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
Supervisor: Dietmar Pfahl, PhD

Tartu 2020

Building a tool for detecting code smells in Android application code

Abstract:

Write abstract text here

Keywords:

List of keywords

CERCS:

CERCS code and name: `https://www.etis.ee/Portal/Classifiers/Details/d3717f7b-bec8-4cd9-8ea4-c89cd56ca46e`

Building a tool for detecting code smells in Android application code

Lühikokkuvõte:

One or two sentences providing a basic introduction to the field, comprehensible to a scientist in any discipline.

Two to three sentences of more detailed background, comprehensible to scientists in related disciplines.

One sentence clearly stating the general problem being addressed by this particular study.

One sentence summarising the main result (with the words “here we show´´ or their equivalent).

Two or three sentences explaining what the main result reveals in direct comparison to what was thought to be the case previously, or how the main result adds to previous knowledge.

One or two sentences to put the results into a more general context.

Two or three sentences to provide a broader perspective, readily comprehensible to a scientist in any discipline, may be included in the first paragraph if the editor considers that the accessibility of the paper is significantly enhanced by their inclusion.

Võtmesõnad:

List of keywords

CERCS:

CERCS kood ja nimetus: `https://www.etis.ee/Portal/Classifiers/Details/d3717f7b-bec8-4cd9-8ea4-c89cd56ca46e`

Contents

1 Introduction

1.1 Research context

Describe what code smells are. Describe how code smells are different from bugs. Shortly about previous research and how we plan to be different.

1.2 Research motivation

Describe why solution proposed in this thesis is useful. Goals of the thesis:

- Develop a tool, describe why it would be useful from different perspectives (developers, project managers, data scientists)
- Extend the body of knowledge about the occurrence of code smells in Android applications (extend the number of code smells, provide analysis results, compare the results with with already published results, additional results for code smells not yet published in the literature)

1.3 Thesis outline

Shortly describe structure of the thesis. What does each chapter tell the reader?

2 Background

2.1 Code smells

Describe code smells in general, what are they, how were they found at first. Describe how to fix code smells. Describe why would you want to fix them.
Add all of the implemented code smells into appendix, here we should bring some examples about the code smells.
Bring some examples from the Fowler's list and then also describe those that we have implemented.

2.2 Related work

Describe existing tools. Discuss their results and implementations. Here we can describe the same 3 tools that were used during the seminar: paprika, infusion and anti patterns code smells plugin for SonarQube.

2.3 SonarQube

Describe what is SonarQube. Describe why was SonarQube chosen as implementation platform. Describe how can SonarQube be exnteded. Describe what does it mean to write a plugin for SonarQube: extension points (sensor/rule), what are the possibilities for the user (enabling/disabling rules), possibility to run both server side and inside an IDE (SonarLint).

3 Method

Describe the tool here.

What did we build?
Plugin for SonarQube that can detect 29 code smells. We need a tool that can scan a large number of applications. This is best achieved when the tool can be run automatically for a given input project and since the corpus is large, the analysis should be performed on the server side by the program and not by the human who would perform a manual check.

In order to fulfil our task of analyzing a large corpus of application to detect the code smells, we would need to build a tool that is stable, scalable and allow 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 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 need 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.

How?
Followed tutorial that is available on SonarQube documentation page (https://docs.sonarqube.org/display/PLUG/Writing+Custom+Java+Rules+101). But since there were some issues (describe issues with classpath, describe how the analysis works), we had to reuse some of the internals of the SonaQube Java module.
Here we also say that there are multiple contexts where the plugin runs. One of the contexts is to run the plugin on the server side, which is supposed to be run during CI/CD pipeline, and another context is to run inside developers IDE to provide instant feedback without the need to compile the code.

To create the plugin, we followed the tutorial provided in the SonarQube documentation [**?**]. 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 extension of Sonar Java plugin [**?**], 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 only 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 would be also 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 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 were available during compilation.

What is the goal?
The goal is to build a tool, that can help developers in static analysis of the code. The tool would detect the list of code smells found in the appendix. Previously we mentioned that there are 2 contexts in which the plugin can be run, and in this thesis we will focus only on the server side of the tool. This is because we are interested in analyzing a large corpus of the applications and this best done on the server side, because manually skimming through all of the detected code smells in the IDE is not an option when you have a corpus of 1000 applications.

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 analysis of the large corpus of applications and it makes no sense to do this manually through the IDE.

Thus, we will implement the plugin that detects the code smells described in subsection 2.1 and can be executed inside a SonarQube instance.

What tools did we use?
SonarQube as a platform to run the analysis - provides nice UI for the end users and also very mature tool in the industry. But most importantly, allows us to analyze the applications and controls the whole flow of the analysis (starting the analysis, running the analysis, creating results, uploading them and then displaying them to the user). So we only need to provide our custom rules and a way to load them.
Scala - language that we used to write the rules in. Scala is a programming language that combines both functional and object oriented approaches. So this was a good choice for me, because I wanted to write the code in a functional way but SonarQube is written in Java and API is designed in an object oriented matter. So Scala provided nice interop with Java API because it supports both functional and imperative approaches like described previously.
Sonar Java plugin - used this as a base, because it provides all necessary tools that are required to parse Java sources into the AST and also provides needed utilities during the analysis (detection of cognitive complexity, number of lines of code etc).

In order to implement the plugin, we used SonarQube as the implementation platform. We decided to go with SonarQube not only because it provides a nice user interface to display the end results of the analysis and is a very mature tool in the industry, but also because it controls the flow of the analysis itself (project configuration, starting the analysis, creating the results, serializing and parsing the results, uploading them to the server and displays them to the user) and provides other neat features like user management and ability to check the projects distribution of code smells.

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.

Previously we also mentioned that plugins are loaded into the SonarQube dynamically and that this plugin must be used with `provided` scope when compiling the plugin. This means that only some of the packages are provided during the runtime and not all of the classes provided by the Sonar Java plugin can be used during the runtime. This is an issue, because the plugin itself provides a lot of utilities that could be reused inside the custom rules (for example cognitive complexity counter, lines of code counter). To overcome this limitation, we extracted those utilities into a separate package and included it in the compiled artifact of our plugin, so that we could use those tools during the analysis execution.

Describe how did we test it?
Mostly unit tests, to test the internal architecture of the plugin. In order to check the rules validity, SonarQube provides a framework to analyze the files the same way that the scanner on the server side in production would do.
So, the rules were validated by writing a code snippet that would be represents a given code smell that we want to detect in a given rule. Then, you can describe which line of the code is invalid (e.g. you want to detect classes that contain) a code smell pattern. Then, the code snippet would contain comment in pattern (provide pattern here and also an example).

Once the plugin has been 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.

In order to satisfy the first phase of the testing, we used Scalatest [**?**] 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 form of specification.

For the second phase, SonarQube provides a testing framework that allows to test the rules in a similar fashion 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 test file in figure 1. This code snippet contains code that a person writing 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 testing phase.

```
package com.example.test;

public class DataClass { // Noncompliant {{Refactor this class so it
    includes more than just data}}
        public String field1;
        public int field2;
    }
```

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

3.1 Selected datasets

Describe how we selected the dataset. We might be using the dataset that was used by the authors of another paper, so that we can compare our results to those that they have already provided. Also mention here that some of the projects in our corpus were not using the build system, so we were not able to analyze them. Here we can say that we excluded them because we could build them, but SonarQube itself cannot analyze projects that do not use build systems.

3.2 Methodology

Describe that we want to perform analysis of projects in order to:

- see how code smells that we implemented are distributed inside analyzed applications
- see how our code smells definitions compare to already published results
- see how code smells not yet published by the literature are distributed inside analyzed applications

Here we also need to describe the method that we will use to analyze the applications. For example for a each application:

1. Build the project
2. Analyze the project
3. Extract what code smells were found / how many were found (statistics)

Also for some code smells we need to determine some parameters statistically (using box plot technique). This section would need to describe how we would fint hose parameters statistically.

4 Results

4.1 Developed tools

Here describe plugin for SonarQube. In introduction we mentioned groups that we think the tool might be used for. So here we provide our ideas how each group can be helped with our tool, provide screenshots or other artifacts that might help our points.
Also describe the results of developing the bulk analyzer, provide simple instructions on how to run this tool, provide output from help command, which will show input parameters and basic instruction on how to run.

4.2 Analysis results

Describe the results that we got from project analysis. Say how those compare to already existing results. Distribution of code smells that are not yet published. How many projects in corpus versus how many were actually successfully analyzed. Statistics on code smell distributions inside analyzed applications.

5 Conclusion

Say that we have created a tool and it works on the projects that we checked, but we dont know if it actually helps, since we did not perform any empirical study. Say that there might be some limitations with the dataset that we have selected. Discuss future work.

Appendix

I. Glossary

II. Licence

Non-exclusive licence to reproduce thesis and make thesis public