UNIVERSITY OF TARTU

Institute of Computer Science

Computer Science Curriculum

Liisa Sakerman

Overview of Code Smell Detection Tools for Android and iOS Applications

Bachelor's Thesis (9 ECTS)

Supervisor: Kristiina Rahkema

Tartu 2020

Overview of Code Smell Detection Tools for Android and iOS Applications

Abstract:

The aim of this thesis is to give an up-to-date overview of code smell detection tools for Android and iOS applications. There are numerous tools available for developers that help detect code smells and refactor code. Most tools detect only some types of code smells and sometimes use different rules and methodologies for detection and analysis. In this thesis, the code smells found by different code smell detection tools are determined and an overview of those tools for Android and iOS applications is given. Tools are then tested on numerous different Android applications.

Keywords: Code smells, code smell detection, mobile applications

CERCS: P170 Computer science, numerical analysis, systems, control

Ülevaade lõhnavat koodi tuvastavatest programmidest Androidi ja iOS-i rakendustarkvaradele

Lühikokkuvõte:

Bakalaureusetöö eesmärk on anda ajakohane ülevaade lõhnava koodi tuvastamise programmidest, mis on suunatud Androidi ja iOS-i keskkondadele. Arendajatele on olemas mitmeid abivahendeid, mis tuvastavad lõhnavat koodi ning aitavad koodi refaktoreerida. Enamik programme tuvastavad ainult osa kõikidest lõhnava koodi tüüpidest ning mõnikord kasutavad erinevaid reegleid ja metoodikaid nende analüüsil ning määramisel. Töös tehakse kindlaks käsitletavate programmide leitud koodilõhnad ning antakse ülevaade nende sobivusest Androidi ja iOS-i rakendustele. Programmid testitakse mitme erineva Androidi rakenduse peal.

Võtmesõnad: Lõhnav kood, lõhnava koodi tuvastamine, mobiilirakendused

CERCS: P170 Arvutiteadus, arvutusmeetodid, süsteemid, juhtimine

Table of Contents

1. Introduction **5**

2. Background and Related Work **7**

2.1. Code Smells 7

2.2. Code Smells Described by Martin Fowler and Kent Beck 7

2.3. Taxonomy of Code Smells 11

2.4. Code Smell Detection Tools 12

2.5. Related Work 13

3. Research Methodology **15**

3.1. Code Smell Detectors for Android Applications 15

3.2. Code Smell Detectors for iOS Applications 18

3.3. Applications Selected for Analysis 19

4. Results and Analysis **20**

4.1. Chosen Code Smell Detectors 20

4.2. Collective Results 22

4.3. Individual Results 24

4.3.1. PMD 24

4.3.2. DesigniteJava 25

4.3.3. CheckStyle 25

4.3.4. Kritika 26

4.3.5. Embold 27

4.3.6. JArchitect 28

4.3.7. SonarQube 29

4.3.8. JDeodorant 30

4.4. Usability 31

4.5. Author’s Experience 32

5. Discussion and Recommendations **34**

6. Conclusion **36**

References **37**

Appendix **44**

I. Results for Tested Code Smell Detectors.. 44

II. Licence.. 45

1. Introduction

Software maintenance is considered to be one of the longest and costliest parts of a software lifecycle [1]. It is a continuous process, that exists in the pre-delivery stage as well as the post-delivery stage of a software product [2]. It is important to design software that is easy to maintain. According to Reshi et al. [3], code smells that often manifest in poor design choices or implementation issues, are found to have serious effect on software maintainability. They note that code smells do not hinder the functionality and performance of the software; however, code full of smells has a higher risk of system failure in the future.

Code smells are characteristics in the source code of a program that are associated with bad design and incorrect programming practices [4]. Fowler [5] has written, that code smells are the first surface indicators of a flaw, that usually corresponds to a deeper problem in the source code. Code smells don't always mean there is a problem in the source code of a program so the code must be analyzed to detect an underlying problem [5].

In the increasingly competitive market of mobile development, developers are pushed to constantly release new versions and updates for mobile applications to retain their customers [6]. Developers may feel the need to adopt bad code design or implementation practices, that can lead to code smells. Numerous scientific papers [7, 8] have proposed solutions and developed programs to fix the issue by automatically identifying code smells and analyzing the effect of those code smells on the system.

Various tools exist for developers, that find code smells and help refactor the code. Most of the existing tools use varying rules and methods to analyze and classify code smells, meaning they cannot identify all types of code smells.

The aim of this thesis is to give an up-to-date overview of code smell detection tools for Android and iOS applications. The code smells found by different code smell detectors are identified, those detectors are then tested on multiple Android mobile applications. Results of the detectors are then compared. Three research questions are answered in this thesis:

RQ1: Do these tools detect the same code smells?

RQ2: Which tools have better usability?

RQ3: Where are the gaps in code smell detection?

The first chapter describes code smells, gives an overview of available code smell detection tools and summarizes previous work on this topic. The second chapter gives an overview of the research methodology describing the selection of code smell detection tools and applications for analysis. The third chapter presents the collective and individual results of the analysis and the usability of the tools is described. The fourth chapter contains discussion and recommendations. The final chapter presents the conclusions.

2. Background and Related Work

This chapter gives an overview of related background information. This includes the definition of code smells and refactoring, descriptions code smells described by Fowler and Beck, code smell detection tools and related work on this topic.

2.1. Code Smells

Martin Fowler and Kent Beck [4] introduced the concept of code smells in 1999, when describing the need for refactoring code. According to them, certain structures in the code often suggest the possibility of refactoring, however there is no set criteria for when it should be done. Fowler [5] describes bad smells in code as something that's easy to spot. They don't always indicate a problem and the code must be analysed in depth to understand, if it should be refactored [5].

Fowler also introduces the practice of refactoring [4]. According to him, refactoring is a controlled technique, where the design and internal structure of the code is changed in a way that it becomes more maintainable as it helps potentially reduce the occurrence of bugs. The changes are done so that the external behaviour of the code is not changed. Refactoring can also help with faster development of the programs. A study conducted by Forrester Consulting [9] found, that mobile-based software applications are released at least on quarterly bases, if not more often. The study concluded, that hence the development cycles for mobile domains are shorter than for regular desktop programs, mobile developers can benefit greatly from refactoring practices.

2.2. Code Smells Described by Martin Fowler and Kent Beck

The code smells analysed in this thesis are the ones introduced by Martin Fowler and Kent Beck. To help programmers understand different possibilities and ways as well as the importance of writing more maintainable code, Fowler and Beck [4] have described 22 different code smells, that most often appear in object-oriented projects. This study focuses on the maintenance related problems of software systems that are most often caused by object-oriented code smells. Android-specific code smells are however most associated with energy efficiency problems of mobile domain instead of maintenance [10]. For this reason, the code smells described by Fowler and Beck were selected as the basis for this thesis. The following are the 22 code smells summarized from Fowler and Beck [4]:

Duplicated Code

Duplicated Code is one of the most often occurring code smells. Duplicated Code happens when the same code structure can be found in more than one place in the code. It can be in the form of the same expression in two methods of the same class or in two sibling subclasses. Code that is similar but not exactly the same or methods that do the same thing with different algorithm are also considered duplicate code [4].

Long Method

Long Methods are more difficult to understand than shorter methods. They often need to be broken into multiple smaller methods. Method can be considered long when it has too many lines of code or it contains multiple conditionals and loops [4].

Large Class

Large Classes often carry too many instance variables. They can also just contain too much code, which can lead to duplicate code and other maintainability issues. Those classes should be broken into multiple classes or subclasses, or interfaces could be extracted to stop the class from trying to do too much [4].

Long Parameter List

Parameter Lists tend to get long when they have passed everything the method needs. However, in object-oriented programs parameter lists can be much smaller when using objects, just enough can be passed for the method to get to the needed data itself. Long Parameter Lists are hard to understand and maintain as the constant change in needed data makes them inconsistent [4].

Divergent Change

Divergent Change is present when one clear point in the system cannot be pinpointed to make a needed change and when one class is changed in several ways for different reasons. It is when many unrelated methods are altered after one change to the class. Each object should be changed only as a result of one kind of alteration. Class extraction can be used to fix this problem and combine all changing aspects to a new class [4].

Shotgun Surgery

Shotgun Surgery is present when making a change, a lot of little changes need to be made in a lot of different classes. In Divergent Change one class goes through many kinds of changes, but with Shotgun Surgery the opposite is done- one change alters many classes. Changes should be moved to a single class [4].

Feature Envy

Feature Envy happens, when a method uses data from other classes more than its own. For example, a method invokes too many getting methods on another object. Then that method must be moved to the other class [4].

Data Clumps

Data items, that appear together in multiple locations in the code, are called Data Clumps. They should be made into their own object. Clumps of multiple fields can be turned into an object and Long Parameter Lists can be slimmed down with introducing parameter object. That way the parameter lists can be shortened and method calling is simplified [4].

Primitive Obsession

Primitive Obsession happens, when instead of separate objects, small tasks are handled with primitives. Data values should be replaced with objects and type codes with classes. Primitives can also be extracted as a class or introduced as a parameter object [4].

Switch Statements

Often the same Switch Statements are present in multiple parts of the code and the problem of duplication arises. If a new clause is added to the Switch Statement, it must be changed in all other Switch Statements as well. Usually, polymorphism should be considered instead of Switch Statements to improve code organization [4].

Parallel Inheritance Hierarchies

This code smell is a special case of Shotgun Surgery. In this case, when a subclass of one class is made, a subclass of another class must also be made. This creates unnecessary duplication. To eliminate this smell, instances of one hierarchy must refer to the instances of the other. Then, the hierarchy in the referred class can be removed [4].

Lazy Class

This is a case of a class, that isn't doing enough. It can be a class that got downsized with refactoring or a class that was added with the intent for later implementation, that never happened. Classes, that are nearly useless, can be deleted by moving their features to another class. Subclasses with few functionalities can be merged with their parent classes [4].

Speculative Generality

Speculative Generality happens, when a part of the code is written in the hopes of later implementation. This creates classes, methods, and fields that are unused, and the code is harder to maintain and understand. In this case, often the only users of a method or a class are test cases [4].

Temporary Field

Sometimes, an object has an instance variable set only in certain circumstances. This kind of code can be difficult to understand, as some variables appear to not be in use. New class can be created for the unused variables or conditional code can be eliminated by creating an alternative component [4].

Message Chains

Message Chains happens, when a series of calls take place in the form of client asking one object for another object, who then asks for another object and so on. Oftentimes this comes in the form of a line of getThis methods or as a sequence of temps. Any changes in this kind of relationships lead to changes in the client [4].

Middle Man

A Middle Man class exists only to delegate work from one class to another. The Middle Man can be removed or it can be turned into a subclass of a real object to extend behaviour [4].

Inappropriate Intimacy

One class uses the internal fields and methods of another class too much. Classes should not know about the implementations of other classes, that way they are easier to maintain. Methods and fields can be moved from one class to another to reduce this problem; classes can also be extracted to separate the commonality. Subclasses and parent classes can be separated when replacing delegation with inheritance [4].

Alternative Classes with Different Interfaces

These are classes that have the same functionalities and purpose, but different signatures. Methods, that do the same but have different names, can be renamed. Methods can be moved to move behaviour and make the signature of the classes the same. With refactoring oftentimes one of the classes can be deleted [4].

Incomplete Library Class

Library Classes can become unsuitable for user needs; however, they cannot be changed to cater to the functions the user needs. To change just a couple of new methods to the library class, foreign method can be introduced. For bigger changes local extension must be introduced [4].

Data Class

Classes, that only have fields and getting and setting methods, are called data classes. They are just containers for data and don't have any additional functionality. Methods of the user class may be moved to the data class to give it more purpose [4].

Refused Bequest

Refused Bequest happens, when subclasses inherit the data of their parents, but they only need some of it and the rest goes unused. If the subclass doesn't support the interface of the superclass, interface must be eliminated. If the inheritance is appropriate, new sibling class can be created and given all the unused methods [4].

Comments

Heavily commented code can be an indicator of problems in the code. Comments act as a cover for deeper issues in the code. When there is a need for comments, usually the code should be restructured or refactored to make it more intuitive [4].

2.3. Taxonomy of Code Smells

Mantyla et al. [11] have proposed categorization for those 22 code smells. The smells are divided into six groups by their similarities and relations to each other. The following are those six classes proposed by Mantyla et al. [11] shortly summarized:

Bloaters

Bloaters represent code, that has grown too much in size and is hard to handle and maintain. The code smells in this category are Long Method, Large Class, Primitive Obsession, Long Parameter List, and Data Clumps [11].

Object-Orientation Abusers

The code smells in this category represent cases where the code doesn't fully use the potential of object-oriented design. This can be in the form of not using subclasses when needed, violating information hiding ideas or misusing inheritance design. These smells are Switch

Statements, Temporary Field, Refused Bequest, Alternative Classes with Different Interfaces, and Parallel Inheritance Hierarchies [11].

Change Preventers

Change Preventers make it harder to modify the code in the long run and violate the rule that possible changes and classes should have a one-to-one relation. The code smells in this category are Divergent Change and Shotgun Surgery [11].

Dispensables

The code smells in this category represent code, that should be removed as it is unnecessary, unused, or not doing enough. These are Lazy Class, Data Class, Duplicate Code, and Speculative Generality code smells [11].

Encapsulators

Encapsulators represent problems in communication between objects and the way data is accessed. The code smells here are Message Chains and Middle Man [11].

Couplers

The code smells in this category are coupling related and present a problem, where coupling is high, which is against object-oriented design principles. These smells are Feature Envy and Inappropriate Intimacy [11].

2.4. Code Smell Detection Tools

Bad smells in the program's source code can either be detected by manual or automated analysis. According to Moha et al. [12] manual detection of smells in the code is a laborious and error-prone activity, especially for larger projects. They explain that smells can exist across multiple classes or methods and searching for them based only on their description, which often leave room for interpretation, is very resource-consuming. Therefore, numerous automated smell detectors have been proposed.

Fernandes et al. [13] found in their study 84 different tools for bad smell detection, 29 of which were available online for download. Two thirds of those tools supported Java, which is the predominant development language for Android applications. Java was followed by C-languages, which are used for iOS applications programming.

Different tools can use different detection techniques for code smell detection. Perocelli et al. [14] found, that most of the automated code smell detection tools are metric-based. Those tools compute a set of code metrics of the source code, combine them into a set of rules and compare the metrics against empirically identified thresholds. A code smell is present, if the threshold is exceeded. They conclude that the results vary greatly depending on the set thresholds and metric counts.

Trees, namely Abstract Syntax Trees (AST) can also be used for code smell detection. The source code of a program is parsed into an abstract syntax tree and the nodes are then compared for code smell findings [15]. Program Dependence Graph (PDG) mapping is a similar approach as it compares statements to find code smells [15]. Textual analysis is also used for smell detection. Conceptual information is extracted from the source code by using the textual analysis techniques and is used to identify code smells [16]. Logic meta–programming is used for getting information about the source code as it helps to express structural relations, that allow the tool to detect bad smells [17]. Machine learning – based techniques [14] have also been used for code smell detection as well as token analysis [18].

2.5. Related Work

The topic of code smells has been relevant since the late 1990s and it became more well-known after the idea featured in the book *Refactoring: Improving the Design of Existing Code* by Fowler [4]. Since then there have been numerous research papers and theses that have touched upon the subject. Since the rise of mobile applications, code smells appearing in Android and iOS environments have become more relevant for developers.

Two related thesis papers were found to analyse the previous findings of code smell detection tools for the mobile domain: a Master’s thesis paper *Detecting Code Smells in Android Applications* by Lim [19] and a Master’s thesis paper *Code Smells in the Mobile Applications Domain* by Verloop [6].

Lim’s study [19] is a 2017 research on the detection performance of existing code smell detection tools, that were not designed with Android in mind. Lim used existing smell detectors on Android apps to explore their detection performance. The empirical study aimed to investigate how well the existing smell detectors perform on Android applications and explored if the smells detected by different detectors vary. Lim considered 9 different Java smell detectors: Checkstyle, DECOR, inFusion, iPlasma, JDeodorant, PMD, Stench Blossom, TACO, JSpIRIT. From those tools, 4 were chosen and tested: DECOR, JDeodorant, TACO, JSpIRIT. It was found that the detectors performed relatively poor in terms of what percentage of found smells were actual code smells and how many true smells were found. The best result was received with the DECOR tool. Furthermore, different tools detected different smells.

Verloop's study [6] in 2013 investigated, how mobile software developers can limit the number of code smells in their applications. One of the research questions was aimed to find out, which tools can be used to find code smells in mobile apps and what is the quality of those tools. To find code smells in Android projects, a selection of code smell detection tools was assessed. At the time of the research, not many detectors were accessible. The following tools were available: JDeodorant, Lint, Checkstyle, PMD, UCDetector, iPlasma, DECOR, Stench Blossom. From this selection, only 4 tools were selected for further analysis: JDeodorant, Checkstyle, PMD, UCDetector. Other tools were dropped as they weren't freely available, compatible with newer versions of Eclipse that is required for Android, or they didn't find any actual code smells.

Reimann et al. [20] constructed a smell catalogue of 13 Android-specific code smells. The catalogue offered rules and definitions of the smells. Another study conducted by Almalki [21] focused on studying the tools made specifically for detecting Android-specific code smells. Only two tools were found, that could identify a limited number of Android code smells. One of those tools was aDoctor, which uses the Abstract Syntax Tree of the application's source code and applies the rules proposed by Reimann et al. The second tool was Paprika, which was able to detect 4 Android-specific code smells from the Reimann et al.'s smell catalogue, in addition to multiple Object Oriented code smells.

As these studies were conducted several years ago, it is expected that some of these findings are not applicable anymore today with the rapid change of the mobile domain. Some mentioned and analysed tools may not be accessible anymore and some may have changed drastically. It is also possible, that some new detectors have been introduced in the form of an academic project or an independent commercial product. There does not exist an up-to-date overview of current code smell detection tools. The aim of this thesis is to give a recent overview of the available technologies for code smell detection and compare the findings with previous work to analyse, what has changed and what are the latest options to achieve well maintained code in the mobile domain.

3. Research Methodology

This chapter gives an overview of the methodology used in the research. Selection of code smell detection tools and applications for the analysis is described.

3.1. Code Smell Detectors for Android Applications

To carry out the experimental part of this thesis, a search for code smell detection tools was performed. For Android application analysis, the tools needed to support mainly Java programming language. Tools for iOS application code smell detection were found, but only overview of them is given. As Mac OS is required for compiling iOS applications, no further testing of those tools could be carried out.

The search was based on Lim's and Verloop's thesis papers [6, 19]. In addition to the code smell detection tools mentioned in those papers, additional search on the Internet was performed. The search was performed on Google with the keywords:

- Detect code smells
- Code smell detector/detectors
- Java smell detector
- Code smell detection tools for Android
- Static code analysis tool

This search showed results for different tools found from numerous sources: previous research projects, forums, blog posts, websites for related commercial products, articles. Every suggested tool, that supported Java and was said to be related to this topic or the search keywords, was written down as a potential result. This search also found detectors used in the two aforementioned thesis papers [6, 19]. Total of 31 potential code smell detection tools for Java that could potentially be run on Android applications were found and are presented in Table 1.

Table 1. Code smell detection tools found for Android applications.

Tool	**Supported Languages**	**Found Code Smells**	**Plug-in/ Standalone Tool**	**Was It Selected for Analysis?**
JDeodorant [22]	Java	Feature Envy**,** Type/State Checking**,** Long Method**,** God Class and Duplicated Code.	Eclipse plug-in	Yes
InFusion [23]	Java, C, C++	Duplicated code, Feature Envy, God Class, etc.	Standalone	No (not accessible)
PMD [24]	Java and 7 other	Duplicated code, Large class, Long Method, Long Parameter List etc.	Sandalone	Yes
JSpIRIT [25]	Java	Brain Class, Brain Method, Data Class, Disperse Coupling, Feature Envy and 5 other	Eclipse plug-in	No (not runnable)
Stench Blossom [26]	Java	Feature Envy, Data Clumps, Long Method, Instanceof, Large Class, Message Chains, Switch Statements and Typecast	Eclipse plug-in	No (Eclipse versions since 3.4 are not supported)
iPlasma [27]	Java, C++	Large Class, Feature Envy, Duplicated Code	Standalone Tool	No (not accessable)
Décor [28]	Java	Long Method, No Parameter, No Inheritance, No Polymorphism, Procedural Name, Use Global Variable	Ptidej plug-in (part of a tool suite)	No (not accessable)
aDoctor [29]	Java	15 Android specific design flaws	Standalone Tool	No (no object-oriented code smells found)
Paprika [30]	Java	Blob Class, Swiss Army Knife, Long Method, Complex Class and 13 Android specific code smells	Standalone Tool	No (not working correctly since Java 9)
DesigniteJava [31]	Java	Detects 17 design smells and 10 implementation smells	Standalone Tool	Yes
CheckStyle [32]	Java	Large Class, Long Method, Long Parameter List, Complex Conditional, Empty Catch Clause, Long Statement	Standalone Tool and various plug-ins supported	Yes
Clone Digger [33]	Java, Python	Duplictated Code	Standalone Tool	No (not accessable)
TrueRefactor [34]	Java	Lazy Class, Long Method, Temporary Field, Large Class, Shotgun Surgery	Standalone Tool	No (not accessable)
SYMake [35]	Java, C	Cyclic Dependency, Duplicated Prerequisites, Loop of Revursive Variables, Rule Inclusion etc.	Standalone Tool	No (not accessable)
NiCad [36]	Java, C, C#, etc.	Duplicated code	Standalone Tool	No (finds only duplicated code)
jCosmo [37]	Java	InstanceOf, Switch Statement typecast	Standalone Tool	No (not runnable)
Java Clone Detector [38]	Java	Duplicated Code	Standalone Tool	No (finds only duplicate code)

DECKARD [39]	Java	Duplicated Code	Standalone Tool	No (finds only duplicate code)
ConcernReCS [40]	Java	Primitive Concern Constant, Static Concern Constant, Attribute of a non-Dedicated Type, Divergent Join Point, Element out of Inheritance Tree	Eclipse plug-in	No (not accessable)
Code Bad Smell Detector [41]	Java	Data Clumps, Switch Statements, Speculative Generality, Message Chains, Middle Man	Ecplise plug-in	No (not accessable)
JDepend [42]	Java	Design Quality Metrics	Standalone Tool	No (finds only design quality metrics)
Lint [43]	Java	Missing and unused translations, layout performance problems, unused resources, inconsistent array sizes, accessibility and internationalization problems, icon and usability problems, manifest errors	Eclipse plug-in and Standalone Tool	No (doesn't find object-oriented code smells)
CodeNose [44]	Java	Switch Statements, Long Method, Long Parameter List, Message Chains, Refused Bequest, Feature Envy, Lazy Class and other class size code smells	Eclipse plug-in	No (not accessable)
SonarQube [45]	Java, C, C# and 24 more	Duplicated Code, Long Method, Large Class, Long Parameter List, Lazy Class, Speculative Generality etc.	Standalone Tool	Yes
Kritika [46]	Java, C, C++ and 10 more	Duplicated code, Large class, Long Method, Long Parameter List etc.	Standalone Tool	Yes
SourceMeter [47]	Java, C, C++ and 3 more	Duplicated code, Large class, Long Method, Long Parameter List etc.	Standalone Tool and SonarQube plug-in	No (not runnable)
JArchitecht [48]	Java	Long Method, Large Class, Long Parameter List etc.	Standalone Tool	Yes
Qulice [49]	Java	Combines PMD, CheckStyle and FindBugs results	Standalone Tool	No (not runnable)
PVS-Studio [50]	Java, C, C++, C#	Finds bugs and security weaknesses	Standalone Tool, plug-in for multiple programs	No (no object-oriented code smells found)
Embold [51]	Java, C, C++ and 11 more	Duplicated Code, Long Method, Large Class, Long Parameter List, Shotgun Surgery, Feature Envy, Message Chains, Data Class	Standalone Tool	Yes
Detekt [52]	Kotlin	Large Class, Long Method, Long Parameter List, Complex Method etc.	IntelliJ, SonarQube plug-in	No (not runnable)

3.2. Code Smell Detectors for iOS Applications

In addition to code smell detection tools for Android applications, available code smell detectors for iOS applications were found. The search was performed on Google similarly to the search for Android detectors and in addition to the keywords used in the previous search, these keywords were used:

- code smell detection tools iOS
- code smell detection tools swift
- code smell detection tools objective-c

The results for the search are presented in Table 2.

Table 2. Code smell detection tools found for iOS applications.

Tool	**Supported Languages**	**Found Code Smells**	**Plug-in/ Standalone Tool**
OCLint [53]	Objective-C, C++, C	Long Method, Long Parameter List, Possible Bugs, Redundant Code, Complicated Code, Unused Code etc.	Standalone Tool
SonarQube [45]	Swift etc.	Duplicated Code, Long Method, Large Class, Long Parameter List, Lazy Class, Speculative Generality etc.	Standalone Tool
Swift Lint [54]	Swift	Long Parameter List, Long Method, Cyclomatic Complexity etc.	Standalone Tool, Third Party Plug-ins
Codebeat [55]	Swift, Objective-C etc.	Long Parameter List, Long Method, Cyclomatic Complexity etc.	Standalone Tool
SwiftFormat [56]	Swift	Reformats Swift code, removes deviations from the standard Swift	Standalone Tool, XCode Extension

3.3. Applications Selected for Analysis

In order to carry out the analysis of the selected code smell detection tools, Android applications were selected to test the detectors. The criteria for the app selections was, that the apps needed to be open source. The source code needed to be freely accessible to be able to run the code smell detectors. The selected apps needed to be of different sizes to allow comparison for the analysis. However, no considerably large projects were selected, as the analysis would have been too extensive. In addition, it was important that all the apps selected were different from each other in their description in order to ensure the code that was analysed was diverse. The Android applications were selected from the F-Droid [57] platform, which is a repository for free and open source software for the Android platform. This repository was selected, as it has a wide range of open source applications, and it was also used in previous research projects [6, 19]. Lines of code for the applications were calculated using the CLOC tool [58]. It was important, that the apps had relatively fresh updates, as they had to be runnable. 10 applications were selected from the F-Droid repository to be used in this thesis and they are described in Table 3.

Table 3. Applications selected for analysis

Application Name	**Description**	**Line Count**	**Google Play Installs**
Notepad	Note taking app	3043	100 000+
Solitaire	Klondike Solitaire game	4129	500+
Goodtime	Productivity timer	8325	100 000+
AnkiDroid	A flashcard-based study aid	51300	5 000 000+
Alarmio	Alarm clock	4651	10+
Authorizer	Password manager	30837	N/A
LaunchTime	Alternative homescreen/ launcher	18007	500+
Missed Notifications Reminder	Reminder for missed notifications	8580	50 000+
Open Note Scanner	Scanner for handwritten notes/ documents	2786	10 000+
PDF Converter	Images to PDF converter	13964	10 000+

4. Results and Analysis

This section presents the results of the experimental part of the thesis. Results for every code smell detector are summarized and their usability is evaluated. In addition, the collective results are given.

4.1. Chosen Code Smell Detectors

Out of the 31 tools found, 8 detectors were chosen for further analysis. Other detectors were not accessible, not correctly runnable on newer versions of needed programs, or did not find any actual code smells. The selected tools were: JDeodorant, PMD, DesigniteJava, CheckStyle, Kritika, Embold, SonarQube, JArchitect.

JDeodorant

JDeodorant [22] is an Eclipse plug-in, that finds code smells in Java code. The tool detects five code smells: Feature Envy, Type Checking, Long Method, God Class, and Duplicated Code. A third-party plug-in called IntelliJDeodorant for IntelliJ IDEA [58] was used in this study, as the original Eclipse plug-in was not fully functional due to needing older Eclipse IDEs to work, but the analysed applications were built for newer environments. IntelliJDeodorant was based on tsantalis/JDeodorant, the original Eclipse plug-in. IntelliJDeodorant version 2019.3-1.1 was used in this study. To analyze a project, the project needed to be loaded to IntelliJ IDEA and then the JDeodorant tool could be run on the project. JDeodorant doesn't calculate metrics and use threshold values to identify code smells, instead it finds the code smells by searching for the refactoring needs in the code. Long Method code smell is found by looking for opportunities to perform Extract Method refactoring, Large Class code smell is found by looking for Extract Class refactorings, and Feature Envy is present, when Move Method refactoring could be done.

PMD

PMD [24] is a static source code analyzer that finds common programming flaws. It supports Java and 7 other programming languages. PMD offers built-in checks for the code as well as the feature to write and customize the rules for the code check. PMD finds code smells like Long Method, Large Class, Long Parameter List, Data Class, Short Identifiers, Long Identifiers, Empty Catch Clause, and numerous other metrics. PMD also includes CPD, the copy-paste detector, for duplicated code detection. PMD version 6.19.0 was used in this study. To find code smells PMD calculates metrics and compares the values to set thresholds. If the threshold is crossed, a violation is marked.

DesigniteJava

DesigniteJava [31] is a code quality assessment tool, that detects different design and implementation smells including Long Method, Large Class, Long Parameter List, and Feature Envy. It finds main object-oriented code metrics in Java code. The Designite tool is also available for C#. DesigniteJava tool is runnable from command line and its output is created as a new file. DesigniteJava Enterprise version 1.1.2 with academic license was used in this study. DesigniteJava has set threshold values for all class-level and method-level metrics. If the threshold is crossed, a violation is marked.

CheckStyle

CheckStyle [32] is a development tool for Java, that automates the process of checking the code and ensures that coding standards are met. It finds class and method design problems and formatting issues, including code smells like Long Method, Large Class, Long Parameter List. It is runnable from command line and additionally plug-ins written by third-parties are supported. CheckStyle version 8.29 in the form of an IntelliJ plug-in [60] was used in this study. CheckStyle finds code smells by calculating different metrics and comparing those with set threshold values. If the threshold is crossed, a violation is marked.

Kritika

Kritika [46] is an online tool, that detects code smells, errors and best practises violations using different other static code analysis tools and custom proprietory rules. It supports Java and 12 additional programming languages. Kritika uses incremental approach to code analysis and allows rules to be configurable. To run an analysis on Kritika, the code must be imported from a public GitHub repository. Kritika version 07.2017 was used in this study. As Kritika uses PMD for code smell detection, the detection technique is the same as described for PMD and the smells found by Kritika are the same as for PMD.

Embold

Embold [51] is an online static code analysis tool, that finds design and code issues and assesses code quality while also recommending solutions. It supports Java and 13 other programming languages. In addition to proprietary analyzer, Embold implements PMD and SpotBugs tools. Embold analyses code’s structural design, finds common code metrics, identifies duplicate code, and implements unique code rules to find code smells like Duplicated Code, Long Method, Large Class, Long Parameter List, Shotgun Surgery, Feature Envy, Message Chains, and Data Class. Code can be imported through Git and SVN or from connected GitHub or

BitBucket account. Embold version 1.7.9 was used in this study. It is not specified, what detection technique Embold uses for code smell findings.

SonarQube

SonarQube [45] is an online automated tool for static code analysis. It supports Java and 26 other programming languages. SonarQube detects bugs, vulnerabilities, and code smells such as Duplicated Code, Long Method, Large Class, Long Parameter List, Lazy Class, Speculative Generality. To analyze a project, a scanner must be installed and configured. SonarQube Community Edition version 8.1.0 was used in this study. SonarQube calculates metrics and compares them to the set theshold values. If the threshold is crossed, a violation is marked.

JArchitect

JArchitect [48] is a tool for analyzing Java code base, that offers a wide range of different features. It allows to analyze code structure, specify code design rules and manage changes. JArchitect supports Code Query over LINQ, that allows to write custom rules and query code in order to detect flaws like structure problems and complex implementations. It finds numerous code metrics and estimates technical debt. Code smells like Long Method, Large Class, Long Parameter List etc. can be found with JArchitect. A project can be uploaded to JArchitect directly from the source code or from different IDE projects. JArchitect version v2019.2 was used in this study. JArchitect calculates metrics and compares them to set threshold values. If the threshold is crossed, a violation is marked.

4.2. Collective Results

The types of code smells the chosen code smell detection tools promised to find and did find were searched and analysed. As only the code smells described by Fowler and Beck will be found in this thesis, those were the smells searched. Detectors and the Fowler et al. described code smells they find are described in Table 4. These 8 chosen detectors collectively promised to detect 10 different code smells. All these detectors can detect Long Method and Large Class code smells, all but two can also detect Long Parameter List code smell. Four of them can find Duplicated Code and three search for Feature Envy. Other smells are only found by one or two detectors. Most of the found smells are Bloaters or Dispensables, which means the tools find too large or unnecessary code. One detected smell belongs in the Change Preventers category, one in Encapsulators, and one represents also the Couplers group. No Object-Oriented Abusers category code smells are found by the selected tools, so the issues related to object-oriented design go unnoticed by these detectors.

Table 4. Overview of code smells found by selected detectors

Tool/Code Smells	JDeodorant	PMD	Designite Java	CheckStyle	Kritika	Embold	SonarQube	JArchitecht
Duplicated Code		X			X	X	X	
Long Method	X	X	X	X	X	X	X	X
Large Class	X	X	X	X	X	X	X	X
Long Parameter List		X	X	X	X		X	X
Shotgun Surgery						X		
Feature Envy	X		X			X		
Lazy Class							X	
Speculative Generality							X	
Message Chains						X		
Data Class		X			X			

The chosen code smell detectors were tested on the Android applications to analyse their usability and usefulness in finding code smells. It was expected, that the detectors, that find the same code smells, give the same results. However, differences can occur from different definitions for code smells as the rules to find them may have been implemented differently for different detectors. The threshold values can also differ for some code smells in different detectors. Every code smell detector was tested on the chosen applications. For tools that offered customization, rulesets with only Fowler et al. code smells were applied, therefore, only Fowler et al. code smells are presented and analysed in the results. For tools, that didn’t offer customizable configurations, Fowler et al. code smells are found from the results, presented and analysed. A collective table with each of the tools’ result is found in the Appendix I. It was marked, how many different code smells each tool found for every tested application. If a tool was expected to find some type of code smell, but it did not for some application, then the results are marked with a “0”. If the tool was not expected to find some type of code smell, then that cell of a table is left empty.

4.3. Individual Results

Individual results are presented here for every detector to analyse their performance, did they meet the expectations and where were the threats to validity.

4.3.1. PMD

PMD allows customized rulesets for code smell findings. A customized ruleset was applied to find the code smells described by Fowler and Beck: DataClass rule for Data Class code smell, ExcessiveMethodLength rule with a threshold of 100 lines of code for Long Method code smell, ExcessiveParameterList rule with a threshold of 10 parameters for Long Parameter List code smell, ExcessiveClassLength rule with a threshold of 1000 lines of code and TooManyMethods rule with a threshold of 10 methods for Large Class code smell, as according to Fowler's definition the Large Class code smell can pose a problem if they carry too many instance variables or have too many lines of code. The threshold values were the default values for PMD and were not changed.

The tool was able to identify all the expected code smells on different applications with the correct thresholds. The results were presented in the output text file, that was automatically created after the command line command was run. The results contain complete paths to the faulty class and the line number, where the code smell exists. For ExcessiveClassLength rule the line number presented is the line, that contains the class name declaration and for TooManyMethods rule it's the line, that contains the curly opening bracket of the class. This sometimes created difference between different tools' results in regards to the line numbers, when in reality the code smells and classes they were referencing were the same. As the Large Class, Long Method, and Long Parameter List code smells identification depends on the minimum value set as the threshold, the results can vary from tool to tool.

CPD, the copy-paste detector used the Karp-Rabin string matching algorithm for clone detection, that compared the string's hash values. The default token length for duplication was 100. The results were faulty at times. Test classes were also included in the check, so PMD and CPD checked all the code presented. For Missed Notification Reminder application, the CPD presented a duplication, that was with a test class. That should not be counted as a duplication, as it is not present in the logic of the source code and is not run in the production. For Notepad application, one of the found duplication blocks was partially incorrect. CPD presented the duplication starting from line 34 for the class SaveButtonDialogFragment.java and the class BackButtonDialogFragment.java, however the lines no. 34 were different in these classes. Due to not just comparing the strings for the duplicate findings, CPD can miss quite long clone code blocks, that other tools have found and pointed out and that could possibly benefit from refactoring.

4.3.2. DesigniteJava

DesigniteJava doesn’t allow customizable rulesets. It found different architecture, design and implementation smells in addition to object-oriented code metrics. The tool was expected to find Feature Envy code smell, Long Method code smell with the threshold of 100 lines of code, Long Parameter List code smell with the threshold of 5 parameters, Insufficient Modularization as Large Class code smell with the threshold of 30 methods from which 20 are public and 100 as the Weighted Methods per Class threshold. The threshold values were the default values for DesigniteJava and were not changed.

Upon running the command from command line, a new folder was created to the output path with Excel files of the results. As there was no grouping option, the file had to be searched with keywords in order to find specific code smells. The Excel file had to be modified by the user for the correct formatting. The tool was able to identify all the expected code smells on different applications with the correct thresholds. Feature Envy and Insufficient Modularization code smells was classified under Design Smells and Long Method and Long Parameter List code smells were under Implementation Smells.

DesigniteJava didn’t find some Long Method code smells, that other tools did with the same threshold. For example, the Notepad application has methods onActivityCreated and onOptionsItemSelected in NoteEditFragment class, that have 114 and 138 lines of code respectively. As the threshold for the Long Method code smell in DesigniteJava is 100, then those methods should have been represented in the results. The reason could be that DesigniteJava counts the effective lines of code (ELOC) instead of all lines of code, that was expected. The ELOC counts were under 100 for both of those methods. Feature Envy code smell was found by checking, which methods reached into other classes data. When for example Embold seemed to have a threshold for how many method usages from other classes was considered a code smell, then DesigniteJava checked, how much logic was used from other classes and how much was from the method’s own class. DesigniteJava check included test classes as well.

4.3.3. CheckStyle

CheckStyle allows customizable rulesets for code smell findings. A customizable ruleset was created and applied for code smell detection. Custom ruleset included MethodLength module with a threshold of 150 lines of code to find Long Method code smell, ParameterLength module with a threshold of 7 parameters to find Long Parameter List code smell and JavaNCSS module with a threshold of 1500 non commenting lines to find Large Class code smell. CheckStyle does not have a module that corresponds exactly to the Large Class code smell; JavaNCSS module was used for this as its rule description says it finds classes that are too large and need breaking into smaller units, which fits the idea of a Large Class code smell. JavaNCSS module summarizes two metrics: non-commentary lines of a class including its nesting classes and the member variable declarations. JavaNCSS module also detected methods, that had too high complexity. This module's results were also counted as Long Method code smell with the threshold of 50 non commenting lines. The threshold values were default values for CheckStyle and were not changed.

IntelliJ plug-in for CheckStyle was used for code smell detection. Customized ruleset was uploaded to IntelliJ IDEA and the scan was run. The results were presented in IntelliJ IDEA in the CheckStyle toolbar. Results report showed the total number of found code smells, the total number of files the code smells were found in, and code smells grouped by classes, where the code smell description and the rule violation was shown. Line number, where the code smell exists, was also shown in the results and when the code smell was clicked, it showed the rule violation in the code.

CheckStyle in majority of cases didn't find Large Class code smell due to the threshold for it being too high. Most classes had the NCSS count around 50, when the threshold was 1500. The threshold for MethodLength being 150 lines of code was also higher than most other tools', which caused CheckStyle to not report many Long Method code smells, that other tools with lower thresholds had found. However, the NCSS rule applied also found more complex methods, so this rule often found methods that could benefit from refactoring, but their length in lines was below 150 and so were not found by MethodLength rule. As the high complexity and long length of a method often are in close association, both rules many a times reported the same method. Long Parameter List code smell was found correctly. CheckStyle doesn't check files that are marked as test sources root classes, so it does not find code smells in test classes. Classes, that were inside a folder named "test", but weren't classified as test folder in IntelliJ, were still checked.

4.3.4. Kritika

Kritika allowed customized ruleset creation. It implemented different well-known static code analysis tools including PMD. It was possible to pick rules from different rule profiles and the same PMD rules were applied: DataClass rule for Data Class code smell, ExcessiveMethodLength rule with a threshold of 100 lines of code for Long Method code smell, ExcessiveParameterList rule with a threshold of 10 parameters for Long Parameter List code smell, ExcessiveClassLength rule with a threshold of 1000 lines of code and TooManyMethods rule with a threshold of 10 methods for Large Class code smell. When for PDM the threshold values were changeable, they were not for Kritika; Kritika used the default threshold values for PMD. Kritika promised to find duplications using abstract syntax trees, so CPD, that was the clone code detector for PMD, was not used here. The threshold value for duplications had to be set by the user. Minimum value of 100 lines of code was used, as the same value

The analysed code was uploaded to Kritika from GitHub repositories. Then the scan was run on the code and the results were presented. The complete path to the code smell was presented in the results and line numbers for the code smells were show following the same logic as PMD. When the code smell was clicked, it showed the code smell's exact location in the code. Kritika found all the same code smells as PMD, which was to be expected. No duplications were found for any of the analysed applications, when in reality, multiple blocks of duplications were present in different applications.

4.3.5. Embold

Embold didn't allow customizable rulesets. It found different metrics, numerous code smells called "Anti-patterns" and duplications in addition to numerous other features. The tool was expected to find Duplicated Code code smell, God Class and Fat Interface anti-patterns that according to their descriptions can be counted as Large Class code smell, Brain Method anti-pattern that can be counted as Long Method code smell, Feature Envy code smell, Shotgun Surgery code smell, and Message Chain code smell. The threshold values for the code smells were not specified, so it is not known, what threshold values Embold uses for its code inspection. Embold also has Code Issues feature, that uses other existing static code analysis tools including PMD. It was expected that code smells found by PMD were also found by Embold with the default thresholds, as the rulesets and thresholds for PMD were not customizable in Embold.

The code had to be uploaded to Embold from public Git repository. Then the scan was run on the code. Results were presented under the Design Issues tab: complete path of the code smell was shown and when the code smell was clicked, the exact location of the code smell was shown in the code. Code Issues were presented under the Code Issues tab.

In addition to finding large classes, Fat Interface anti-pattern also found large interfaces, no other tool did that. Brain Method anti-pattern in a lot of cases didn't find the Long Method code smells, that other tools found. Reason for that is that Brain Method anti-pattern calculated method complexity, executed lines of code, maximum nesting depth and number of accessed variables (NOAV) and made the decision based on those four metrics, while the majority of other tools only calculated the number of lines of code. For this reason, even if the method was long in terms of lines of code but it wasn’t complex enough to be considered a Brain Method, Embold didn’t classify it as one. For Feature Envy findings, Embold seemed to have a threshold for how many other class’s methods a method could use before it was considered a code smell. Methods that accessed 5 or more methods from other classes were considered as Feature Envy code smell, however this threshold was not specified anywhere by Embold. Some Feature Envy code smells were present in the results, but no relations to other classes were shown in the anti-pattern visualization. For example, _prepareFiles method in Anki2Importer.java class in AnkiAndroid application was shown to have Feature Envy code smell, but on further inspection no relations to other classes were presented in the visualization. It’s unclear, why the code smell was found, if no relations to other classes are shown. Embold was the only one out of the tested code smell detectors, that found Shotgun Surgery and Message chain code smells. Shotgun Surgery was by calculating the dependency count and number of constructor methods (NOCM). The thresholds for those metrics are not specified by Embold. Message Chain detects long sequences of outgoing methods or function calls and presents the chain size. The threshold size for Message Chain code smell is also not specified.

Embold was expected to find code smells according to PMD rules too, as PMD was one of the implemented code smell detection tools Embold uses to find code issues. However, no PMD results for the code smells used in this thesis were found for any of the applications tested.

4.3.6. JArchitect

JArchitect offered customizable code smell findings. JArchitect uses Code Query over LINQ. The queries are changeable and new queries can be added. Project Rules have different code quality and metrics related query sets, including Quality Gates, Code Smell, Architect, and Design. If the query has results relating to potential problem in the code, a warning is presented. The tool had Code Queries “Avoid types too big” and “Avoid types with too many methods”, that found Large Class code smell with default thresholds of 200 lines of code and 20 methods respectively. Code Query “Avoid methods too big, too complex” found Long Method code smell. The query was modified and only the method length check was applied, as the complexity of the method is not counted as a part of a Long Method code smell. The default threshold for that query was 35 lines of code. Code Query “Avoid methods with too many parameters" was expected to find Long Parameter List code smell with the default threshold of 7 parameters.

The source folder was uploaded to the system and analysis was run. The Code Query "Avoid methods too big, too complex" was then modified. The results were presented in the tool's dashboard, but the report was also downloadable. Full path to the code smell was presented in the results and when the path was clicked, the exact location of the smell was shown in the source code.

JArchitect found all expected code smells with the correct thresholds. Differences from other tools' results occurred due to slightly different threshold values: Long Method code smell was found with the threshold of 35 lines of code, when other tools' had values such as 100 or 150 lines of code. This caused JArchitect to present more Long Method code smells than other tools with higher threshold values. Large Class code smell threshold being 20 methods was however higher than some other tools' threshold. This resulted in JArchitect not reporting some classes, that other tools had considered to be a Large Class code smell. Large Parameter List was found correctly.

4.3.7. SonarQube

SonarQube doesn't offer customizable rulesets, but it is possible to customize the rules themselves and change the threshold values. It detects different code vulnerabilities like bugs and code smells. Code smells were described in the form of statements. SonarQube was expected to find Duplicate Code smell with the statements "Methods should not have identical implementations" and "Source files should not have any duplicated blocks", Long Method smell with the statement "Methods should not have too many lines" with the threshold value of 75 lines, Large Class smell with the statements "Classes should not have too many methods" with the threshold value of 35 methods and "Classes should not have too many fields" with the threshold value of 20 fields, Long Parameter List smell with the statement "Methods should not have too many parameters" with the threshold value of 7 parameters, Lazy Class smell with the statement "Classes should not be empty" and Speculative Generality smell with the statements "Methods should not be empty", "Sections of code should not be commented out", "Unused method parameters should be removed" and "Unused local variables should be removed".

SonarQube server had to be started on the computer, where the analysis was carried out. All the analysed applications used Gradle as their build technology. To run the analysis with Gradle, org.sonarqube plugin had to be declared in the applications build.gradle file. Then the analysis was executed from the command line. Analyzed projects were shown in the SonarQube server.

The results were presented under the project's Code Smells page. A smell was presented and when clicked, the exact location of the smell was shown in the code, no full path to the code smell was shown.

SonarQube was able to find duplicated lines, duplicated blocks and duplicated files. It didn't find any Long Method code smells for any of the analysed applications, even though it should have, as the threshold for this code smell was lower than some other tools' that found this code smell in the applications. The description of the rule "Methods should not have too many lines" states that the threshold value of 75 applies for maximum authorized lines in a method. This could mean, that the rule disregards the empty, commented or non-logic containing lines of the method. SonarQube also didn't find any Large Class code smell, this could be caused by the fact that the threshold values for those rules were higher, than the other tools' and so no instances of classes that had more than 35 methods or 20 fields were present in the analysed applications. Long Parameter List code smell was found correctly. SonarQube was only tool from the tested detectors, that found Lazy Class and Speculative Generality code smells. It was able to find empty classes, that's considered to be Lazy Class code smell. Unused method parameters, unused local variables, empty methods and sections of code that were commented out were found by SonarQube, that can be counted as Speculative Generality.

4.3.8. JDeodorant

JDeodorant didn't offer customizable rulesets. It detected design problems and offered refactioring opportunities. JDeodorant was expected to find Long Method, God Class, that will be counted as a Large Class code smell, and Feature Envy code smells. JDeodorant didn't calculate metrics and use threshold values to identify code smells like other tools, instead it found the code smells by searching for the refactoring needs in the code. Long Method code smell was found by looking for opportunities to perform Extract Method refactoring, Large Class code smell was found by looking for Extract Class refactoring and Feature Envy was present, when Move Method refactoring could be done.

The code was imported to IntelliJ integrated development environment. The analysis could then be run on the project. The results were presented in the JDeodorant plug-in view. Full path of the code smell was presented in the results and when the code smell was clicked, the exact location of the smell was shown in the code.

Since JDeodorant didn't find code smells by calculating metrics, the results differentiated on occasion greatly from other tools' results. More Long Method and Large Class instances were found for numerous applications than what other tools had found. The Large Class and Long

Method code smell instances that were found by other tools, were also found by JDeodorant on most cases. This means, that majority of Long Method and Large Class code smell occurrences that were found by calculating metrics and comparing them to threshold values, could benefit from extraction refactoring

4.4.Usability

One of the goals of this thesis was to evaluate the usability of the tested code smell detection tools. By installing and using the tools for code smell detection, observations were made in regards to their usefulness, complexity and the features they offer. Criteria proposed by Fernandes et al. [61] were used in this section to evaluate the tools' usability. Table 5 was created according to that criteria.

Table 5. Usability features of the tools

Feature	**PMD**	**DesigniteJava**	**CheckStyle**	**Kritika**	**Embold**	**JArchitect**	**SonarQube**	**JDeodorant**
Result Export	X	X			X	X		
Highlight Smell Occurrences			X	X	X		X	X
Allow Detection Settings	X		X	X		X	X	
Graph Visualization					X	X	X	
Detected Smell Filtering					X	X	X	

Features, that tools support, are marked in the table with a "X". For Result Export feature it was expected that tools supported some sort of version of the export of analysis results, for example in text format, CSV file format or in some other file format.

Result export is important for easier analysis and sharing the results for example with development team or other roles that benefit from improving the code in a development team. It is also helpful for comparing the results of different bad smell detectors that proved necessary in the making of this thesis. Result export was supported by PMD and DesigniteJava, which presented their results immediately in a separate auto-generated file as they were run from the command line. It was also supported by Embold and JArchitect, that offered to export the results after the analysis was run on the code in their environment.

Highlighting Smell Occurrences feature allows observing the exact occurrences and location of the bad smell in the source code. It betters the tools as it shows the smell in the context of the whole method or class and so it is easier for the developer to decide if it really needs to be removed. It proved to be useful, when checking if two found code smells were same in comparing different tools for this thesis. It also simplifies the refactoring of the code as it could be done immediately in the user interface (UI) of the tool. This feature was offered by CheckStyle and JDeodorant, that were IntelliJ and Eclipse plug-ins respectively, so they were able to show the smells in the source code as the program was imported to the IDE. The feature was also supported by Embold, Kritika and SonarQube, which were the tools, that had a separate UI or were online tools.

Detection Setting configuration was another desirable feature. This supported the possibility to change the list of smells that are found or change the threshold values for specific smells. It's a useful feature to have, because often code smell detectors find a large number of smells, but only some of them are needed. This was the case for this thesis, as only the object-oriented smells described by Fowler and Beck were investigated, so it was good, if the tool allowed to search for only specific smells. Also, in the case of metric-based code smells, the threshold values are subjective, and the users might want to change them according to the context of the development or their needs. PMD, CheckStyle and JArchitect supported this feature. Kritika allowed customizable rulesets but not changeable threshold values and SonarQube allowed the creation of new rules but not customizable rulesets.

Graph Visualization was a feature, that bettered the presentation of the results and improved the understanding of refactoring needs for specific classes or the whole project in general. The visualization could have been in the form of different graphical elements such as graphs, charts or heat maps. This was offered only by Embold, JArchitect and SonarQube.

Detected Smell Filtering feature allows to hide or show only a part of the found smells or offers some other type of visualization setting. It would be useful when navigating through found smell lists and on searching for a specific smell. This feature was also offered by only Embold, JArchitect and SonarQube.

None of the tools offered all the five desired features. Embold, JArchitect and SonarQube supported most of them, four features out of five. DesigniteJava and JDeodorant however supported only one of the five useful features.

4.5. Author's Experience

Kritika and Embold were online tools, which means they didn't need to be downloaded. This saved the user a considerable amount of time as they didn't have to read through the download manual, prepare their computer with necessary supporting programs or wait for the downloading process to finish, which is often the case with downloadable computer programs. They allowed the projects to be uploaded from online repositories, which was also convenient. As online tools they were very easy to use, the user interface sides were intuitive for both of them. Kritika was more simplistic in design, Embold supported more features.

SonarQube interface was a webpage as well, however the program needed to be downloaded and the server had to be started from the command line. Due to the user interface being in the form of a web page, it still offered positive user experience when using the tool as it had conventional layout. Each analysed application had to be uploaded through the command line, which can be difficult, if the user is not very familiar with that. In addition, the version of SonarQube used, needed specific newer build versions for the application which oftentimes created conflicts and hindered the usability.

PMD and DesigniteJava were downloadable programs, that didn’t offer user interface, but instead were runnable also from command line and presented their results in a text file or an excel file form. That means the user wasn’t able to filter the results very well or search for a specific code smells. The tools also didn’t highlight the found code smells in the code, which often was inconvenient, especially with bigger project where there were a lot of code smells to filter through. The downloading process was simple however the usage was more complicated; the analysis had to be run from the command line using command line parameters.

CheckStyle and JDeodorant were IntelliJ plug-ins. They were downloaded and then configured in the IntelliJ IDE as a plug-in. The process wasn’t time-consuming. As they were plug-ins, there was no way of filtering the smells nor was there the feature to export the code smells. The plug-in format, however, allowed the tools to highlight smell occurrences in the source code.

JArchitect was downloaded as a program and it offered a graphical user interface. Because the tool supported a large number of different features, it took some time for the user to learn how to properly use it, as they had to familiarize themselves with the user manual and documentation. Compared to other tested detectors, the code smell detection rules for JArchitect were fully customizable which offered the user a lot more freedom with the detection analysis. It provided the user with all the usability features but highlighting the smell occurrences, so it was overall a good tool to use for analysis.

5. Discussion and Recommendations

The goal of this thesis was to give a current overview of different programs and tools that can be used for code smell detection. Three questions were answered in this thesis and hère the results to those questions are discussed.

The first question was whether different code smell detection tools detect the same code smells. This question was answered in Section 4.2. Out of the 22 object-oriented code smells described by Fowler and Beck, the eight detectors chosen for this study collectively found 10 different code smells, however not every tool found all 10. Long Method and Large Class code smells were found by all those detectors, Long Parameter List was also found by the majority of the tools. Most of the detected smells were Bloaters and Dispensables. These code smells are one of the more easier code smells to find by calculating metrics based on the source code. They are also the more common code smells as they were the smells that occurred the most as the conclusive table showed. Therefore it is to be expected, that all or the majority of code smell detection tools are able to find those smells. Other code smells were represented in only couple analyzed detectors. No tool was able to detect code smells related specifically to object-oriented design misuse. Embold and Sonarqube detected the most code smells, both found six different code smells.

The second question was which tools have better usability. This question was answered in Section 4.4 and Section 4.5. Code smell detector was considered to have good usability, if it had result export feature, was able to highlight smell occurrences, allowed detection settings, supported graph visualization, and offered detected smell filtering. Embold, JArchitect and SonarQube tools had four out of five of those desired features, no tool supported all five. These tools were also easy-to-understand and intuitive in usage. DesigniteJava and JDeodorant however supported only one of those five features. Missing usability functionalities can hinder the code smell detection tools' user-friendliness. One of the more important features is the option to customize the rulesets and change the rules and threshold values. As code smells are often subject to interpretation, it is important for the users to be able to configure the rules. Results export, smell filtering and graph visualizations are important features for presenting and comparing the results. Hightlighting the smell occurrences is convenient for the user as it helps to keep track of the current code. When the code smell detector is easy to use, handy and practical, more users are inclined to improve the quality of their code. Code smell detectors that are more difficult to use or that don't offer wished features hinder the user-friendliness and that can result in users not improving their code quality altogether.

The third question was where are the gaps in code smell detection. As it was found in Section 4.2, no tools detected code smells that were related to object-oriented design problems. This means the code that misuses inheritance and information hiding principles goes unnoticed. If a program doesn't implement object-oriented concepts or implements them incorrectly, it can hinder the code's flexibility and lead to more serious problems in addition to maintenance issues. What is more, it became apparent that the majority of code smells analyzed in this study are somewhat subjective, that means their definitions are not entirely fixed. Different code smells implementations can differ from tool to tool, the threshold values can also be different for alternative code smell detectors. This means, that there isn't a firm unity in code smell detection. In addition, many missing functionalities like not being able to export results or non-changeable threshold values greatly hinder the user experience side of the tools. For developers and other development roles to want to better their code, the detectors should have good usability and be user-friendly.

Through author's own experience with the eight tested tools, the tools they recommend most based on this research are JArchitect and Embold. JArchitect offered the user the most freedom with their customizable detection rules. The tool's graphical user interface can be difficult at first, however the user manual is a good source of help. The results export files were thorough and results were visualised for better overview. Embold offered an online smell detection and the analysed applications were uploaded from online repositories. The tool was able to detect a lot of smells in the applications and it highlighted the smell occurrences in the source code. Both of these tools offered numerous other features and metrics calculations in addition to code smell findings.

Further research could be done on the detection performance of the studied tools. That is to find out, which tools have more accurate results in code smell detection. The same kind of testing of code smell detectors as was carried out in this thesis could also be performed on code smell detection tools for iOS applications. In addition to this, an experimental study in developing a new code smell detector is also an option for continuing research. The implementation of the ability to detect more code smells related to object-oriented issues should be considered. As of now, no tools were found that were able to detect those smells. Usability ought to be evaluated critically and user-friendliness should be an important factor when designing the code smell detection tools.

6. Conclusion

The aim of this thesis was to give an up-to-date overview of code smell detection tools for Android and iOS applications. A search was performed to find different code smell detectors for Android and iOS applications, descriptions of them were given. A selection of eight detectors were chosen to be tested on 10 Android applications and three research questions were answered in this study.

The first question was whether different code smell detection tools detect the same code smells. It was found that code smells vary from tool to tool. Different detectors find different code smells, however Long Method, Large Class and Long Parameter list code smells were found by most tools. No tool was able to find code smells related to object-oriented issues in the code.

The second question was which tools have better usability. Out of the 8 code smell detectors for Android applications tested in this thesis, Embold, JArchitect and SonarQube were the most user-friendly, as they supported four out of five desired usability functionalities.

The third question was where are the gaps in code smell detection. No tool was able to find code smells related to object-oriented design problems, so this is considered to be an aspect of further research. It became apparent that code smell definitions and their finding techniques can be different for alternative code smell detectors. This means that numerous aspects of code smell findings are open to interpretation. Many tested detectors didn’t offer various helpful usability functions, that hindered their overall ease of use.

In author’s opinion, out of the eight tested tools JArchitect and Embold had the best usability. JArchitect offered fully customizable code smell detection rules and thorough result export feature; Embold offered easy code smell detection as an online tool while still supporting numerous code smell findings and other metric calculations.

References

[1] Nasir, Z., Abbasi, A., Z., "A framework for software maintenance and support phase", *International Conference on Information and Emerging Technologies*, Karachi, 2010, pp. 1-6. Available from: https://doi.org/10.1109/ICIET.2010.5625668.

[2] ISO/IEC/IEEE International Standard for Software Engineering - Software Life Cycle Processes - Maintenance, ISO/IEC 14764:2006 (E) IEEE Std 14764-2006 Revision of IEEE Std 1219-1998, pp. 1-58, 2006. Available from: https://doi.org/10.1109/IEEESTD.2006.235774.

[3] Reshi, J. A., Satwinder, S., "Software Maintenance: Role of metrics and Smells in object oriented softwares", *National Conference on Computer Engineering Problem Optimization - CEPO'16*, Bathinda, India, 2016. Available from: https://www.academia.edu/34314426/Software_Maintenance_Role_of_metrics_and_Smells_in_object_oriented_softwares [Accessed 14th March 2020].

[4] Fowler, M., Beck, K., Brant, J., Opdyke, W., Roberts, D., *Refactoring. Improving the Design of Existing Code.* Boston: Addison-Wesley Professional, 1999.

[5] Fowler, M., *CodeSmell*, 2006. Available from: https://martinfowler.com/bliki/CodeSmell.html [Accessed 3rd December 2019].

[6] Verloop, D., *Code Smells in the Mobile Applications Domain.* Delft University of Technology, Software Engineering Research Group, Department of Software Technology, Faculty EEMCS, Master's Theses, 2013. Available from: https://repository.tudelft.nl/islandora/object/uuid%3Abcba7e5b-e898-4e59-b636-234ad3fdc432.

[7] Palomba, F., Nucci, D. D., Panichella, A., Zaidman, A., Lucia, A. D., "Lightweight Detection of Android-specific Code Smells: the aDoctor Project", *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER),* Klagenfurt, 2017, pp. 487-491. Available from: doi: https://doi.org/10.1109/SANER.2017.7884659.

[8] Cordy, J., Roy, C., "The NiCad clone detector", *2011 IEEE 19th International Conference on Program Comprehension*, Kingston, ON, 2011, pp. 219-220. Available from: https://doi.org/10.1109/ICPC.2011.26.

[9] Podoler, Y., *Regression Testing and Tools*, 2018. Available from: https://www.testcraft.io/ebook-automated-regression-testing/?utm_campaign=regression%20testing%20eBook&utm_source=hs_automation&utm_medium=email&utm_content=64566764&_hsenc=p2ANqtz--YC3a3JXPBmz7Ov5_4BvKFGU3j3S0i4NXyFi0w20NGDyxsePt6vIk8ILSkC8Wo6UDSdmUI41ijHxcHx9WCq4InbzSeqw&_hsmi=64566764 [Accessed 5th April 2020].

[10] Palomba, F., Nucci, D. D., Panichella, A., Zaidman, A., Lucia, A. D., "Lightweight Detection of Android-specific Code Smells: the aDoctor Project", *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, Klagenfurt, 2017, pp. 487-491. Available from: https://doi.org/doi:10.1109/SANER.2017.7884659.

[11] Mantyla, M., Vanhanen, J., Lassenius, C., "A taxonomy and an initial empirical study of bad smells in code", *Proceedings of the International Conference on Software Maintenance,* 2003. ICSM 2003. Amsterdam, The Netherlands, 2003, pp. 381-384. Available from: https://doi.org/10.1109/ICSM.2003.1235447.

[12] Moha, N., Gueheneuc, Y., Duchien, L., Meur, A. L., "DECOR: A Method for the Specification and Detection of Code and Design Smells", *IEEE Transactions on Software Engineering*, vol. 36, no. 1, pp. 20-36, Jan.-Feb. 2010. Available from: https://doi.org/10.1109/TSE.2009.50.

[13] Fernandes, E., Oliveira, J., Vale, G., Paiva, T., Figueiredo, E., "A review-based comparative study of bad smell detection tools", *Proceedings of the 20th International Conference on Evaluation and Assessment in Software Engineering (EASE '16)*, Association for Computing Machinery, Article 18, New York, NY, USA, 2016, pp. 1–12. Available from: https://doi.org/10.1145/2915970.2915984.

[14] Pecorelli, F., Palomba, F., Nucci, D. D., Andrea, D. L., "Comparing heuristic and machine learning approaches for metric-based code smell detection", *Proceedings of the 27th International Conference on Program Comprehension (ICPC '19).* IEEE Press, 2019, 93–104. Available from: https://doi.org/10.1109/ICPC.2019.00023.

[15] Kulkarni, C., Joshi, S. D., "A Review on Code Smell Techniques Using Nesting Structure Tree", *International Journal of Computational Intelligence and Informatics,* 2016; 5 (3): pp. 6-10. Available from:

https://www.periyaruniversity.ac.in/ijcii/issue/Vol5No4March2016/IJCII%205-4-330.pdf [Accessed 27 April 2020].

[16] Palomba, F., "Textual Analysis for Code Smell Detection," *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Florence, 2015, pp. 769-771, Available from: https://doi.org/10.1109/ICSE.2015.244.

[17] Bravo, F. M., *A Logic Meta-Programming Framework for Supporting the Refactoring Process*, Vrije Universiteit Brussel, Faculty of Sciences, In Collaboration with Ecole des Mines de Nantes, Master's Thesis, 2003. Available from: http://www.informatik.uni-bremen.de/st/lehre/Arte-fakt/Seminar/papers/09/When%20to%20apply/A%20Logic%20Meta-Programming%20Framework%20for%20Supporting%20the%20Refactoring%20Process.pdf [Accessed 25th April 2020].

[18] Sharma, T., Efstathiou, V., Louridas, P., Spinellis, D., On the Feasibility of Transfer-learning Code Smells using Deep Learning. *ACM Trans. Softw. Eng. Methodol*. 1, 1, Article 1, 2019. Available from: https://arxiv.org/abs/1904.03031.

[19] Lim, D., *Detecting Code Smells in Android Applications*. Delft University of Technology, Software Engineering Research Group, Department of Software Technology, Faculty EEMCS, Master's Theses, 2018. Available from https://repository.tudelft.nl/islandora/object/uuid%3Abab69ac3-07b7-4dae-8b80-670443af2faa?collection=education.

[20] Reimann, J., Brylski, M., Aßmann, U., „A Tool-Supported Quality Smell Catalogue For Android Developers", *Softwaretechnik-Trends*, 34, Technische Universität Dresden, Software Technology Group, 2014. Available from: https://pdfs.semanticscholar.org/7ce5/32e0e933037c5f410a9e3cbc07f5513c3078.pdf [Accessed 3rd December 2019].

[21] Almalki, K. S., *Bad Droid! An in-depth empirical study on the occurrence and impact of Android specific code smells*. Rochester Institute of Technology, Department of Software Engineering, B. Thomas Golisano College of Computing and Information Sciences, Master's Thesis, 2018. Available from: https://scholarworks.rit.edu/cgi/viewcontent.cgi?article=11087&context=theses [Accessed 3rd December 2019].

[22] Tsantalis, N., *JDeodorant*. GitHub repository. Available from: https://github.com/tsantalis/JDeodorant [Accessed 9th November 2019].

[23] Eclipse Plugins, Bundles and Products - Eclipse Marketplace. *inFusion Hydrogen*. 2011. Available from: https://marketplace.eclipse.org/content/infusion-hydrogen [Accessed 9th November 2019].

[24] PMD Source Code Analyzer. *PMD An extensible cross-language static code analyzer*. Available from: https://pmd.github.io [Accessed 5th November 2019].

[25] Vidal S., *JSpIRIT*. Santiago Vidal home page. Available from: https://sites.google.com/site/santiagoavidal/projects/jspirit [Accessed 7th November 2019].

[26] Nagpal, R., *Stench Blossom*. GitHub wiki page. Available from: https://github.com/DeveloperLiberationFront/refactoring-tools/wiki/Stench-Blossom [Accessed 7th November 2019].

[27] Marinescu, C., Marinescu, R., Mihancea, P. F., Ratiu, D., Wettel, R., "iPlasma: An Integrated Platform for Quality Assessment of Object-Oriented Design", *Proceedings of the 21st IEEE International Conference on Software Maintenance - Industrial and Tool volume*, ICSM 2005, 25-30 September 2005, Budapest, Hungary, pp. 77-80. Available from: https://www.researchgate.net/publication/221308133_iPlasma_An_Integrated_Platform_for_Quality_Assessment_of_Object-Oriented_Design [Accessed 6th November 2019].

[28] Moha, N., Gueheneuc, Y., Duchien, L., Meur, A. Le, "DECOR: A Method for the Specification and Detection of Code and Design Smells", *IEEE Transactions on Software Engineering*, vol. 36, no. 1, pp. 20-36, Jan.-Feb. 2010. Available from: https://doi.org/10.1109/TSE.2009.50.

[29] Palomba, F., *aDoctor*, GitHub repository. Available from: https://github.com/fpalomba/aDoctor [Accessed 9th November 2019].

[30] Hefft, G., *Paprika*, GitHub repository. Available from: https://github.com/GeoffreyHecht/paprika [Accessed 8th November 2019].

[31] Sharma, T., *DesigniteJava*, Designite web page. Available from: https://www.designite-tools.com/designitejava [Accessed 5th November 2019].

[32] CheckStyle web page. Available from: https://checkstyle.org [Accessed 9th November 2019].

[33] Bulychev, P., *Clone Digger*, SourceForge web page. Available from: https://sourceforge.net/projects/clonedigger [Accessed 7th November 2019].

[34] Griffith, I., Wahl, S., Izurieta, C., *TrueRefactor: An Automated Refactoring Tool to Improve Legacy System and Application Comprehensibility*. Computer Science Department, Montana State University, Bozeman, MT, 2011. Available from: https://pdfs.semanticscholar.org/6c3c/fde93911d5c4e9474b67cf2faa01c8938d66.pdf [Accessed 8th November 2019].

[35] Tamrawi, A., Nguyen, H. A., Nguyen, H. V., Nguyen, T. N., "SYMake: a build code analysis and refactoring tool for makefiles", *2012 Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, Essen, 2012, pp. 366-369. Available from: http://doi.org/10.1145/2351676.2351749.

[36] Cordy, J. R., Roy, C. K., *NiCad clone detector*, Txl home page. Available from: https://txl.ca/txl-nicaddownload.html [Accessed 7th November 2019].

[37] Emden, E. van, Moonen, L., "Assuring software quality by code smell detection", *2012 19th Working Conference on Reverse Engineering*, Kingston, ON, 2012. Available from: https://doi.org/10.1109/WCRE.2012.69.

[38] Davis, I., *Java clone detector*. University of Waterloo SWAG: Software Architecture Group home page. Available from: https://www.swag.uwaterloo.ca/jcd [Accessed 10th November 2019].

[39] Jiang, L., Misherghi, G., Su, Z., Glondu, S., “Deckard: Scalable and accurate tree-based detection of code clones”, *ICSE ’07: Proceedings of the 29th international conference on Software Engineering*. Washington, DC, USA: IEEE Computer Society, 2007, pp. 96–105. Available from: http://dx.doi.org/10.1109/ICSE.2007.30.

[40] Alves, P., Santana, D., Figueiredo, E., "ConcernReCS: Finding code smells in software aspectization", *2012 34th International Conference on Software Engineering (ICSE)*,

Zurich, 2012, pp. 1463-1464. Available from: https://doi.org/10.1109/ICSE.2012.6227063.

[41] *Code Bad Smell Detector*, SourceForge web page. Available from: https://sourceforge.net/projects/cbsdetector [Accessed 8th November 2019].

[42] Clark, M., *JDepend*, GitHub repository. Available from: https://github.com/clarkware/jdepend [Accessed 8th November 2019].

[43] *Android Lint*, Android Studio Project Site. Available from: http://tools.android.com/tips/lint [Accessed 9th November 2019].

[44] Slinger, S., *Code Smell Detection in Eclipse*, Delft University of Technology, Faculty of Electrical Engineering, Mathematics and Computer Science Department of Software Technology, Software Engineering Group, Software Evolution Research Lab, Thesis Report, 2005. Available from: https://pdfs.semanticscholar.org/d692/3e3de76f9cb4dd3d6dae241e8aacc97bc055.pdf?_ga=2.204081232.1363212833.1581860123-1337955312.1581860123 [Accessed 10th November 2019].

[45] SonarQube web page. Available from: https://sonarqube.org [Accessed 10th November 2019].

[46] Kritika web page. Available from: https://kritika.io [Accessed 10th November 2019].

[47] SourceMeter web page. Available from: https://www.sourcemeter.com [Accessed 10th November 2019].

[48] JArchitect web page. Available from: https://jarchitect.com [Accessed 10th November 2019].

[49] *Source Code Quality Police (Qulice)*, Qulice web page. Available from https://www.qulice.com [Accessed 10th November].

[50] PVS-Studio web page. Available from: https://viva64.com/en/pvs-studio/ [Accessed 16th November 2019].

[51] Embold web page. Available from: https://embold.io [Accessed 16th November 2019].

[52] *Detekt*, GitHub repository. Available from: https://github.com/detekt/detekt [Accessed 16th November 2019].

[53] OCLint web page. Available from: http://oclint.org/ [Accessed 15th December 2019].

[54] *SwiftLint*, GitHub repository. Available from: https://github.com/realm/SwiftLint#installation [Accessed 15th December 2019].

[55] Codebeat web page. Available from: https://codebeat.co [Accessed 15th December 2019].

[56] Lockwood, N., *SwiftFormat*, GitHub repository. Available from: https://github.com/nicklockwood/SwiftFormat [Accessed 15th December 2019].

[57] F-Droid web page. Available from: https://f-droid.org [Accessed 12th December 2019].

[58] Danial, A., *cloc*, GitHub repository. Available from: https://github.com/AlDanial/cloc [Accessed 19th December 2019].

[59] *IntelliJDeodorant*, GitHub repository. Available from: https://github.com/JetBrains-Research/IntelliJDeodorant [Accessed 19th December].

[60] *CheckStyle-IDEA*, JetBrains web page. Available from: https://plugins.jetbrains.com/plugin/1065-checkstyle-idea [Accessed 6th December 2019].

[61] Fernandes, E., Oliveira, J., Vale, G., Paiva, T., Figueiredo, E., "A review-based comparative study of bad smell detection tools", *Proceedings of the 20th International Conference on Evaluation and Assessment in Software Engineering (EASE '16).* Association for Computing Machinery, New York, NY, USA, 2016, Article 18, 1–12. Available from: https://doi.org/10.1145/2915970.2915984.

Appendix

I. Results for Tested Code Smell Detectors

		Duplicated Code	Long Method	Large Class	Long Parameter List	Shotgun Surgery	Feature Envy	Lazy Class	Speculative Generality	Message Chains	Data Class
Notepad	PMD	5	4	5	0						0
	DesigniteJava		2	4	0		0				
	CheckStyle		2	0	0						
	Kritika	5	4	5	0						0
	Embold	17	1	4		1	0			1	
	JArchitect		5	3	0						
	SonarQube	7	0	0	0			0	4		
	JDeodorant		0	0			0				
Solitaire	PMD	4	4	4	1						2
	DesigniteJava		3	4	12		5				
	CheckStyle		6	0	3						
	Kritika	4	4	4	1						2
	Embold	12	1	5		2	1			13	
	JArchitect		0	3	3						
	SonarQube	7	0	0	3			0	30		
	JDeodorant		70	2			7				
Goodtime	PMD	0	5	4	0						2
	DesigniteJava		1	3	2		1				
	CheckStyle		4	0	0						
	Kritika	0	5	4	0						2
	Embold	3	0	6		1	0			6	
	JArchitect		8	3	2						
	SonarQube	3	0	0	2			2	2		
	JDeodorant		16	5			0				
AnkiDroid	PMD	37	26	48	0						12
	DesigniteJava		37	40	2		0				
	CheckStyle		21	1	3						
	Kritika	37	26	48	0						12
	Embold	72	6	46		76	7			266	
	JArchitect		76	36	3						
	SonarQube	54	0	0	3			3	52		
	JDeodorant		136	50			0				
Alarmio	PMD	4	2	2	0						0
	DesigniteJava		1	2	0		0				
	CheckStyle		1	0	0						
	Kritika	4	2	2	0						0
	Embold	6	0	6		0	0			4	
	JArchitect		4	1	0						
	SonarQube	4	0	0	0			0	26		
	JDeodorant		8	7			1				
Authorizer	PMD	140	22	25	3						4
	DesigniteJava		11	12	10		0				
	CheckStyle		8	0	2						
	Kritika	140	22	25	3						4
	Embold	131	1	31		24	1			69	
	JArchitect		37	15	2						
	SonarQube	643	0	0	2			0	92		
	JDeodorant		0	0			0				
LaunchTime	PMD	0	17	8	0						0
	DesigniteJava		6	8	16		1				
	CheckStyle		3	1	1						
	Kritika	0	17	8	0						0
	Embold	2	3	10		6	0			26	
	JArchitect		15	7	1						
	SonarQube	3	0	0	1			0	16		
	JDeodorant		56	17			2				
Missed Notifications Reminder	PMD	1	3	8	1						2
	DesigniteJava		0	5	13		0				
	CheckStyle		2	0	1						
	Kritika	1	3	8	1						2
	Embold	2	0	6		4	1			1	
	JArchitect		6	2	1						
	SonarQube	2	0	0	1			0	4		
	JDeodorant		24	18			0				
Open Note Scanner	PMD	0	4	2	0						2
	DesigniteJava		1	1	0		1				
	CheckStyle		1	0	0						
	Kritika	0	4	2	0						2
	Embold	3	0	2		0	1			0	
	JArchitect		1	2	0						
	SonarQube	2	0	0	0			0	12		
	JDeodorant		6	7			0				
PDF Converter	PMD	15	4	17	1						0
	DesigniteJava		3	4	10		2				
	CheckStyle		0	0	3						
	Kritika	15	4	17	1						0
	Embold	58	0	18		27	0			35	
	JArchitect		8	9	3						
	SonarQube	22	0	0	3			0	25		
	JDeodorant		15	20			0				

II. Licence