

# Small Scale Analysis of Source Code Quality with regard to Native Android Mobile Applications

MARTIN GJOSHEVSKI AND TINA SCHWEIGHOFER, University of Maribor

---

The popularity of smart phones and mobile applications is growing every day. Every day, new or updated mobile applications are being submitted to different mobile stores. The rapidly increasing number of mobile applications has led to an increase in the interest in overall source code quality. Therefore, we will present a case study, where we analyzed different open source Android mobile applications from different domains and different sizes. They were analyzed using the SonarQube platform, based on the SQALE method. We were aiming to research the overall code quality, the connection between lines of code and technical depth and the most common issues facing mobile applications. The results show that the majority of applications tend to have similar code issues and potential difficulties when it comes to maintenance or updates.

General Terms: Mobile application testing

Additional Key Words and Phrases: source code quality, analysis, technical depth, SQALE

---

## 1. INTRODUCTION

In light of the popularity that smart phones and mobile applications have achieved, and intrigued by the constant increase in the number of mobile applications submitted to mobile stores (Statista, 2015), we raised the question of whether this trend has an impact on the overall quality of source code. We were concerned that many mobile application developers may have been focused mostly on fast builds and releases and neglected testing and good design practices, and that this could lead to further difficulties down the road with regard to maintenance and the long-term success of the applications.

Two separate studies in the field of source quality in mobile applications (Syer, Nagappan, Adams, & Hassan, 2014) and (Pocatilu, 2006) have verified that using classical object oriented metrics and approaches for measuring the code quality of desktop and web applications can be used to measure the source code quality of mobile phone applications. A study performed at the University of Maribor (Jošt, Huber, & Hericko, 2013), dealt with a similar issue: two hypothesis were tested, of which the first -- which claimed that using object oriented metrics for the analysis of mobile application code quality does not deter from their usage for the source code quality of desktop applications -- was confirmed. However, the second hypothesis, which claimed that results from analyzing source code quality of equivalent mobile applications developed for different platforms would be undifferentiated, was rejected. The previously mentioned study (Syer, Nagappan, Adams, & Hassan, 2014), has also proven that classical relations between metrics, like “*high coupling low cohesion*,” which are eligible for the source code quality of desktop and web applications and are eligible for mobile applications as well. Their study also found that the claim that “*more code, less quality*” was true in 3 out of 5 applications.

## 2. BACKGROUND

Static code analysis is a technique for the evaluation of code quality that is performed by analyzing the source code or the binary code, with no need of actually running the code (Michael & Laurie, 2007). The advantage of using tools for static code analysis has been recognized by numerous companies and according to the VDC Research Group (Girad & Rommel, 2013), the market share is expected to grow, on

---

Authors' addresses: M. Gjoshevski, Faculty of Electrical Engineering and Computer Science, University of Maribor, Smetanova 17, 2000 Maribor, Slovenia; email: martin.gjoshevski@student.um.si; T. Schweighofer, Faculty of Electrical Engineering and Computer Science, University of Maribor, Smetanova 17, 2000 Maribor, Slovenia; email: tina.schweighofer@um.si.

Copyright © by the paper's authors. Copying permitted only for private and academic purpose.

In: Z. Budimac, M. Heričko (eds.): Proceedings of the 4th Workshop of Software Quality, Analysis, Monitoring, Improvement, and Applications (SQAMIA 2015), Maribor, Slovenia, 8.-10.6.2015. Also published online by CEUR Workshop Proceedings (CEUR-WS.org, ISSN 1613-0073)

average, by 15% annually. This amount of growth and concomitant demand has led to well-developed tools for static code analysis that have become powerful and feasible.

One of the key aspects that defines usable and good tools for static code analysis is that the representation of the results has to be easy to read and understand by the developers (Gomes, Morgado, Gomes, & Moreira, 2015). One way of representing the result is by using technical debt(TD) which is a metaphor reflecting technical compromises that can yield short-term benefits but may hurt the long-term health of a software system(Li, Avgeriou, & Liang, 2015).

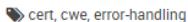
## 2.1 Standards, methods and tools

In this field, there are some relevant standards and methods that support the evaluation of software code quality. ISO/IEC 9126 and its successor ISO/IEC 25010:2011 directly address the issue of system and software quality. We have used ISO/IEC 9126 as a quality model for our analysis, which was performed with SonarQube, based on the SQALE method (Letouzey J.-L. I., 2012).

Software quality assessment, based on life cycle expectations (SQALE), is a method developed by DNV ITGS France that is platform independent and is applicable to any kind of development methodology. The SQALE method estimates the technical debt, which can be presented with custom units, such as cost in money or the time required to remove all the found issues. One of the main four concepts is the quality model, which is organized in three levels. The first and the second level present characteristics such as testability, maintainability etc. and their related sub-characteristics. The third level defines the non-functional requirements or simply the rules, which are usually dependent on the programming language used. The analysis model defines the time required for fixing an issue determined by a rule (Letouzey J.-L. , 2012). Figure 1 gives an example of a defined rule in the SonarQube platform.

---

**Throwable and Error should not be caught**  
 squid:S1181

 Blocker  Available Since August 30 2013 SonarQube (Java)

Reliability > Exception handling Constant/issue 20min

Throwable is the superclass of all errors and exceptions in Java.

Error is the superclass of all errors, which are not meant to be caught by applications.

Catching either Throwable or Error will also catch OutOfMemoryError and InternalError, from which an application should not attempt to recover.

Only Exception and its subclasses should be caught.

**Noncompliant Code Example**

```
try { /* ... */ } catch (Throwable t) { /* ... */ }
try { /* ... */ } catch (Error e) { /* ... */ }
```

---

**Compliant Solution**

```
try { /* ... */ } catch (Exception e) { /* ... */ }
try { /* ... */ } catch (RuntimeException e) { /* ... */ }
try { /* ... */ } catch (MyException e) { /* ... */ }
```

---

See

- [MITRE, CWE-396 - Declaration of Catch for Generic Exception](#)
- [CERT, ERR07-J - Do not throw RuntimeException, Exception, or Throwable](#)

---

Fig. 1. Rule definition in SonarQube

SonarQube is an open source platform that provides a central place for managing source code quality for single or multiple projects. Its extension mechanism allows for the addition of new uncovered

programming languages, tools and features that in conjunction with the default features and tools, such as visual reporting and across projects, time machine etc. provides a genuinely flexible and powerful tool for source code quality analysis (S.A, 2015).

### 3. CASE STUDY – ANALYSIS OF SOURCE CODE QUALITY

#### 3.1 Case study

A case study is a suitable type of research methodology in the field of software engineering, because it studies contemporary phenomena in its natural context. Primarily, the case study was used for exploratory purposes, but nowadays it is also used for descriptive purposes. Case studies are by definition conducted in a real world environment, and thus have a high degree of realism (Runeson, Hostl 2008).

A case study usually combines five major process steps. The first is case study design, where objectives are defined and a case study is planned. This phase is followed by preparation for data collection, which includes definitions of procedures and protocols. The third step is collecting the evidence, and this is followed by an analysis of the collected data. The final step is the reporting phase of the case study (Runeson & Höst, 2009).

#### 3.2 Objectives of the analysis

The aim of our study was a comparison of the results obtained using the SQALE method with the help of the SonarQube platform, where we analyzed open source mobile applications. We addressed the following research questions:

- RQ1: Is there a major deviation between the overall qualities of the mobile applications source code?
- RQ2: Do lines of code have a direct influence on the technical depth of the mobile applications?
- RQ3: What are the most common issues that occur in the analyzed mobile applications?

#### 3.3 Selection of mobile applications – preparation phase

Our analysis was based on open source mobile applications, so we looked for applications in the F-Droid repository (F-Droid, 2015). F-Droid is a catalogue of open source applications for the Android platform. We selected 30 applications randomly. With the selection we attempted to cover and select applications that cover a vast range of characteristics. Our selection contained applications that belonged to different domain categories, were developed by different developers and differentiated between the lines of code. The complete list of analyzed applications can be seen in Table 1.

Table 1. Selection of applications

<b>Id</b>	<b>Application name</b>	<b>https://github.com</b>	<b>Commit</b>	<b>LOCs</b>
1	K-9 Mail	/k9mail/k-9	c36d2d7a5e5b27f7cfd6892109f9ceb5e49006df	55,603
2	Book Catalogue	/eleybourn/Book-Catalogue	31706472e49500224fd1ed5c9dd15fd253a82081	39,670
3	ChatSecureAndroid	/guardianproject/ ChatSecureAndroid	2a6b6c06fda94f588ad98c97896d0049dd1c2484	36,178
4	TextSecure	/WhisperSystems/TextSecure/	27b5bf54cc2ddd809eedcbb627dbda30d77b2eae	34,429
5	OpenConnect VPN client	/cernekee/ics-openconnect	c8dac1eae12793af78f4bbdc785ebd44e5c016d	25,731
6	AndBible	/mjdenham/and-bible	841eb835d72e2d3488b80bcef754132fa56d6c00	23,478
7	OwnCloud	/owncloud/android	20cd00b22c86ef0ab03ae0b8cd8eedd63844981f	17,660
8	A Comic Viewer	robotmedia/droid-comic-viewer	e9f98610962288b53379f1ac5f889d222f6463e5	16,195
9	Ushahidi	ushahidi/Ushahidi_Android	8c02fecb7dd5b52ef116520c24d4e509e3676c4e	15,950
10	AFWall	/ukanth/afwall	261a7b5919af4459d38a9a6b229f7e22b500c4de	14,006
11	Geopaparazzi	/geopaparazzi/geopaparazzi	0b0e8b7d4285e06e3d0c4a05d804e3328106b7ae	10,892
12	Dudo	/goldenXcode/dudo	9a52e28dfc48e5eb9fc85433072f51203612856	10,643
13	Transdroid	/erickok/transdroid	fef815720e3f46dccc8bb8692f944ba510052ee1	10,430
14	Car Report	*Bitbucket/frigus02/car-report/	b14724ffbe9f73c0654a06c592fbf7e0189ea87e	9411
15	Prey	prey/prey-android-client	9b27da9fb213871ee70744aa493fc5973dc08924	8,788
16	BART Runner	dougkeen/BartRunnerAndroid	5c1ae735e1a301a7141b6e46a26b5864c934778b	5,918
17	oandbackup	/jensstein/oandbackup	a15cbb1c5d91057f9f056f2a71889dc91cabfbe3	5,901

18	LucidBrowsere	/powerpoint45/Lucid-Browser	6d7f86a4f64358135852811e6d9bd39cc5144cb1	5,035
19	CampFahrplan	/tuxmobil/CampFahrplan	a69af4e1f2f147307ae051e3b55649f5345971c8	4,627
20	Aarrddict	/aarrddict/android	17f8892d35483ea6bcdd8dfafe6ef4d5cfbad566	4,008
21	retroarch	/libretro/RetroArch	7e903c248fc3e561133fa9c715db709c766c5984	3,636
22	swftp	/softlayer/swftp	46eabe8bbb06dcf54d5901595eb1b4286afbe5d4	3,098
23	runningLog	/gjoshevski/runningLog	553e9066c84d5e81eb68e176148ec8ea109836ee	3,058
24	OpenDocument	/pear/OpenDocument	8c50b445541a1b3debbd02153de3b853b7e0de8c	2,891
25	bysykklist	/rogerkk/bysykklist-oslo	f714c8e533c099da8d5f35c35e442e428b516cfd	2,006
26	AsciiCam	/dozingcat/AsciiCam	340349e80b6e7cb7f4bb24f164b2165c29bd6060	1,963
27	Wifikeyboard	/darth10/wifikeyboard	e99ee117a00bc9cfd82cc6593b0dce690bcae27	1,909
28	TuxRider	/drodin/TuxRider	36220e5c2f5404bd453f8cb621a1646dd8cf20a4	1,535
29	Presentation	/feelinglucky/Presentation	ef5cc53210283eebad7f2d0bcbe6f6e014b7be17	1,375
30	Submarine	/niparasc/papanikolis-submarine	06140eb34e69a28094628a7f0ac0ff0b01bf2ed3	556

### 3.4 Setup of the SonarQube quality profile – preparation phase

Our quality profile, in accordance with the SQALE quality model, was in compliance with all Android Lint rules (Lint, 2015). Android Lint is the official tool for validating the code and configuration of android projects. It is usually part of the IDEs that have capabilities for Android development (Eclipse, IntelliJ). Our rules table consisted of 140 lint rules. In addition to the Android Lint rules, we also included the default SonarQube rules created for Java development, known as *Sonar way* (SonarQube, 2015). We have included this set of rules because all of the applications were native applications, written in Java. The *Sonar Way* quality profile contains a total of 197 rules. In the end, we had a complete set of 237 rules. The rules were categorized in line with the ISO/IEC 9126 standard. Table 2 displays the mode used.

Table 2. Quality model

<i>Portability</i>	Compiler
<i>Maintainability</i>	Understandability Readability
<i>Security</i>	API abuse Errors Input validation and representation Security features
<i>Efficiency</i>	Memory use Processor use
<i>Changeability</i>	Architecture Data Logic
<i>Reliability</i>	Architecture Data Exception handling Instruction Logic Synchronization Unit tests coverage
<i>Testability</i>	Unit level

## 4. ANALYSIS OF COLLECTED DATA

After an analysis, the collected data was graded via the SQALE method. The results are presented in Figure 2. As can be seen, the applications are aligned based on lines of code (LOC). The overall result reveals technical depth and is presented in the number of days that are needed in order to remove all of the technical depth. We can see that the applications with the higher number of LOC have more technical depth, and that applications with less LOC effectively do not have technical depth. But in between, by observation, we cannot find a rule that applications with more lines of code have more technical depth.

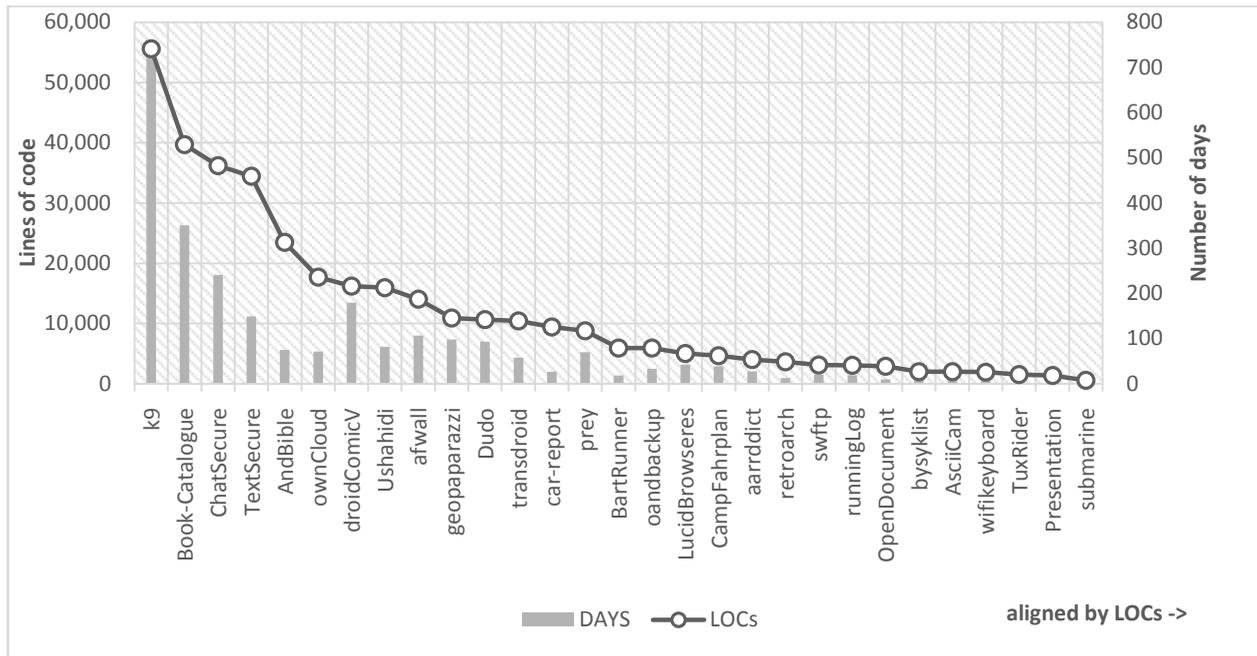


Fig. 2. Initial results

In order to find an answer, supported by data, that would explain the correlation between LOC and technical depth, we presented a new variable. Driven by the assumption that technical depth is correlated with LOC, we calculated an index for each application. The index shows the time required to remove the technical debt for 1,000 lines of code. Our indexes are calculated with the formula presented in Figure 3.

$$i_x = \frac{h_x}{l_x} \times 1000$$

Fig. 3. Normalization formula

Where:

$i_x$  – is the index that represents the time required to eliminate the technical debt for 1000 lines of code.

$h_x$  – is the time required to eliminate the technical debt in the application X.

$l_x$  – is the total number of lines of code (LOC) for the application X.

After applying the formula we get the results presented in Figure 4. The graph shows mobile applications aligned by LOC and time (in days) required to eliminate the technical debt for 1000 lines of code. From the obtained results, based on the data and the applied formula, we were able to see that 13.4% of mobile applications have reached a score that we can categorize as weak and was greater than the upper control limit, which in our case was 9.73 days. On the contrary, 16.6% of the applications received a score that was better than the lower control limit, which was 2.95 days. The average score was 6.79 days, which reveals that the applications had a fair share that can be improved. The upper control limit, lower control limit and mean were calculated based on the results obtained from the analysis. As can we see from the chart, both applications with large LOC and applications with small LOC scored similar scores, so we can conclude, that lines of code do not have a linear correlation with technical depth.

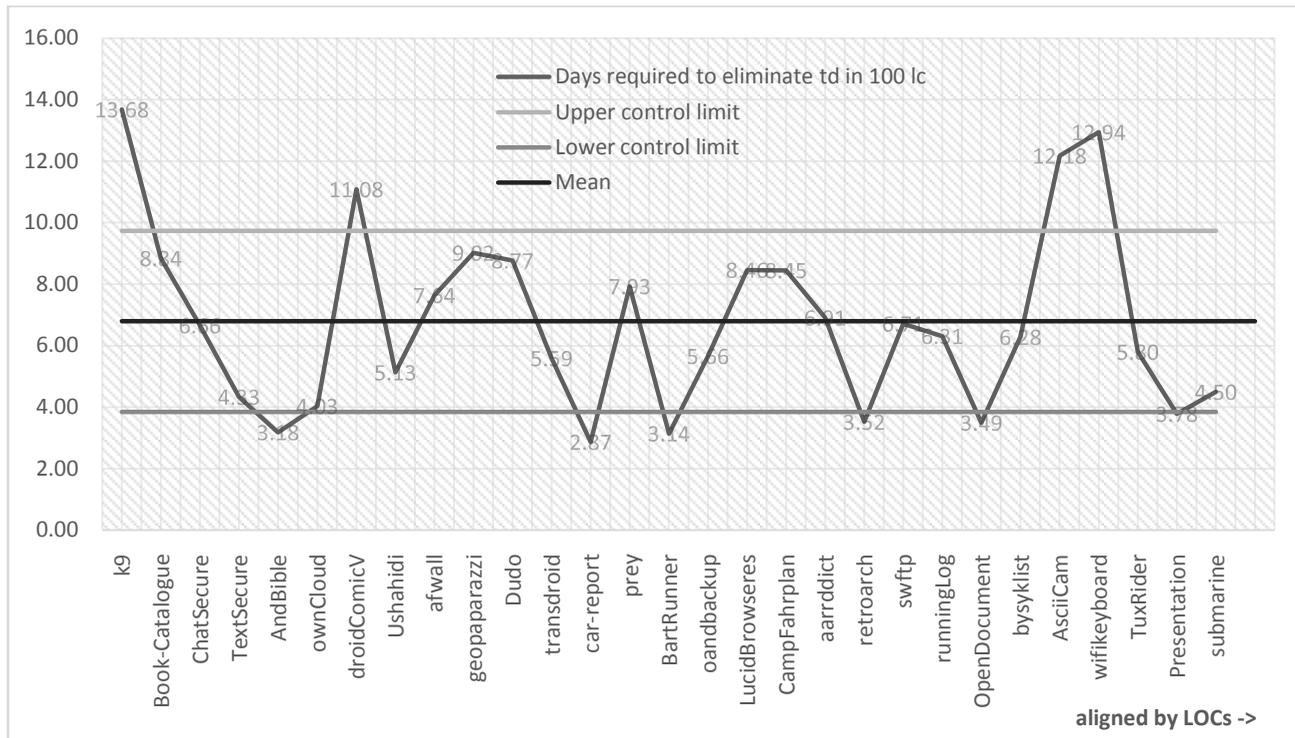


Fig. 4. Control chart

#### 4.1 Problematic code

We also searched for the most common issues that occur in analyzed mobile applications. The detected issues according to our quality profile, based on ISO/IEC 9126, were categorized. Shares were divided among the categories maintainability, changeability, reliability, security, portability, efficiency and testability. The data revealed that the most critical categories were maintainability and changeability, and roughly 91% of all detected issues were categorized in one of these categories. In addition to the mentioned categories, reliability also took a significant share. All shares of the detected issues are presented in the pie chart in Figure 5.

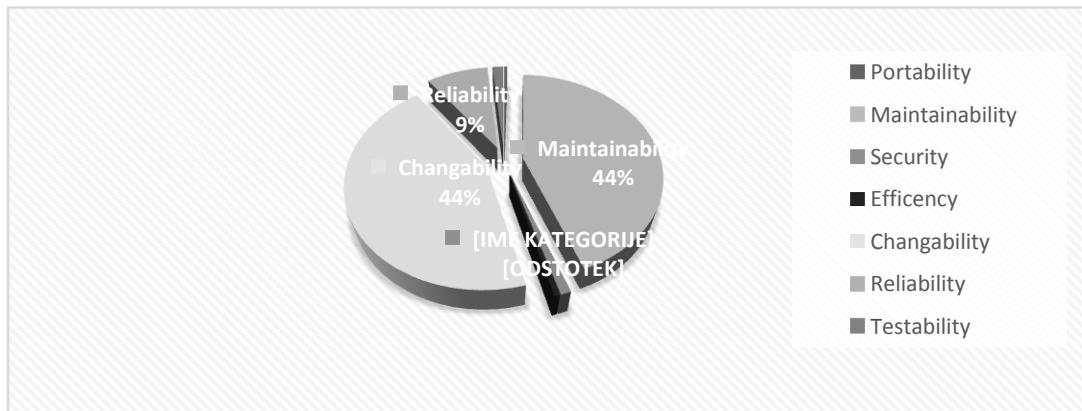


Fig. 5. Code issues by characteristics

## 5. CASE STUDY RESULTS

**RQ1.** The first question was aimed at finding if a major deviation occurred between the overall results of the analyzed mobile applications source code. As we can see in Figure 4, the score of 70% of the applications is within the control boundaries and lines of code do not have a direct impact on the overall score. This roughly gives an answer to our question. Another indicator that can support the claim that no major deviation occurs between the overall results are the results from Figure 5, which clearly show that the weakness detected throughout all of the applications belong to the same categories. We believe that this similarity could be due to the fact that most of the developers use IDEs that have code suggestions and validation tools, such as Android Lint, included in their default configuration.

**RQ2.** The research question was aimed at finding if lines of code in mobile applications have any significant influence on technical depth. We analyzed the applications and presented the technical depth of each mobile application in days. The data was then normalized using the presented formula in Figure 2. We took into consideration that the indexes were within tolerable boundaries and that in most cases the applications had similar code quality. Based on this, the graph in Figure 4 shows that there is no visible correlation between lines of code and technical depth in the analyzed mobile applications.

**RQ3.** The last question was aimed at finding the most common issues that occurred in analyzed mobile applications. From the 237 rules that we analyzed with our code, we detected only 79. This violations actually represent the 3rd level of the SQALE quality model. In order to get a clear answer to the presented research question, we represented the data with a bar chart. The most commonly detected issue, with a nearly 25% occurrence, was the visibility modifier, followed by other issues such as: avoidance of commented out-lines of code, magic number should not be used, unused private methods and others. All issues, and their respective shares, are presented in Figure 5.

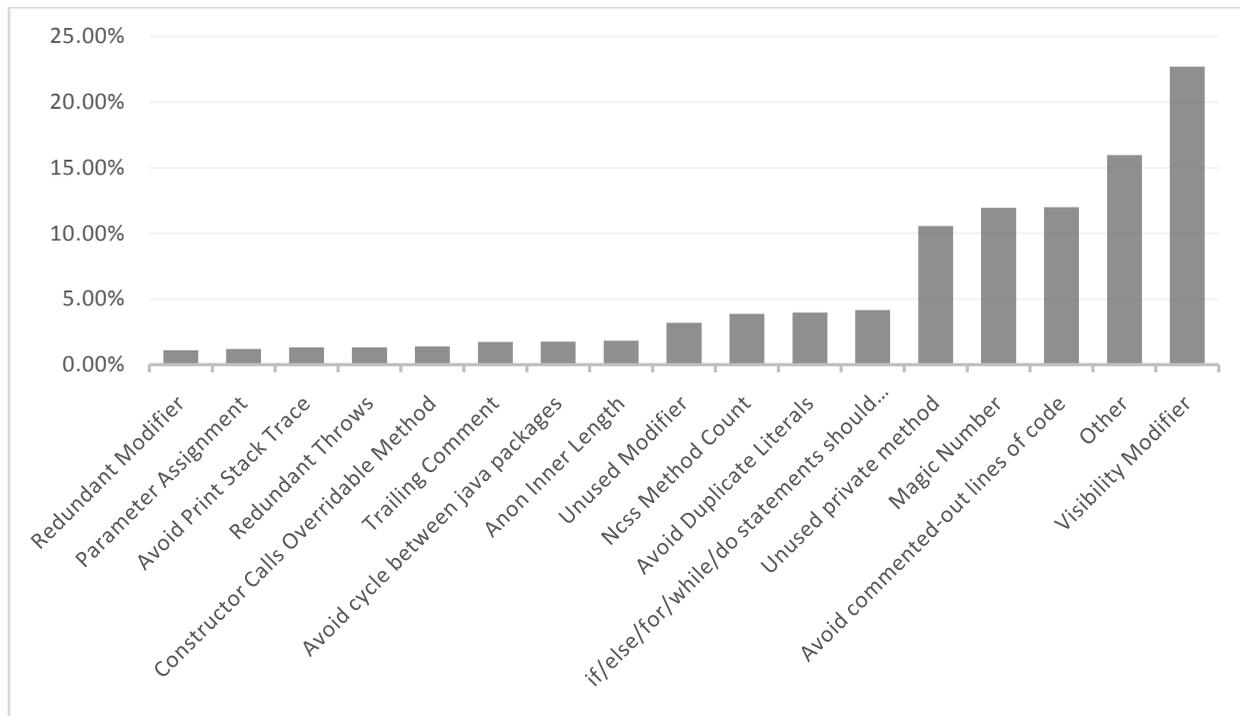


Fig. 5. Most common issues

## 6. DISCUSSION

Even though we were concerned that code quality would suffer due to the rapid development and race for a share of the growing market, our analysis has given us a results that were not as critical as we expected. In our opinion, there is lot of room for improvement, but the current state of things is bearable and in our subjective opinion is only going to get better, mostly because trends show that the industry is more and more concerned with code quality. This is leading to better quality in the product itself.

This analysis has pointed to a few issues that constantly occur in every project. By dealing with this handful of problems, we could significantly decrease the number of detected issues and can lower the technical debt by *more than 40%*.

We would also like to share our experience of working with the open source platform SonarQube, and would like to stress that it provided a pleasant experience. The platform itself is a powerful tool for managing code quality and in our opinion this tool or similar tools (List of tools for static code analysis, 2015) should become standard practice for managing code quality, which will reflect on the overall quality of the product.

## REFERENCES

- F-Droid. (2015, 3 3). Retrieved from Free and Open Source Android App Repository: <https://f-droid.org/>
- Girad, A., & Rommel, C. (2013). The Global Market for Automated Testing and Verification Tools. VDC Group.
- Gomes, I., Morgado, P., Gomes, T., & Moreira, R. (2015, 2 23). An overview on the Static Code Analysis approach in. Retrieved from <http://paginas.fe.up.pt/>: <http://paginas.fe.up.pt/~ei05021/TQSO%20-%20An%20overview%20on%20the%20Static%20Code%20Analysis%20approach%20in%20Software%20Development.pdf>
- Jošt, G., Huber, J., & Hericko, M. (2013). Using Object Oriented Software Metrics for Mobile Application Development. SQAMIA.
- Letouzey, J.-L. (2012, January 27). The SQALE Method - Definition Document - V 1.0.
- Letouzey, J.-L. I. (2012). Managing technical debt with the SQALE method. *IEEE Software*, pp. 44-51.
- Li, Z., Avgeriou, P., & Liang, P. (2015). A systematic mapping study on technical debt and its management. *Journal of Systems and Software*, 193 - 220.
- Lint. (2015, 3 3). Retrieved from Android: <http://tools.android.com/tips/lint>
- List of tools for static code analysis. (2015, January 10). Retrieved from Wikipedia: [http://en.wikipedia.org/wiki/List\\_of\\_tools\\_for\\_static\\_code\\_analysis](http://en.wikipedia.org/wiki/List_of_tools_for_static_code_analysis)
- Michael, G., & Laurie, W. (2007). Toward the Use of Automated Static Analysis Alerts for Early Identification. ICIMP, 18-23.
- Pocatilu, P. (2006). Influencing Factors of Mobile Applications. *ECONOMY INFORMATICS*, 102-104.
- Runeson, P., & Höst, M. (2009). Guidelines for conducting and reporting case study research in software engineering. *Empirical Software Engineering*, 131-164.
- S.A, S. (2015, 2 6). Documentation for SonarQube 4.5 and 4.5.x LTS. Retrieved from SonarQube: <http://docs.sonarqube.org/display/SONARQUBE45/Documentation>
- SonarQube. (2015, 3 3). Retrieved from Java Plugin - SonarQube - Confluence: <http://docs.sonarqube.org/display/SONAR/Java+Plugin>
- Statista. (2015, February 25). Retrieved from Number of available apps in the Google Play Store 2015 | Statistic: <http://www.statista.com/statistics/266210/number-of-available-applications-in-the-google-play-store/>
- Syer, M. D., Nagappan, M., Adams, B., & Hassan, A. E. (2014). Studying the relationship between source code quality and mobile platform dependence. *Software Quality Journal*, Volum.