Maintainability Metrics for Android Applications in Kotlin: An Evaluation of Tools

Lisa-Marie Andrä
FH Campus Vienna
Favoritenstraße 226
1100 Vienna, Austria
+43 1 606 68 77-6600

lisa-marie.andrae@stud.fh-campuswien.ac.at

Bernhard Taufner
FH Campus Vienna
Favoritenstraße 226
1100 Vienna, Austria
+43 1 606 68 77-2134

bernhard.taufner@fh-campuswien.ac.at

Sigrid Schefer-Wenzl
FH Campus Vienna
Favoritenstraße 226
1100 Vienna, Austria
+43 1 606 68 77-2141

sigrid.schefer-wenzl@fh-campuswien.ac.at

Igor Miladinovic
FH Campus Vienna
Favoritenstraße 226
1100 Vienna, Austria
+43 1 606 68 77-2131

igor.miladinovic@fh-campuswien.ac.at

ABSTRACT

Over the last decade, Android has become the largest mobile platform, providing a huge and ever increasing amount of apps in the Google Play store. Since 2017, a growing number of Android apps have been written using the Kotlin programming language. Maintainability is an important attribute of software quality. It determines how well maintenance and code modification can be done. This is particularly important for Android apps that are frequently updated to meet changing user requirements.

Several metrics can provide information about the maintainability of a system. They can be calculated with different analysis tools. This paper analyzes the maintainability of Kotlin applications. In particular, we investigate which metrics are relevant for the maintainability of Kotlin applications and to what extent analysis tools support the calculation of these metrics. Our results show that these tools are very limited with respect to the supported programming languages. Especially for Android applications written in Kotlin, the majority of maintainability metrics cannot be calculated with current tools.

CCS Concepts

- **Software and its engineering**---Software notations and tools--
-Software maintenance tools; 500.
- **Software and its engineering**---Software notations and tools--
-General programming languages---Language types---Object
oriented languages; 500.

Keywords

Android; Clean Code; Code Analysis Tools; Maintainability; Metrics.

This work is licensed under a Creative Commons Attribution International 4.0 License.

ESSE 2020, November 6–8, 2020, Rome, Italy
© 2020 Association for Computing Machinery.

ACM ISBN 978-1-4503-7762-1/20/11...\$15.00

<https://doi.org/10.1145/3393822.3432334>

1. INTRODUCTION

Software engineering consists of several fundamental phases including specification, design, implementation, validation, deployment and evolution. Consequently, software still needs to be maintained in the form of enhancements and modifications after successful deployment. These changes are necessary to correct faults or failures and to ensure the survivability of a system or application. As requirements change over time, it is important that software is easily changeable. The fact that a piece of software is maintainable also implies that the code is easily understandable. Otherwise, it would take more time for (new) developers to maintain the software, which in turn means additional costs.

Software maintainability can be structured in several categories, including analyzability, changeability, stability and testability (see, e.g. [1, 2]). Analyzability measures the capability of specifying faults or failures in software. Changeability is defined as the ability for changes in software products. Stability is about avoiding unexpected effects of changes made to the software product. Testability refers to the ability of testing and validating the adapted software product. Improving maintainability leads to a reduction of the time needed to maintain (modify) software for improving performance, correcting faults, changing attributes or adjusting to environment changes [1]. There are different possibilities and approaches to make a piece of software maintainable. These include various coding conventions, which should be followed.

To get information about various code properties, including maintainability metrics, static analysis tools can be used. They often work directly via the IDE. Such tools can, for example, calculate certain metrics or check if certain rules are followed for supported programming languages.

The open source programming language Kotlin, which was developed in 2011, can be applied for different cases, including Android development. Google reports that more than 60% of the world's top 1000 apps in the Play Store use Kotlin. JetBrains says that Kotlin code is written faster and it is more understandable for others. In addition, there is an interoperability with Java. According

to JetBrains, Kotlin is easy to learn. The code is safe, which is facilitated by the error detecting Kotlin compiler. [2],[3]

In this paper, we focus on the topic of maintainability, specifically on the maintainability of Android applications developed with Kotlin. We discuss what maintainability is in general, why it is important and how it can be improved. In this context, concepts of clean coding and different metrics are presented. We have evaluated several available tools for maintainability metrics calculation. The outcome was that none of them is suitable for Kotlin applications.

This paper is structured as follows. In Section 2, some metrics and clean code conventions are described including an analysis of their relationship. In Section 3, the approach of finding a suitable tool for calculating Kotlin code metrics is presented. Section 4 summarizes the main findings and gives an outlook on future work.

2. MAINTAINABILITY

2.1 Metrics

Whereas maintenance refers to the process of modifying a piece of software, maintainability reflects the ease with which a piece of software can be modified [4]. Maintainability is essential as a quality attribute but not easily achieved [1]. Maintainability can be cost-intensive, yet it is a crucial factor in determining whether a system can be sustained over a long time. Software must be maintainable in order to support future changes, which are often made by programmers who did not develop it initially. The effort and the costs to maintain a system are related to its quality [5]. They decrease with higher code understandability and quality [6].

Different metrics can be used to determine and measure maintainability of software; some are particularly important for Android applications[1,7]. Table 1 shows the most relevant maintainability metrics for Android and their descriptions.

Table 1. Maintainability metrics

Metric	Description
NOM	Number of Methods
NOAT	Total Number of Attributes
COH or LCOM	Lack of Cohesion
CSO	Class Size (Operations)
Query	Number of Queries
WMC	Weighted Method per Class
CLOC	Comments Lines of Code
Command	Number of Commands
CSA	Class Size (Attributes)
SLOC	Source Lines of Code

Many of the metrics mentioned in Table 1 are useful to obtain information about the scope of an application. These metrics are e.g. NOM, NOAT, Query, Command and SLOC. If there are two or more versions of an application, these metrics can also be used to find out to what extent the application has grown from one version to another. SLOC can be calculated for a whole project or individual parts of an application, which facilitates the determination of file sizes, classes or even individual functions. In

general, individual parts of an application are easier to understand if they consist of fewer lines of code.

In order to find out the size of individual classes, CSO and CSA are helpful. Generally, a smaller class is better than a larger one (see Section 2.2.3). Therefore, it is desirable to measure low values for CSO and CSA.

COH, sometimes also abbreviated LCOM, describes how strongly the methods within a class are related. Thus, the lower the value, the better.

WMC stands for the complexities of all methods of a class added together. The higher the value, the worse for understandability and maintainability of a piece of software.

2.2 Clean Code Conventions

Clean code conventions aim to improve maintainability. However, there is not only one exact definition of clean code. Robert C. Martin states in [8] different interpretations of clean code from several well-known experts in programming technology. Let us consider two prominent examples. Bjarne Stroustrup, the inventor of C++ said the following: *"I like my code to be elegant and efficient. The logic should be straightforward to make it hard for bugs to hide, the dependencies minimal to ease maintenance, error handling complete according to an articulated strategy, and performance close to optimal so as not to tempt people to make the code messy with unprincipled optimizations. Clean code does one thing well"* [8]. The important parts are elegant and efficient code, straightforward logic, minimal dependencies and optimal performance. Another interpretation of clean code is by Dave Thomas, founder of OTI, who said that *"Clean code can be read, and enhanced by a developer other than its original author"* [8]. These two interpretations already indicate the connection between clean code and maintainability. The next subsections concentrate on several parts of programming to have clean code. They show that the main goal of clean code is readability and understandability – both are highly relevant for maintainability.

2.2.1 Conventions for functions

There are two rules concerning functions. The first rule is that functions have to be as **small** as possible [9]. Robert C. Martin suggests in [8] that a function should not be longer than 20 lines.

The second key rule is that *"Functions should do **one thing**. They should do it well. They should do it only"* [8]. The problem is to find out what one thing is. A first hint is the name of the function. If it is hard to find a descriptive name, which truly reveals the intent of the function and not only a part of it, this could be an indication that the function does more than one thing. The next indication of fulfilling the one thing rule is the **separation of command and query**, since the task of a specific function is either to execute or to answer something, but never both. For example, it is better for **try/catch blocks** to extract the content of try and catch into separate functions so that the actual functionality is not mixed up with **error handling**. In addition, error handling itself is one thing and should be in a separate function.

2.2.2 Conventions for comments

In general, functionality should be described with the code and not with comments whenever possible. The reason for this is that comments are not always changed when modifying the corresponding code and they might not be correct. However, there are some cases where comments need to be used, including legal comments, comments for clarification and warning of consequences, and TODO comments. [8]

2.2.3 Conventions for classes

The first rule, as with functions, is that classes should be as **small** as possible [9]. In order to know how small, responsibilities are counted as a guideline. The **Single Responsibility Principle** (SRP) should be practiced. Consequently, a class should only have one responsibility and only one reason for change [10]. Naming the class can help to find out whether it has too many responsibilities and is therefore too large. It should be possible to describe a class with 25 words, without the use of *if*, *and*, *or* or *but* [8].

2.3 Mapping between Clean Code Conventions and Metrics

In this section, we consider which maintainability metrics are relevant for each clean code convention. This should help to measure to what extent clean code conventions have been used in an application. Table 2 maps the clean code conventions from Section 2.2 and the corresponding metrics from Section 2.1.

Table 2. Relationship between Clean Code conventions and metrics

Clean Code Convention	Mapping Metric(s)	Effectuated Metric	Considerations
Classes should be as small as possible and adhere to the SRP	CSO and CSA	COH	Methods should follow the principle
Methods should be small and not complex	SLOC per method	WMC	
Separation of Command and Query	Command, Query and NOM	Not really a metric, but a calculation: $ NOM - (Command + Query) $	Calculating the total number of Methods (NOM), the number of commands and the number of queries. If NOM is not equal to commands + queries, there is no clear separation between commands and queries.
Comments should only be used for specific reasons	CLOC	Comment ratio	It must be taken into account if there are whole classes or functions that are commented out. If this is the case, these have to be deleted for the measurement. If the whole

			content of a file is commented out, this file should be excluded from the measurement.
--	--	--	--

3. ANALYZING AN ANDROID APPLICATION WRITTEN IN KOTLIN

To calculate all chosen metrics for an Android application written in Kotlin, we aimed to use the Android Studio plugin *MetricsReloaded* [11]. For this purpose, a new metrics profile was created with the following preferences:

Project Metrics:

Comment lines of code, Comment ratio, Lines of Java, Lines of Kotlin, Lines of XML, Non-comment lines of code, Number of Java files, Number of Kotlin files, Number of XML files, Number of classes and Number of methods

Class Metrics:

Class size (attributes), Class size (operations), Class size (operations + attributes), Lack of Cohesion of Methods, Non-comment lines of code, Number of commands and Number of queries

Method Metrics:

Non-comment lines of code

The numbers and lines of the different file types (Java, Kotlin and XML) serve to provide an overview of the calculations. In fact, there should be mainly Kotlin files (as the program logic is mainly written in Kotlin), a few Java files and no XML files (since they should be excluded from the measurement). To verify this, the mentioned metrics are quite useful.

After the metrics profile was completed, the measurements were started. At first, the metrics were calculated for the whole project. This was too much data with foreign code and resource files included. Therefore, the decision was made to apply the plugin to a specific package of the project, which contained all features of the applications. In this way, foreign code and any resource files were excluded from the calculation so that the amount of results should be processible.

After the limitation to the specific package, the metric results were very poor—only project level metrics were calculated but no class or method level metrics. We found out that the *MetricsReloaded* plugin does not have the ability to calculate class or method metrics for Kotlin source code. It can calculate the number of Kotlin files and the Lines of Kotlin for the whole project and for modules, but no other metrics and not for packages, classes or methods.

This is also evident when examining the source code of *MetricsReloaded*, which is provided on GitHub and linked on the official plugin page of JetBrains¹: Listing 1 shows a snippet from *KotlinMetricProvider.java* while Listing 2 displays a snippet from *JavaMetricProvider.java*. In comparison, the snippets show that only four metrics can be calculated for Kotlin, which refer to a

¹<https://github.com/BasLeijdekkers/MetricsReloaded/tree/master/stockmetrics/src/com/sixrr/stockmetrics>

project or a module. For Java, however, metrics can be calculated for classes, interfaces, methods, modules, packages and projects. Which metrics can be seen on GitHub?

```
public class KotlinMetricProvider implements MetricProvider {

    @NotNull
    @Override
    public List<Metric> getMetrics() {
        final List<Metric> metrics = new ArrayList<>(4);
        metrics.add(new NumKotlinFilesProjectMetric());
        metrics.add(new NumKotlinFilesModuleMetric());
        metrics.add(new LinesOfKotlinProjectMetric());
        metrics.add(new LinesOfKotlinModuleMetric());
        return metrics;
    }
}
```

Listing 1. Snippet from *KotlinMetricProvider.java*.

```
public class JavaMetricProvider implements MetricProvider {

    @NotNull
    @Override
    public List<Metric> getMetrics() {
        final List<Metric> metrics = new ArrayList<>(255);
        initializeClassMetrics(metrics);
        initializeInterfaceMetrics(metrics);
        initializeMethodMetrics(metrics);
        initializeModuleMetrics(metrics);
        initializePackageMetrics(metrics);
        initializeProjectMetrics(metrics);
        return metrics;
    }
}
```

Listing 2. Snippet from *JavaMetricProvider.java*.

Having discovered that *MetricsReloaded* cannot calculate the required metrics for Kotlin, we started the search for another plugin or tool that can calculate all the required metrics and supports Kotlin. Most plugins or tools do not state clearly which metrics and which languages are supported. Therefore, each plugin had to be tested.

The first alternative plugin tested was *CodeMR* [12]. It supports Kotlin, thus fulfilling one criterion. The plugin was applied to the features package, which immediately threw an error, because the free version is limited to a specific number of classes. The plugin generates an HTML report, including some graphics. Unfortunately, the required metrics could not be calculated. For example, it is not possible to calculate CSO, CSA, SLOC per method, Command and Query.

The next alternative considered was *SonarQube* [13]. For this tool a server is needed and the setup is more complex. However, the necessary metrics are not supported so that this tool failed as well.

Among many other plugins tested, were *Code Cleaner with Code Climate CLI* [14], *detect* [15], *PMDPlugin* [16] and *CodeMetrics* [17]. None of them fulfilled our requirement, i.e. calculating all the required metrics for Kotlin applications. Some of them are not able to calculate any metrics at all.

After all available plugins for IntelliJ IDEs had been tested, tools for other IDEs were investigated. In this context, the extensions for Visual Studio Code were examined, but also without success.

Following this, no alternatives could be found, and the search was aborted with the result that there is no tool or plugin for Kotlin that can make a sufficient statement about the maintainability of a Kotlin program based on metrics related to the clean code conventions.

4. CONCLUSION

Maintainability depends on whether certain coding conventions have been used during the development of a software. This also has a strong influence on the readability of a code and it determines whether and how well a software can be maintained and extended. If the code is difficult to maintain and expand, this is associated with high follow-up costs.

The practical investigations of the various tools and plugins for calculating certain metrics have shown that for Java code there is a very good plugin for IntelliJ based IDEs. This plugin, *MetricsReloaded*, offers the possibility to calculate many metrics. However, this plugin has only very limited support for Kotlin; a suitable alternative could not be found. It can be concluded that there is currently no suitable tool or plugin for calculating maintainability metrics for Kotlin applications, which are defined by the clean code conventions.

These plugins are heavily dependent on individual developers. The tools of the individual developers are often poorly documented. It is rarely stated which metrics and programming languages are supported. This information has to be obtained through practical tests, which costs time and effort.

In our future work we plan to design and develop a tool or extend an existing tool to calculate the required metrics and determinants as to what extent the clean code conventions are followed. This tool should perform some of the calculations on the fly so that high quality code is generated from the start, saving time and costs. Furthermore, as soon as there is the possibility to calculate the various metrics, mappings between clean code conventions and metrics will be evaluated and enhanced to improve the estimation of maintainability of an application.

5. REFERENCES

- [1] Ahmad A. Saifan and Areej Al-Rabadi. Evaluating maintainability of android applications. 2017.
- [2] Kotlin – Using Kotlin for Android Development <https://kotlinlang.org/docs/reference/android-overview.html> [Online; accessed 18-August-2020].
- [3] V. Oliveira, Leopoldo Teixeira, and Felipe Ebert. On the Adoption of Kotlin on Android Development: A Triangulation Study. IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER), London, ON, Canada, Canada, Feb. 2020.
- [4] IEEE standard glossary of software engineering terminology. IEEE Std 610.12-1990, pages 1–84, Dec 1990.
- [5] I. Heitlager, T. Kuipers, and J. Visser. A practical model for measuring maintainability. In 6th International Conference on the Quality of Information and Communications Technology (QUATIC 2007), pages 30–39, 2007.
- [6] S. Velioglu and Y. E. Selçuk. An automated code smell and anti-pattern detection approach. In 2017 IEEE 15th International Conference on Software Engineering Research, Management and Applications (SERA), pages 271–275, June 2017.

- [7] A. Kaur, K. Kaur, and K. Pathak. A proposed new model for maintainability index of open source software. In Proceedings of 3rd International Conference on Reliability, Infocom Technologies and Optimization, pages 1–6, Oct 2014.
- [8] Robert C. Martin. Clean Code. A Handbook of Agile Software Craftsmanship. Pearson Education, 2009.
- [9] Henning Grimeland Koller. Effects of clean code on understandability: An experiment and analysis. Master’s thesis, 2016.
- [10] Volker Dusch. Save time by applying clean code principles. 06 2013.
- [11] MetricsReloaded – plugin for IntelliJ IDEs | JetBrains. <https://plugins.jetbrains.com/plugin/93-metricsreloaded> [Online; accessed 18-August-2020].
- [12] CodeMR. <https://www.codemr.co.uk/> [Online; accessed 18-August-2020].
- [13] SonarQube. <https://www.sonarqube.org/> [Online; accessed 18-August-2020].
- [14] Code Cleaner with Code Climate CLI – plugin for IntelliJ IDEs | JetBrains. <https://plugins.jetbrains.com/plugin/13306-code-cleaner-with-code-climate-cli> [Online; accessed 18-August-2020].
- [15] detekt - plugin for IntelliJ IDEs | JetBrains. <https://plugins.jetbrains.com/plugin/10761-detekt> [Online; accessed 18-August-2020].
- [16] PMDPlugin – IntelliJ IDEs | JetBrains. <https://plugins.jetbrains.com/plugin/1137-pmdplugin> [Online; accessed 18-August-2020].
- [17] CodeMetrics - plugin for IntelliJ IDEs | JetBrains. <https://plugins.jetbrains.com/plugin/12159-codemetrics> [Online; accessed 18-August-2020].