developers. The participants were only said that the master's thesis is about code complexity, code of a Java Android application, and that the survey aims to find out what opinion the software engineering community has about it. There is no time limit set, but the participant gets informed that the questionnaire takes up to ten minutes. The survey was conducted completely anonymously. This introduction can also be seen in the Appendix.

The threshold for stopping the survey was an amount of 50 valid questionnaires, which means that they need to be filled out completely and question 13 is ticked correctly. It was manually stopped at an amount of 62 valid questionnaires. In about two days, the questionnaire was started 83 times in total and 13 of them were not completed as summarized in Table 15.

51 https://survey2.edu.uni-graz.at/778328/lang-en [accessed on 4 December 2020]

Started Questionnaires in Total	83
Completed Questionnaires	70
Valid Questionnaires	62

Table 15: Statistics about the available questionnaires of the survey which was active for about two days.

The dataset was exported via a web page in various file formats, whereby the Microsoft Excel Binary File Format (XLS)[52] was used for further evaluation in Microsoft Excel explained below.

4.2.2 Evaluation of the Survey

Firstly, after data collection, we cleaned the data by removing all 13 incomplete questionnaires. Moreover, we discarded those eight questionnaires where the attention test failed, and the question for checking that, question C3, also. Then, we extended each question by the information, whether it is of type slider or Likert, and whether its question text or statement is negated or not. The answers are automatically encoded in values 0 ("Don't know"), and 1 ("Strongly agree") to 5 ("Strongly disagree"). To handle answers of type "Don't know" correctly, we exchanged zero values by blank cells. By this, the value is automatically not included in various calculations explained below. Next, we inverted all answers whose questions were negated regarding the expected outcome. That means, that value 5 gets 1, value 4 gets 2, and vice versa, and for slider values, the numbers get inverted. Furthermore, slider values are shifted by 50 to range from zero to 100.

Now, we conducted a combination of confirmatory data analysis (CDA), and exploratory data analysis (EDA) (Turkey, 1977). On the one hand, we check our expectation of having as low results as possible for all questions after cleaning and preparing the data, but, on the other hand, we also search for new patterns the data can give us by examining it exploratory (Martinez, Martinez, & Solk, 2010).

First, we calculated the average ("AVG"), standard deviation ("STD"), median ("MED"), and mode ("MOD") for the answers to each question as can be seen in Table 16. As the mode for answers to slider questions is not as interesting as in which section the most answers are given, the according section is listed instead. The sections are 0 to 19, 20 to 39, 40 to 60, 61 to 80, and 81 to 100. We set the third section equally to an answer "Neither agree nor disagree" in Likert scale questions, which is encoded by 3, in further considerations if necessary.

52 https://docs.microsoft.com/en-us/openspecs/office_file_formats/ms-xls/ [accessed on 4 December 2020]

id	A1	A2	A3	A4	A5	B1	B2	B3	B4	B5	C1	C2	C4	C5	D1	D2	D3	D4	D5
1	1	1	0	1	3	1	4	2	1	1	1	2	1	0	4	1	3	2	2
2	1	2	0	21	2	1	2	2	1	1	1	0	1	0	12	2	3	2	1
3	1	1	0	0	4	1	3	1	2	1	1	50	1	0	0	3	2	2	1
4	1	1	0	30	2	1	3	2	2	1	1	0	2	10	35	2	1	3	2
5	4	4	0	25	4	2	2	2	1	2	2	0	2	20	10	4	1	2	1
6	2	2	19	9	5	2	3	2	2	1	1	3	1	15	11	4	2	1	1
7	3	1	31	62	2	1	2	3	1	4	2	78	2	100	55	2	4	4	4
8	2	2	0	0	5	1	4	1	1	1	2	0	1	0	0	3	2	1	2
9	1	1	25	10	4	1	4	3	2	1	1	22	2	22	21	4	4	2	2
11	1	1	0	0	4	2	3	2	3	2	2	4	2	0	0	4	3	3	2
13	1	1	0	0	5	1	5	1	1	1	1	93	4	0	100	2	1	2	1
14	1	1	24	75	3	1	2		2	1	1	81	1	0	0	4	2	1	2
15	1	1	0	0	2	1	2	2	2	1	1	0	1	18	0	1	4	2	1
16	5	1	0	100	2	1	3	1	1	2	1	0	1	75	100	4	1	1	1
17	1	1	0	0	5	5	5	1	1	1	2	0	1	0	0	1	2	1	1
21	1	1	0	0	2	1	1	1	1	1	1	100		0	100	2	2	1	1
22	1	1	0	75	4	2	2	1	2	2	1	0	1	0	0	2	2	2	1
23	1	1	0	0	2	2	2	1	2	2	1	0	2	17	19	3	2	2	4
25	1	1	0	0	4	2			2	2	1	60	1	10	25	3	3	2	2
28	1	2	0	0	4	2	2	2	2	1	2	28		20	11	2	4	2	2
29	2	1	0	25	5	2		2	1	1	2	66	3	35	70	4	3	1	2
30	2	1	15	0	4	2	4	2	2	2	2	24	2	16	22	2	3	2	2
32	1	2	0	17	4	3	4	2	2	2	2	0	3	20	31	4	2	3	3
33	2	1	20	22	3	1	4	2	1	2	1	9	3	32	23	2	3	1	1
35	1	1	5	100	5	1	5	5	1	1	1	0	1	100	8	5	1	1	1
36	1	1	75	40	4	2	1	1	2	1	1	0	2	0	0	2	4	2	1
37	1	1	35	65	2	1	1	2	1	1	1	15	5	35	45	4	2	1	2
38	1	1	0	0	5	3	3	2	2	2	1	0	3	100	0	3	1	2	2
40	5	1	10	10	4	1	4	1	2	2	2	17	2	0	20	2	2	1	2
41	1	1	0	0	5	1	5		1	1	1	100	1	50	0	4	1	1	1
45	1	1	0	0	3	2	4	2	2	3	2	65	3	50	50	3	5	2	3
46	2	2	0	0	2	2	3	2	1	3	1	28	2	81	40	2	2	3	2
47	1	3	10	15	4	2	3	2	2	1	1	25	2	10	15	4	3	1	2
49	5	1	74	0	5	3	2	3	1	1	2	0	2	21	8	3	2	3	1
50	1	1	0	0	4	2	5		2	2	2	35	3	50	40	4	2	3	2
51	2	2	25	25	4	2	3	4	2	2	2	35	3	25	30	3	2	2	2
55	1	1	5	100	2	1	2	1	2	3	1	0	1	2	2	2	4	5	3
56	2	2	17	16	2	3	4	1	1	2	2	11	3	9	30	4	1	4	4
57	2	2	0	0	3	3	3	4	1	4	2	0	2	0	76	2	2	3	2
58	2	2	0	0	2	5	2	2	2	2	1	0	1	0	0	1	2	2	1
59	1	2	0	0	3	2	2		4	1	2	19	2	0	27	2	2	2	2
60	1	1	20	90	3	1		1	2	1	1	6	1	25	9	2	1	1	1
61	2	2	29	13	3	2	3	1	1	2	2	20	1	14	10	3	2	1	2
62	2	2	0	20	3	1	3	2	1	1	1	0	2	0	0	4	2	1	1
63	2	2	0	0	5	3	5	1	3	2	2	0	2	10	50	2	3	2	2
64	5	1	60	25	4	2	2	2	2	2	2	35	1	20	20	2	3	2	1
65	1	2	11	0	3	2		2	2	2	2	60	3	25	0	3	2	4	2
66	5	1	0	0	4	1	1	2	1	1	2	0	1	0	0	2	2	1	2
67	2	2	0	10	3	2	2	2	2	1	2	20	2	50	80	2	1	2	2
68	1	3	80	75	4	2	5	3	2	1	1	25	4	10	10	2	3	2	1
69	1	1	20	17	3	3		1	2	2	1	10	1	0	21	4	3	2	1
70	1	1	50	66	4	2	2	1	1	2	1	4	2	18	0	4	3	2	3
71	2	1	0	13	4	2	1	1	1	1	2	15	2	0	20	2	1	2	1
76	2	1	0	65	4	2	4	2	1	2	4	30	4	0	25	4	2	2	2
77	3	3	29	63	4	2	3	2	3	2	2	50	2	26	38	4	2	2	2
78	1	1	0	0	4	3	4	1	2	1	1	0	2	0	0	2	4	2	2
79	2	2	20	20	3	3	4	3	1	1	2	0	2	15	20	3	2	3	4
81	2	2	0	0	4	3	4	2	2	2	2	0	2	0	0	3	2	2	2
83	1	2	0	20	5	2	2	1	1	1	2	25	2	0	20	2	2	2	1
84	5	1	9	11	1	1	2	1	1	1	1	2	2	36	67	4	2	1	1
85	1	1	0	0	4	2		1	1	2	2	0	2	0	21	4	2	2	2
86	1	1	0	0	3	3	5		2	1	4	100	4	83	0	3	4	2	

AVG	1.79	1.45	11.58	21.79	3.52	1.92	3.04	1.82	1.63	1.6	1.56	22.13	2	20.56	23.4	2.84	2.34	1.98	1.8
STD	1.22	0.66	19.64	29.83	1.04	0.9	1.21	0.87	0.65	0.73	0.66	29.26	0.95	26.98	26.67	1	0.97	0.87	0.83
MED	1	1	0	10	4	2	3	2	2	1	1.5	9.5	2	12	19.5	3	2	2	2
MOD	1	1	<20	<20	4	2	2	2	2	1	1	<20	2	<20	<20	2	2	2	2

?	0	0	0	0	0	0	6	6	0	0	0	0	2	0	0	0	0	0	1
1	35	39	46	39	1	22	5	22	28	32	31	37	20	38	31	4	11	18	24
2	18	19	11	10	12	27	17	26	30	25	29	13	26	14	18	24	29	32	29
3	2	3	2	1	14	11	13	5	3	3	0	4	9	4	6	13	13	8	4
4	1	1	3	8	24	0	13	2	1	2	2	3	4	1	4	20	8	3	4
5	6	0	0	4	11	2	8	1	0	0	0	5	1	5	3	1	1	1	0

Table 16: The cleaned data including calculations of average, standard deviation, median, and mode. The mode of questions of type slider is meant to be that section where the most values are included (here always the section below 20). On the bottom the occurrences of answers for each question are presented. For Likert questions these are "?", which represents "Don't know", and 1-5, for slider questions these are 1 for 0-19, 2 for 20-39, 3 for 40-60, 4 for 61-80, and 5 for 81-100.

On the bottom of Table 16, the occurrences of answers for each question are presented, including "Don't know" answers, indicated by "?", for Likert scale questions. As already mentioned above, for Likert scale questions the number of occurrences for answers given in the described sections are counted, where 1 refers to section 0 to 19, 2 to 20 to 39, 3 to 40 to 60, 4 to 61 to 80, and 5 to 81 to 100. A boxplot of all questions is shown in Figure 44, whereby the section values 1 to 5 are used for slider questions. As a result, the graphical representation is more consistent.

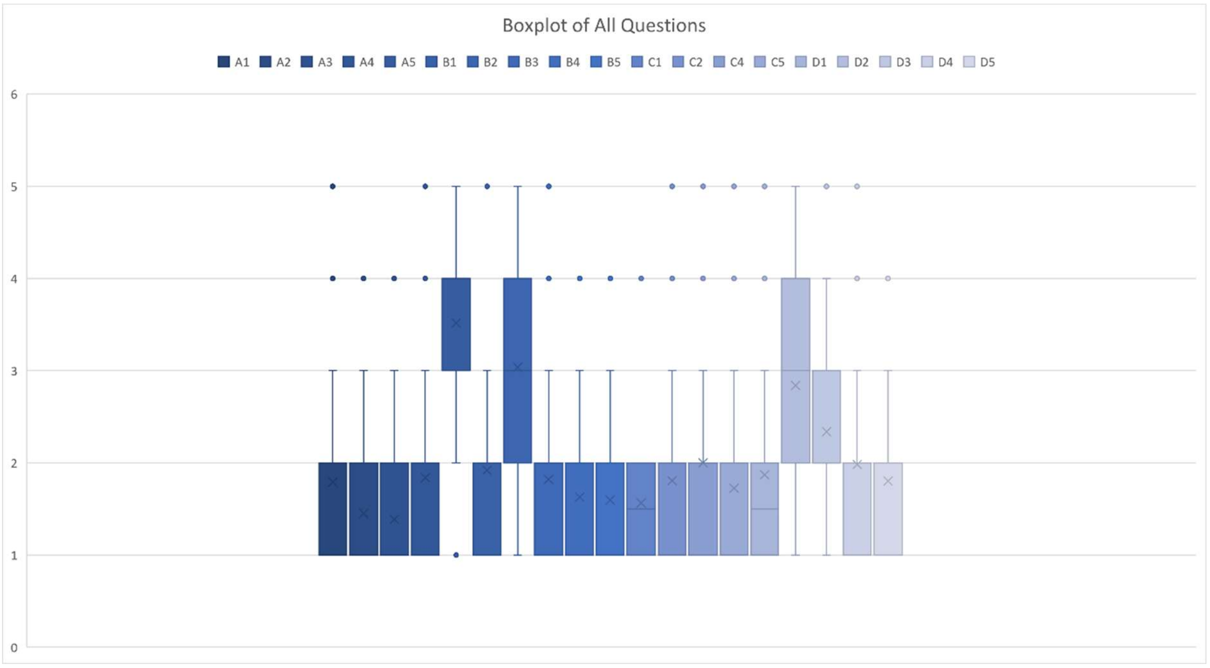
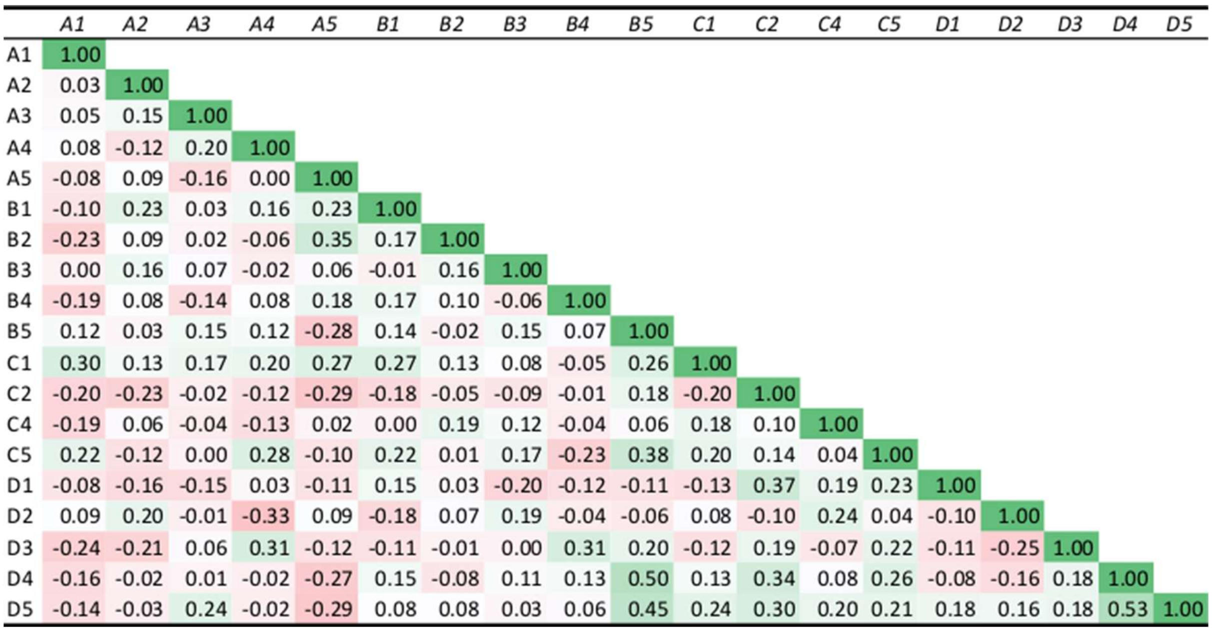


Figure 44: Boxplot of the results for all questions, whereby the section values 1-5 are used for slider questions for easier comparison.

Next, we calculated the correlation between the questions, which we present in a correlation matrix including the values shown in Figure 45, and the correlation between the participants, which is only shown by a heatmap in Figure 46. Green fields indicate a positive correlation, and red fields indicate a negative correlation. To calculate the values, we used Analysis ToolPak[53] by Microsoft[54] as an add-in in Microsoft Excel. For this calculation, we removed any questionnaires containing a "Don't know", which are in total 13 entries.

[53] https://support.microsoft.com/en-us/office/load-the-analysis-toolpak-in-excel-6a63e598-cd6d-42e3-9317-6b40ba1a66b4 [accessed on 5 December 2020]
[54] https://www.microsoft.com/en-us/ [accessed on 5 December 2020]



	A1	A2	A3	A4	A5	B1	B2	B3	B4	B5	C1	C2	C4	C5	D1	D2	D3	D4	D5
A1	1.00																		
A2	0.03	1.00																	
A3	0.05	0.15	1.00																
A4	0.08	-0.12	0.20	1.00															
A5	-0.08	0.09	-0.16	0.00	1.00														
B1	-0.10	0.23	0.03	0.16	0.23	1.00													
B2	-0.23	0.09	0.02	-0.06	0.35	0.17	1.00												
B3	0.00	0.16	0.07	-0.02	0.06	-0.01	0.16	1.00											
B4	-0.19	0.08	-0.14	0.08	0.18	0.17	0.10	-0.06	1.00										
B5	0.12	0.03	0.15	0.12	-0.28	0.14	-0.02	0.15	0.07	1.00									
C1	0.30	0.13	0.17	0.20	0.27	0.27	0.13	0.08	-0.05	0.26	1.00								
C2	-0.20	-0.23	-0.02	-0.12	-0.29	-0.18	-0.05	-0.09	-0.01	0.18	-0.20	1.00							
C4	-0.19	0.06	-0.04	-0.13	0.02	0.00	0.19	0.12	-0.04	0.06	0.18	0.10	1.00						
C5	0.22	-0.12	0.00	0.28	-0.10	0.22	0.01	0.17	-0.23	0.38	0.20	0.14	0.04	1.00					
D1	-0.08	-0.16	-0.15	0.03	-0.11	0.15	0.03	-0.20	-0.12	-0.11	-0.13	0.37	0.19	0.23	1.00				
D2	0.09	0.20	-0.01	-0.33	0.09	-0.18	0.07	0.19	-0.04	-0.06	0.08	-0.10	0.24	0.04	-0.10	1.00			
D3	-0.24	-0.21	0.06	0.31	-0.12	-0.11	-0.01	0.00	0.31	0.20	-0.12	0.19	-0.07	0.22	-0.11	-0.25	1.00		
D4	-0.16	-0.02	0.01	-0.02	-0.27	0.15	-0.08	0.11	0.13	0.50	0.13	0.34	0.08	0.26	-0.08	-0.16	0.18	1.00	
D5	-0.14	-0.03	0.24	-0.02	-0.29	0.08	0.08	0.03	0.06	0.45	0.24	0.30	0.20	0.21	0.18	0.16	0.18	0.53	1.00

Figure 45: Correlation between questions with rounded values without duplicated values.

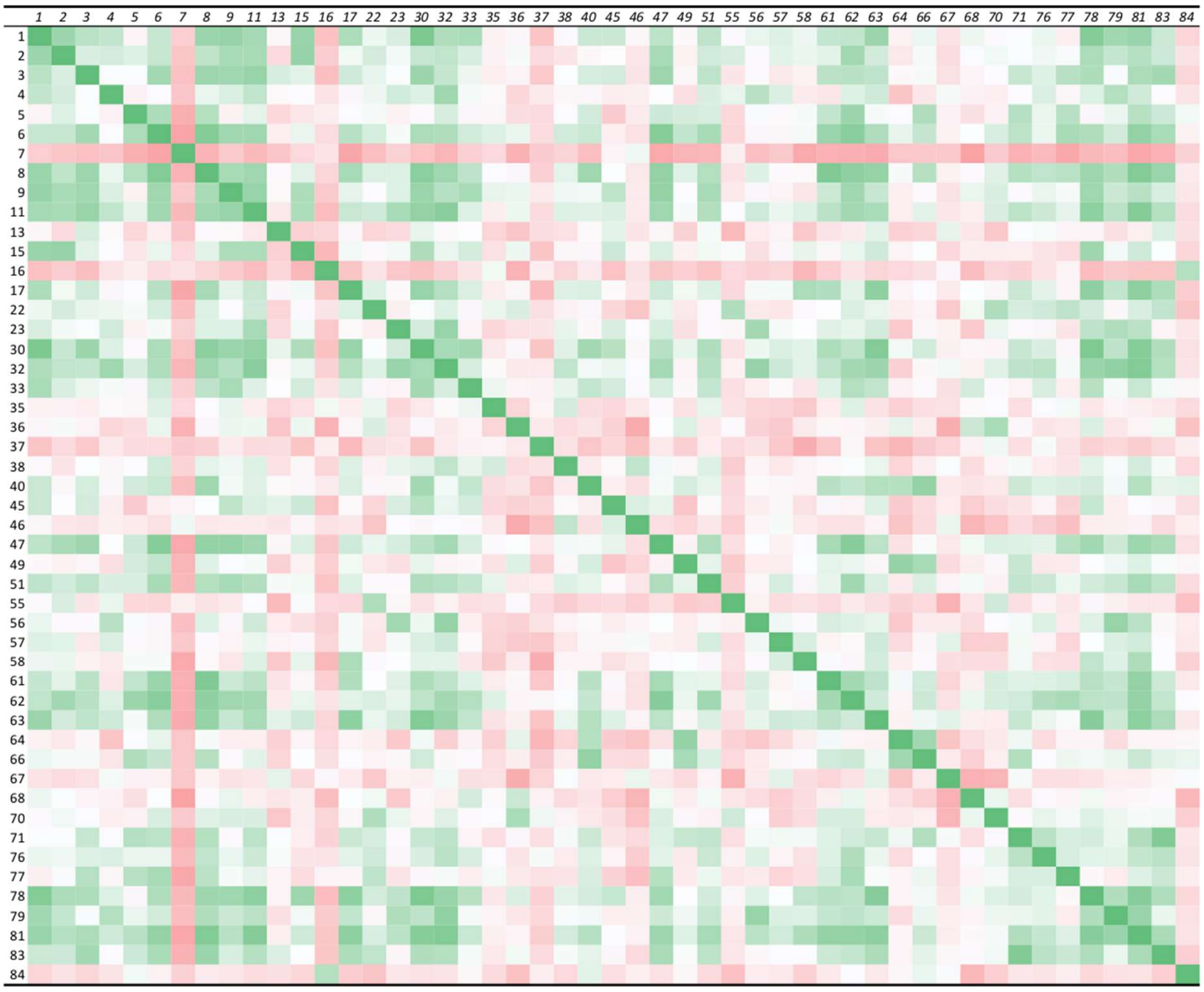


Figure 46: Heatmap created by the correlation matrix of the participants of the survey.

To compare the overall conformity with the statements made, we summarized all answers made including “Don’t know” answers also and plotted a pie chart presented in Figure 47.

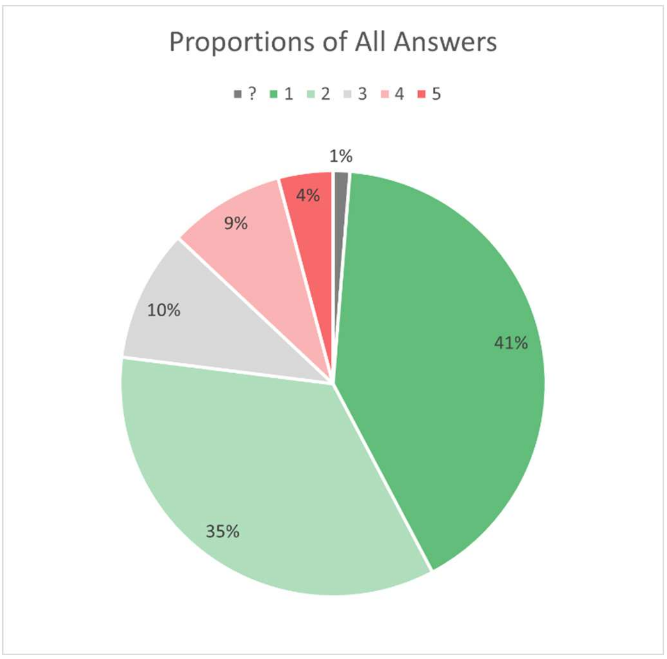


Figure 47: This pie chart shows the proportions of all answers made including "Don't know" marked as “?”. Agreement to the statements is indicated by green, disagreement is coloured red. The grey section shows a quite neutral opinion.

Finally, we used a stacked column diagram to display the opinion of the community to each question in Figure 48.

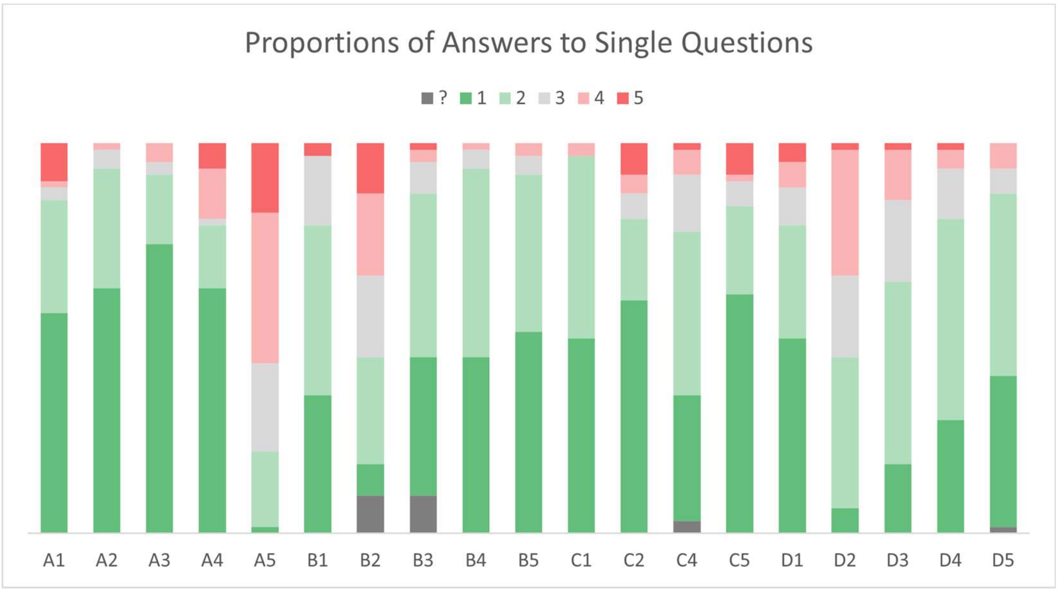


Figure 48: Stacked column diagram to show the answers made to each question.

4.3 Discussion of the Results

Throughout the whole two years, the main focus of the project was to add features. Figure 42 shows that all total complexity metrics of the project follow the same trend. While the number of files and the lines of code get more, the total complexity gets also more. They all follow the same trend. Regarding the average for method complexity metrics, at the beginning of our evaluation, the average complexity of the methods and classes was reduced. However, when looking at the complexity trend for the following snapshots, the complexity is raised in general except for OCavg. Nevertheless, after that, the complexity is reduced in version 6 for all complexity metrics including number of files and lines of code. As Figure 39 shows, there was started code refactoring actively, and the unused resources were removed. The greatest reduction of complexity, however, can be recognized in the last time interval from version 6 to version 7. Even if the total complexity values and the number of files raise, not only all average metrics scores shrink as fast as never before, but also the lines of code get lower. The reduction of complexity is even for OCavg the strongest. When comparing the trend of the total complexities from version 2 to version 5, it is nearly equal to that from version 6 to version 7. However, while the average complexity values have at version 5 their maximum, except for OCavg, that from version 6 to version 7 falls as strong as never before and reaches everywhere the minimum. So, the overall complexity was reduced extremely.

An overview of the different trends in average complexities gives Figure 41. There it can be seen at a glance how strong the complexity is reduced from version 6 to version 7 and that they reach the minimum in the end.

When viewing the MOOD metrics in Figure 49 of the Appendix, it can be seen that the development team managed to improve polymorphism which coincides with the implementation of introducing many inherited classes. This can be seen by comparing version 6 with version 7. With the knowledge given in 4.1.1, we can say, that we got a lower *PF*, which is preferable, a higher *MHF*, which means abstraction was improved but the functionality was lowered, and finally, a higher *MIF* and *AIF,* which means that inheritance was increased. However, the *AHF* is reduced further which should be noticed as the attributes should be hidden optimally 100%. A solution for that would be the refactoring technique *Encapsulate Variable* to change all protected member variables to private and implementing getter and setter methods, instead. As a result, *AHF* and *MIF* would be increased while *AIF* and *MHF* would be decreased as explained above. So, this solution was taken wrongly according to the values. The *CF* was improved by the code changes as the value is lower now. However, some of the decisions regarding inheritance seem to be made too extreme.

Regarding the survey, a variety of answers and opinions is given as can be seen in Figure 46 showing the heatmap of the correlation between the different questionnaires. However, there are some people included who have a completely different opinion, like IDs 7, 16, 37, or 84. However, as there are many green fields also, there are many people who share the same thoughts. If looking at Figure 47 and Figure 48, respectively, it can be clearly seen that the community answered the questionnaire according to our ideas mostly. This is also confirmed by the boxplot in Figure 44. However, there are some questions where we expected another outcome. These are questions A5, B2, D2, and D3.

Starting with the most contrary answer, A5 which says, "Class member names should always have a special prefix like 'm' or '_', as for example 'private int mWeight', or 'private String _name'. According to Martin (2008), this question should be answered with "Strongly disagree". However, as this is a subjective question, this is only a suggestion. In C# for example, it is quite common using the "_" as a prefix for private member variables. However, this has no further advantage. According to online resources and discussions, it is suggested to "not use a prefix for member variables (_, m_, s_, etc.). If you want to distinguish between local and member variables you should use "this." in C# and "Me." in VB.NET." (BradA, 2005). It is further explained that the reasons for not using Hungarian notation or prefixes are on the one hand consistency of the code appearance and, on the other hand, clean readable source code, which is also confirmed by other sources (RickHos, 2004). According to Peter Ritchie (2007), classes should be simple and therefore, it is not necessary to enforce using a prefix, it is just important is using meaningful names. Moreover, he says that it is harder being consistent with such a notation than without. However, all in all, it is up to the developers using it or not. In the case of an older codebase where this notation is used already, it should be followed further due to consistency. Regarding the question within the questionnaire, it seems that the community believes using a prefix for member variables is cleaner than not using it, or they just prefer it. The world "always" within the question supports this assumption.

In comparison to the usage of view related and file-related Hungarian notation, questioned in A3 and A4, where the community approved, this seems to be contrary to the statement of not using prefixes for member variables. However, the intention of prefixing files is the automatic sorting by *Android Studio* and hence the possibility of grouping the files. Furthermore, private member variables are only visible in one file. Classes should be as simple as possible, and not having that much member variables, which was also confirmed by the community answering question C2. So, the scope of such variables is limited. However, regarding the usage of view related Hungarian notation, it is an advantage for faster finding identifiers which are not within the same file. In case we know we want to set the title of a news article, we firstly type "tv_" for meaning TextView and using then a descriptive name like "title". Due to consistency, "tvTitle" seems the right choice. The same idea for scoping can also be said for drawable resources. All in all, it is important to have a naming convention, confirmed by A1, and A2, the used Hungarian notation for view related code and files seems to be the right choice, but the naming of member variables should be discussed further with the developers of the project due to the contrary opinion of the community.

The next question whose result is different as expected is question B2 with an average of about 3. Even if most of the participants have answered "Disagree" to the statement "Clean code includes comments explaining the code. If a method was extracted before, the code of it should be better inlined back into the calling method! After that, a comment should explain the inlined code.", eight of them strongly agree to that, 13 agree, and further 13 neither agree nor disagree. The result of this statement, which was asked in inverted logic, seems to be contrary to the results given in D1, where 31 participants strongly confirmed the version where methods were extracted from the body of a long method. Moreover, the low correlation of 0.13 to C1, given in Figure 45, with the statement "It is often hard to understand a long method. Extracting parts of it to new methods is a good idea as the new meaningful method names can explain the extracted parts if chosen well.", also contradicts the result in B2. However, some of the participants may be detracted from the extraction of the method by the first sentence saying that clean code includes comments explaining the code. Despite this, not only this opinion but also the opinion to extract method is contrary to the clean code referring to Martin (2008) explained above. As he says, blocks should have at maximum two lines of code, which implies extracting methods, and comments should not be used for documenting code. Descriptive function names should be used instead as tried to show in the listing of question D1. Moreover, according to the complexity measure explained in 4.1.1, the cyclomatic complexity should be lower for the extracted method variant. This hypothesis is confirmed by an experiment made shown in Figure 50 in the Appendix. The cyclomatic method complexity was calculated for both variants. As can be seen, the values for *ev(G)*, *iv(G)*, and *v(G)* are reduced for the average of the whole class by only extracting methods in this single long method from 1.13, 1.3, 1.4 to 1.09, 1.19, and

1.26, respectively. This means a reduction by 3.54%, 8.46%, and 10% for the corresponding metrics for the whole class. To end with, the implementation seems to be the right choice, leading to the assumption that the community build their opinion upon the statement "Clean code includes comments explaining the code.". This should have been questioned separately to check their opinion about the usage of comments within code.

"The effort of code refactoring does not always pay off." was stated D2 in this negated question compared to our assumption. As displayed in Figure 48, the opinion of the participants is not that extreme, as only four participants strongly disagree, and only one strongly agrees here. However, as can be seen in the heatmap in Figure 46, this participant with ID 35, taken from Table 16, seems to be quite contrary to the great portion of other participants, so, this should not be taken into account in a great manner. However, as the extreme wording "always" is included in the statement, this inconclusive result is perspicuous. All in all, 28 people disagree with the statement, while 21 agree, so the majority has voted for the more thoughtful answer.

This question can be compared to other questions about code refactoring, maintainability, quality, and code complexity. Examples are B5 with the statement "Complex code that is hard to understand needs always more time for maintenance and bug fixing.", D4, "There is a clear link between maintenance costs and understandability of code.", and D5, "I think refactoring is an important task in software engineering and should be done regularly to increase the quality of the code.". Looking again at Figure 45, these three questions correlate with each other (D5-D4 by 0.53, D4-B5 by 0.5, and D5-B5 by 0.45). Hence, we can say that the community thinks that complex code with low understandability leads to more maintenance effort, according to B5, that they think that maintainability and understandability are related clearly, approved in D4, and that code refactoring is an important task to increase code quality. Hence, they agree that code refactoring helps reducing code complexity and as a result, the maintenance effort and the maintenance costs shrink in the end. On the other hand, we can conclude that the community says that code, which is hard to understand is complex, and in case of complex code, the maintenance effort increases. As they voted that the refactored versions of C2, C5, D1 are better regarding the asked quality attributes for code complexity, and these questions correlate with D5, the corresponding values are 0.3, 0.21, and 0.18, we can say that the refactoring decisions seem to be taken appropriately.

The average of D3 "Code complexity is related to the testing effort." with 2.34, displayed in Table 16, seems to be quite high, so it is not that clear. However, those who confirmed this statement, also confirmed with a correlation of 0.31 the statement B4 "The more possible paths there are within a method, the more tests are required to achieve 100% code coverage in testing.". So, the community confirms that a higher testing effort is related to code complexity which is further related to more paths within a method, which means that long methods are more complex and hence, they are harder to test. This fits the cyclic complexity definition presented in Table 12 meaning the higher it is, the more paths are within the control flow graph, and the more tests are needed to reach a code coverage of 100% in testing. Interestingly, C4 asking for the greater testing effort of long methods in comparison to short methods, agreed by the community with an average of 2. However, question B1 asking for the cleanness of long methods seems to be not answered that clearly as "cleanness" is a subjective word. But, as the average of 1.92, and the median and mode of 2 each show, as displayed in Table 16, they tend to agree to the statement. B1 however, correlates with C1 by 0.27 where it is meant that it is a good idea extracting methods and choosing well-defined names for the extracted methods for documentation reasons. So, this seems to be appropriate.

The last interesting correlation is that between B3 asking for the attitude about using polymorphism instead of using repeated switch statements with the same cases, and the implementation of an abstract class factory to replace these code duplications asked by the question C5 of type slider. The correlation given in Figure 45 of 0.17 seems to be quite low, but it is not contrary, though. The answers of each are in line with our intuition, so the average values of both are quite low, being 1.82 for B3, and 20.56 for C5. This means, that the community confirms not only the theory described above but also our decision to introduce that pattern during the implementation while refactoring as shown for example in Listing 14.

To conclude, the averaged results of the survey are in line with the theoretical discussed issues, and the implemented refactorings and decisions made for this project.

5 Conclusion

The main goal of this master's thesis was to explore an Android application for complex parts, measuring it by complexity metrics, and to find ways to reduce its complexity by well-known refactoring approaches. Not only the quality but also the understandability should be increased. Furthermore, another goal was finding out a way to evaluate the improvement by a questionnaire that is filled out by software engineers. We found a way to compare both results and saw, that we have taken accepted proposals for cleaning up and refactoring the code. These decisions were accepted not only by the community but also led to great complexity reduction. So, the theoretical explained approaches, techniques, and recommendations, including refactoring, clean code, and working with legacy code, can be used by other software developers to improve the quality of their software in future. In the implementation part of this thesis, we described and applied approaches of refactoring and removing code smells in Android Studio. A naming convention and coding standard, the removal of unused code and resources, and the exchange of raster graphics with vector graphics were implemented and described including showing the results. Regarding the refactoring and clean code, preparatory and comprehension refactoring were used to implement some new features like a collapsible header and a dynamic tabbed layout view, new profile pages for leagues and clubs, a new event view page, new news, and some more.

However, some things need to be considered in the future. Prefixing member variables should be discussed within the developer team. But, as Hungarian notation seems to be also preferred, that should be discussed in future. Consistency should be important. Moreover, according to the MOOD metrics, it seems that sometimes the using of inheritance was too excessively. So, when designing for patterns, these metrics should always be considered during development and checked. Moreover, during the evaluation, we learned that we should all member variables encapsulate even for parent classes to achieve an *AHF* of 100%, which means using “private” instead of “protected”. Regarding the questionnaire, we recognized that the questions should be asked as clear as possible and that they should really ask only for one single thing. A negative example was question B2, where people seemed to be distracted by the first sentence. However, this led us to the idea that we should ask for the opinion of the community regarding the usage of comments for documentation reasons.

5.1 Limitations

Due to limited time resources, we could not cover testing in great detail in this master's thesis. However, as for refactoring testing is really important, even if used safe automatic code refactorings, this should always be done for reasons of clean code. Furthermore, because the approach is only applied to a single application, the results should be considered advisedly but should encourage to repeat such experiments with the same approach. Since the implementation was applied in a real software system, the results are only trends and it cannot be said conclusively, that this refactoring or that clean code was responsible for the reduction of the metrics. However, as the other developers followed adding features to nearly the same degree as the two years before, according to the trend the result can be regarded as correct.

5.2 Future Work

As this subjective evaluation technique of comparing quantitative measures with subjective opinions can be repeated to any time of development, it is interesting, how this evolves when the refactoring is kept up. For future questionnaires, however, further questions regarding clean code should be included like the usage of comments for documenting functionality, the usage of member prefixes in different languages asked separately, or demography would also be interesting. Not only the community is interesting for evaluating the improvements of the code, but also code experts. Hence, another survey can be done by interviewing domain experts of the code and asking them for the evaluation of the refactorings. Additionally, a type of A/B testing for refactored code could bring insights to which parts effectively helped people to progress faster. This should be accompanied by thinking aloud testing to capture the thoughts of the developer. Another variant to see if the productivity increases or decreases, timing issues would also be interesting not only on an experimental level but also in real work life.

Another interesting aspect would be refactoring or rewriting the code from Java to the emerging Kotlin[55] while covering the code with tests and without changing any behaviour.

[55] https://kotlinlang.org/ [accessed on 9 December 2020]

6 Bibliography

Aguiar, A., Restivo, A., Correia, F., Ferreira, H., & Dias, J. (2019). Live Software Development: Tightening the Feedback Loops. *Proceedings of the Conference Companion of the 3rd International Conference on Art, Science, and Engineering of Programming. 22*, pp. 1-6. Genova: Association for Computing Machinery. doi:10.1145/3328433.3328456

Almutairi, A. (2018). A Comparative Study on Steganography Digital Images: A Case Study of Scalable Vector Graphics (SVG) and Portable Network Graphics (PNG) Images Formats. *International Journal of Advanced Computer Science and Applications, 9*(1), 170-175.

Apdevries. (2020, May 19). *Question types - LimeSurvey Manual*. Retrieved December 4, 2020 from Home page - LimeSurvey - Easy online survey tool: https://manual.limesurvey.org/Question_types

Basu, A. (2015). *Software Quality Assurance, Testing and Metrics.* Delhi: PHI Learning Pvt. Ltd.

Bavota, G., De Lucia, A., Di Penta, M., Oliveto, R., & Palomba, F. (2015). An experimental investigation on the innate relationship between. *The Journal of Systems and Software, 107*, 1-14.

Bavota, G., De Lucia, A., Marcus, A., & Oliveto, R. (2014). Automating Extract Class Refactoring: An Improved Method and Its Evaluation. *Empirical Software Engineering, 19*(6), 1617-1664.

Beck, K. (1997). *Smalltalk Best Practice Patterns.* Upper Saddle River: Prentice Hall.

Beck, K., & Andres, C. (2004). *Extreme Programming Explained: Embrace Change* (2nd ed.). Boston: Addison-Wesley Professional.

Beck, K. [KentBeck]. (2012, September 26). for each desired change, make the change easy (warning: this may be hard), then make the easy change [Tweet]. Retrieved December 2, 2020 from Twitter: https://twitter.com/KentBeck/status/250733358307500032

BradA. (2005, January 26). *Internal Coding Guidelines | Microsoft Docs*. Retrieved December 6, 2020 from Developer tools, technical documentation and coding examples | Microsoft Docs: https://docs.microsoft.com/en-us/archive/blogs/brada/internal-coding-guidelines

Brar , H., & Kaur, P. (2015). Differentiating Integration Testing and Unit Testing. *2015 2nd International Conference on Computing for Sustainable Global Development (INDIACom)* (pp. 796-798). New Delhi: IEEE.

Cdorin. (2018, September 2017). *Overview - LimeSurvey Manual*. Retrieved December 4, 2020 from Home page - LimeSurvey - Easy online survey tool: https://manual.limesurvey.org/Overview

Chacon, S., & Straub, B. (2014). *Pro Git* (2nd ed.). Berkely: Apress.

Cheon, Y. (2014). Writing Self-testing Java Classes with SelfTest. *Departmental Technical Reports (CS), Paper 831*. doi:http://digitalcommons.utep.edu/cs_techrep/831

Clark, M. (2006, February 20). *JUnit FAQ*. From JUnit: http://junit.sourceforge.net/doc/faq/faq.htm

CodeMR. (2020). *CodeMR | Static Code Analysis and Software Quality Features*. Retrieved 29 November, 2020 from CodeMR | Measure, analyze, improve software code quality: https://www.codemr.co.uk/features/

Debray, S. K., Evans, W., Muth, R., & De Sutter, B. (2000). Compiler Techniques for Code Compaction. *ACM Transactions on Programming Languages and Systems, 22*(2), 378-415.

Djoudi, L., & Jalby, W. (2018). Automatic Analysis for Managing and Optimizing Performance-Code Quality. *Proceedings of the 2008 Workshop on Static Analysis* (pp. 30-38). Tucson: Association for Computing Machinery. doi:10.1145/1394504.1394508

Du Bois, B., Demeyer, S., & Verelst, J. (2004). Refactoring - Improving Coupling and Cohesion of Existing Code. *11th Working Conference on Reverse Engineering* (pp. 144-151). Delft: IEEE. doi:10.1109/WCRE.2004.33

Farcic, V., & Garcia, A. (2015). *Test-Driven Java Development.* Birmingham: Packt Publishing.

Feathers, M. (2004). *Working Efficiently with Legacy Code.* Upper Saddle River: Prentice Hall.

Figma. (2020). *Design, prototype, and gather feedback all in one place with Figma*. Retrieved November 25, 2020 from Figma: the collaborative interface design tool.: https://www.figma.com/design/

Fowler, M. (2007, June 20). *DesignStaminaHypothesis.* Retrieved November 26, 2020 from martinfowler.com: https://martinfowler.com/bliki/DesignStaminaHypothesis.html

Fowler, M. (2018). *Refactoring: Improving the Design of Existing Code* (2nd ed.). Boston: Addison-Wesley Professional.

Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1994). *Design Patterns: Elements of Reusable Object-Oriented Software* (Vol. 1). Boston: Addison-Wesley.

Google LLC. (2019). Application Fundamentals. *Android documentation for app developers*. Mountain View: Author. Retrieved November 21, 2020 from https://developer.android.com/guide/components/fundamentals

Google LLC. (2019). Layout resource. *Android documentation for app developers*. Mountain View: Author. Retrieved November 17, 2020 from https://developer.android.com/guide/topics/resources/layout-resource#idvalue

Google LLC. (2020). AAPT2. *Android documentation for app developers*. Mountain View: Author. Retrieved November 17, 2020 from https://developer.android.com/studio/command-line/aapt2

Google LLC. (2020). Add multi-density vector graphics. *Android documentation for app developers*. Mountain View: Author. Retrieved November 22, 2020 from https://developer.android.com/studio/write/vector-asset-studio

Google LLC. (2020). Prepare for release. *Android documentation for app developers*. Mountain View: Author. Retrieved November 21, 2020 from https://developer.android.com/studio/publish/preparing

Google LLC. (2020). Resources. *Android documentation for app developers*. Montain View: Author. Retrieved November 17, 2020 from https://developer.android.com/reference/android/content/res/Resources

Griswold, W., & Opdyke, W. (2015). The Birth of Refactoring: A Retrospective on the Nature of High-Impact Software Engineering Research. *IEEE Software, 32*(6), 30-38. doi:10.1109/MS.2015.107

Harrison, R., Counsell, S., & Nithi, R. (1998). An Evaluation of the MOOD Set of Object-Oriented Software Metrics. *IEEE Transactions on Software Engineering, 24*(6), 491-496. doi:10.1109/32.689404

Hennessy, J., & Patterson, D. ([1990] 2007). *Computer Architecture: A Quantitative Approach.* San Francisco: Morgan Kaufmann Publishers.

Hunt, A., & Thomas, D. (2000). *The Pragmatic Programmer.* Reading: Addison Wesley.

IEEE Standards Coordinating Committee. (1990). IEEE Standard Glossary of Software Engineering Terminology. *IEEE Std 610.12-1990*, 1-84. doi:10.1109/IEEESTD.1990.101064

JetBrains. (2020, September 21). *Implementing a Parser and PSI / IntelliJ Platform SDK DevGuide*. Retrieved December 3, 2020 from JetBrains: Essential tools for software developers and teams: https://jetbrains.org/intellij/sdk/docs/reference_guide/custom_language_support/implementing_parser_and_psi.html

Kafura, D., & Reddy, G. (1987). The Use of Software Complexity Metrics in Software Maintenance. *IEEE Transactions on Software Engineering, SE-13*(3), 335-343. doi:10.1109/TSE.1987.233164

Kataoka, Y., Imai, T., Andou, H., & Fukaya, T. (2002). A Quantitative Evaluation of Maintainability Enhancement by Refactoring. *International Conference on Software Maintenance* (pp. 576-585). Montreal, Quebec: IEEE. doi:10.1109/ICSM.2002.1167822

Kaur, K., Minhas, K., Mehan, N., & Kakkar, N. (2009). Static and Dynamic Complexity Analysis of Software Metrics. *World Academy of Science, Engineering and Technology, 3*(8), 1936-1938.

Kerievsky, J. (2005). *Refactoring to Patterns.* Boston: Addison-Wesley.

Kim, M., Zimmermann, T., & Nagappan, N. (2012). A Field Study of Refactoring Challenges and Benefits. *Proceedings of the ACM SIGSOFT 20th International Symposium on* *the Foundations of Software Engineering. 50*, pp. 1-11. New York: Association for Computing Machinery.

Kuchana, P. (2004). *Software Architecture Design Patterns in Java.* Boca Raton: CRC Press LLC.

Legowski, G. (1996, September 18). *Hungarian Notation*. Retrieved December 6, 2020 from DTU Compute: http://www.imm.dtu.dk/~alan/hungarian.html

Leijdekkers, B., & Sixth and Red River Software. (2020, April 18). *MetricsReloaded - Plugins | JetBrains*. Retrieved November 27, 2020 from Plugins | JetBrains: https://plugins.jetbrains.com/plugin/93-metricsreloaded

Leitch, R., & Stroulia, E. (2003). Assessing the Maintainability Benefits of Design Restructuring Using Dependency Analysis. *Proceedings. 5th International Workshop on Enterprise Networking and Computing in Healthcare Industry (IEEE Cat. No.03EX717)* (pp. 309-322). Sydney: IEEE. doi:10.1109/METRIC.2003.1232477

Likert, R. (1932). A Technique for the Measurement of Attitudes. *Archives of Psychology, 22*(140), 1-55.

Linares-Vásquez, M., Cortés-Coy, L., Aponte, J., & Poshyvanyk, D. (2015). ChangeScribe: A Tool for Automatically Generating Commit Messages. *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, *2* (pp. 709-712). Florence: IEEE. doi:10.1109/ICSE.2015.229

Lopian, E. (2018, May 15). *Defining Legacy Code - DZone DevOps*. Retrieved December 4, 2020 from DZone: Programming & DevOps news, tutorials & tools: https://dzone.com/articles/defining-legacy-code

Mackinnon, T., Freeman, S., & Craig, P. (2001). Endo-Testing: Unit Testing with Mock Objects. In G. Succi, & M. Marchesi (Ed.), *XP eXamined* (pp. 287-301). Boston: Addison-Wesley.

Martin, R. C. (2017). *Clean Architecture: A Craftman's Guide to Software Structure and Design* (Vol. 1). Upper Saddle River: Prentice Hall.

Martin, R. C. (2008). *Clean Code: A Handbook of Agile Software Craftmanship.* Upper Saddle River: Prentice Hall.

Martinez, W., Martinez, A., & Solk, J. (2010). *Exploratory Data Analysis with MATLAB* (Vol. 2). Boca Raton: CRC Press.

McCabe, T. (1976). A Complexity Measure. *IEEE Transactions on Software Engineering, SE-2*(4), 308-320. doi:10.1109/TSE.1976.233837

Meyer, M. (2014). Continuous Integration and Its Tools. *IEEE Software, 31*(3), 14-16. doi:10.1109/MS.2014.58

Molnar, A., & Motogna, S. (2017). Discovering Maintainability Changes in Large Software Systems. *Proceedings of the 27th International Workshop on Software Measurement and 12th International Conference on Software Process and Product Measurement* (pp. 88-93). New York: Association for Computing Machinery.

Moser, R., Sillitti, A., Abrahamsson, P., & Succi, G. (2006). Does Refactoring Improve Reusability? In M. Morisio (Ed.), *Reuse of Off-the-Shelf Components. ICSR 2006. Lecture Notes in Computer Science. 4039*, pp. 287-297. Berlin: Springer. doi:10.1007/11763864_21

Nazir, M., Khan, R., & Mustafa, K. (2010). A Metrics Based Model for Understandability Quantification. *Journal of Computing, 2*(4), 90-94.

Nuutila, E., & Soisalon-Soininen, E. (1994). On finding the strongly connected components in a directed graph. *Information Processing Letters, 49*(1), 9-14.

Oracle. (1999, April 20). *Code Conventions for the Java Programming Language: 9. Naming Conventions*. Retrieved November 2020, 25 from Oracle | Integrated Cloud Applications and Platform Services: https://www.oracle.com/java/technologies/javase/codeconventions-namingconventions.html

Oracle. (2020). *What Is an Exception? (The Java™ Tutorials >Essential Classes > Exceptions)*. Retrieved December 3, 2020 from Oracle Help Center: https://docs.oracle.com/javase/tutorial/essential/exceptions/definition.html

Ratzinger, J., Fischer, M., & Gall, H. (2005). Improving Evolvability through Refactoring. *Proceedings of the 2005 International Workshop on Mining Software Repositories, 30*(4), 1-5. doi:10.1145/1083142.1083155

Refactoring.Guru. (2020). *Catalog of Refactoring*. Retrieved December 3, 2020 from Refactoring and Design Patterns: https://refactoring.guru/refactoring/catalog

RickHos. (2004, September 7). *Variable names in C# Part 2 | Microsoft Docs*. Retrieved December 6, 2020 from Developer tools, technical documentation and coding examples | Microsoft Docs: https://docs.microsoft.com/en-us/archive/blogs/rickhos/variable-names-in-c-part-2

Ritchie, P. (2007, June 17). *The Religion of Class Member Prefixing – Peter Ritchie's MVP Blog*. Retrieved December 6, 2020 from Msmvps – The WordPress blogging site for current and ex- Microsoft MVPs.: https://blogs.msmvps.com/peterritchie/2007/06/17/the-religion-of-class-member-prefixing/

Romano, S., Vendome, C., Scanniello, G., & Poshyvanyk, D. (2020). A Multi-Study Investigation into Dead Code. *IEEE Transactions on Software Engineering*, 46(1), 71-99.

Saifan, A., & Al-Rabadi, A. (2017). Evaluating Maintainability of Android Applications. *2017 8th International Conference on Information Technology* (pp. 518-523). Amman: IEEE.

Saifan, A., Alsghaier, H., & Alkhateeb, K. (2018). Evaluating the Understandability of Android Applications. *International Journal of Software Innovation, 6*(1), 44-57. doi:10.4018/IJSI.2018010104

Sakshica, & Gupta, K. (2015). Various Raster and Vector Image File Formats. *International Journal of Advanced Research in Computer and Communication Engineering, 4*(3), 268-271.

Spohrer, K., Kude, T., Schmidt, C., & Heinzl, A. (2013). Knowledge Creation In Information Systems Development Teams: The Role Of Pair Programming And Peer Code Review. *European Conference on Information Systems 2013. Paper 213.* Atlanta: AIS Electronic Library.

Sun Microsystems Inc. (1997). JavaBeans. *1.01-A*. (G. Hamilton, Ed.) Mountain View. Retrieved December 6, 2020 from https://download.oracle.com/otn-pub/jcp/7224-javabeans-1.01-fr-spec-oth-JSpec/beans.101.pdf

Turkey, J. (1977). *Exploratory Data Analysis.* Reading: Addison Wesley.

W3C. (2020, November 16). *Link Purpose (Link Only)*. Retrieved November 2020, 25 from w3.org: https://www.w3.org/TR/UNDERSTANDING-WCAG20/navigation-mechanisms-link.html

Wang, Y. (2009). What Motivate Software Engineers to Refactor Source Code? Evidences from Professional Developers. *2009 IEEE International Conference on Software Maintenance* (pp. 413-416). Edmonton: IEEE. doi:10.1109/ICSM.2009.5306290

Wang, Y. (2009). *What motivate software engineers to refactor source code? evidences from professional developers - IEEE Conference Publication.* Retrieved December 2, 2020 from IEEE Xplore: https://ieeexplore.ieee.org/mediastore_new/IEEE/content/media/5290639/5306271/5306290/5306290-fig-1-source-large.gif

Xie, Y., Wolf, W., & Lekatsas, H. (2003). Profile-driven Selective Code Compression. *2003 Design, Automation and Test in Europe Conference and Exhibition* (pp. 462-467). Munich: IEEE. doi:10.1109/DATE.2003.1253652

Yu, Y., Wang, Y., Mylopoulos, J., Liaskos, S., Lapouchnian, A., & Do Prado Leite, J. (2005). Reverse Engineering Goal Models from Legacy Code. *13th IEEE International Conference on Requirements Engineering (RE'05)* (pp. 363-372). Paris: IEEE.

Appendix

A. Content of the Exported Coding Style Settings in CodeStyle.xml

```
<code_scheme name="fanat" version="173">
  <option name="RIGHT_MARGIN" value="180" />
  <JavaCodeStyleSettings>
    <option name="IMPORT_LAYOUT_TABLE">
      <value>
        <package name="android" withSubpackages="true" static="false" />
        <emptyLine />
        <package name="com" withSubpackages="true" static="false" />
        <emptyLine />
        <package name="junit" withSubpackages="true" static="false" />
        <emptyLine />
        <package name="net" withSubpackages="true" static="false" />
        <emptyLine />
        <package name="org" withSubpackages="true" static="false" />
        <emptyLine />
        <package name="java" withSubpackages="true" static="false" />
        <emptyLine />
        <package name="javax" withSubpackages="true" static="false" />
        <emptyLine />
        <package name="" withSubpackages="true" static="false" />
        <emptyLine />
        <package name="" withSubpackages="true" static="true" />
        <emptyLine />
      </value>
    </option>
  </JavaCodeStyleSettings>
  <codeStyleSettings language="JAVA">
    <option name="IF_BRACE_FORCE" value="3" />
    <option name="DOWHILE_BRACE_FORCE" value="3" />
    <option name="WHILE_BRACE_FORCE" value="3" />
    <option name="FOR_BRACE_FORCE" value="3" />
  </codeStyleSettings>
  <codeStyleSettings language="XML">
    <indentOptions>
      <option name="CONTINUATION_INDENT_SIZE" value="4" />
    </indentOptions>
    <arrangement>
      <rules>
        <section>
          <rule>
            <match>
              <AND>
                <NAME>xmlns:android</NAME>
                <XML_ATTRIBUTE />
                <XML_NAMESPACE>^$</XML_NAMESPACE>
              </AND>
            </match>
          </rule>
        </section>
        <section>
          <rule>
            <match>
              <AND>
                <NAME>xmlns:.*</NAME>
                <XML_ATTRIBUTE />
                <XML_NAMESPACE>^$</XML_NAMESPACE>
              </AND>
            </match>
            <order>BY_NAME</order>
          </rule>
        </section>
        <section>
          <rule>
            <match>
              <AND>
                <NAME>.*:id</NAME>
                <XML_ATTRIBUTE />
                <XML_NAMESPACE>http://schemas.android.com/apk/res/android</XML_NAMESPACE>
              </AND>
            </match>
          </rule>
        </section>
        <section>
```

```
          <rule>
            <match>
              <AND>
                <NAME>.*:name</NAME>
                <XML_ATTRIBUTE />
                <XML_NAMESPACE>http://schemas.android.com/apk/res/android</XML_NAMESPACE>
              </AND>
            </match>
          </rule>
        </section>
        <section>
          <rule>
            <match>
              <AND>
                <NAME>name</NAME>
                <XML_ATTRIBUTE />
                <XML_NAMESPACE>^$</XML_NAMESPACE>
              </AND>
            </match>
          </rule>
        </section>
        <section>
          <rule>
            <match>
              <AND>
                <NAME>style</NAME>
                <XML_ATTRIBUTE />
                <XML_NAMESPACE>^$</XML_NAMESPACE>
              </AND>
            </match>
          </rule>
        </section>
        <section>
          <rule>
            <match>
              <AND>
                <NAME>.*</NAME>
                <XML_ATTRIBUTE />
                <XML_NAMESPACE>^$</XML_NAMESPACE>
              </AND>
            </match>
            <order>BY_NAME</order>
          </rule>
        </section>
        <section>
          <rule>
            <match>
              <AND>
                <NAME>.*</NAME>
                <XML_ATTRIBUTE />
                <XML_NAMESPACE>http://schemas.android.com/apk/res/android</XML_NAMESPACE>
              </AND>
            </match>
            <order>ANDROID_ATTRIBUTE_ORDER</order>
          </rule>
        </section>
        <section>
          <rule>
            <match>
              <AND>
                <NAME>.*</NAME>
                <XML_ATTRIBUTE />
                <XML_NAMESPACE>.*</XML_NAMESPACE>
              </AND>
            </match>
            <order>BY_NAME</order>
          </rule>
        </section>
      </rules>
    </arrangement>
  </codeStyleSettings>
</code_scheme>
```

Listing 18: The defined coding standard exported by Android Studio as XML. © by SFA Sport GmbH

B. Results of Code Complexity Metrics Calculations

Version	Hash	Date
1	e9620d752a6ee75ffcee0446b3ffb68dbcc8202f	22-Nov-18
2	733b185262b78e83e4e6b844eeb7ba30b9784e9f	22-Mar-19
3	eae0e9514cc8f4cdd5829bf4b5b87bb8bdbe03fb	19-Jul-19
4	0baddeefd78e27ff5ba72194012c80a0f32d6013	15-Nov-19
5	93245af07066d20f9c9561b783c051e28eb02135	24-Mar-20
6	f5645ab0c32a00b54205d54fffea969a6d289309	28-Jul-20
7	ed9ede3e660618c1c09bd18ea338c984033456ec	24-Nov-20

Table 17: The corresponding hashes within the repository[56] for each version.

Complexity Class			Complexity Method						Dependency Class					Dep. Package		Files & LOC		MOOD					
Java																							
OCavg Ø	WMC ∑	WMC Ø	ev(G) ∑	ev(G) Ø	iv(G) ∑	iv(G) Ø	v(G) ∑	v(G) Ø	Cyclic Ø	Dcy Ø	Dcy* Ø	Dpt Ø	Dpt* Ø	PDcy Ø	PDpt Ø	Files ∑	LOC ∑	AHF	AIF	CF	MHF	MIF	PF
2.02	7735	22.68	3827	1.31	6289	2.15	7606	2.6	84.27	8.35	197.9	5.93	170.93	8.51	8.51	309	47221	85.61	54.86	5.45	39.77	28.39	63.65
1.97	9638	21.32	4860	1.29	7938	2.11	9505	2.53	118.59	8.4	262.75	6.03	230.28	8.92	8.92	391	59796	84.21	51.98	4.51	40.15	29.57	60.81
1.95	10322	21.59	5208	1.29	8456	2.09	10189	2.52	128.17	8.67	281.48	6.24	246.67	9.1	9.1	407	64149	84.75	50.96	4.49	40.1	29.72	59.95
1.95	10415	21.97	5312	1.29	8585	2.08	10362	2.52	124.08	8.6	276.33	6.19	241.36	9.05	9.05	403	63936	85.08	50.02	4.49	39.55	28.65	61.19
1.94	10652	22.01	5455	1.29	8894	2.11	10740	2.55	113.68	8.51	265.5	6.03	231.96	8.98	8.98	414	65058	84.41	50.12	4.32	39.18	29.36	59.24
1.9	9383	21.82	4783	1.28	7850	2.09	9374	2.5	117.21	8.55	242.61	6.1	217.82	9.04	9.04	371	56134	82.68	77.49	4.76	37.99	28.29	77.92
1.82	9423	20.18	5146	1.26	7930	1.94	9432	2.31	136	8.28	271.22	5.93	245.45	9.3	9.28	393	54844	82.31	80.06	4.23	42.58	30.65	61.62
Game																							
OCavg Ø	WMC ∑	WMC Ø	ev(G) ∑	ev(G) Ø	iv(G) ∑	iv(G) Ø	v(G) ∑	v(G) Ø	Cyclic Ø	Dcy Ø	Dcy* Ø	Dpt Ø	Dpt* Ø	PDcy Ø	PDpt Ø	Files ∑	LOC ∑	AHF	AIF	CF	MHF	MIF	PF
3.14	2166	39.38	734	1.6	1691	3.68	2172	4.72	111.27	12.36	242.58	2.31	177.82	11.29	2.29	53	13317	98.21	65.95	48.3	53.52	33.52	165.4
3.15	2314	40.6	771	1.59	1821	3.76	2344	4.84	151.25	12.88	311.98	2.39	238.28	11.57	2.71	54	14305	97.96	65.35	50.2	53.37	32.77	170.8
3.17	2561	40.65	849	1.58	2025	3.78	2604	4.86	158.1	12.84	336.92	2.49	244.75	12.14	3	57	15650	96.5	64.49	49.6	52.09	31.68	179.8
3.08	2612	42.13	901	1.61	2071	3.7	2674	4.78	149.55	13.15	331.66	2.44	239.84	12.14	2.71	57	15804	96.72	62.97	49.6	51.65	30.66	191.7
2.95	2647	40.72	913	1.56	2121	3.63	2740	4.69	145.23	12.94	326.55	2.38	237.23	12.83	2.83	60	16161	95.92	63.32	46.2	50.8	31.31	206.4
2.94	2439	43.55	860	1.57	1938	3.53	2519	4.59	138.43	13.11	289.91	2.45	223.64	12.17	3	53	14629	95.31	80.41	51.6	51.57	31.88	230.4
2.11	2104	31.4	966	1.29	1645	2.19	2037	2.71	178.88	10.31	339.46	2.19	258.04	12.33	3.17	49	12101	95.47	80.9	32.8	64.17	33.53	106.4
Entity																							
OCavg Ø	WMC ∑	WMC Ø	ev(G) ∑	ev(G) Ø	iv(G) ∑	iv(G) Ø	v(G) ∑	v(G) Ø	Cyclic Ø	Dcy Ø	Dcy* Ø	Dpt Ø	Dpt* Ø	PDcy Ø	PDpt Ø	Files ∑	LOC ∑	AHF	AIF	CF	MHF	MIF	PF
1.66	259	19.92	96	1.23	175	2.24	198	2.54	117.69	13.62	256.46	1.92	184.54	17.5	6	13	1749	87.32	34.15	227	36.83	50.94	100
1.65	262	16.38	99	1.22	179	2.21	202	2.49	131.06	11.56	271.19	1.81	220.31	17.5	6	13	1778	85.71	34.71	222	36.06	52.4	5750
1.72	285	17.81	102	1.23	186	2.24	220	2.65	140.06	11.69	285.25	1.94	236.94	18.5	6.5	13	1877	84.77	36.2	224	36.69	51.18	6000
1.73	287	17.94	102	1.23	188	2.27	222	2.67	137.25	11.75	283	2	232.56	18.5	6.5	13	1863	84.77	36.2	226	36.69	51.18	6000
1.72	279	17.44	100	1.2	180	2.17	219	2.64	132.75	11.69	272.25	2.06	236.94	18	6.5	13	1827	84.72	37.2	224	36.19	52.77	5850
1.69	277	17.31	100	1.18	182	2.14	220	2.59	128.25	11.38	247.5	2.38	215.06	17	8	13	1830	85.43	85.19	218	35.75	52.46	6450
1.38	252	15.75	147	1.06	198	1.42	226	1.63	111.81	11	268.12	2.56	206.44	18.67	7	13	1502	79.45	90.34	210	59.69	55.38	304.4
Adapter																							
OCavg Ø	WMC ∑	WMC Ø	ev(G) ∑	ev(G) Ø	iv(G) ∑	iv(G) Ø	v(G) ∑	v(G) Ø	Cyclic Ø	Dcy Ø	Dcy* Ø	Dpt Ø	Dpt* Ø	PDcy Ø	PDpt Ø	Files ∑	LOC ∑	AHF	AIF	CF	MHF	MIF	PF
2.75	815	17.72	422	1.74	652	2.69	839	3.47	99.78	10.04	234.5	2.2	168.65	34	18	20	4538	86.23	2.37	129	16.33	10.71	236.1
2.6	1186	17.19	615	1.64	956	2.55	1228	3.27	148.58	10.14	321.25	2.22	237.01	27	13	31	6549	83.37	17.03	85.6	27.87	14.9	125.2
2.53	1355	16.94	702	1.64	1079	2.52	1393	3.25	155.62	10.19	329.69	2.2	257.09	26	13	33	7687	86.3	15.29	81.4	27.95	13.82	134.1
2.57	1377	17.43	694	1.63	1108	2.59	1423	3.33	154.43	10.24	331.01	2.18	248.77	26	13	32	7774	86.99	15.03	85.9	28.44	13.78	135.4
2.74	1403	18.22	711	1.65	1192	2.77	1545	3.59	128.96	9.74	315.04	2.13	214.96	21.5	11.5	32	7623	86.28	13.25	78.6	21.5	13.84	117.8
2.42	1074	16.52	614	1.6	919	2.39	1175	3.06	147.35	9.48	303.92	2.2	230.45	20.5	12	31	5855	82.63	75.4	74	20.7	12.07	239.5
2.4	1059	15.57	610	1.55	906	2.3	1122	2.85	165.06	9	335.74	2.5	254.37	15	12	36	5836	83.49	76.04	58.1	22.46	21.16	115.2
GameFragment																							
OCavg Ø	WMC ∑		ev(G) ∑	ev(G) Ø	iv(G) ∑	iv(G) Ø	v(G) ∑	v(G) Ø															
3.24	285		87	1.98	230	5.23	290	6.59															
3.2	352		105	1.88	283	5.05	371	6.62															
3.25	357		105	1.88	287	5.12	378	6.75															
3.15	346		105	1.88	281	5.02	370	6.61															
3.01	304		88	1.63	288	5.33	387	7.17															
3.17	241		79	1.84	265	6.16	347	8.07															
1.32	215		149	1.09	190	1.39	201	1.47															

Table 18: The calculated metrics for versions 1 to 7 each.

[56] https://gitlab.com/fan-at/app/android

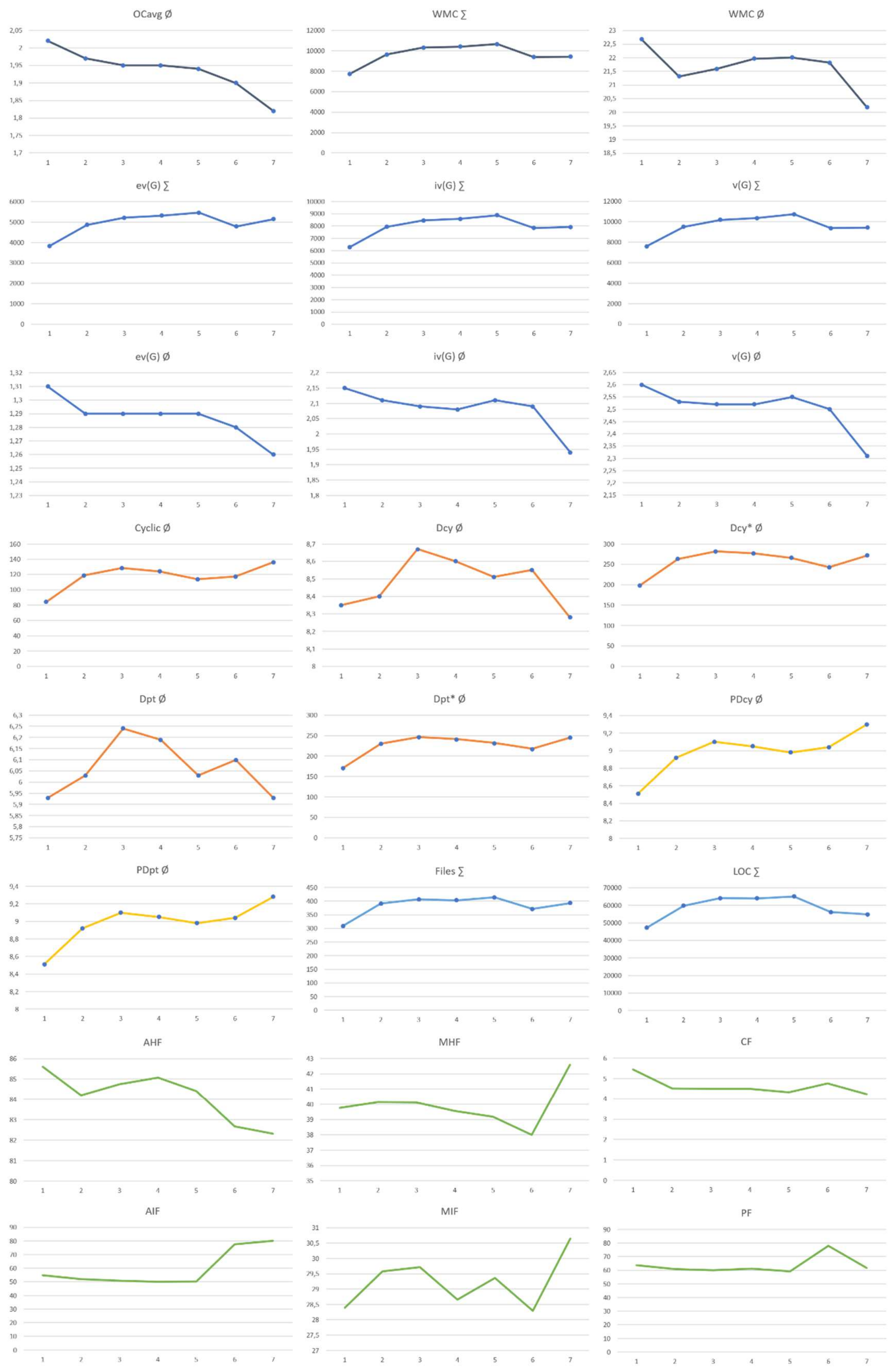
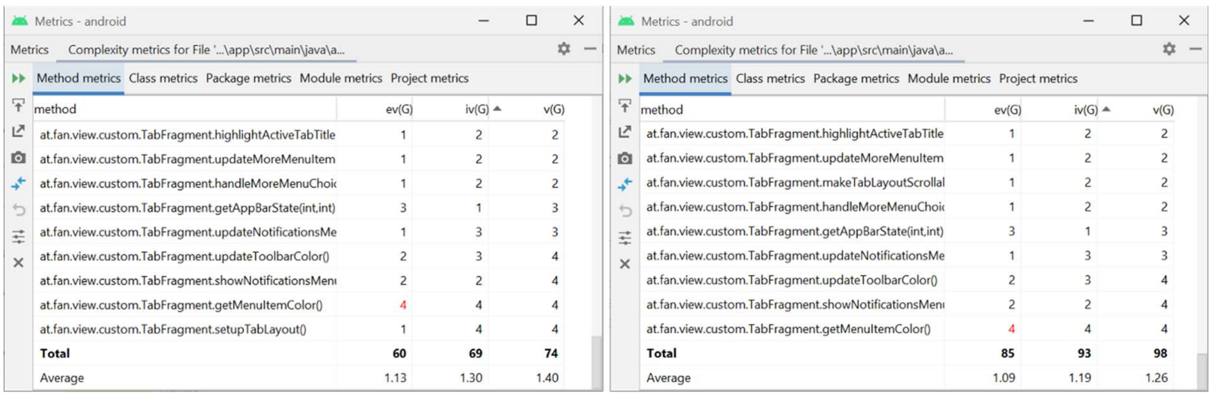


Figure 49: Diagrams of all metrics taken for the project including complexity, dependencies, number of Java files, lines of code, and the MOOD metrices.



Metrics - android

Metrics Complexity metrics for File '...\app\src\main\java\a...

Method metrics Class metrics Package metrics Module metrics Project metrics

method	ev(G)	iv(G)	v(G)
at.fan.view.custom.TabFragment.highlightActiveTabTitle	1	2	2
at.fan.view.custom.TabFragment.updateMoreMenuItem	1	2	2
at.fan.view.custom.TabFragment.handleMoreMenuChoic	1	2	2
at.fan.view.custom.TabFragment.getAppBarState(int,int)	3	1	3
at.fan.view.custom.TabFragment.updateNotificationsMe	1	3	3
at.fan.view.custom.TabFragment.updateToolbarColor()	2	3	4
at.fan.view.custom.TabFragment.showNotificationsMenu	2	2	4
at.fan.view.custom.TabFragment.getMenuItemColor()	4	4	4
at.fan.view.custom.TabFragment.setupTabLayout()	1	4	4
Total	**60**	**69**	**74**
Average	1.13	1.30	1.40

Metrics - android

Metrics Complexity metrics for File '...\app\src\main\java\a...

Method metrics Class metrics Package metrics Module metrics Project metrics

method	ev(G)	iv(G)	v(G)
at.fan.view.custom.TabFragment.highlightActiveTabTitle	1	2	2
at.fan.view.custom.TabFragment.updateMoreMenuItem	1	2	2
at.fan.view.custom.TabFragment.makeTabLayoutScrollal	1	2	2
at.fan.view.custom.TabFragment.handleMoreMenuChoic	1	2	2
at.fan.view.custom.TabFragment.getAppBarState(int,int)	3	1	3
at.fan.view.custom.TabFragment.updateNotificationsMe	1	3	3
at.fan.view.custom.TabFragment.updateToolbarColor()	2	3	4
at.fan.view.custom.TabFragment.showNotificationsMenu	2	2	4
at.fan.view.custom.TabFragment.getMenuItemColor()	4	4	4
Total	**85**	**93**	**98**
Average	1.09	1.19	1.26

Figure 50: Experiment to calculate the cyclic complexity for survey question D1. The refactored method to the right results in a lower complexity only by refactoring one long method.

C. Questionnaire of Survey

Introduction to the Questionnaire

The aim of this survey is to find out what opinion the software engineering community has about code complexity.

Dear participant,

Thank you for taking part in my survey.

My name is Lorenz Kofler and I am student at the Technical University of Graz and University of Graz. Currently, I am doing my Master's thesis on complexity of code.

This survey is about programming, so you should have programming experience. If this is not the case, I thank you anyway that you wanted to participate.

The survey will take up to ten minutes. Some of the questions are built upon code of a Java Android application, but you do not need any prior knowledge. Performance issues are not relevant in any question, in doubt. The data obtained with this survey will only be used for scientific purposes. All data is collected anonymously, so no conclusions can be drawn about certain people. You are welcome to pass this questionnaire on to other software developers.

If you have any questions, please do not hesitate to contact me (lorenz.kofler@student.tugraz.at). Thank you for your support!

Best regards,

Lorenz Kofler

PS: As code is presented within this survey, you should use a desktop browser. In case the font size is too small, please zoom into the survey. Thank you.

There are 20 questions in this survey

Content of Questionnaire

* 1 **In a software project, it is important that the developers agree on a naming convention.**

Choose one of the following answers

- ◯ Strongly disagree
- ◯ Disagree
- ◯ Neither agree nor disagree
- ◯ Agree
- ◯ Strongly agree
- ◯ Don't know

That means that they write e.g. all variables in lowerCamelCase, or constants in UPPERCASE with "_" as separator between words, etc.

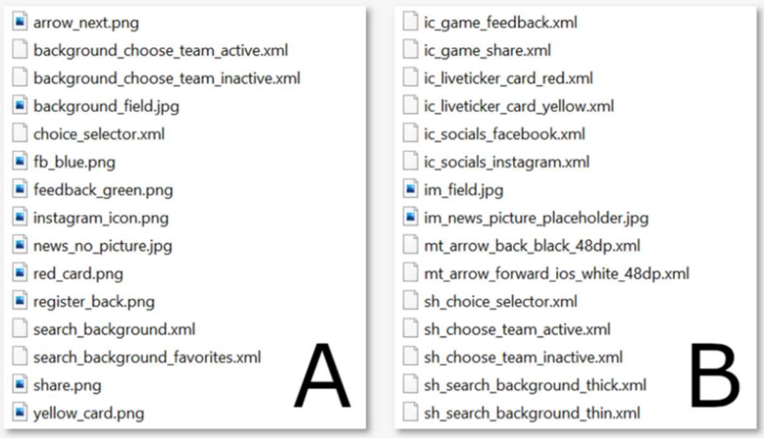
* 2 **File names should not follow a certain naming convention.**

Choose one of the following answers

- ◯ Strongly disagree
- ◯ Disagree
- ◯ Neither agree nor disagree
- ◯ Agree
- ◯ Strongly agree
- ◯ Don't know

3

Which of the both drawable folders follows a better naming convention?



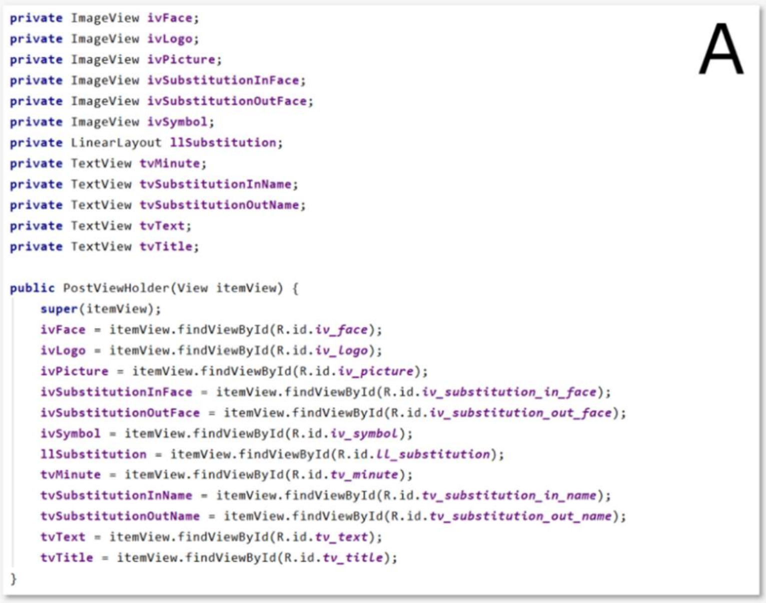
A	B
arrow_next.png	ic_game_feedback.xml
background_choose_team_active.xml	ic_game_share.xml
background_choose_team_inactive.xml	ic_liveticker_card_red.xml
background_field.jpg	ic_liveticker_card_yellow.xml
choice_selector.xml	ic_socials_facebook.xml
fb_blue.png	ic_socials_instagram.xml
feedback_green.png	im_field.jpg
instagram_icon.png	im_news_picture_placeholder.jpg
news_no_picture.jpg	mt_arrow_back_black_48dp.xml
red_card.png	mt_arrow_forward_ios_white_48dp.xml
register_back.png	sh_choice_selector.xml
search_background.xml	sh_choose_team_active.xml
search_background_favorites.xml	sh_choose_team_inactive.xml
share.png	sh_search_background_thick.xml
yellow_card.png	sh_search_background_thin.xml

Please click and drag the slider handles to enter your answer.

0: No difference -50: A +50: B

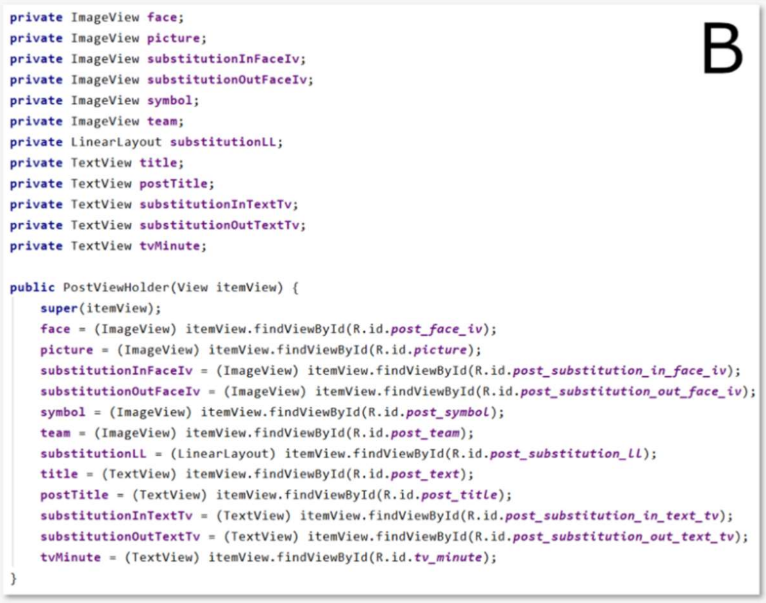
4

Which of the two examples follows a better naming convention for the handling of views, i.e. text boxes, images, etc.? The entries are sorted first by type and second by name in alphabetical order.



A

```
private ImageView ivFace;
private ImageView ivLogo;
private ImageView ivPicture;
private ImageView ivSubstitutionInFace;
private ImageView ivSubstitutionOutFace;
private ImageView ivSymbol;
private LinearLayout llSubstitution;
private TextView tvMinute;
private TextView tvSubstitutionInName;
private TextView tvSubstitutionOutName;
private TextView tvText;
private TextView tvTitle;

public PostViewHolder(View itemView) {
    super(itemView);
    ivFace = itemView.findViewById(R.id.iv_face);
    ivLogo = itemView.findViewById(R.id.iv_logo);
    ivPicture = itemView.findViewById(R.id.iv_picture);
    ivSubstitutionInFace = itemView.findViewById(R.id.iv_substitution_in_face);
    ivSubstitutionOutFace = itemView.findViewById(R.id.iv_substitution_out_face);
    ivSymbol = itemView.findViewById(R.id.iv_symbol);
    llSubstitution = itemView.findViewById(R.id.ll_substitution);
    tvMinute = itemView.findViewById(R.id.tv_minute);
    tvSubstitutionInName = itemView.findViewById(R.id.tv_substitution_in_name);
    tvSubstitutionOutName = itemView.findViewById(R.id.tv_substitution_out_name);
    tvText = itemView.findViewById(R.id.tv_text);
    tvTitle = itemView.findViewById(R.id.tv_title);
}
```

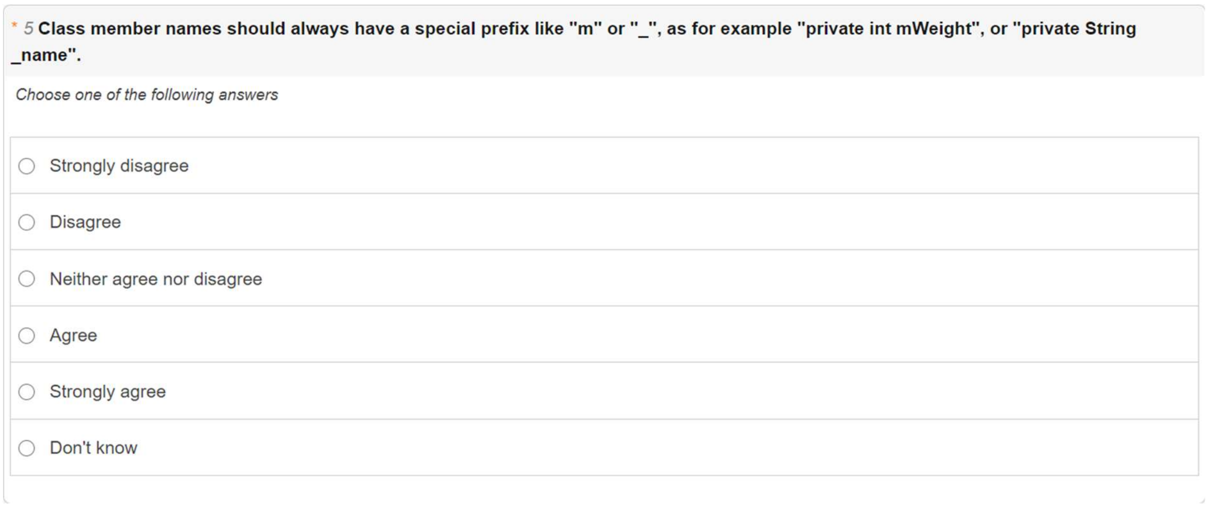


B

```
private ImageView face;
private ImageView picture;
private ImageView substitutionInFaceIv;
private ImageView substitutionOutFaceIv;
private ImageView symbol;
private ImageView team;
private LinearLayout substitutionLL;
private TextView title;
private TextView postTitle;
private TextView substitutionInTextTv;
private TextView substitutionOutTextTv;
private TextView tvMinute;

public PostViewHolder(View itemView) {
    super(itemView);
    face = (ImageView) itemView.findViewById(R.id.post_face_iv);
    picture = (ImageView) itemView.findViewById(R.id.picture);
    substitutionInFaceIv = (ImageView) itemView.findViewById(R.id.post_substitution_in_face_iv);
    substitutionOutFaceIv = (ImageView) itemView.findViewById(R.id.post_substitution_out_face_iv);
    symbol = (ImageView) itemView.findViewById(R.id.post_symbol);
    team = (ImageView) itemView.findViewById(R.id.post_team);
    substitutionLL = (LinearLayout) itemView.findViewById(R.id.post_substitution_ll);
    title = (TextView) itemView.findViewById(R.id.post_text);
    postTitle = (TextView) itemView.findViewById(R.id.post_title);
    substitutionInTextTv = (TextView) itemView.findViewById(R.id.post_substitution_in_text_tv);
    substitutionOutTextTv = (TextView) itemView.findViewById(R.id.post_substitution_out_text_tv);
    tvMinute = (TextView) itemView.findViewById(R.id.tv_minute);
}
```

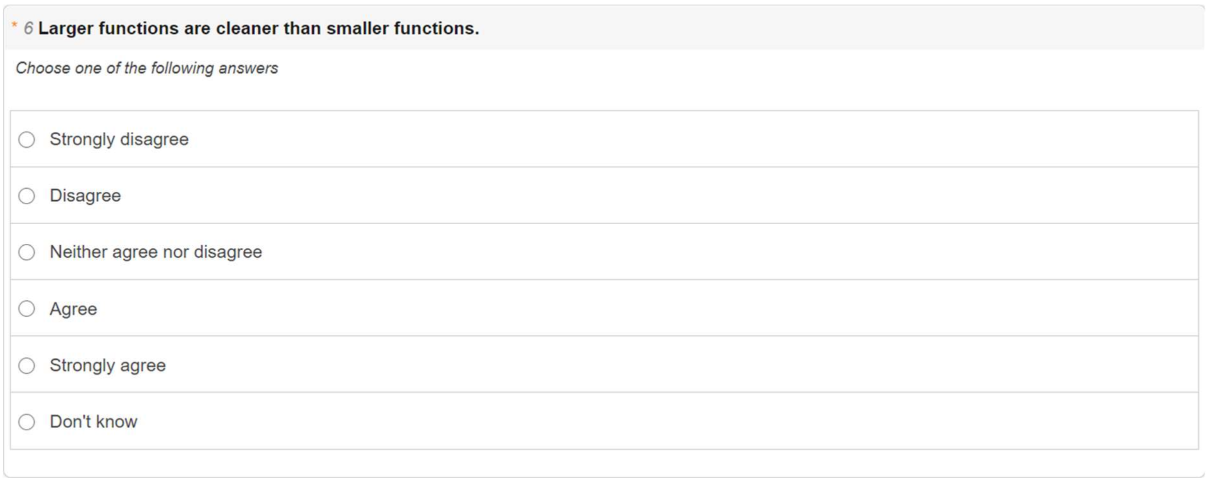
Please click and drag the slider handles to enter your answer.

0: No difference -50: A +50: B


* 5 **Class member names should always have a special prefix like "m" or "_", as for example "private int mWeight", or "private String _name".**

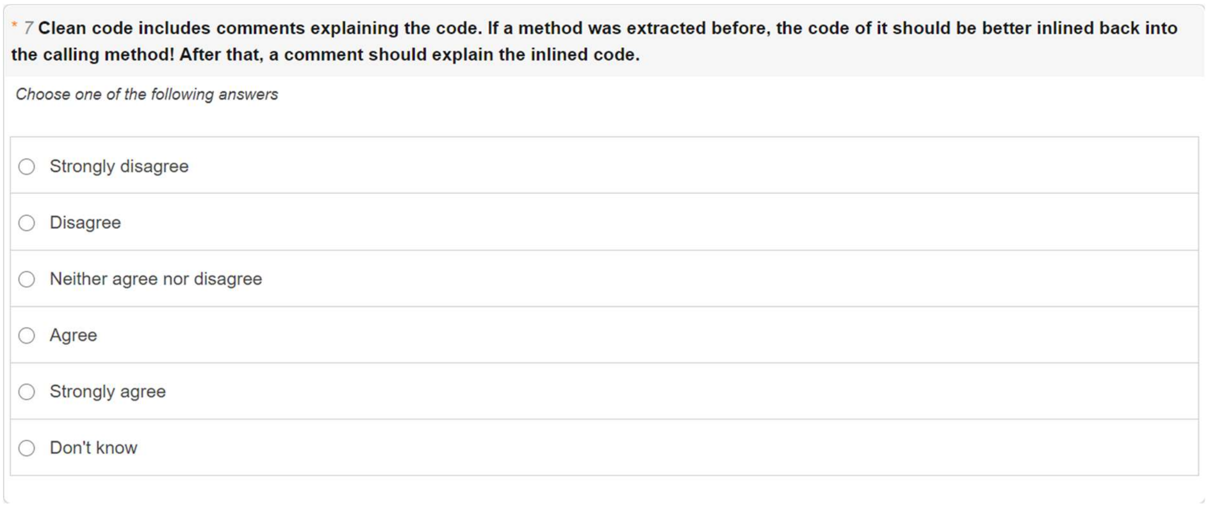
Choose one of the following answers

- ○ Strongly disagree
- ○ Disagree
- ○ Neither agree nor disagree
- ○ Agree
- ○ Strongly agree
- ○ Don't know


* 6 **Larger functions are cleaner than smaller functions.**

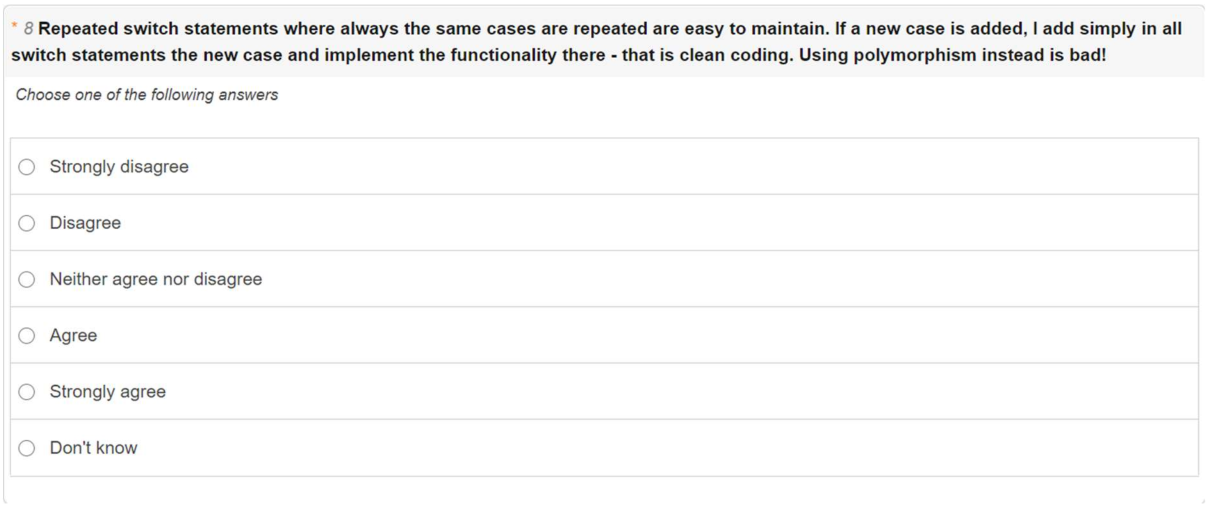
Choose one of the following answers

- ○ Strongly disagree
- ○ Disagree
- ○ Neither agree nor disagree
- ○ Agree
- ○ Strongly agree
- ○ Don't know


* 7 **Clean code includes comments explaining the code. If a method was extracted before, the code of it should be better inlined back into the calling method! After that, a comment should explain the inlined code.**

Choose one of the following answers

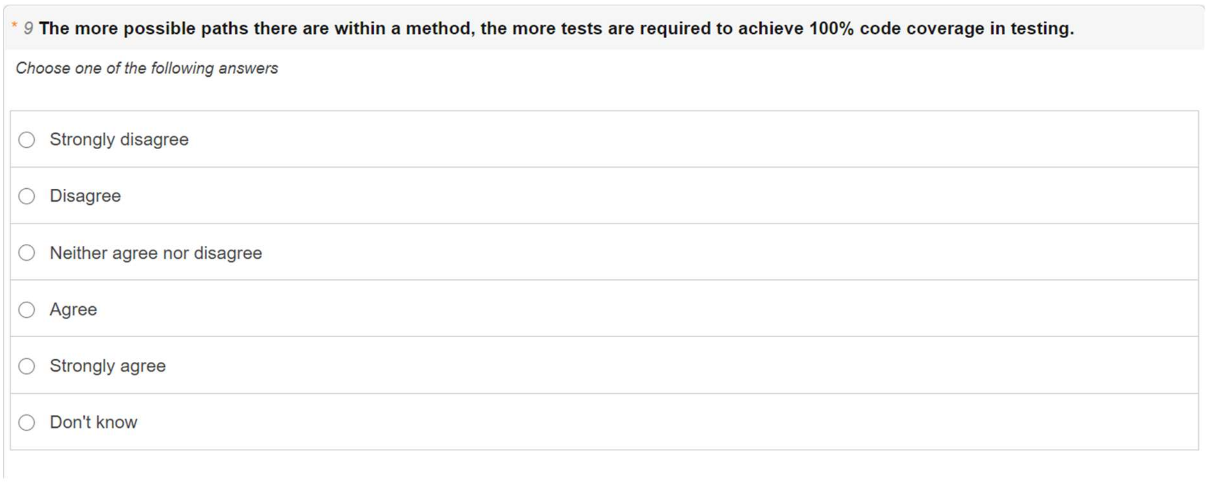
- ○ Strongly disagree
- ○ Disagree
- ○ Neither agree nor disagree
- ○ Agree
- ○ Strongly agree
- ○ Don't know



* 8 **Repeated switch statements where always the same cases are repeated are easy to maintain. If a new case is added, I add simply in all switch statements the new case and implement the functionality there - that is clean coding. Using polymorphism instead is bad!**

Choose one of the following answers

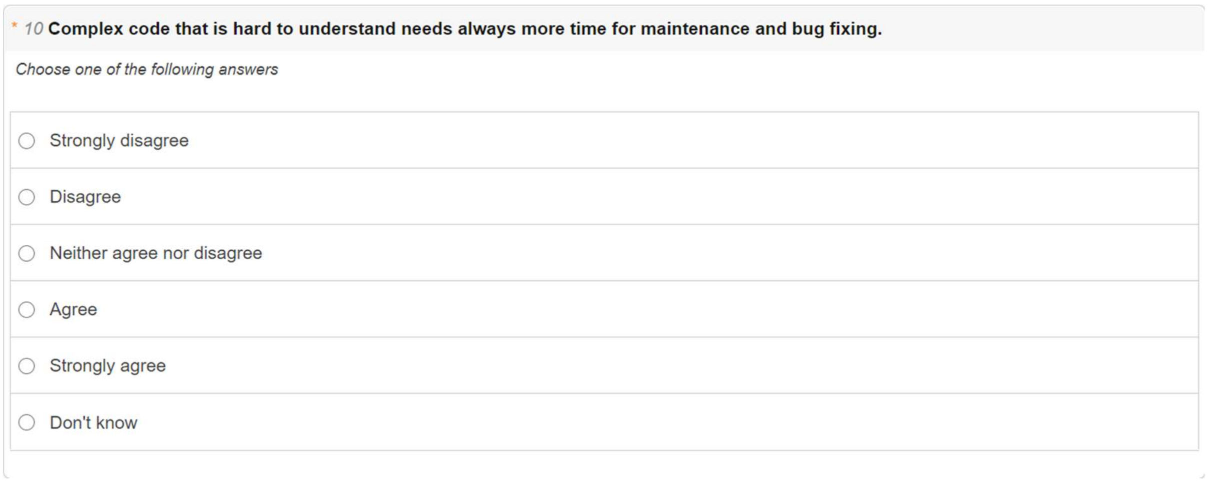
○ Strongly disagree
○ Disagree
○ Neither agree nor disagree
○ Agree
○ Strongly agree
○ Don't know



* 9 **The more possible paths there are within a method, the more tests are required to achieve 100% code coverage in testing.**

Choose one of the following answers

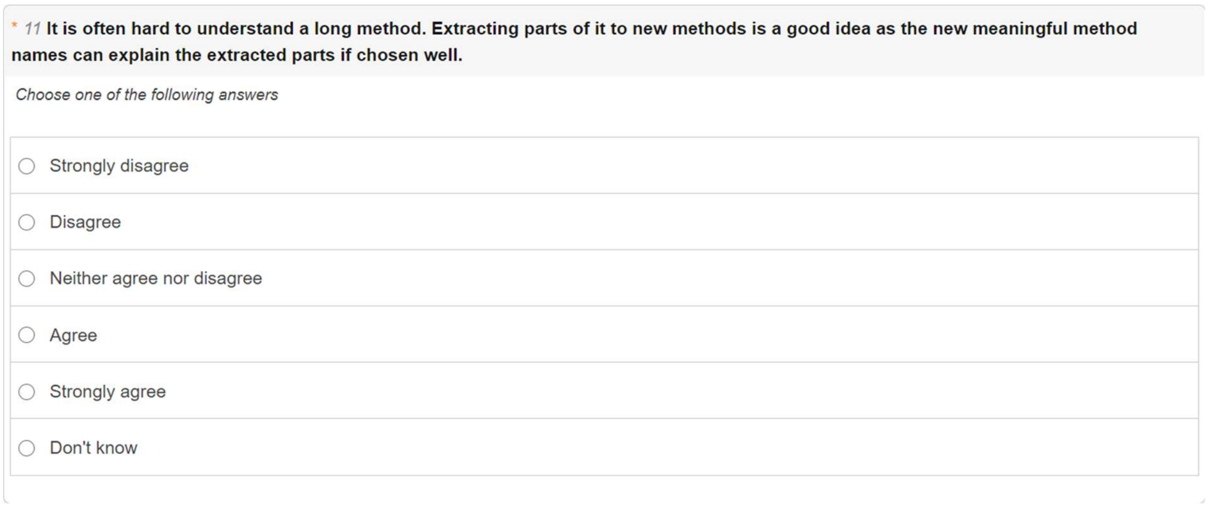
○ Strongly disagree
○ Disagree
○ Neither agree nor disagree
○ Agree
○ Strongly agree
○ Don't know



* 10 **Complex code that is hard to understand needs always more time for maintenance and bug fixing.**

Choose one of the following answers

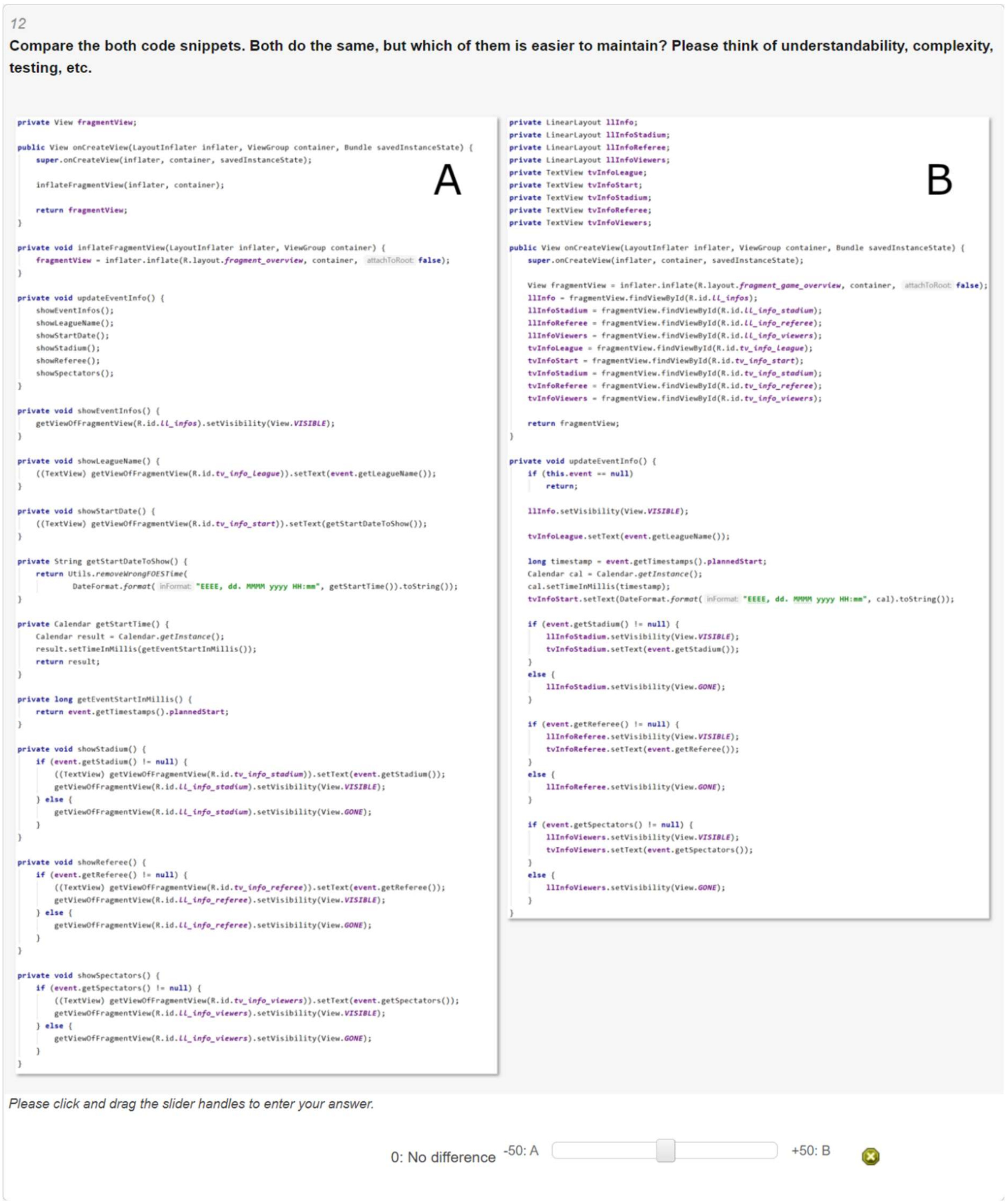
○ Strongly disagree
○ Disagree
○ Neither agree nor disagree
○ Agree
○ Strongly agree
○ Don't know



* 11 **It is often hard to understand a long method. Extracting parts of it to new methods is a good idea as the new meaningful method names can explain the extracted parts if chosen well.**

Choose one of the following answers

- ○ Strongly disagree
- ○ Disagree
- ○ Neither agree nor disagree
- ○ Agree
- ○ Strongly agree
- ○ Don't know



12

Compare the both code snippets. Both do the same, but which of them is easier to maintain? Please think of understandability, complexity, testing, etc.

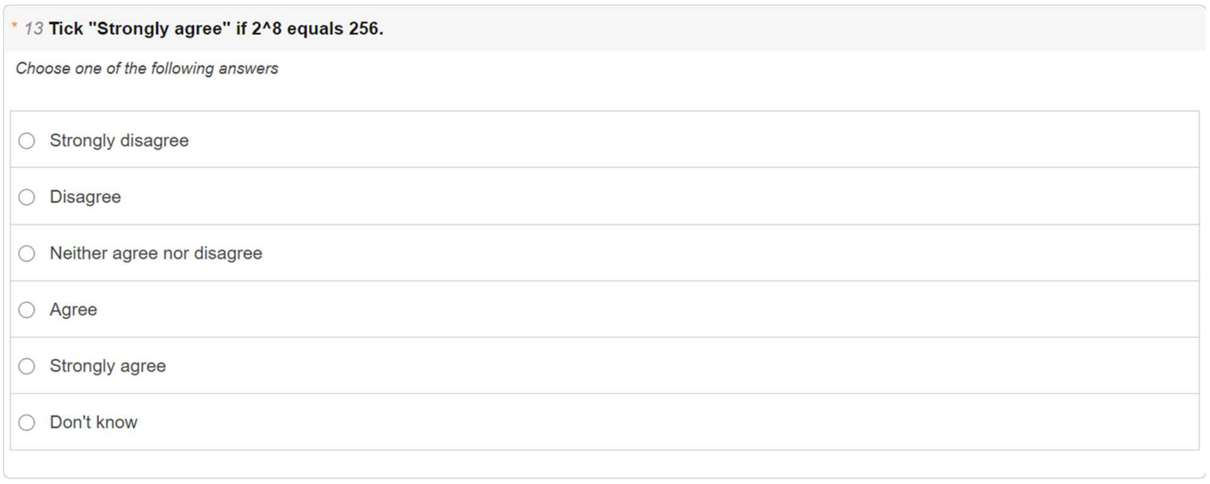
A

```java
private View fragmentView;

public View onCreateView(LayoutInflater inflater, ViewGroup container, Bundle savedInstanceState) {
    super.onCreateView(inflater, container, savedInstanceState);

    inflateFragmentView(inflater, container);

    return fragmentView;
}

private void inflateFragmentView(LayoutInflater inflater, ViewGroup container) {
    fragmentView = inflater.inflate(R.layout.fragment_overview, container, attachToRoot: false);
}

private void updateEventInfo() {
    showEventInfos();
    showLeagueName();
    showStartDate();
    showStadium();
    showReferee();
    showSpectators();
}

private void showEventInfos() {
    getViewOfFragmentView(R.id.ll_infos).setVisibility(View.VISIBLE);
}

private void showLeagueName() {
    ((TextView) getViewOfFragmentView(R.id.tv_info_league)).setText(event.getLeagueName());
}

private void showStartDate() {
    ((TextView) getViewOfFragmentView(R.id.tv_info_start)).setText(getStartDateToShow());
}

private String getStartDateToShow() {
    return Utils.removeWrongFOESTime(
            DateFormat.format( inFormat: "EEEE, dd. MMMM yyyy HH:mm", getStartTime()).toString());
}

private Calendar getStartTime() {
    Calendar result = Calendar.getInstance();
    result.setTimeInMillis(getEventStartInMillis());
    return result;
}

private long getEventStartInMillis() {
    return event.getTimestamps().plannedStart;
}

private void showStadium() {
    if (event.getStadium() != null) {
        ((TextView) getViewOfFragmentView(R.id.tv_info_stadium)).setText(event.getStadium());
        getViewOfFragmentView(R.id.ll_info_stadium).setVisibility(View.VISIBLE);
    } else {
        getViewOfFragmentView(R.id.ll_info_stadium).setVisibility(View.GONE);
    }
}

private void showReferee() {
    if (event.getReferee() != null) {
        ((TextView) getViewOfFragmentView(R.id.tv_info_referee)).setText(event.getReferee());
        getViewOfFragmentView(R.id.ll_info_referee).setVisibility(View.VISIBLE);
    } else {
        getViewOfFragmentView(R.id.ll_info_referee).setVisibility(View.GONE);
    }
}

private void showSpectators() {
    if (event.getSpectators() != null) {
        ((TextView) getViewOfFragmentView(R.id.tv_info_viewers)).setText(event.getSpectators());
        getViewOfFragmentView(R.id.ll_info_viewers).setVisibility(View.VISIBLE);
    } else {
        getViewOfFragmentView(R.id.ll_info_viewers).setVisibility(View.GONE);
    }
}
```

B

```java
private LinearLayout llInfo;
private LinearLayout llInfoStadium;
private LinearLayout llInfoReferee;
private LinearLayout llInfoViewers;
private TextView tvInfoLeague;
private TextView tvInfoStart;
private TextView tvInfoStadium;
private TextView tvInfoReferee;
private TextView tvInfoViewers;

public View onCreateView(LayoutInflater inflater, ViewGroup container, Bundle savedInstanceState) {
    super.onCreateView(inflater, container, savedInstanceState);

    View fragmentView = inflater.inflate(R.layout.fragment_game_overview, container, attachToRoot: false);
    llInfo = fragmentView.findViewById(R.id.ll_infos);
    llInfoStadium = fragmentView.findViewById(R.id.ll_info_stadium);
    llInfoReferee = fragmentView.findViewById(R.id.ll_info_referee);
    llInfoViewers = fragmentView.findViewById(R.id.ll_info_viewers);
    tvInfoLeague = fragmentView.findViewById(R.id.tv_info_league);
    tvInfoStart = fragmentView.findViewById(R.id.tv_info_start);
    tvInfoStadium = fragmentView.findViewById(R.id.tv_info_stadium);
    tvInfoReferee = fragmentView.findViewById(R.id.tv_info_referee);
    tvInfoViewers = fragmentView.findViewById(R.id.tv_info_viewers);

    return fragmentView;
}

private void updateEventInfo() {
    if (this.event == null)
        return;

    llInfo.setVisibility(View.VISIBLE);

    tvInfoLeague.setText(event.getLeagueName());

    long timestamp = event.getTimestamps().plannedStart;
    Calendar cal = Calendar.getInstance();
    cal.setTimeInMillis(timestamp);
    tvInfoStart.setText(DateFormat.format( inFormat: "EEEE, dd. MMMM yyyy HH:mm", cal).toString());

    if (event.getStadium() != null) {
        llInfoStadium.setVisibility(View.VISIBLE);
        tvInfoStadium.setText(event.getStadium());
    }
    else {
        llInfoStadium.setVisibility(View.GONE);
    }

    if (event.getReferee() != null) {
        llInfoReferee.setVisibility(View.VISIBLE);
        tvInfoReferee.setText(event.getReferee());
    }
    else {
        llInfoReferee.setVisibility(View.GONE);
    }

    if (event.getSpectators() != null) {
        llInfoViewers.setVisibility(View.VISIBLE);
        tvInfoViewers.setText(event.getSpectators());
    }
    else {
        llInfoViewers.setVisibility(View.GONE);
    }
}
```

Please click and drag the slider handles to enter your answer.



* 13 **Tick "Strongly agree" if 2^8 equals 256.**

Choose one of the following answers

○ Strongly disagree

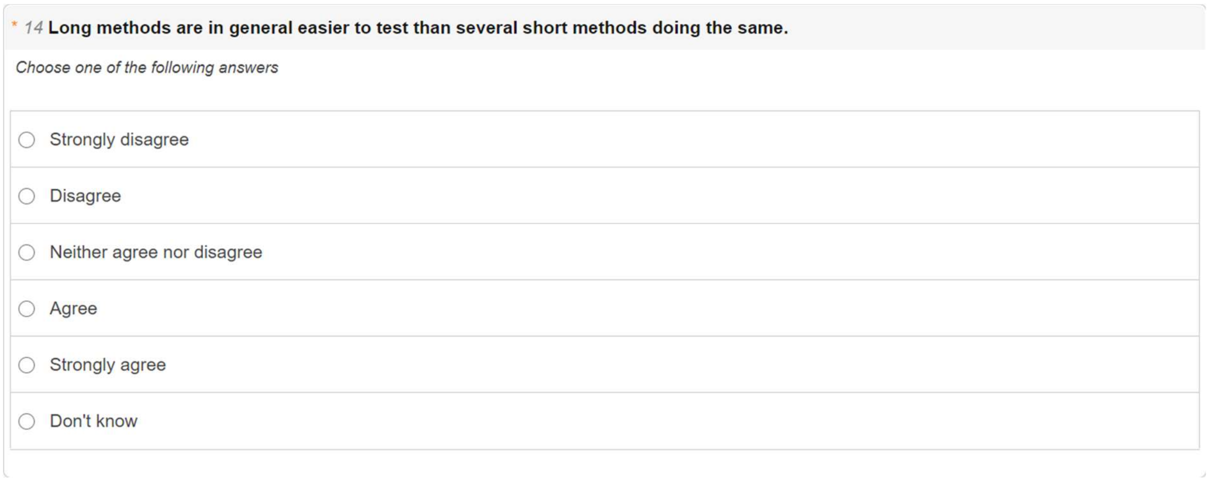
○ Disagree

○ Neither agree nor disagree

○ Agree

○ Strongly agree

○ Don't know



* 14 **Long methods are in general easier to test than several short methods doing the same.**

Choose one of the following answers

○ Strongly disagree

○ Disagree

○ Neither agree nor disagree

○ Agree

○ Strongly agree

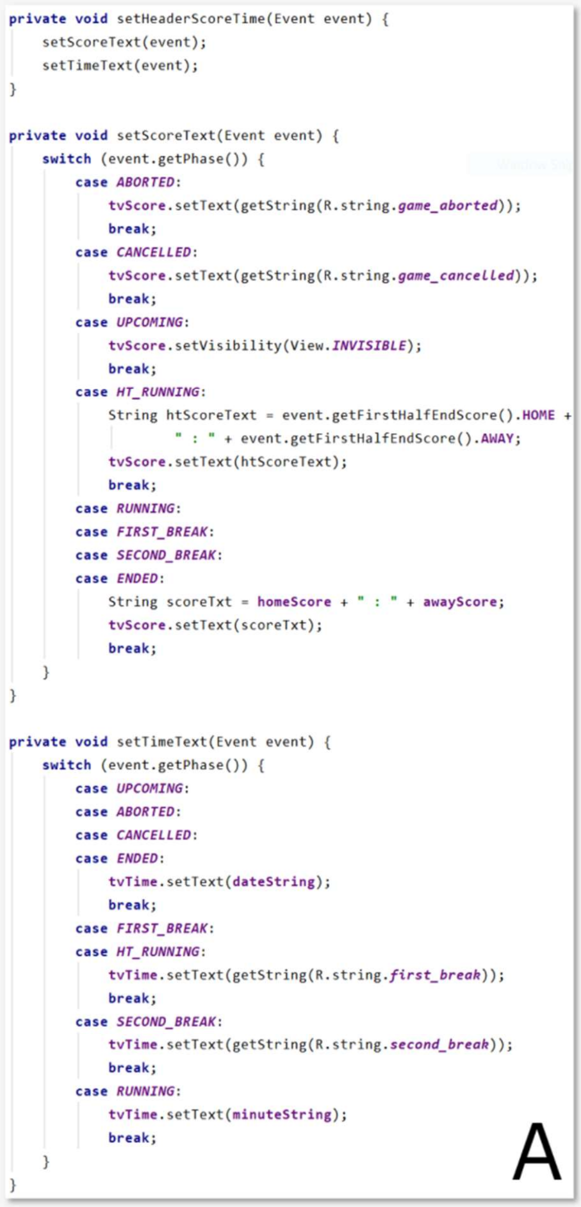
○ Don't know

15

In A there are two switch statements with the same cases. Calling "setHeaderScoreTime()" sets the score and the time depending on the phase of the event.

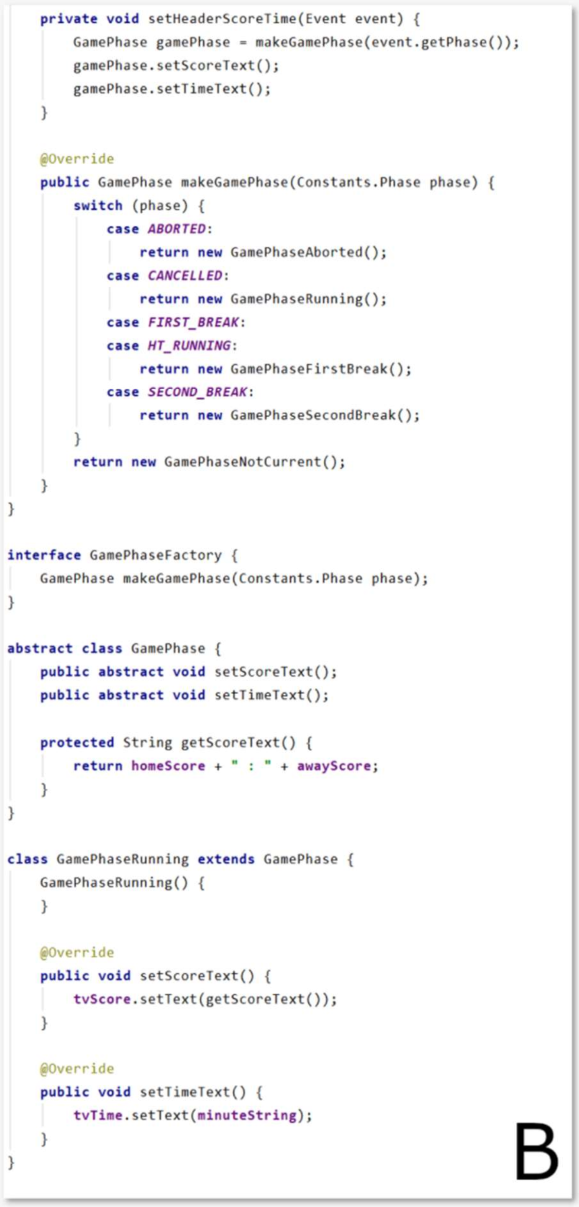
In B there is created a GamePhase object depending on the phase of the event. Then its implementation of setting the score and time is called (only the implementation of the case "RUNNING" can be seen). This is polymorphism.

Which one is cleaner?



```java
private void setHeaderScoreTime(Event event) {
    setScoreText(event);
    setTimeText(event);
}

private void setScoreText(Event event) {
    switch (event.getPhase()) {
        case ABORTED:
            tvScore.setText(getString(R.string.game_aborted));
            break;
        case CANCELLED:
            tvScore.setText(getString(R.string.game_cancelled));
            break;
        case UPCOMING:
            tvScore.setVisibility(View.INVISIBLE);
            break;
        case HT_RUNNING:
            String htScoreText = event.getFirstHalfEndScore().HOME +
                    " : " + event.getFirstHalfEndScore().AWAY;
            tvScore.setText(htScoreText);
            break;
        case RUNNING:
        case FIRST_BREAK:
        case SECOND_BREAK:
        case ENDED:
            String scoreTxt = homeScore + " : " + awayScore;
            tvScore.setText(scoreTxt);
            break;
    }
}

private void setTimeText(Event event) {
    switch (event.getPhase()) {
        case UPCOMING:
        case ABORTED:
        case CANCELLED:
        case ENDED:
            tvTime.setText(dateString);
            break;
        case FIRST_BREAK:
        case HT_RUNNING:
            tvTime.setText(getString(R.string.first_break));
            break;
        case SECOND_BREAK:
            tvTime.setText(getString(R.string.second_break));
            break;
        case RUNNING:
            tvTime.setText(minuteString);
            break;
    }
}
```

A

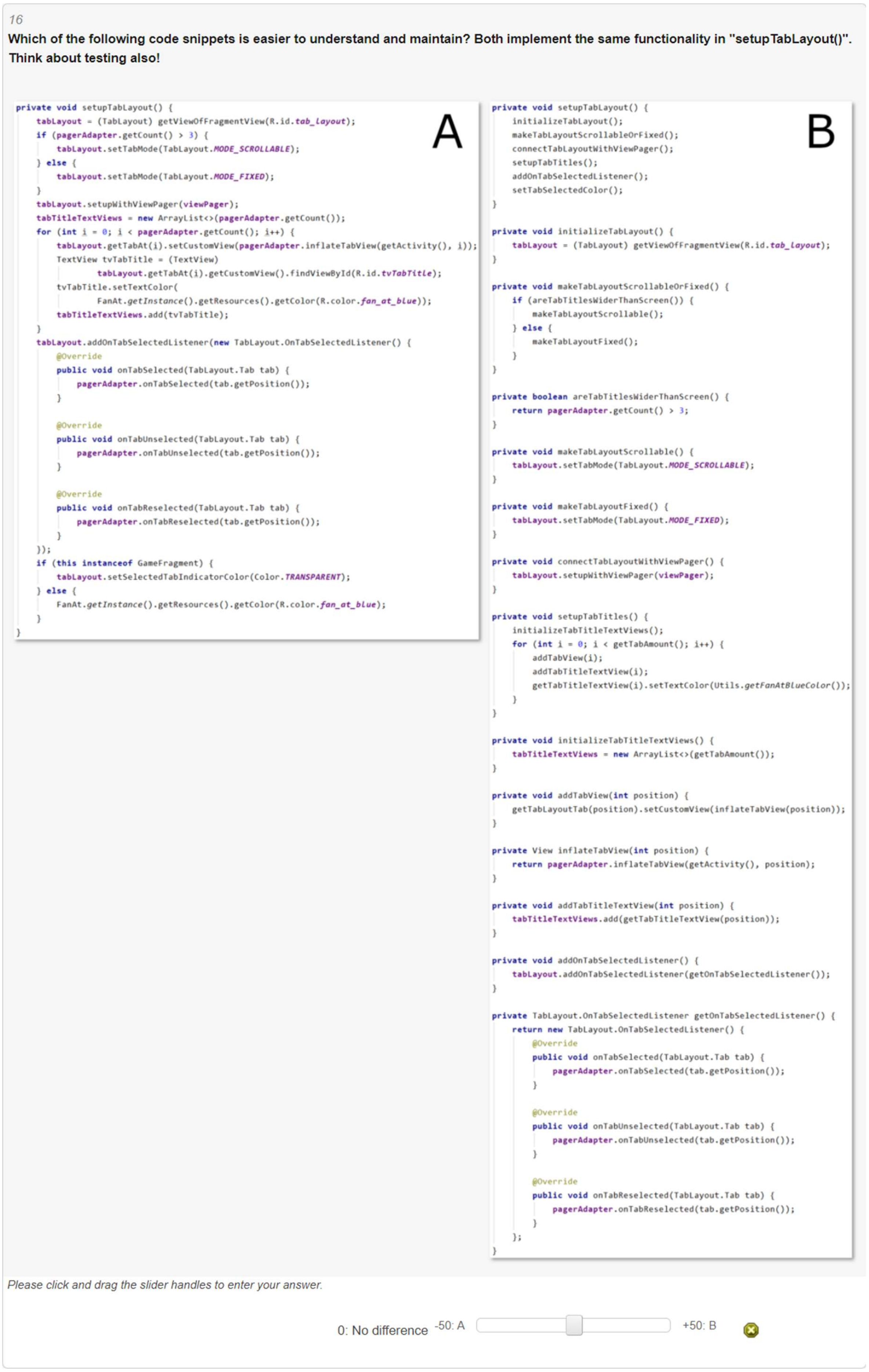


```java
    private void setHeaderScoreTime(Event event) {
        GamePhase gamePhase = makeGamePhase(event.getPhase());
        gamePhase.setScoreText();
        gamePhase.setTimeText();
    }

    @Override
    public GamePhase makeGamePhase(Constants.Phase phase) {
        switch (phase) {
            case ABORTED:
                return new GamePhaseAborted();
            case CANCELLED:
                return new GamePhaseRunning();
            case FIRST_BREAK:
            case HT_RUNNING:
                return new GamePhaseFirstBreak();
            case SECOND_BREAK:
                return new GamePhaseSecondBreak();
        }
        return new GamePhaseNotCurrent();
    }
}

interface GamePhaseFactory {
    GamePhase makeGamePhase(Constants.Phase phase);
}

abstract class GamePhase {
    public abstract void setScoreText();
    public abstract void setTimeText();

    protected String getScoreText() {
        return homeScore + " : " + awayScore;
    }
}

class GamePhaseRunning extends GamePhase {
    GamePhaseRunning() {
    }

    @Override
    public void setScoreText() {
        tvScore.setText(getScoreText());
    }

    @Override
    public void setTimeText() {
        tvTime.setText(minuteString);
    }
}
```

B

Please click and drag the slider handles to enter your answer.

0: No difference -50: A +50: B



16

Which of the following code snippets is easier to understand and maintain? Both implement the same functionality in "setupTabLayout()". Think about testing also!

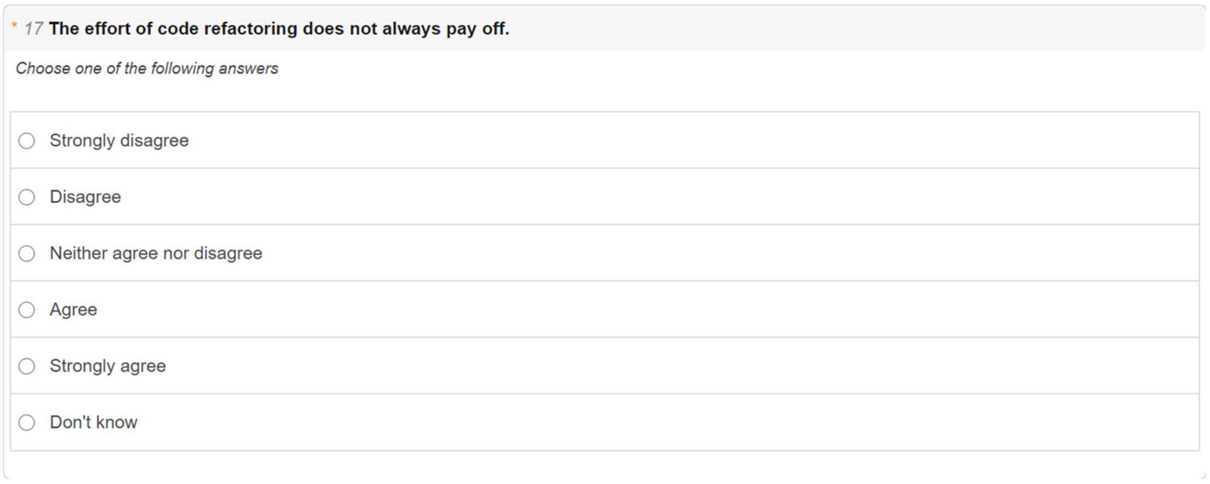
A

```java
private void setupTabLayout() {
    tabLayout = (TabLayout) getViewOfFragmentView(R.id.tab_layout);
    if (pagerAdapter.getCount() > 3) {
        tabLayout.setTabMode(TabLayout.MODE_SCROLLABLE);
    } else {
        tabLayout.setTabMode(TabLayout.MODE_FIXED);
    }
    tabLayout.setupWithViewPager(viewPager);
    tabTitleTextViews = new ArrayList<>(pagerAdapter.getCount());
    for (int i = 0; i < pagerAdapter.getCount(); i++) {
        tabLayout.getTabAt(i).setCustomView(pagerAdapter.inflateTabView(getActivity(), i));
        TextView tvTabTitle = (TextView)
                tabLayout.getTabAt(i).getCustomView().findViewById(R.id.tvTabTitle);
        tvTabTitle.setTextColor(
                FanAt.getInstance().getResources().getColor(R.color.fan_at_blue));
        tabTitleTextViews.add(tvTabTitle);
    }
    tabLayout.addOnTabSelectedListener(new TabLayout.OnTabSelectedListener() {
        @Override
        public void onTabSelected(TabLayout.Tab tab) {
            pagerAdapter.onTabSelected(tab.getPosition());
        }

        @Override
        public void onTabUnselected(TabLayout.Tab tab) {
            pagerAdapter.onTabUnselected(tab.getPosition());
        }

        @Override
        public void onTabReselected(TabLayout.Tab tab) {
            pagerAdapter.onTabReselected(tab.getPosition());
        }
    });
    if (this instanceof GameFragment) {
        tabLayout.setSelectedTabIndicatorColor(Color.TRANSPARENT);
    } else {
        FanAt.getInstance().getResources().getColor(R.color.fan_at_blue);
    }
}
```

B

```java
private void setupTabLayout() {
    initializeTabLayout();
    makeTabLayoutScrollableOrFixed();
    connectTabLayoutWithViewPager();
    setupTabTitles();
    addOnTabSelectedListener();
    setTabSelectedColor();
}

private void initializeTabLayout() {
    tabLayout = (TabLayout) getViewOfFragmentView(R.id.tab_layout);
}

private void makeTabLayoutScrollableOrFixed() {
    if (areTabTitlesWiderThanScreen()) {
        makeTabLayoutScrollable();
    } else {
        makeTabLayoutFixed();
    }
}

private boolean areTabTitlesWiderThanScreen() {
    return pagerAdapter.getCount() > 3;
}

private void makeTabLayoutScrollable() {
    tabLayout.setTabMode(TabLayout.MODE_SCROLLABLE);
}

private void makeTabLayoutFixed() {
    tabLayout.setTabMode(TabLayout.MODE_FIXED);
}

private void connectTabLayoutWithViewPager() {
    tabLayout.setupWithViewPager(viewPager);
}

private void setupTabTitles() {
    initializeTabTitleTextViews();
    for (int i = 0; i < getTabAmount(); i++) {
        addTabView(i);
        addTabTitleTextView(i);
        getTabTitleTextView(i).setTextColor(Utils.getFanAtBlueColor());
    }
}

private void initializeTabTitleTextViews() {
    tabTitleTextViews = new ArrayList<>(getTabAmount());
}

private void addTabView(int position) {
    getTabLayoutTab(position).setCustomView(inflateTabView(position));
}

private View inflateTabView(int position) {
    return pagerAdapter.inflateTabView(getActivity(), position);
}

private void addTabTitleTextView(int position) {
    tabTitleTextViews.add(getTabTitleTextView(position));
}

private void addOnTabSelectedListener() {
    tabLayout.addOnTabSelectedListener(getOnTabSelectedListener());
}

private TabLayout.OnTabSelectedListener getOnTabSelectedListener() {
    return new TabLayout.OnTabSelectedListener() {
        @Override
        public void onTabSelected(TabLayout.Tab tab) {
            pagerAdapter.onTabSelected(tab.getPosition());
        }

        @Override
        public void onTabUnselected(TabLayout.Tab tab) {
            pagerAdapter.onTabUnselected(tab.getPosition());
        }

        @Override
        public void onTabReselected(TabLayout.Tab tab) {
            pagerAdapter.onTabReselected(tab.getPosition());
        }
    };
}
```

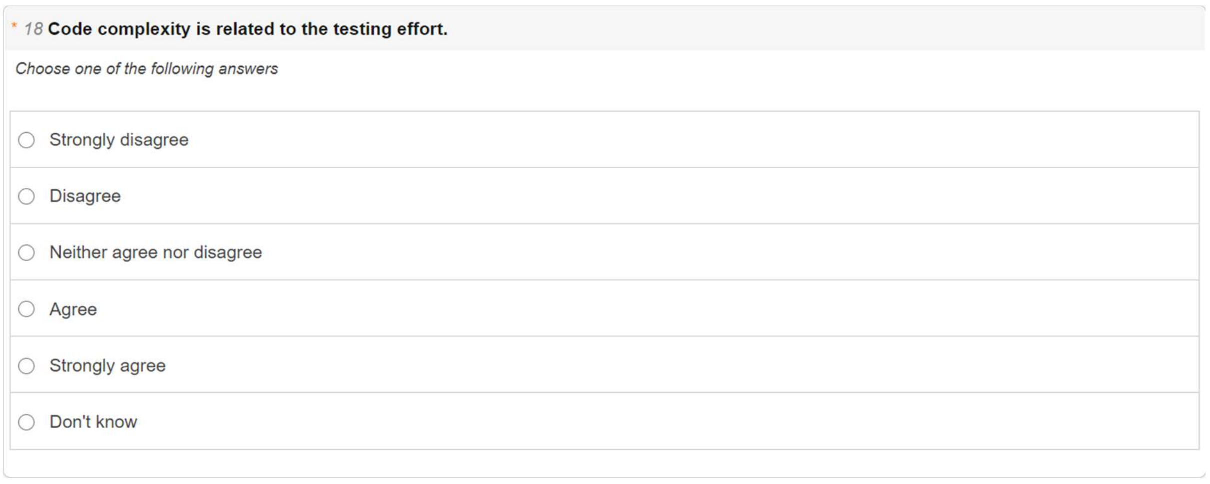
Please click and drag the slider handles to enter your answer.



* 17 **The effort of code refactoring does not always pay off.**

Choose one of the following answers

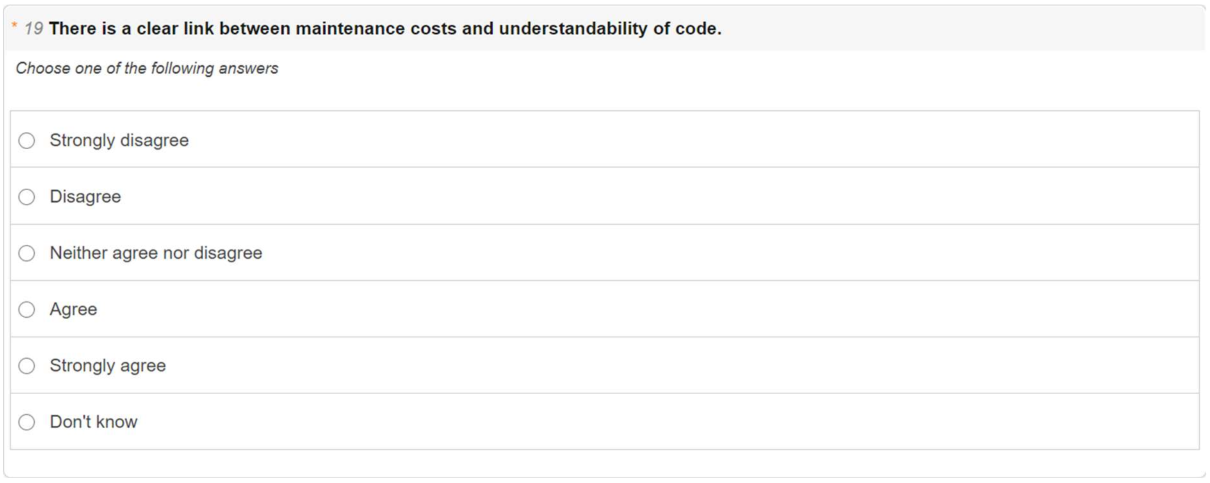
- ○ Strongly disagree
- ○ Disagree
- ○ Neither agree nor disagree
- ○ Agree
- ○ Strongly agree
- ○ Don't know



* 18 **Code complexity is related to the testing effort.**

Choose one of the following answers

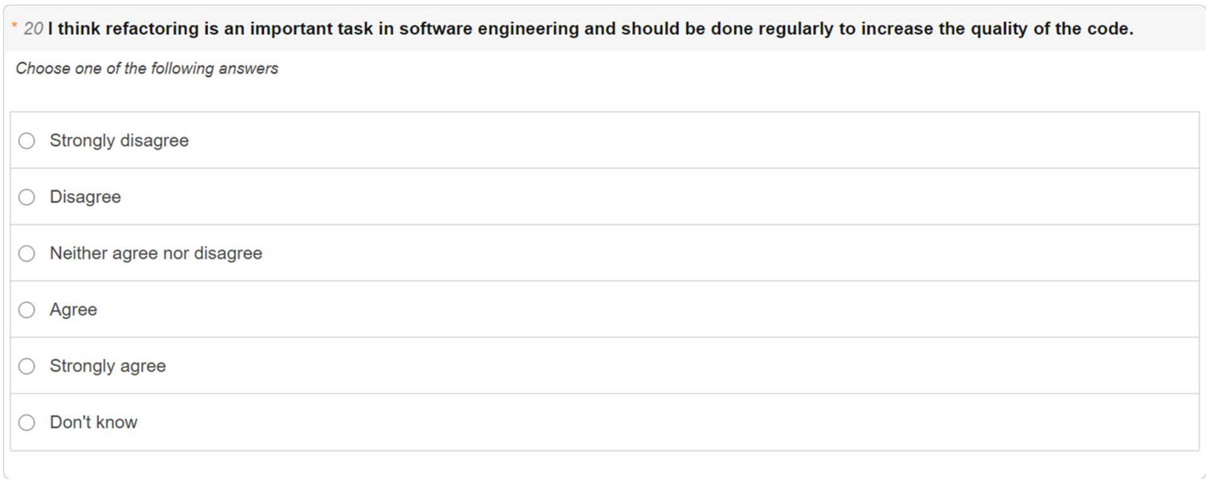
- ○ Strongly disagree
- ○ Disagree
- ○ Neither agree nor disagree
- ○ Agree
- ○ Strongly agree
- ○ Don't know



* 19 **There is a clear link between maintenance costs and understandability of code.**

Choose one of the following answers

- ○ Strongly disagree
- ○ Disagree
- ○ Neither agree nor disagree
- ○ Agree
- ○ Strongly agree
- ○ Don't know


* 20 **I think refactoring is an important task in software engineering and should be done regularly to increase the quality of the code.**

Choose one of the following answers

- ○ Strongly disagree
- ○ Disagree
- ○ Neither agree nor disagree
- ○ Agree
- ○ Strongly agree
- ○ Don't know

Closing Words

Thank you for taking part in my survey.

If you have any questions, please do not hesitate to contact me (lorenz.kofler@student.tugraz.at).