

Prioritizing Corrective Maintenance Activities for Android Applications: An Industrial Case Study on Android Crash Reports

Valentina Lenarduzzi¹ , Alexandru Cristian Stan¹ ,
Davide Taibi^{1,2} , Gustavs Venters¹ , and Markus Windegger³

¹ Free University of Bolzano-Bozen, 39100 Bolzano-Bozen, Italy
{vlenarduzzi, astan, dtaibi, g.venters}@unibz.it

² Pervasive Computing Department, Tampere University of Technology,
Tampere, Finland

davide.taibi@tut.fi

³ SASAbus, 39100 Bolzano-Bozen, Italy
windegger@sasabz.it

Abstract. *Context:* Unhandled code exceptions are often the cause of a drop in the number of users. In the highly competitive market of Android apps, users commonly stop using applications when they find some problem generated by unhandled exceptions. This is often reflected in a negative comment in the Google Play Store and developers are usually not able to reproduce the issue reported by the end users because of a lack of information.

Objective: In this work, we present an industrial case study aimed at prioritizing the removal of bugs related to uncaught exceptions. Therefore, we (1) analyzed crash reports of an Android application developed by a public transportation company, (2) classified uncaught exceptions that caused the crashes; (3) prioritized the exceptions according to their impact on users.

Results: The analysis of the exceptions showed that seven exceptions generated 70% of the overall errors and that it was possible to solve more than 50% of the exceptions-related issues by fixing just six Java classes. Moreover, as a side result, we discovered that the exceptions were highly correlated with two code smells, namely “Spaghetti Code” and “Swiss Army Knife”. The results of this study helped the company understand how to better focus their limited maintenance effort. Additionally, the adopted process can be beneficial for any Android developer in understanding how to prioritize the maintenance effort.

Keywords: Continuous monitoring · Software quality · Technical debt

1 Introduction

In the quickly evolving world of mobile applications, users can nowadays choose among a large variety of alternatives. On the one hand, this huge market boosts the competition among similar apps. On the other hand, users can easily switch to competitor applications if some features are missing or – as commonly happens – not responding correctly. The cost of acquiring lost customers is always high, and losing

customers because of bugs and issues – which are often related to code exceptions – occurs very frequently.

Android applications sometimes crash or are irresponsive, and several users commonly report this with negative comments in the Google Play Store, complaining about non-functioning parts such as unresponsive buttons or other issues. This behavior is often due to uncaught exceptions happening [23, 24]. We consider uncaught exceptions those exceptions not caught in “try/catch” blocks that may occur at any time due to programming errors that the developers did not expect to deal with (e.g., “OutOfMemoryError”, “NullPointerException”) [17].

In this paper, we report on an industrial case study about an Android application developed by SASA bus SpA AG, an Italian public transportation company¹. Since SASA bus had limited resources available for the maintenance of their application, our aim was to support them in the identification of better ways to invest their maintenance effort in order to prevent the greatest possible number of issues related to code exceptions and to have them lose the lowest possible number of customers. Therefore, our goal was to understand which code exception(s) generated most of the crashes, or generally what the bugs in the application were, to allow focusing corrective maintenance on solving the most relevant errors with the little resources available.

Crash reports are collected directly from the Google Play Store, which provides a system that collects for the developers the latest exceptions that generated a fatal error of the application.

The results of our study showed that six uncaught exceptions occurred in more than 70% of the crashes and that it is possible to reduce this high percentage by investing a limited amount of effort. Furthermore, from the manual inspection of the code that threw the exceptions, we observed the prominent presence of two code smells [3] (“Spaghetti Code” and “Swiss Army Knife”) in similar exceptions. Moreover, this result confirms the findings of Microsoft, who reported that 50% of crashes are caused by 1% of bugs [18].

This paper is structured as follows: Sect. 2 describes the continuous exception monitoring approaches. Section 3 reports the case study. Section 4 discusses the results while Sect. 5 draws conclusions and provides an outlook on future work.

2 The Exception Monitoring System

Although the trend towards DevOps and continuous exception monitoring tools is growing constantly², to the best of our knowledge the literature on exception monitoring is still very limited [1, 2].

Exception monitoring tools have been introduced recently and are usually collocated in the operational side of the DevOps pipeline [19]; however, the majority of these instruments are crash reporting systems: when a fatal error in the application

¹ SASA bus. www.sasabus.it.

² <https://trends.google.com>.

happens, users get a summary message and are asked whether they are willing to report the details to the development team.

However, developers get only crash-related data, while non-fatal errors, which are often triggered by unhandled exceptions, remain unreported, hence not getting analyzed and solved. Recently, some tools aimed at dealing with this issue have been introduced on the market: Most of them offer ways to integrate exception-collecting plugins within the client to be monitored, while others simply require the addition of dependencies or libraries to the source code. When an unhandled exception occurs, some contextual data is collected on the client and forwarded to a server (on the premises or in the cloud), which stacks and categorizes the exceptions.

One of the most frequently adopted exception monitoring tools is Firebase, the default crash report provided in the Google Play Store. Firebase sends crash reports to developers, reporting a list of fatal crashes and the exceptions that caused them. Furthermore, Firebase also allows tracking uncaught exceptions, even though this feature requires instrumenting the code by adding a custom logger to the classes and substantially modifying all the source code.

One of the goals of our research is to improve the resolution of such problems quickly and promptly by capturing also concealed problems related to exceptions in the software, categorizing them, and solving the issues accordingly.

In Table 1, we present some of the most commonly adopted exception monitoring tools on the market.

Table 1. The most common exception management tools

Tool	Supported programming language	Exception categorization	Open source	Link
<u>OverOps</u>	Java, Scala, Closure, .NET	Yes	No	www.overops.com
<u>Airbrake</u>	All major ones	Yes	Yes	http://airbrake.io
<u>Sentry</u>	All major ones	Yes	Yes	www.sentry.io
<u>Rollbar</u>	All major ones	Yes	Yes	www.rollbar.com
<u>Raygun</u>	All major ones	Yes	Yes	www.raygun.org
<u>Honeybadger</u>	All major ones	Yes	Yes	www.honeybadger.io
<u>Stackhunter</u>	Java	Yes	No	www.stackhunter.io
<u>Bugsnag</u>	All major ones	Yes	Yes	www.bugsnag.com
<u>Exceptionless</u>	All major ones	Yes	Yes	www.exceptionless.com
<u>Firebase</u>	Android, iOS	Yes	No	https://firebase.google.com

3 The Case Study

In this section, we report on our case study, which aimed at (1) understanding how to manage and capture exceptions, and which exceptions need to be focused on in order to solve most of the errors, and (2) at reducing the frequency of exceptions.

In the following sub-sections, we will present the study process we adopted: First, we will describe the study goal as well as the questions and metrics to be collected; then we will describe the design and the execution of the study; and finally, we will present the results we obtained.

3.1 Study Goal, Questions and Metrics

According to our expectations, we formulated the goal of the case study according to the GQM approach [16] as:

*Analyze the unhandled exceptions
for the purpose of prioritizing them
with respect to their frequency
from the point of view of the end user
in the context of the SASABus application*

Therefore, we formulated our research questions as follows and further derived the relative metrics.

RQ1: Which are the most frequently recurring exceptions in our Android application? In this RQ, we aim at identifying the most frequent exceptions in order to rank them based on frequency. We expect to have a subset of frequent exceptions that generate most of the errors.

M1: Total number of exceptions. This metric considers all the exceptions that occurred in running apps.

M2: Number of unique exceptions.

M3: Number of occurrences of the same exception.

M4: Number of occurrences of the same exception from the same class.

As an example, consider the exceptions listed in Table 2. In this case, the total number of exceptions (M1) is six. The number of unique exceptions M2 is three (E1, E2, E3), and the number of occurrences of the same exception (M3) is three for E1, two for E2, and one for E3. However, taking into account M4, we see that E1 was raised twice in class C1 and once in class C2, while exception E2 was raised twice, both times in C1.

Table 2. Example of exceptions raised by an application

Date	Exception	Class
01/01/2017 08:00	E1	C1
01/01/2017 08:30	E1	C2
01/01/2017 09:00	E1	C1
02/01/2017 09:30	E2	C1
02/01/2017 10:30	E2	C1
02/01/2017 11:30	E3	C4

RQ2: Which and how many exception-related issues should be fixed in order to reduce the number of exceptions by 70%?

With this RQ, we analyze the frequency of the exceptions and identify those exceptions that need to be addressed in order to reduce their occurrence by 70%. This RQ was required by the industrial partner, who seeks to reduce their number of unhandled exceptions and to increase user satisfaction, as well as to reduce the number of negative comments due to some unresponsive features caused by the exceptions.

As a consequence of RQ1, we expect to be able to solve 70% of the exceptions by fixing only 20% of the issues.

As an example, considering the exceptions reported in Table 1, addressing exception E1 in class C1 and E2 in class C2 would be more beneficial than solving E1 in C2 or E3 in C4.

RQ3: Which classes generate more exceptions?

This RQ aims at understanding whether some classes deserve more attention or need to be refactored.

In our example, we can identify C1 as a class worth getting the developers' attention.

3.2 The Analyzed Application

This case study is based on the analysis of the SASA bus Android application, available on the Google Play Store³. The application has been downloaded by nearly 50 K users and is currently used by more than 10 K users. The source code is also available as Open Source⁴. The app was developed by SASA bus with the aim of providing bus timetables, delays and locations of buses, and other travel-related information. The software is developed with an agile process, and it migrating flawlessly from an ad-hoc process to SCRUM from several years [13].

Figure 1 shows the SASA bus application on the Google Play Store.

3.3 Results

In order to answer to our research questions, we analyzed the SASA bus Android crash reports from April 2013 to March 2017 (we chose this time range because the Google Play Store does not save reports older than four years).

Before analyzing the individual result of each RQ, we confirmed the normality of the exception distribution by means of the Shapiro-Wilk test and analyzed the descriptive statistics.

As for **RQ1**, the SASA bus application collected 1208 exceptions in four years, with an average of 0.83 exceptions per day. Peaks in the number of exceptions per day were reached during high season (e.g., Christmas and summer time), with a maximum of eight exceptions per day.

³ SASA bus on the Google Play Store. <https://play.google.com/store/apps/details?id=it.sasabz.android.sasabus&hl=en>.

⁴ SASA bus GitHub repository. <https://github.com/SASA bus/SASA bus>.



Fig. 1. The SASAbus application on the Google Play Store

Table 3. Frequency of exceptions

Java exception	Frequency	%
java.lang.RuntimeException	433	35.8
java.lang.NullPointerException	212	17.5
java.lang.ClassCastException	124	10.3
java.lang.UnsupportedOperationException	106	8.8
java.lang.IllegalStateException	94	7.8
java.lang.ClassNotFoundException	57	4.7
android.view.WindowManager\$BadTokenException	45	3.7
java.lang.IllegalAccessException	26	2.2
android.database.sqlite.SQLiteDatabaseCorruptException	23	1.9
java.lang.IndexOutOfBoundsException	23	1.9
java.lang.IllegalArgumentException	21	1.7
java.lang.InstantiationException	16	1.3
Native crash	8	0.7
java.lang.StringIndexOutOfBoundsException	7	0.6
android.database.sqlite.SQLiteException	4	0.3
java.lang.OutOfMemoryError	4	0.3
android.util.AndroidRuntimeException	3	0.2
java.io.IOException	1	0.1
java.lang.NoSuchMethodError	1	0.1
Total	1208	

We identified 19 types of unique exceptions with different frequencies (Table 3). Two exceptions were caught in more than 50% of the cases, namely “java.lang.RuntimeException” and “java.lang.NullPointerException”, while other exceptions occurred rarely.

Taking into account the classes that had more exceptions, we can see that 44.8% of the exceptions were thrown by three Java classes (Table 4). However, the same exceptions were often thrown by different classes.

Table 4. Frequency of exceptions per class that raised more than 10 exceptions

Java class	Frequency	%
android.graphics.Bitmap	223	18.5
android.os.Looper	211	17.5
java.util.Collections\$UnmodifiableCollection	106	8.8
it.sasabz.sasabus.ui.busstop.NextBusFragment\$2	81	6.7
dalvik.system.BaseDexClassLoader	57	4.7
android.support.v4.app.Fragment	44	3.6
android.support.v4.app.FragmentManagerImpl	41	3.4
android.view.ViewRootImpl	33	2.7
it.sasabz.sasabus.opendata.client.model.BusStation	31	2.6
it.sasabz.sasabus.ui.busstop.NextBusFragment\$1	31	2.6
it.sasabz.sasabus.ui.busstop.NextBusFragment\$7	28	2.3
it.sasabz.android.sasabus.classes.adapter.MyWayListAdapter	27	2.2
java.lang.Class	26	2.2
android.database.sqlite.SQLiteConnection	23	1.9
android.support.v7.widget.RecyclerView\$n	21	1.7
it.sasabz.sasabus.ui.busschedules.BusScheduleDetailsFragment	19	1.6
it.sasabz.sasabus.ui.busschedules.BusSchedulesFragment\$4	17	1.4
it.sasabz.sasabus.ui.routing.SearchResultsFragment	17	1.4
it.sasabz.sasabus.ui.busstop.NextBusFragment\$6	16	1.3
java.lang.reflect.Constructor	16	1.3
android.app.LoadedApk	13	1.1
android.view.ViewRoot	12	1
it.sasabz.android.sasabus.classes.services.CheckUpdate	11	0.9
it.sasabz.android.sasabus.fcm.a.e	10	0.8
org.mapsforge.map.reader.MapDatabase	10	0.8

The analysis of **RQ2** and **RQ3** led us to the identification of the exceptions that should be addressed by the developers. As requested by SASAbus, our goal was to identify the classes that generated 70% of the exceptions.

Table 5 reports the frequency of the exceptions per class, where the “Freq.” column reports the number of times the exception was thrown by the class reported in the column “Java Class”, while “Rel Freq.” reports how often an exception of the same type was thrown by the selected class. For reasons of space, Table 5 only reports the

Table 5. Frequency of exceptions per class. Package named is omitted for reasons of space.

Java classes	Exception	Freq.	Rel freq.	Abs. freq.
Bitmap	RuntimeException	222	51%	18%
Looper	RuntimeException	211	49%	17%
NextBusFragment	ClassCastException	124	100%	10%
Collections \$UnmodifiableCollection	UnsupportedOperationException	106	100%	9%
BaseDexClassLoader	ClassNotFoundException	57	100%	5%
NextBusFragment	NullPointerException	55	26%	5%
MyWayListAdapter	NullPointerException	54	25%	4%
BusScheduleDetailsFragment	NullPointerException	38	18%	3%
SearchResultsFragment	NullPointerException	34	16%	3%
BusStation	NullPointerException	31	15%	3%
Class	IllegalAccessException	26	100%	2%
Constructor	InstantiationException	16	100%	1%

name of the class, omitting the package. Package names can be easily associated from Table 4. As an example, “RuntimeException” was thrown 433 times, (222 times in class “Bitmap.java” and 211 times in class “Looper.java”). Finally, column “Abs Freq.” reports the impact of the exception over the total number of occurrences. For instance, RuntimeException, thrown 222 times in the class Bitmap, accounted for 18% of the 1208 exceptions.

Summing up, the first seven exceptions occurred in nearly 70% of the cases. Furthermore, one method of the class “NextBusFragment” threw 124 “ClassCastExceptions” and 55 “NullPointerExceptions”. Hence, focusing corrective maintenance on the first six classes of this table would fix the issues that generated 70% of the exceptions.

4 Discussion

A first analysis of the source code forced us to consider the importance of adopting a continuous exception monitoring system. In the last four years of this code’s evolution, 32 classes generated a total of 1208 exceptions, and just six of them threw nearly 70% of the exceptions. Maintenance is an expensive task [4] and, considering the limited resources available to SASA bus, they will now be able to focus their maintenance effort especially on the most critical issues, solving the biggest number of issues with the lowest possible effort.

Based on the results obtained, we manually inspected the source code of these six classes and found interesting side results.

One of the main side results is that in most of the classes generating exceptions, we identified code smells, a set of structural software characteristics that indicate code or design problems that can make software hard to evolve and maintain [3].

Both of the classes that generated “`RuntimeException`” have a “Spaghetti Code” structure, while the other classes were generally very complex or structured as a “Swiss Army Knife”. Although this result is not statistically significant and was not obtained with a rigorous method, we are planning to train SASAbus developers in code smells and code quality practices to help them write cleaner code, which would be easier to maintain in the future. However, the result can be already beneficial from the development team, that could start to create a knowledge base of their development practices, reporting the bad practices that generated the exception. As example, we recommended to start keeping track of the data with the Agile Experience Factory [5].

Based on the results provided by this work, SASAbus decided to adopt an exception monitoring tool. Starting in May 2017, they began using Sentry because of its open source license, which allows on-premise installations to collect the data, and because of the ease of configuration, which requires only the addition of one dependency library to the code base without the need to instrument the source code.

The results of this work contribute to understanding the importance of a continuous exception monitoring approach.

As for threats to validity, in our work we prioritized the exceptions based on their frequency, since they were all related to crash reports that caused the shutdown of the application. A continuous monitoring approach that reports uncaught exceptions should also be able to classify them, i.e., not only provide their frequency, but also relate them to their severity. The results obtained are useful for SASAbus but not statistically significant, since they were not obtained using a rigorous method. These results and the approach we adopted can be validated in future studies, also applying defect prediction techniques.

5 Conclusion

In this work, we presented an industrial case study aimed at understanding the most frequent exceptions raised by an Android application adopted by more than 10 K users.

We analyzed the crash reports and the exceptions from the last four years, identifying seven types of exceptions that occurred in more than 70% of the cases.

A manual code inspection enabled us to understand that most of the exceptions were generated by bad code practices, mainly code smells, which decrease the readability, understandability, and maintainability of the code, hence causing bugs and issues [7, 12].

The results of this work helped SASAbus to clearly understand where to focus their corrective maintenance effort, working on the most frequent issues, as well as solving most of the problems with the least effort possible. Before the analysis of their exceptions, the application crashed nearly once a day because of uncaught exceptions, and several users reported the unresponsive features on the Google Play Store, such as unresponsive buttons or other exception-related issues. The results of the newly installed continuous monitoring system will be visible in the near future, when developers will be able to not only detect the most frequent exceptions, but also to intervene based on the severity of the issue.

The main lesson learned from this work is that, thanks to a continuous monitoring tool and statistical analysis of the exceptions, companies can easily understand how to better address part of the corrective maintenance effort.

A classic fault prediction approach based on code analysis, might have indicated the high possibility of running into problems when delivering the app in a proactive way. Therefore, we are planning to apply prediction models, both based on internal and on external metrics, similar to these we already applied in [6, 9, 10, 14, 15] paired with this approach.

We are currently working on analyzing change proneness in classes that generated issues, including bugs and fixes manually reported by users and developers in the issue tracker and not actually related to exceptions [12].

As described in our previous work [8, 11], future work will include the analysis of static metrics and code smells of this application, as well as understanding possible correlations between code smells and issues. Another future work will be the analysis of comments and the relations between negative comments, crashes and exception, so as to better address maintenance activities. Moreover, other approach for the automatic discover of android crashes such as [20, 21, 22] could be adopted to better focus the maintenance activities.

References

1. Krall, A., Probst, M.: Monitors and exceptions: how to implement Java efficiently. *Concur. Pract. Exp.* **10**(11–13), 837–850 (1998)
2. Turner, L.D., Owhoso, V.: Use ERP internal control exception reports to monitor and improve controls. *Manag. Account. Q.* **10**(3), 41–50 (2009)
3. Fowler, M.: *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Longman Publishing Co., Inc., Boston (1999)
4. Lenarduzzi, V., Sillitti, A., Taibi, D.: Analyzing forty years of software maintenance models. In: *International Conference on Software Engineering (ICSE 2017)*, Buenos Aires, Argentina (2017)
5. Taibi, D., Lenarduzzi, V., Diebold, P., Lunesu, I.: Operationalizing the experience factory for effort estimation in agile processes. In: *International Conference on Evaluation and Assessment in Software Engineering (EASE 2017)*, Karlskrona, Sweden, 15–16 June 2017. <http://doi.org/10.1145/3084226.3084240>
6. Lavazza, L., Morasca, S., Taibi, D., Tosi, D.: Predicting OSS trustworthiness on the basis of elementary code assessment. In: *International Symposium on Empirical Software Engineering and Measurement (ESEM 2010)*, Bolzano-Bozen, Italy, 16–17 September 2010. <http://doi.org/10.1145/1852786.1852834>
7. Taibi, D., Janes, A., Lenarduzzi, V.: Towards a lean approach to reduce code smells injection: an empirical study. In: Sharp, H., Hall, T. (eds.) *XP 2016*. LNBP, vol. 251, pp. 300–304. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-33515-5_30
8. Lenarduzzi, V., Stan, A.C., Taibi, D., Tosi, D., Venters, G.: A dynamical quality model to continuously monitor software maintenance. In: *11th European Conference on Information Systems Management (ECISM 2017)*, Genoa, Italy, 14–15 September 2017
9. Tosi, D., Lavazza, L., Morasca, S., Taibi, D.: On the definition of dynamic software measures. In: *International Symposium on Empirical Software Engineering and Measurement (ESEM 2012)*, pp. 39–48. ACM, New York (2012). <http://doi.org/10.1145/2372251.2372259>

10. Taibi, D., Lavazza, L., Morasca, S., Tosi, D.: An empirical investigation of perceived reliability of open source Java programs. In: Proceedings of the 27th Annual ACM Symposium on Applied Computing (SAC 2012), Riva del Garda, Italy, 26–30 March 2012. <http://doi.org/10.1145/2245276.2231951>
11. Taibi, D., Janes, A., Lenarduzzi, V.: How developers perceive code smells and antipatterns in source code: a replicated study. *Inf. Softw. Technol. J. (IST)* **92**, 223–235 (2017). <https://doi.org/10.1016/j.infsof.2017.08.008>
12. Janes, A., Lenarduzzi, V., Stan, A.C.: A continuous software quality monitoring approach for small and medium enterprises. In: Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering Companion (ICPE 2017 Companion), L'Aquila, Italy (2017). <http://doi.org/10.1145/3053600.3053618>
13. Lavazza, L., Morasca, S., Taibi, D., Tosi, D.: Applying SCRUM in an OSS development process: an empirical evaluation. In: Sillitti, A., Martin, A., Wang, X., Whitworth, E. (eds.) XP 2010. LNBIP, vol. 48, pp. 147–159. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-13054-0_11
14. Bianco, V.D., Lavazza, L., Morasca, S., Taibi, D., Tosi, D.: The QualiSPo approach to OSS product quality evaluation. In: Proceedings of the 3rd International Workshop on Emerging Trends in Free/Libre/Open Source Software Research and Development, pp. 23–28. ACM, New York (2010). <http://doi.org/10.1145/1833272.1833277>
15. Morasca, S., Taibi, D., Tosi, D.: Towards certifying the testing process of open-source software: new challenges or old methodologies? In: Proceedings of the 2009 ICSE Workshop on Emerging Trends in Free/Libre/Open Source Software Research and Development, FLOSS 2009, pp. 25–30 (2009). <http://doi.org/10.1109/FLOSS.2009.5071356>
16. Caldiera, G., Rombach, H.D., Basili, V.: Goal question metric approach. In: Encyclopedia of Software Engineering, pp. 528–532. Wiley, New York (1994)
17. Java Oracle: Uncaught Exceptions Documentation. <https://docs.oracle.com/javase/7/docs/api/java/lang/Thread.UncaughtExceptionHandler.html>. Accessed May 2017
18. Rooney, P.: Microsoft's CEO: 80-20 Rule Applies To Bugs, Not Just Features. CRN News, 03 October 2002. <http://www.crn.com/news/security/18821726/microsofts-ceo-80-20-rule-applies-to-bugs-not-just-features.htm>
19. Delgado, N., Gates, A.Q., Roach, S.: A taxonomy and catalog of runtime software-fault monitoring tools. *IEEE Trans. Softw. Eng.* **30**(12), 859–872 (2004)
20. Moran, K., Linares-Vásquez, M., Bernal-Cárdenas, C., Vendome, C., Poshyvanyk, D.: Automatically discovering, reporting and reproducing android application crashes. In: 2016 IEEE International Conference on Software Testing, Verification and Validation (ICST). IEEE (2016)
21. White, M., Linares-Vásquez, M., Johnson, P., Bernal-Cárdenas, C., Poshyvanyk, D.: Generating reproducible and replayable bug reports from android application crashes. In: 2015 IEEE 23rd International Conference on Program Comprehension (ICPC). IEEE (2015)
22. Agarwal, S., Mahajan, R., Zheng, A., Bahl, V.: Diagnosing mobile applications in the wild. In: Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks. ACM (2010)
23. Kechagia, M., Spinellis, D.: Undocumented and unchecked: exceptions that spell trouble. In: Proceedings of the 11th Working Conference on Mining Software Repositories. ACM (2014)
24. Cinque, M., Cotroneo, D., Testa, A.: A logging framework for the on-line failure analysis of android smart phones. In: Proceedings of the 1st European Workshop on AppRoaches to MOBiquitous Resilience. ACM (2012)