UNIVERSITY OF TARTU
Institute of Computer Science
Computer Science Curriculum

Stanislav Mõškovski

Building a tool for detecting code smells in Android application code

Master's Thesis (30 ECTS)

Supervisor: Kristiina Rahkema, MSc
Co-supervisor: Dietmar Pfahl, PhD

Tartu 2020

Building a tool for detecting code smells in Android application code

Abstract:
In recent years, quality of the code behind the presentation layer has become increasingly important since a high number of applications are in maintenance mode. Maintaining complex applications is hard and modifying code that is difficult to understand may introduce new bugs. Poor design or implementation choices that contribute to technical debt are called code smells. Static analyzers are tools used to detect code smells and other vulnerabilities inside software applications. In this paper, we analyze static analysis tools and also develop an alternative, which tries to overcome the shortcomings of the previous tools by providing more code smell detection rules. As a result, we developed a plugin for SonarQube that analyzes applications written in Java programming language. Finally, we verified our tool by performing an empirical study on a selected corpus of applications.

Keywords:
Static code analyzer, Code smells, Android, SonarQube

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

Koodilõhnade tuvastamise tööriista loomine Android platvormi rakendustele

Lühikokkuvõte:

Viimastel aastatel on esitluskihi taga oleva koodi kvaliteet muutunud üha olulisemaks, kuna suur hulk rakendusi on hoolduse faasis. Keerukate rakenduste hooldamine on liialt töömahukas ja raskesti mõistetava koodi muutmine võib põhjustada uusi vigu. Kehva disaini või implementeerimise otsuseid, mis põhjustavad tehnilist võlgnevust, nimetatakse koodilõhnadeks. Staatilised analüsaatorid on tööriistad, mida kasutatakse koodilõhnade ja muude tarkvara haavatavuste tuvastamiseks. Selles toos uurime staatilise analuusi tooriistu ja arendame välja ka alternatiivse lahenduse, mis puuab ule saada eelnevate vahendite probleemidest suutes tuvastada suuremat arvu koodilohnu. Tulemusena töötasime välja SonarQube'i pistikprogrammi, mis analüüsib Java programmeerimiskeeles kirjutatud rakendusi. Lõpuks kontrollisime oma tööriista, viies läbi empiirilise uuringu valitud rakenduste korpusel.

Võtmesõnad:

Staatiline koodi analüüs, koodilõhn, Android, SonarQube

CERCS: P170 Arvutiteadus, arvutusmeetodid, süsteemid, juhtimine (automaatjuhtimisteooria)

Contents

1 Introduction **6**
1.1 Research context . . . 6
1.2 Research motivation . . . 6
1.3 Thesis outline . . . 6

2 Background **7**
2.1 Code smells . . . 7
2.2 Related work . . . 8
2.2.1 Tool "inFusion" . . . 8
2.2.2 Tool "paprika" . . . 8
2.2.3 SonarQube plugin "Anti patterns code smells" . . . 9
2.3 SonarQube . . . 10

3 Method **12**
3.1 SonarQube plugin development . . . 12
3.2 Development tools used . . . 13
3.3 Testing . . . 13
3.4 Datasets . . . 14
3.5 Analysis procedure . . . 14

4 Results **17**
4.1 Developed plugin . . . 17
4.1.1 Implementation . . . 17
4.1.2 Translating queries into SonarQube rules . . . 20
4.1.3 Testing . . . 21
4.2 Analysis results . . . 21

5 Discussion **27**
5.1 Encountered problems . . . 27
5.2 Issues and limitations . . . 28
5.3 Comparison to published results . . . 29
5.4 Potential usages . . . 32
5.4.1 Developers . . . 32
5.4.2 Project managers . . . 33
5.4.3 Data scientists . . . 36

6 Conclusion **37**

References **40**

Appendix **41**
I. Implemented code smells 41
II. Licence 43

1 Introduction

1.1 Research context

Code smells are patterns in the code that are symptoms of poor design choices. As opposed to software bugs, code smells do not make the problem run incorrectly but instead, lead to increased fault-proneness and, in the long run, decrease software maintainability.

In [11, 3, 10] researchers developed numerous static code analyzers to detect code smells inside Java applications. However, some of the tools are not available anymore due to being commercial and closed source. Others are only able to detect a small number of code smells or work with compiled applications instead of the source files.

In this paper, we developed a tool that is open-source and implements 26 code smells defined by Fowler. We also performed an evaluation of the tool, where we performed an analysis of 240 applications and then compared our results to already published results in [11].

1.2 Research motivation

From the industry perspective, we wanted to develop a tool that would assist different roles in the software development process. The main focus is the developers, can receive real-time feedback about the quality of the code that they produce. The project managers can have an overview of the project status and knowledge of any potential vulnerabilities, bugs, or code smells. The data scientist could look into correlations between various code smells and their occurrences.

From the academic perspective, we wanted to extend the body of knowledge about the occurrence of code smells in Android applications by extending the number of detected code smells, providing analysis results and then comparing them against results published in the literature. Additionally, we wanted to provide an overview of code smells occurrences for the code smells not yet published in the literature.

1.3 Thesis outline

The paper consists of 4 main sections. Section 2 provides an introduction to code smells, reviews the state of the art implementations of other static code analyzers, and gives an overview of our implementation platform. Section 3 describes the developed tool, tools used for the development, and also walks through the process of the tool verification procedure. Section 4 talks about the results of the paper: the developed tool and the results that we found during the analysis of the applications. Section 5 discusses the results of the analysis, compares our results with already published results and discusses the issues and limitations of our tool. Furthermore, we discuss the potential usages of the tool for various roles in the software development process.

2 Background

2.1 Code smells

In the era of agile software development, the quality of the code produced by developers plays a very important role. It is crucial because developers have to be able to change the code quickly due to evolving business requirements. However, due to time constraints and constant changes in the codebase, the code may become less maintainable. Often developers make changes that allow them to implement the current requirements quickly, but if the requirements change in the future, they would have to refactor large portions of code that they developed earlier. Applying solutions that may cause future code refactoring is called technical debt [22].

Often technical debt comes in pieces that can be universally unified into recognizable patterns. Such technical debt patterns that can be applied to most software applications are called code smells. According to [24], code smells are symptoms of poor design and implementation choices, which lead to increased fault-proneness and, eventually, decrease software maintainability. Decreased maintainability leads to costs when implementing new features or making changes to existing ones.

Since code smells come in patterns that can be recognized by humans, we could also teach machines to recognize those patterns in the code that humans write. The software that can perform static code analysis to recognize code smells and problematic patterns in the code is called a static analyzer. A static analyzer performs an analysis of the code without actually executing it [25].

Fowler describes various code smells in his book [5]. The book describes 22 code smells and discusses how the code smells can be fixed by applying different patterns. Let's look into some examples of the code smells defined by Fowler.

Data class — a class that only contains data and no business logic that interacts with the data. The class usually contains only public fields or private fields that only have getters and setters.

Cyclic dependencies – classes that depend on each other. This is an issue because this makes dependency injection impossible, since the dependency graph forms a cycle, and it is not possible to determine which class should be instantiated first. Dependencies should be representable as an acyclic graph as defined by Fowler.

Feature envy – code smell where a class is interested in the data of the other classes more than its data. Usually occurs when a class gets the data externally and tries to compute some value. Such approach goes against OOP principles, which require that a class should use its internal state to compute the values.

In this paper, we implemented the code smells defined by Fowler. The full list of implemented code smells can be found in the appendix I.

2.2 Related work

In this section, we will look into various tools that were developed to detect code smells and see how they performed during static analysis on corpora of applications. We will look into three tools: "inFusion", "paprika" and a SonarQube plugin called "anti patterns code smells" and see how accurate they are at detecting the code smells.

2.2.1 Tool "inFusion"

In [11], Mannan et al. used a tool called "inFusion" to analyze a large corpus of Android and Java desktop applications. The corpus that authors of [11] used consisted of 750 open-source Java applications and 500 Android projects including in total over 22 million lines of code. According to Mannan et al., the tool was chosen for multiple reasons. Firstly, this tool has a very high precision (84%) and recall (100%) rate (and also F-measure of 91.3%) according to [1]. Secondly, according to the authors of the research, this tool scales to analyzing large applications. Finally, the tool used to be popular among researchers during the conduction of this study.

The main issue with this tool is that it used to be commercial and is no longer available nowadays. Due to the tool not being available, we are not able to check the exact code definitions used by it and thus, we cannot easily replicate the results that were achieved in [11].

Additionally, since the tool was commercial, it would be highly unlikely that the users of the software would be able to define their code smells to extend the definitions provided by the software vendor.

2.2.2 Tool "paprika"

Paprika is a tool developed by Hecht et al. in [10]. This tool analyzes APK files, which is the format in which Android applications are distributed. One important issue is that this tool analyzes compiled artifacts of the application and not its source code. It is also worth mentioning that Android applications run on Dalvik virtual machines and not on Java virtual machines. This is crucial because it is mentioned in [10] the bytecode for those virtual machines is different and during decoding from one bytecode to another, some information is lost. Mannan et al. investigated the differences between the analysis of the source code and the analysis of the decompiled source code from the `.apk` archive. This step is required to actual Java code representation because some of the code smells may have been introduced during the compilation process by the compiler and we would like to avoid those. Losing information during bytecode conversion and code obfuscation together produce quite significant information loss.

According to Hecht et al., the paprika analysis consists of four steps. First, various metrics are extracted from the artifact of the application such as the application name and/or packages. Paprika builds a model with the following attributes: `App`, `Class`,

Method, Attribute, Variable, ExternalClass, ExternalMethod. These entities are linked between each other via different relations to produce a meaningful model that can be analyzed later. It is also worth mentioning that the model is built incrementally using the SOOT framework as described in [10].

Secondly, the existing Paprika model has to be converted into a graph model. Hecht et al. used a graph database to achieve this. They chose Neo4j [15] as a database implementation because Neo4j [15] combined with Cypher provides great performance for Java applications and can satisfy scalability requirements.

Next, anti-patterns in the code are detected using graph queries. As mentioned before, the Cypher query language is used to query the database and detect anti-patterns. Paprika detects the following code smells and anti-patterns: long method, complex class, member ignoring method, leaking inner class, UI overdraw, heavy broadcast receivers. The exact code definitions and more detailed descriptions can be found in [10].

Finally, the software quality score is computed. According to Hecht et al., the score is based on the number of anti-patterns detected. This score is computed for multiple versions of the same application to see how the number of code smells changes with the evolution of the application.

To sum up, we can see that this application is built to analyze applications at scale, but it is only limited to Android applications. Another disadvantage is that the number of code smells supported is quite small.

2.2.3 SonarQube plugin "Anti patterns code smells"

In [11, 10], the authors present standalone tools that were developed from scratch. In [3], a plugin for SonarQube has been developed that can detect 19 code and 4 architectural smells.

Internally the "Anti patterns code smells" plugin uses DECOR tool to detect code smells. According to Fontana et al., DECOR has 100% recall rate as well as 80% precision for the detection of code smells. The plugin is able to detect the following code smells: "anti-singleton", "base class knows derived class", "base class should be abstract", "blob", "class data should be private", "complex class", "duplicated code", "functional decomposition", "large class", "lazy class", "long method", "long parameter list", "many field attributes but not complex", "message chains", "refused parent bequest", "spaghetti code", "swiss army knife" and "tradition breaker". The plugin is also able to detect the following architectural code smells: "unstable dependency", "hub-like dependency", "cyclic dependency", "multiple architectural smells". The exact code smell definitions and descriptions can be found in the original paper [3].

According to Fontana et al., the plugin was able to analyze 103 projects in 35 days from the Qualitas Corpus Repository [23] on a Linux machine with 4 core and 16GB of RAM.

2.3 SonarQube

SonarQube [19] is a platform that automates the process of code reviewing to detect code smells, bugs and vulnerabilities in the source code.

SonarQube is often used as a stage in the CI/CD pipeline which sets a *quality gate* for the developers. Quality gate consists of *rules* which are defined by the *profile*. A rule is a definition that describes a certain pattern in the source code (such as code smells, bug, or a vulnerability) which must be avoided as it might cause maintainability issues or introduce new bugs and vulnerabilities. The set of rules that are used for a project can be configured individually for each project.

There are multiple reasons why we chose SonarQube as an implementation platform. Firstly, SonarQube is one of the largest ecosystems that perform static code analysis for a wide variety of languages. The framework comes with numerous predefined code smells that are very common in the industry. SonarQube is open-source, has been in development since 2006, and has over 29,000 commits in its Github repository[20]. Therefore, we can say that this framework is pretty mature. Furthermore, SonarQube is also one of the most popular tools in the industry to perform static code analysis.

Besides, SonarQube also provides a custom API that allows developers to define their code smells without the need for writing the code analysis framework from scratch. The most common way to extend SonarQube is to develop a plugin that relies on the SonarQube API. In [8], a comprehensive guide is provided by the SonarQube community which explains how to create a SonarQube plugin that analyzes source code written in Java.

In short, writing a plugin for SonarQube consists of the following steps:

Define the rules – define a set of rules and implement them using the SonarQube API. A rule operates on the abstract syntax tree of the program a marks all the tree nodes where it finds an issue.

Create rule loader – create a mechanism that will load the plugins and prepare them for the analysis execution. This usually includes loading descriptions of the rules into memory and registering them with SonarQube. Registration allows SonarQube to include the rules in the UI, which in turn allows the end-user to enable/disable specific rules based on his/her preferences.

Test the implementation – write unit tests using Sonar testing framework to verify that the plugin actually detects the correct patterns in the code.

Deploy the plugin – install the plugin to the SonarQube instance. This allows the users to run the plugin rules on actual projects to detect issues in their source code.

Finally, after all the aforementioned steps have been completed, the user can enable the rules and run the static analysis.

Previously we mentioned that SonarQube is often used as a step during CI/CD pipeline. As the reader might know, this is quite late in the development process, and it would be much more beneficial if the issues would be discovered earlier. Fortunately, SonarQube provides a solution called SonarLint [18]. SonarLint is an extension for the integrated development environment (IDE), which can be synchronized with an instance of SonarQube to provide real-time analysis of the code. This allows the developers to see and fix the issues before their run and commit their code, which greatly speeds up the development process.

3 Method

3.1 SonarQube plugin development

To fulfill our task of analyzing a large corpus of application to detect the code smells, we built a tool that is stable, scalable and allows us to aggregate the results of the analysis in an organized manner. Moreover, we needed a framework that would allow us to analyze a large corpus of applications programmatically since starting the analysis manually for every project under observation would be inefficient and unproductive.

For this task, we decided to use the SonarQube platform because it is a de facto tool in the industry to use for static analysis of the applications. Not only that, but SonarQube provides possibilities to write custom rules by writing custom plugins. Since we needed to implement code smells that are not yet defined by the SonarQube, we decided to extend the tool by writing a plugin that can detect the code smells that are described in subsection 2.1.

To create the plugin, we followed the tutorial provided in the SonarQube documentation [8]. The documentation provides guidelines on how to create a plugin with custom Java rules, how to test the plugin and how to register rules with the SonarQube so that it would find them during runtime of the application. The documentation relies on an extension of Sonar Java plugin [21], which provides an API for the Java languages abstract syntax tree (AST) and basic interface to create rules, which would be used during the analysis.

However, this tutorial only focuses on running on an instance of SonarQube and not SonarLint, which is an extension to run the plugins inside the integrated development environment (IDE). This is relevant because both SonarQube and SonarLint rely on Sonar compute engine, which means that you can write a plugin for either of those tools and it would be usable in both of them.

During the runtime of SonarQube, plugins can be installed dynamically, either from the marketplace or they can also be loaded from the `/opt/sonarqube/extensions/plugins/` directory as stated in the documentation. This means that plugins will be loaded dynamically during the execution of the analysis and since the documentation states the Sonar Java plugin must be included with `provided` scope during compilation (which means that it will not be included in the final compiled artifact of the plugin), it means that not all of the classes might be available at the runtime that was available during compilation.

Previously we mentioned that there are multiple contexts in which the analysis can be run (SonarQube and SonarLint), however, in this thesis, we will only focus on SonarQube because we are interested in the analysis of the large corpus of applications and it makes no sense to do this manually through the IDE.

Thus, we implemented the plugin that detects the code smells described in subsec-

tion 2.1 and can be executed inside a SonarQube instance.

3.2 Development tools used

To implement the plugin, we used SonarQube as the implementation platform. We decided to go with the SonarQube platform for two primary reasons. Firstly, SonarQube is a very mature tool in the industry. Secondly, it controls the flow of the analysis (project configuration, starting the analysis, creating the results, serializing and parsing the results, uploading them to the server and displays them to the user), so we can focus on writing the custom code smell detection rules. Finally, it provides other features such as a web API, which can be used for exporting the analysis results.

Scala was chosen as the language for the plugin implementation. We chose Scala, because it combines features of both functional and object-oriented languages and since SonarQube API is written in Java, it allowed for nice interoperability between Java and Scala. Moreover, Scala provides some features that are not available in Java natively, such as implicit classes and method parameters, and pattern matching.

As a plugin implementation base, we used Sonar Java plugin since it is recommended base when writing plugins for the Java language and as mentioned previously, it provides an API to use Java AST.

3.3 Testing

Once the plugin was implemented, we need to also test it to test the operation of the plugin. Testing of the plugin can be split into two phases: testing of plugin internal architecture, such as rule loading, registration, rule metadata loading and testing of rule definitions, and verifying that the rule detects code smells if provided with a valid code snippet.

To satisfy the first phase of the testing, we used Scalatest [17] to write basic tests and to verify that plugins internal components operate correctly. We chose Scalatest because it is a standard unit testing framework for Scala projects and it allows developers to write unit tests in the form of specification.

For the second phase, SonarQube provides a testing framework that allows similarly testing the rules that they would have been used in production. In this framework, testing is performed as follows: firstly, you need to specify a rule which you would like to test by creating an instance of the rule and passing it to the framework, and secondly, you need to provide a code snippet that contains a code smell that you would like to verify with the rule.

We can see an example of such a test file in figure 1. This code snippet contains code that a person writing the test would consider a code smell for a particular rule and as seen on line 3, we can specify which line the plugin should report by placing a comment on that line in form of line comment: `// Noncompliant {{Optional message}}`. Then

the framework will verify if the plugin reports an issue in the provided code snippet and whether the plugin reported the same line as marked in the source file and if the message was the same as provided in the source file. This approach allows us to verify the code smells without the need to run the analysis separately on the server and detect most of the issues with the definitions during the testing phase.

```
1   package com.example.test;
2
3   public class DataClass { // Noncompliant {{Refactor this
       class so it includes more than just data}}
4   public String field1;
5   public int field2;
6   }
```

Figure 1. Example of a data class code smell test file.

3.4 Datasets

To evaluate the plugin and see how it performs, we needed to perform an analysis of existing applications and then perform statistical analysis to determine if the results are statistically significant. The analysis itself has two main objectives. Firstly, we wanted to see how existing (previously implemented by the literature) code smells were distributed inside analyzed applications. Secondly, we wanted to see how new code smells (previously not implemented by the literature) are distributed inside analyzed applications.

For this purpose, we used a corpus consisting of 1509 applications that were published in [6] by Góis Mateus and Martinez. This corpus consists of Java and Kotlin applications for the Android platform. In our analysis, we analyzed applications developed in Java and ignored ones implemented in Kotlin.

The main reason why we selected this corpus is that it provides links to the original repositories. This is important because our tool relies on the analysis of the source code and not compiled `.apk` binaries that are deployed to the Android platform. This way we were able to clone the original repositories and perform source code analysis of the applications.

3.5 Analysis procedure

Once the tool has been implemented and tested internally, we needed to evaluate it on some kind of corpus. As mentioned previously in section 3.4, we had already selected the dataset, so we then proceeded with evaluation of our plugin on the selected dataset. The analysis procedure consisted of the following steps:

1. Clone the projects from the corpus
2. For each project, identify the build system that is used
3. Build the projects
4. Perform the analysis to find the thresholds for the rules
5. Update the thresholds for the rules inside the plugin
6. Perform analysis to detect the code smells inside the corpus
7. Export the results from SonarQube
8. Perform the analysis of the data using R

As a first step in the analysis, we needed to create a local copy of the source code to analyze it. Since the corpus provides direct links to the git repositories, each repository could be fairly easily cloned using `git clone` command.

Next, we needed to identify which build system the project uses. SonarQube supports Gradle [7] and Maven [4] build systems. This step was fairly straight forward, we needed to check which file is present in the cloned project directory. For Gradle projects, we checked if a file named `build.gradle` was present in the directory tree of the project. For Maven projects, we checked if a file named `pom.xml`was present in the directory tree of the project. If none of the aforementioned files were present, this meant that the project could not be built and analyzed and thus was excluded from the analysis.

After the build system for the project had been identified, we needed to build the projects in order to get compiled classes of the projects. This might seem like a redundant step since previously we said that we are performing the analysis of the source files of the application, but this is one of the requirements of SonarQube, which needs the compiled files to perform the analysis.

Some of the rules (such as "Long method" or "Blob class") require thresholds for code smell detection. Those thresholds have to be found manually for each analyzed corpus so that the detections could be as accurate as possible.

Exact thresholds, their values, and calculation formulas are presented in section 4. In terms of the analysis procedure, this meant that we had to run the first iteration of the analysis on the projects to collect those thresholds.

Once the required data for the thresholds have been gathered, we needed to calculate the values for the thresholds and then update them inside the plugin, so that it would use new values when detecting code smells.

After the thresholds were updated, we could finally run the analysis of the applications. This means that we had to run the SonarQube from the command line inside the project directory for each project. This procedure would check the application for each of our

rules and then report results back to SonarQube so that the analysis results would be available inside the SonarQube itself.

Finally, when the applications were analyzed, we needed to export the results. SonarQube provides a REST API, which allowed us to export the results into the format that we could further analyze. For further analysis, we exported the results of the analysis into the CSV file. This allowed us to explore the results of the analysis using R and the results of the result analysis can be seen in section 4.2.

To the reader, it might seem that most of the steps of this procedure can be automated and this would be a correct observation. Since most of the steps of the analysis could be easily automated, we also created a tool that performed most of the steps automatically. This tool is also open source and can be found under the following Github repository [12].

4 Results

4.1 Developed plugin

4.1.1 Implementation

As a result of development, we created a plugin that can detect 26 code smells. The resulting implementation contains 119 classes and 6476 lines of Scala code. This number includes tests but does not include test resources (Java files that were used to test code smell detection).

The basic flow of the plugin operation can be seen in figure 3. Firstly, we have to register our plugin inside a SonarQube instance. Secondly, we have to load our rules and descriptions of the rules into the instance of SonarQube. Finally, the user can create a new profile with our rules or add the rules to an existing profile and then run the analysis using the rules that we provide inside our plugin.

As mentioned previously in section 3.1, the plugin itself operates on the Java AST. As an example, we provided an implementation of "Complex class" rule which can be seen in figure 2. This code smell inspects all class nodes which are found in the AST, calculates the complexity of all methods in the class, and reports an issue if the complexity of the class is greater than the configured threshold.

```
@Rule(key = "ComplexClass")
class ComplexClass
extends JavaRule
with ComplexityAccessor
with ContextReporter {

private var context: JavaFileScannerContext = _

private var veryHighClassComplexity: Double = _

override def scanFile(
javaFileScannerContext: JavaFileScannerContext): Unit = {
this.context = javaFileScannerContext

veryHighClassComplexity = config
.flatMap(
_.getDouble(
ConfigurationProperties.COMPLEX_CLASS_VERY_HIGH_COMPLEXITY.
    key))
.orElse(
ConfigurationProperties.COMPLEX_CLASS_VERY_HIGH_COMPLEXITY.
    defaultValue.toDouble)

scan(context.getTree)
}

override def scannerContext: JavaFileScannerContext = context

override def visitClass(tree: ClassTree): Unit = {
val classComplexity = tree.members.asScala
.filter(_.is(Kind.METHOD))
.map(_.asInstanceOf[MethodTree])
.map(complexity)
.sum

report(
s"Complex class: class complexity $classComplexity is higher
    than configured: $veryHighClassComplexity",
tree,
classComplexity >= veryHighClassComplexity
)

super.visitClass(tree)
}
}
```

Figure 2. Implementation of "Complex class" rule.

The high-level architecture of the plugin can be seen in figure 3. The plugin initialization is started when SonarQube loads the plugin into memory. When the plugin is loaded, the plugin registers its extensions and delegates all of the further work to the extensions. Currently there are 2 extensions, rules registrar (`SonarAcademicRulesRegistrar`) and sensor (`SonarAcademicSensor`).

The purpose rules registrar is to load all of the rules, their descriptions and then register those rules with the SonarQube so that the rules could be enabled or disabled by the user in the SonarQube user interface and further be used in the analysis.

The sensor acts as a bridge between SonarQube and rules which require context during their runtime. The idea is that sensor receives a whole project as an input, as opposed to single files which are the case for the regular rules, and then feeds individual files to the sensor rules. The difference between regular rules and sensor rules is explained in section 4.1.2.

After the plugin has been loaded, SonarQube delegates the analysis of single files to the rules. For sensor rules, SonarQube delegates the whole project, and then the sensor performs analysis of single files for all sensor rules.

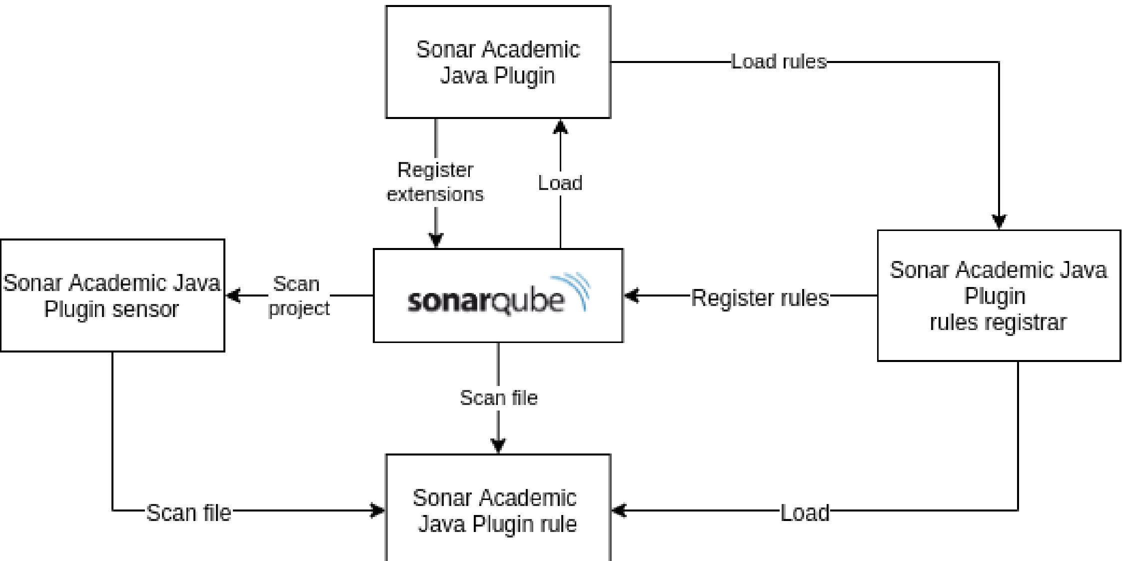

Figure 3. High level architecture of Sonar Academic Java Plugin.

The plugin is open source and full implementation can be found in the Github repository [14]. The implementation contains not only the rules for the code smell detection but also additional classes that are required for the correct operation and rule registration inside the SonarQube instance.

4.1.2 Translating queries into SonarQube rules

As the code smells described by Fowler are quite abstract, we needed more concrete definitions to implement the code smells. To get a more concrete implementation of the code smells we used the paper by Rahkema and Pfahl. In [16], code smells are defined as Neo4j queries. However, SonarQube uses a different pattern for code smell detection.

In SonarQube, code smells are detected by applying visitor pattern [9] on the parsed AST of Java source code. Code smells are detecting by visiting specific nodes in the tree (such as classes or methods) and analyzing the visited nodes to find the code smell pattern. So to translate Neo4j queries into the definitions that apply to SonarQube, we used the following algorithm that can be seen in algorithm 1.

Firstly, we needed to determine the context of the rule, such as if the rule should detect methods, classes, or variables. Then, we needed to decide if the rule required could be detected in a single scan of the context. For example, if we need to decide whether or not the method has X amount of parameters, we would need to visit this single method only a single time and we do not need to know anything about other methods. But in some cases, we need to account for other nodes in the tree. For example, in `Shotgun Surgery` we need to detect method invocations that can be present in other classes.

Algorithm 1: Translation Neo4j queries into SonarQube rules

Input: Neo4j query

Result: Neo4j query translated into SonarQube implementation

```
1 foreach Neo4j query q do
2     context ← DetermineContext(q)
3     if rule r classes required for context > 1 then
4         SensorRule(context, r);
5     else
6         RegularRule(context, r);
7     ;
```

As we can see from the algorithm 1, there are 2 types of rules. Regular rules (defined as `RegularRule` in the algorithm) require the context only of a single class under analysis. Those are the rules, where we can decide whether a specific code smell exists or not just by looking at a single node and its sub-nodes. Example of such rules are `Data class`, `Message chains` or `Blob class`.

However, for some rules, we need more than a context of a single file. Examples of such rules are `Brain method` and `Cyclic dependencies`. To give more freedom to plugin writers, SonarQube provides an additional way to write custom rules - sensors. `Sensor` is an interface provided by SonarQube, which defines only a single method

execute. This is called by the scanner during the analysis and the behavior of this method is defined by the implementation. So, we created own sensor that contains stateful rules. The sensor accepts Java files, parses them into the AST, adds type information, and then passes the AST to the rules. Rules scan the tree passed by the sensor. After all of the rules have been scanned, sensor calls method afterAllScanned, indicating that all of the input classes have been scanned and allows rules to report any issues detected during the analysis. Visualization of the sensor analysis can be seen in algorithm 2.

Algorithm 2: Performing analysis using sensor

Input: Sensor context
Result: Analysis results reported to SonarQube server

```
1  c ← CreateClassPath;
2  foreach Java file j do
3      AST ← ParseJavaFile(j);
4      AstWithSymbolicModel ← UpdateSymbolicModel(j, c);
5      foreach Rule r do
6          Scan(r, AstWithSymbolicModel);
7      ;
8  foreach Rule r do
9      AfterAllScanned();
10 UploadResults();
```

4.1.3 Testing

As mentioned in section 3.3, we also performed internal testing before applying the tool to real projects. We performed unit testing using Scalatest [17]. To measure line coverage, we used codecov [2] which allowed us to measure code coverage during every build in the continuous integration (CI) environment.

In the end, we had at least a single positive test case (a code pattern where our analyzer should detect a code smell) for every rule that we have implemented and we achieved line coverage of 97.25% with 92 test cases.

4.2 Analysis results

As a result, 240 applications were successfully analyzed. This number is vastly different from the total amount of applications (1509) that are present in the corpus for multiple reasons. Firstly, some projects were not using a build system (Gradle or Maven). This is

an issue because to compile the project we would need to specify the classpath manually and this is quite difficult to do programmatically. Moreover, even if we managed to compile the projects, SonarQube is provided as a plugin for the build systems which means that we would still not be able to run the analysis. Secondly, some projects required extra configuration before building, which is also not possible to do programmatically because each project is different and requires manual configuration.

The analysis was performed on a machine with `Intel(R) Core(TM) i5-8250U CPU @ 1.60GHz` and 16GB of RAM. The statistics gathering for the thresholds took 441 minutes in total with 1.84 minutes on the average per project. However, running a project with all of the rules for code smell detections took 565 minutes with 2.35 minutes on average per project. Note that this includes only the analysis itself and does not factor in the process of building and compiling the source application.

In total, we analyzed 1450793 lines of code. On average, each application had 6044 lines of code with a median of 2982 lines of code. The largest application contained 52358 lines of code and the smallest application had 73 lines of code. The total number of detected code smells is 109602 which means that there are 0.08 code smells on average per line of code.

In section 3.5 we mentioned collecting the thresholds for some of the code smells. The main idea for calculating the thresholds for the corpus is that this way we can reduce the amount of false-positive detections and only detect the cases where the presence of the code smell is evident.

Table 1 shows the statistics that were used to calculate the thresholds for the rules. The first column shows the node type which was used to calculate the attribute. The second column displays the attribute name that was calculated for the specific node. Columns Q_1, Q_2 and Q_3 display different quartiles for the collected values.

As mentioned in section 3.5, the statistics were gathered from the analyzed applications. To obtain the thresholds, we had to run the threshold collection 2 times, since during the first analysis we did not exclude the anonymous classes. We decided that anonymous classes should be excluded from the threshold collection since in Android applications anonymous classes are used for listeners and thus occur very often because listeners are used to react to user inputs and mobile applications are mostly about working with user input.

As can be seen, some of the values are missing from the table 1. Due to time constraints, we were not able to collect all of the required statistics for all of the code smells and since we were missing some of the values, we reused the values that were used by Rahkema and Pfahl in [16].

The complete list of calculated thresholds can be seen in table 2. Values in *italics* were not calculated based on table 1 due to missing values and thus were reused from the paper by [16]. These were the thresholds that were used for code smell detection when retrieving the results. The column "Code smell name" displays the rule for which

Location	**Event type**	Q_1	Q_2	Q_3
Class	attributes	1	2.0000000	5.000000
Class	cohesion	-1	-1.0000000	0.000000
Class	comments	-	-	-
Class	complexity	0	3.0000000	12.500000
Class	complexityRatio	0	0.8888889	2.467376
Class	coupling	-	-	-
Class	instructions	4	16.0000000	45.000000
Class	methods	1	3.0000000	7.000000
Method	calls	-	-	-
Method	chainLength	2	2.0000000	3.000000
Method	complexity	0	0.0000000	2.000000
Method	instructions	1	2.0000000	8.000000
Method	parameters	0	1.0000000	1.000000
Method	switchStatements	0	0.0000000	0.000000
Interface	numberOfMethods	1	1.0000000	2.000000

Table 1. Values used for threshold calculation.

the threshold was calculated. The column “Variable” displays the variable for which the threshold was calculated in the rule. The column “Mapping from threshold table” displays the “Location” and “Event type” from table 1 that was used to calculate the value. The column “Formula” shows the formula that was used to calculate the value. The column “Value” shows the final value that was used for “Variable” for the rule inside “Code smell name”.

Code smell name	**Variable**	**Mapping from table 1**	**Formula**	**Value**
Long method	veryHighNumberOfInstructions	Method: instructions	$Q_3 + (Q_3 - Q_1) * 1.5$	18.5
Blob class	veryHighLackOfCohesionInMethods	Class: cohesion	$Q_3 + (Q_3 - Q_1) * 1.5$	1.5
Blob class	veryHighNumberOfMethods	Class: methods	$Q_3 + (Q_3 - Q_1) * 1.5$	16
Blob class	veryHighNumberOfAttributes	Class: attributes	$Q_3 + (Q_3 - Q_1) * 1.5$	11
Shotgun surgery	veryHighNumberOfCallers	Method: calls	$Q_3 + (Q_3 - Q_1) * 1.5$	*2.5*
Switch statements	veryHighNumberOfSwitchStatements	Method: switchStatements	$Q_3 + (Q_3 - Q_1) * 1.5$	0
Lazy class	mediumNumberOfInstructions	Class: instructions	Q_2	16
Lazy class	mediumCouplingBetweenObjectClasses	Class: coupling	Q_2	*0*
Message chains	veryHighNumberOfChainedMessages	Method: chainLength	$Q_3 + (Q_3 - Q_1) * 1.5$	4.5
Comments	veryHighNumberOfComments	Class: comments	$Q_3 + (Q_3 - Q_1) * 1.5$	*29.5*
Divergent change	veryHighNumberOfCalledMethods	-	$Q_3 + (Q_3 - Q_1) * 1.5$	*2.5*
Long parameter list	veryHighNumberOfParameters	Method: parameters	$Q_3 + (Q_3 - Q_1) * 1.5$	2.5
Middle man	lowNumberOfInstructionsMethod	Method: instructions	$\max(Q_1 - (Q_3 - Q_1) * 1.5, 0)$	0
Inappropriate intimacy	highNumberOfCallsBetweenClasses	-	Q_3	*5*
Brain method	highNumberOfInstructionsForClass	Class: instructions	Q_3	45
Brain method	highCyclomaticComplexity	Method: complexity	Q_3	2
God class	veryHighWeightedMethodCount	Class: complexity	$Q_3 + (Q_3 - Q_1) * 1.5$	6.16844
Primitive obsession	veryHighPrimitiveVariableUse	-	$\mathrm{Q}_3+(\mathrm{Q}_3 - Q_1) * 1.5$	*6*
Complex class	veryHighClassComplexity	Class: complexity	$Q_3 + (Q_3 - Q_1) * 1.5$	31.25
Swiss army knife	veryHighNumberOfMethods	Interface: numberOfMethods	$Q_3 + (Q_3 - Q_1) * 1.5$	3.5

Table 2. Calculated thresholds based on values from table 1.

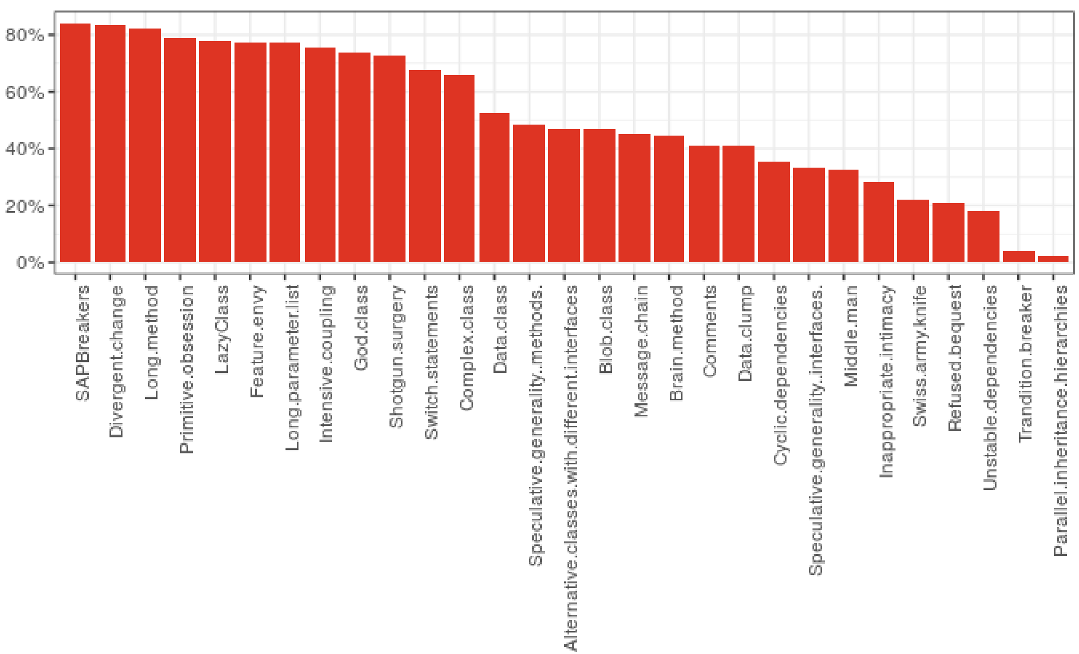

Figure 4. How many applications contain code smell X?

Now that we have shown the general information about the analysis process and thresholds, we can finally go over the results that we found. Figure 4 shows how many of the applications contain each code smell.

As we can see 29 out of 30 code smells have been detected. This means that each implemented code smell has been detected at least once. The only code smell missing from the figure 4 is “Missing template method”, which has not been detected in any of the analyzed applications.

The rule that has been present in the highest amount of applications is “SAPBreaker” which has been present in 84% (201) of the applications. The least frequent code smell is “Parallel inheritance hierarchies” which has been present in only 3% (6) of the applications.

Figure 5, displays the distribution of detected code smells inside analyzed applications.

Here we can see that “Divergent change” makes up a significant part of the distribution which makes up to 28% (30718 detections) of total detected code smells. It is followed by the “SAPBreakers” with detection rate of 9% (10341) detections and “Long method” with detection rate of 8.6% (8343 detections).

The least frequent code smells, excluding “Missing Template Method” due to the fact that it has 0 detections, are the following: “Unstable dependencies” with detection

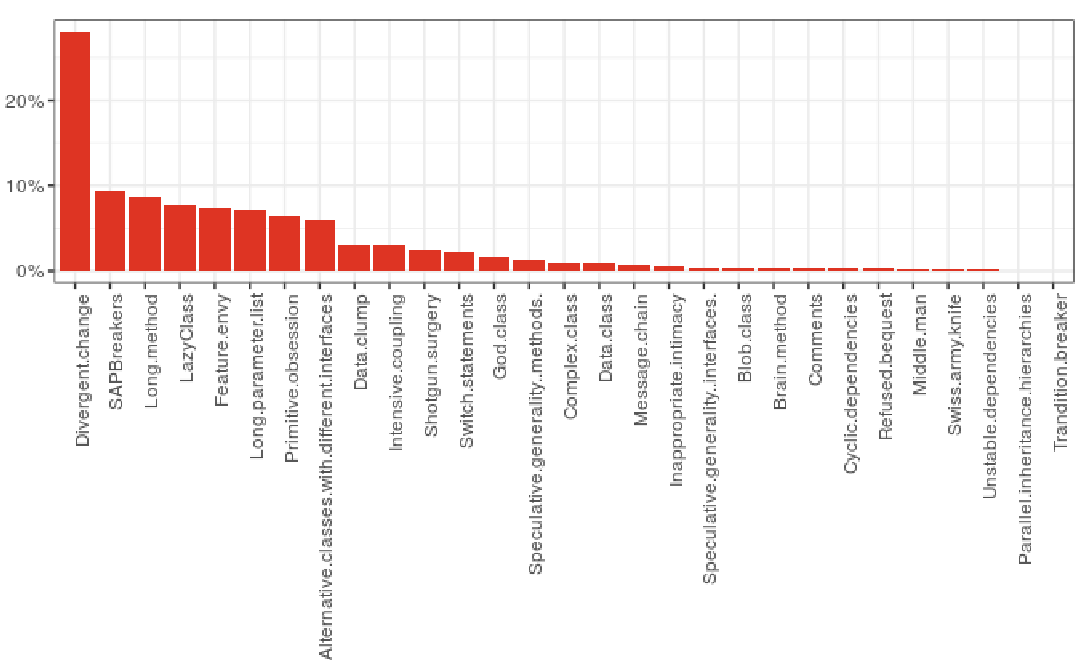

Figure 5. Distribution of code smells inside the analyzed applications.

rate of 0.9% (109 detections total), "Parallel inheritance hierarchies" with detection rate of 0.6% (70 detections total) and "Tradition breaker" with detection rate of 0.1% (17 detections total).

5 Discussion

5.1 Encountered problems

In this section, we will discuss the difficulties encountered during the development of the plugin.

The first problem is related to the analysis model of the SonarQube. Currently, SonarQube assumes that all the code smells can be detected during a single analysis of an AST node. The issue is that a large number of code smells implemented by us require the context of other AST nodes. For example, to find cyclic dependencies, we first need to detect all the dependencies of a class and then, after all the dependencies of all the classes inside of the application have been detected, we would transform the dependencies into a graph and look for the cycles in the graph.

Fortunately, SonarQube provides a possibility to implement sensors, which can fully control the execution flow of the analysis. By using sensors, we were able to create a custom execution pipeline for the rules that require the context of the project and not just a single node. This approach allowed us to solve the original problem, but lead to another one.

The second problem is related to the API that SonarQube plugin base exposes. As mentioned in section 3.1, we followed the guide provided by the SonarQube community to implement our plugin. This includes the usage of the base dependency which includes the API for accessing the AST generated by SonarQube. Additionally, this dependency also contains other utilities, such as various visitors which can be used to detect various measures inside the code such as complexity or count the lines of code. The problem is that some classes are available only during the compilation phase of the application and are missing from the runtime environment when the plugin is loaded for the analysis execution.

As mentioned previously, we used sensors to implement the pipeline for the rules that have state. The sensor pipeline has to be built from scratch as the input that you receive are source files of the project and not parsed AST as it is with normal rules. This means that we needed to parse the files into the AST by ourselves, and it was not that simple to do due to some classes missing from the runtime.

One of the solutions could be to include the full library inside the compiled artifact and bring all those classes with the plugin itself. This would mean we would also bring the implementations of the AST tree nodes and this would cause `ClassCastException-s` at runtime of the plugin because of how Java Virtual Machine (JVM) handles classes loaded by different class loaders: even though the classes are effectively the same if they were loaded by the different class loaders JVM cannot guarantee that they are indeed the same, so casting between those types is not allowed.

To fix this issue, we forked the plugin base dependency and all the classes that are

exposed by the API were left out of it, to fix the aforementioned problem with class loaders. Then we could use our dependency to parse the input files to AST the same way SonarQube would do it and use all the utilities that are present in the SonarQube base dependency for our own needs.

The library is also open source and can be found in the Github repository [13].

5.2 Issues and limitations

After the implementation has been completed, there are still some issues that we would like to fix in the future.

Firstly, some rules have been implemented as normal SonarQube rules, when they should be implemented as sensor rules. As an example, let us look into `Shotgun surgery`. This rule was implemented as a regular SonarQube rule although it has state and this results in a situation where we have to check for issues after every scanned class because we never know whether this class is the last we would scan. This was implemented so because the implementation of the rule was done before we developed the internal sensor "framework", and there was no reason to rewrite it because it works the way it is implemented now. Nonetheless, it would be more efficient to rewrite this rule as a sensor rule.

Secondly, we did not have time to implement nice descriptions of the rules. SonarQube UI provides a possibility to include a description for the rule that the user would read to understand the idea behind the rule and also see compliant / non-compliant code examples as it can be seen in figure 6. Additionally, on that page, we can see the type of the detection (which in our case is "Code smell") and the severity of the detection, which is set by the user when activating the rule for the specific case (which is set to "minor" by default for our rules).

Currently, we assume that the user is familiar with the rules and does not require any assistance in understanding how a given rule works, but it would still be nice to have complete and concise descriptions about rule implementations.

Finally, there as an issue with threshold calculation that has been mentioned in section 4.2. As we did not calculate some thresholds by ourselves, they might be drastically different for Java applications from Swift applications.

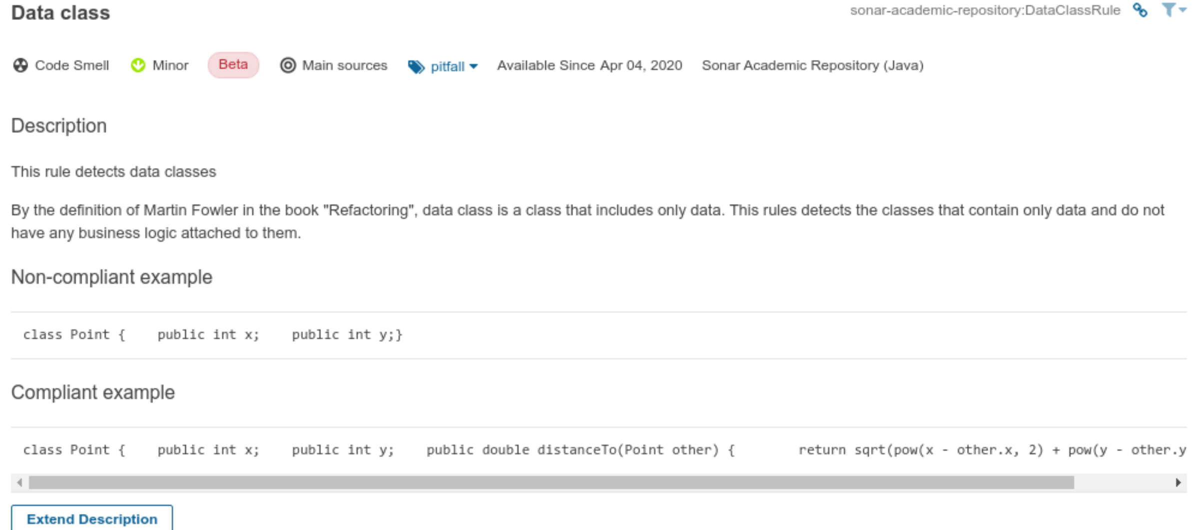

Figure 6. SonarQube rule description example.

5.3 Comparison to published results

In section 4 we introduced the results that we collected when analyzing our corpus of the applications. Previously, Mannan et al. in [11] also investigated code smells in Android applications. The main reason why we want to compare the tool described in this paper to the tool used in [11] is that the tool used in that paper is no longer available. As it was closed-source, commercial solution for code smell analysis, we would like to know how our implementation compares to the one in [11].

As Mannan et al. did not provide any concrete distribution of the detected code smells inside analyzed applications, we had to rely on figure 2 in [11] in order to get the values collected as a result of the analysis. Table 3 shows the values that we were able to extract from the figure which we used for comparison with our results. The figure contains two datasets ("mobile" and "desktop" applications) and we used the values for the "mobile" dataset, since we were analyzing applications for the Android platform.

As in our paper, we analyzed more code smells than in [11], we had to normalize our results so that we could compare the distribution only of those code smells that are present in [11]. Table 4 shows normalized distribution of the code smells (only the ones that are present in [11]).

A comparison of distributions of both papers can be seen in figure 7. It is important to note that we left some of the code smells out of the comparison. The reason is that we did not implement those rules in our plugin, as those are already natively supported by SonarQube. The rules that we excluded from the comparison are "external duplication", "sibling duplication" and "internal duplication".

Code smell name	Percentage of occurrence (%)
Data class	18
Data clump	15.5
Cyclic dependencies	13
Feature envy	9.5
External duplication	9
SAPBreaker	9
God class	5.5
Sibling duplication	4.5
Divergent change	4
Message chains	2.5
Intensive coupling	2.4
Tradition breaker	2
Unstable dependencies	1.75
Refused bequest	1.5
Blob class	1
Shotgun surgery	0.5
Internal duplication	0.25

Table 3. Distribution of detected code smells from [11].

Code smell name	Percentage of the occurrence (%)
Data class	0.85217423
Data clump	5.32531265
Cyclic dependencies	0.54297926
Feature envy	12.69273389
SAPBreaker	16.37011240
God class	3.07424410
Divergent change	48.62751306
Message chains	1.24267849
Intensive coupling	5.06252968
Tradition breaker	0.02691151
Unstable dependencies	0.17255026
Refused bequest	0.50498654
Blob class	0.66170651
Shotgun surgery	4.21719171

Table 4. Distribution of selected code smells for this paper.

From this comparison, we can see that the number of detections for some code smells is pretty similar and quite different for others. For instance, the number of detections for “blob class” and “feature envy” is pretty similar, while “data class” and “divergent change” are quite different.

There are multiple reasons why the results are different but here are the ideas on why the difference more noticeable in some occasions than others.

Implementation might be different. This means that the code smell definitions in [11] might be different from our definition and thus we are detecting different things or detecting the same things differently. Since the tool from [11] is closed-source then we are not able to compare the implementations.

Corpora are different. The source datasets were not provided by Mannan et al., so we are not able to verify our tool against the same dataset. If we used the same dataset, the results might have been more similar or, on the contrary, different.

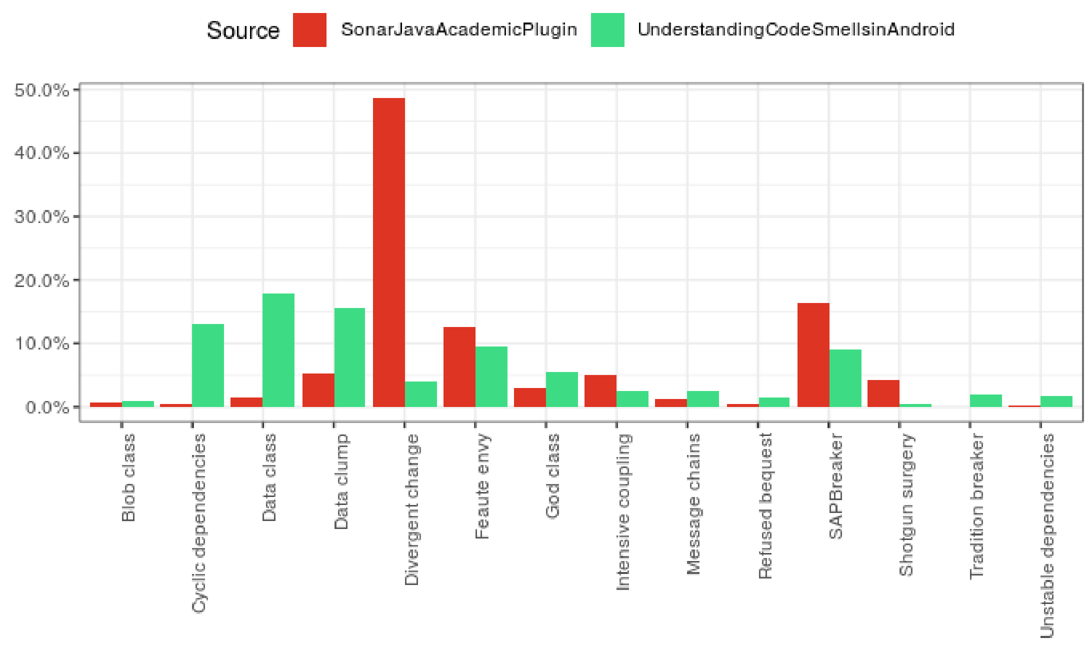

Figure 7. Distribution of detected code smells in our paper versus [11].

As the reader might have noticed, there are various reasons why the results might be different. Nonetheless, different results on different datasets do not mean that one tool is more useful than the other. In the next chapter, we will look into the potential usages of the tool developed by us.

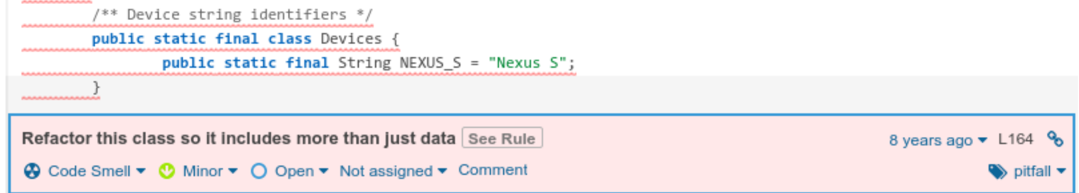

Figure 8. Example of a reported issue on the server side.

5.4 Potential usages

In section 1 we mentioned various roles in software development process who might be interested in the results of an analysis of a software project. In the next subsections, we will go over the ways how each of the focus groups can use the plugin.

5.4.1 Developers

Developers are our main target group. SonarQube also can be integrated into the IDE and be used from there. Unfortunately, this requires the premium edition (paid) of SonarQube and we only have access to the community edition (free) so we are not able to show the screenshots from the IDE. Nonetheless, full information is available on the server-side after the analysis has been run in the CI/CD environment. Figure 8 shows the information that the developer sees when a code smell is reported on the Sonar server side. In the picture we can see the message that was reported by the static analyzer, when it was first introduced, the line it occurs on, what is the type of reported issue (in our case it is "code smell") and severity of the detection.

Additionally, the developer has an option to press the "See rule" button. When opened, this view shows the description of the reported code smell, as it can be seen in figure 9. This allows the developers to quickly access rule definitions to see compliant and non-compliant rule examples.

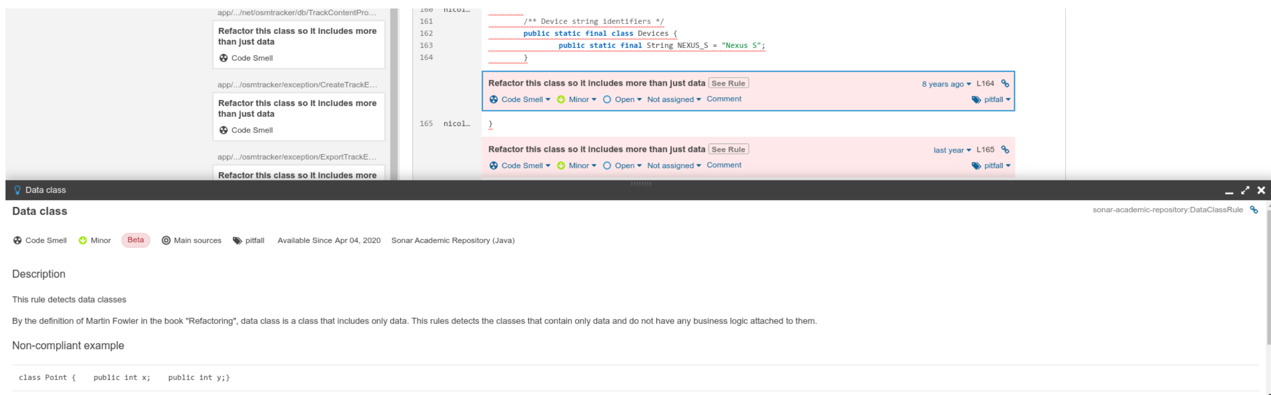

Figure 9. Example of expanded rule description.

5.4.2 Project managers

As mentioned in the previous section, it is possible to integrate the plugin into the IDE. However, not all the roles in software development are related to coding. SonarQube provides a nice user interface, that can be useful for project managers to receive an overview of a project, as seen in figure 10. In this figure, we can see different metrics that are helpful to identify the state of the project. For example, the number of bugs, vulnerabilities, and code smells detected during static code analysis.

The view of figure 10 can be further expanded to see a more detailed overview of detected code smells/vulnerabilities as can be seen in figure 11.

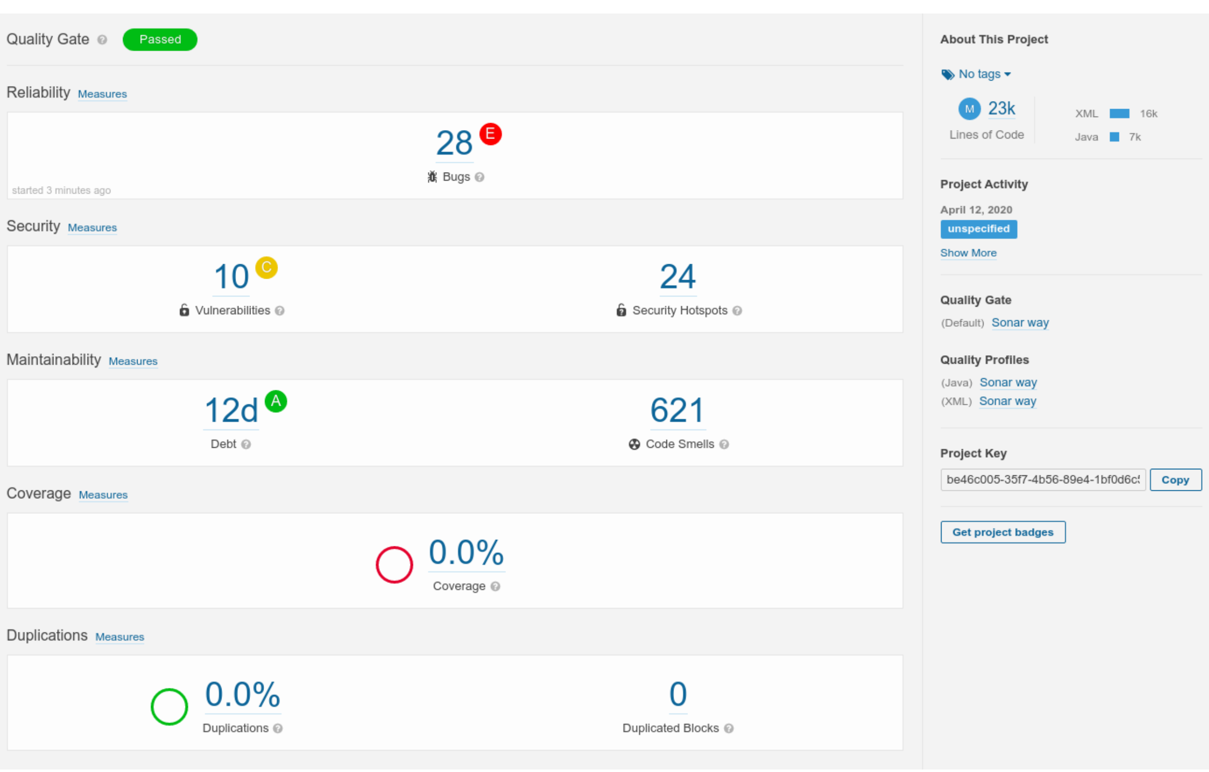

Figure 10. Example of a project overview in the SonarQube user interface.

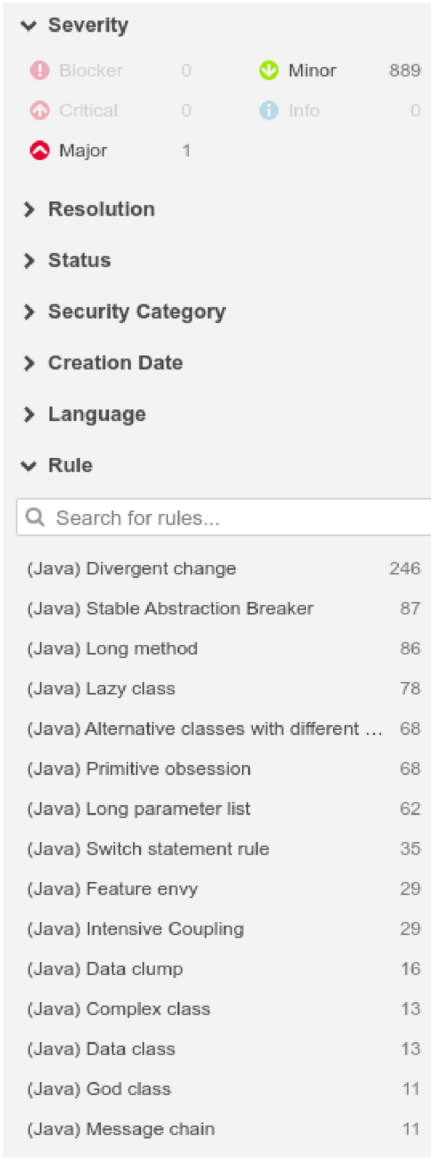

Figure 11. Example of code smell detections grouped by rule.

5.4.3 Data scientists

In the era of big data, a lot of useful information can be extracted from the collected data. Static code analysis is not an exception. If an organization maintains multiple projects in a SonarQube instance, the data from the static analysis could be used to find correlations between different code smells in multiple projects. This could show some trends where certain code smells are popular in multiple teams and could lead to the introduction of new rules to fix the code quality issues.

SonarQube provides a rich set of web services that can be used to extract the data into external applications. The base URI `/api/webservices/list` describes all web services exposed by the current SonarQube instance. For example, the API can be used to export the issues found in the project for further analysis. To achieve this, one would use the `/api/issues/search` endpoint to search for issues inside the projects.

6 Conclusion

In this thesis we developed a static code analyzer for Java applications in a form of a plugin for SonarQube plugin. Then, we performed the analysis of a corpus of applications and analyzed the results. From the results of the analysis, we could see that that tool actually works since it detects the code smells that we have defined. However, we did not perform an empirical study if the tool actually helps the developers to write better code.

We also understand that there might be some limitations with the dataset that we have selected. The dataset that we have selected was not previously used for Java code smell analysis and number of the projects that we have successfully analyzed is not that big. Nonetheless, this dataset is enough to verify that the our tool is able to analyze the applications and is able to do so rather efficiently.

As for the future work, there is still improvement possibilities. Firstly, we would like to implement nice descriptions for the code smells. This would allow the developers to see compliant and non-compliant code examples in order to grasp the idea why a particular code snippet is a code smell. Secondly, we would like to implement the integration with the IDE, such as IntelliJ IDEA or Eclipse. This would allow the developers to see the code smells from the comfort of their own IDE and would allow to detect the code smells during the code writing phase, instead of the compilation time.

Bibliography

[1] Iftekhar Ahmed, Umme Ayda Mannan, Rahul Gopinath, and Carlos Jensen. An empirical study of design degradation: How software projects get worse over time. In *2015 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 1–10. IEEE, 2015.

[2] Codecov. Codecov. URL https://codecov.io/. Accessed: 01.03.2020.

[3] Francesca Arcelli Fontana, Valentina Lenarduzzi, Riccardo Roveda, and Davide Taibi. Are architectural smells independent from code smells? an empirical study. *Journal of Systems and Software*, 154:139–156, 2019. doi: 10.1016/j.jss.2019.04.066. URL https://doi.org/10.1016/j.jss.2019.04.066.

[4] The Apache Software Foundation. Maven. URL https://maven.apache.org/. Accessed: 15.03.2020.

[5] Martin Fowler. Refactoring: Improving the design of existing code. In Don Wells and Laurie A. Williams, editors, *Extreme Programming and Agile Methods - XP/Agile Universe 2002, Second XP Universe and First Agile Universe Conference Chicago, IL, USA, August 4-7, 2002, Proceedings*, volume 2418 of *Lecture Notes in Computer Science*, page 256. Springer, 2002. doi: 10.1007/3-540-45672-4_31. URL https://doi.org/10.1007/3-540-45672-4_31.

[6] Bruno Góis Mateus and Matias Martinez. An empirical study on quality of android applications written in kotlin language. *Empirical Software Engineering*, Jun 2019. ISSN 1573-7616. doi: 10.1007/s10664-019-09727-4. URL https://doi.org/10.1007/s10664-019-09727-4.

[7] Gradle. Gradle build tool. URL https://gradle.org/. Accessed: 15.03.2020.

[8] Michael Gumowski. Writing custom java rules 101 - plugins - doc sonarqube. https://docs.sonarqube.org/display/PLUG/Writing+Custom+Java+Rules+101, apr 2019. Accessed: 15.02.2020.

[9] Refactoring Guru. Visitor. https://refactoring.guru/design-patterns/visitor. Acessed: 01.03.2020.

[10] Geoffrey Hecht, Benomar Omar, Romain Rouvoy, Naouel Moha, and Laurence Duchien. Tracking the Software Quality of Android Applications along their Evolution . In Lars Grunske and Michael Whalen, editors, *30th IEEE/ACM International Conference on Automated Software Engineering* , Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering

(ASE 2015), page 12, Lincoln, Nebraska, United States, November 2015. IEEE . URL https://hal.inria.fr/hal-01178734.

[11] Umme Ayda Mannan, Iftekhar Ahmed, Rana Abdullah M Almurshed, Danny Dig, and Carlos Jensen. Understanding code smells in android applications. In *2016 IEEE/ACM International Conference on Mobile Software Engineering and Systems (MOBILESoft)*, pages 225–236. IEEE, 2016.

[12] Stanislav Mõškovski. Sonar bulk analyzer, . URL https://github.com/Zukkari/sonar-bulk-analyzer. Accessed: 15.03.2020.

[13] Stanislav Mõškovski. Sonar java extracted, . URL https://github.com/Zukkari/sonar-java-extracted. Accessed: 04.04.2020.

[14] Stanislav Mõškovski. Sonar java academic plugin, . URL https://github.com/Zukkari/sonar-java-academic-plugin. Accessed: 15.03.2020.

[15] Inc. Neo4j. Neo4j. URL https://neo4j.com/. Accessed: 29.10.2019.

[16] Kristiina Rahkema and Dietmar Pfahl. Empirical study on code smells in ios applications. In *MOBILESoft 2020 Technical Papers*, 2020.

[17] Scalatest. Scalatest. URL http://www.scalatest.org/. Accessed: 16.02.2020.

[18] SonarLint. Sonarlint. URL https://www.sonarlint.org/. Accessed: 26.04.2020.

[19] SonarQube. Sonarqube, . URL https://www.sonarqube.org/. Accessed: 29.10.2019.

[20] SonarQube. Sonarqube github repository, . URL https://github.com/SonarSource/sonarqube. Accessed: 27.10.2019.

[21] SonarSource. Sonarjava: Java static analyzer for sonarqube & sonarlint. URL https://github.com/SonarSource/sonar-java. Accessed: 15.02.2020.

[22] Techopedia. Technical debt. Available at https://www.techopedia.com/definition/27913/technical-debt (27.10.2019).

[23] Ewan D. Tempero, Craig Anslow, Jens Dietrich, Ted Han, Jing Li, Markus Lumpe, Hayden Melton, and James Noble. The qualitas corpus: A curated collection of java code for empirical studies. In Jun Han and Tran Dan Thu, editors, *17th Asia Pacific Software Engineering Conference, APSEC 2010, Sydney, Australia, November 30 - December 3, 2010*, pages 336–345. IEEE Computer Society, 2010. doi: 10.1109/APSEC.2010.46. URL https://doi.org/10.1109/APSEC.2010.46.

[24] Michele Tufano, Fabio Palomba, Gabriele Bavota, Rocco Oliveto, Massimiliano Di Penta, Andrea De Lucia, and Denys Poshyvanyk. When and why your code starts to smell bad. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ICSE '15, pages 403–414, Piscataway, NJ, USA, 2015. IEEE Press. ISBN 978-1-4799-1934-5. URL http://dl.acm.org/citation.cfm?id=2818754.2818805.

[25] B. A. Wichmann, A. A. Canning, D. L. Clutterbuck, L. A. Winsborrow, N. J. Ward, and D. W. R. M. Marsh. Industrial perspective on static analysis. *Software Engineering Journal*, 10(2):69–75, 1995. doi: 10.1049/sej.1995.0010. URL https://doi.org/10.1049/sej.1995.0010.

Appendix

I. Implemented code smells

1. Data class
2. Message chain
3. Long method
4. Switch statement
5. Shotgun surgery
6. Lazy class
7. Blob class
8. Refused bequest
9. Comments
10. Cyclic dependencies
11. Tradition breaker
12. Divergent change
13. Feature envy
14. Data clump
15. Parallel inheritance hierarchies
16. Speculative generality (methods)
17. Speculative generality (interfaces)
18. Middle man
19. Primitive obsession
20. Brain method
21. Inappropriate intimacy
22. Swiss army knife

23. Missing template method
24. Unstable dependencies
25. Stable abstraction breaker
26. Complex class

II. Licence

Non-exclusive licence to reproduce thesis and make thesis public

I, **Stanislav Mõškovski**,
(author's name)

1. herewith grant the University of Tartu a free permit (non-exclusive licence) to

   reproduce, for the purpose of preservation, including for adding to the DSpace digital archives until the expiry of the term of copyright,

   Building a tool for detecting code smells in Android application code,
   (title of thesis)

   supervised by Kristiina Rahkema and Dietmar Pfahl.
   (supervisor's name)

2. I grant the University of Tartu a permit to make the work specified in p. 1 available to the public via the web environment of the University of Tartu, including via the DSpace digital archives, under the Creative Commons licence CC BY NC ND 3.0, which allows, by giving appropriate credit to the author, to reproduce, distribute the work and communicate it to the public, and prohibits the creation of derivative works and any commercial use of the work until the expiry of the term of copyright.

3. I am aware of the fact that the author retains the rights specified in p. 1 and 2.

4. I certify that granting the non-exclusive licence does not infringe other persons' intellectual property rights or rights arising from the personal data protection legislation.

Stanislav Mõškovski
03.05.2020