Undocumented and Unchecked: Exceptions That Spell Trouble

Maria Kechagia and Diomidis Spinellis

Athens University of Economics and Business
Department of Management Science and Technology
76 Patission str, 104 34, Athens, Greece
{mkechagia, dds}@aueb.gr

ABSTRACT

Modern programs rely on large application programming interfaces (APIs). The Android framework comprises 231 core APIs, and is used by countless developers. We examine a sample of 4,900 distinct crash stack traces from 1,800 different Android applications, looking for Android API methods with undocumented exceptions that are part of application crashes. For the purposes of this study, we take as a reference the version 15 of the Android API, which matches our stack traces. Our results show that a significant number of crashes (19%) might have been avoided if these methods had the corresponding exceptions documented as part of their interface.

Categories and Subject Descriptors

D.2.17.g [**Software Engineering**]: Code documentation; D.2.2.d [**Software Engineering**]: Modules and interfaces; D.2.4.f [**Software Engineering**]: Programming by contract; D.2.4.g [**Software Engineering**]: Reliability

General Terms

Documentation, Design

Keywords

APIs, mobile applications, stack traces, exceptions

> *Ken Thompson has an automobile which he helped design. Unlike most automobiles, it has neither speedometer, nor gas gauge, nor any of the other numerous idiot lights which plague the modern driver. Rather, if the driver makes a mistake, a giant "?" lights up in the center of the dashboard. "The experienced driver," says Thompson, "will usually know what's wrong."* — Anonymous [5]

MSR'14, May 31 – June 1, 2014, Hyderabad, India
Copyright 2014 ACM 978-1-4503-2863-0/14/05...$15.00
http://dx.doi.org/10.1145/2597073.2597089

1. INTRODUCTION

Error messages may be redundant for experts, as they can guess the cause of a problem. However, not all users are experts, and not all experts are infallible. Therefore, in a well-designed system the cause of each error should be unambiguous and well-documented. In this study we give empirical evidence for this assertion, arguing that all exceptions should be documented in the description of API methods used in Java programs.

When a Java program terminates unexpectedly (crashes) the system produces a crash report with a stack trace and runtime metadata. The resulting stack trace is a chain of frames of method calls that leads to an exception and gives information about the crash cause. In Java there are two types of exceptions: checked and unchecked exceptions.[1] Checked exceptions refer to "exceptional conditions that a well-written application should anticipate and recover from." Unchecked exceptions are exceptional conditions that can be either external (`Error`) or internal (`RuntimeException`) and the application usually cannot recover from them.

There is a common practice among API designers to include all the possibly thrown checked exceptions—and some of the unchecked exceptions—in the API reference. Traditionally, designers document unchecked exceptions when developers can overcome from particular execution failures [1, 6]. However, excluding possible exceptions from the API reference may not be productive when a large number of developers from different programming levels uses a common API to build their applications. The documentation of likely thrown exceptions in the description of a method can have a twofold use: guidance on how to use correctly this method and understanding of the causes of possible crashes. Thus, documented exceptions can help in the stability of a program. To the best of our knowledge, there is no empirical study that argues when unchecked exceptions must be documented. Here, we try to show that API methods with undocumented exceptions can be responsible for application crashes.

In particular, we got a data set of 4,900 distinct Java stack traces from 1,800 Android applications crashes coming from a centralized crash report management service. We identified API methods that are probably responsible for crashes by applying a heuristic method in our crash data and we searched for their exceptions in the Android API. We downloaded and parsed the source code of the version 15 of the Android API that matched our data set and we located methods with documented exceptions.

[1] http://docs.oracle.com/javase/tutorial/essential/exceptions

Our findings show that 19% of the examined stack traces had API methods with undocumented exceptions. This means that these crashes could have been avoided if the involved methods had documentation for possible thrown exceptions. Also, contrary to our expectations, we realized that 40% of the documented exceptions in the Android source code are specific unchecked exceptions, namely: `IllegalArgumentException`, `NullPointerException`, `IllegalStateException`.

In the following sections, we first describe our data set (Section 2). Then, we outline our methods (Section 3). In Section 4, we present our results, in Section 5 we make a discussion about our findings, and in Section 6 we refer to the threats to validity of our study. Finally, we end up with related work (Section 7) and our conclusions (Section 8).

2. DATA

For the purposes of this study we used a data set of 4,900 Java stack traces (see `printStackTrace()`)[2] from Android application crashes. The provider of the data set was a Greek startup called BugSense,[3] which is currently a company of Splunk.[4] BugSense offers a centralized crash report management service that collects stack traces from applications that have been crashed and have installed the BugSense SDK.

Specifically, we analyzed 4,902 distinct crash reports from 1,800 different Android applications that run on devices with the Android API level 15 (4.0.3–4.0.4). Our data set was collected on the May of 2012. In addition, we downloaded the Android SDK and used the source code of the Android API (level 15). We parsed 2,171 Java files with online documentation.

Android is an embedded device based on the Linux operating system that can host mobile applications. Briefly, in the bottom layer of Android framework there is the Linux kernel. The kernel is the border between the device and the software and it provides services such as memory management, networking, and power management. In the middle layer there is the Dalvik virtual machine and the Java Native Interface. Dalvik supports the execution of multiple applications on the system. The Java Native Interface is used to perform calls from Java code into native code. Finally, on the top layer there are several Java classes from: 1) core applications (contacts, phone, browser), 2) third-party applications, and 3) the Java Platform (J2SE).

3. METHODS

To identify methods from the Android API that lack critical documentation, we worked in two directions. First, we pinpointed API methods possibly responsible for crashes and linked the methods with the thrown stack trace exceptions (Figure 1). Second, we parsed the source code of the Android API and drew methods with documented exceptions (Figure 2). Thus, having a set of risky API methods and knowing the documented exceptions of the Android API, we were able to list methods that are involved in crashes and have undocumented exceptions. Following we explain our techniques in details.

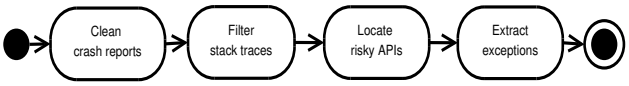

Figure 1: Identification of risky methods

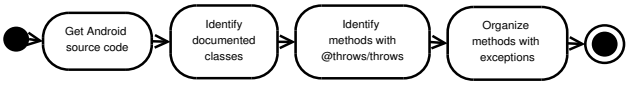

Figure 2: Extraction of API exceptions

3.1 Identification of Risky Methods

The investigation of risky APIs can be made by examining the API reference and finding complicated APIs that lead developers to the violation of the APIs' invariants or preconditions. Also, it can be conducted through fuzz testing by passing random values as method inputs [3]. Here, we empirically identified risky APIs based on their frequent manifestation on application crashes.

In our sample, stack traces consist of method calls from the Android framework that lead to an exception, possibly through an application and an API. We assert that the last instance on an application—API pair is an API call that can lead to an application crash. Thus, by locating such API calls we can find API methods that possibly contribute to application crashes. In Listing 1, `setContentView` is such an example. Then, taking also into account the root exception and the exceptions from the rest frames (for chained stack traces), we could have more concise information about the reason of a crash.

Listing 1: Method calls sequence

```
com.example.Serialize$Looper.run
android.os.Looper.loop
android.os.Handler.dispatchMessage
com.example.SerializeHandler.onMessage
com.example.app.Activity$1.work
android.app.Activity.setContentView
```

Finally, in the beginning of our study, we applied some basic filtering to conform our data set in an appropriate format for analysis and removed meta data from our crash reports.

3.2 Extraction of Documented Exceptions

To identify methods with documented exceptions (checked and unchecked), we downloaded and parsed the source code of the Android API (version 15). From the Java files we kept only the classes that have online documentation in the Android reference.[5] Then, we wrote a Java doclet[6] that identifies methods with `@throws` comments and declared exceptions in their signatures; we excluded the private methods as they do not appear in the online documentation. Thus, we got a set of methods with documented exceptions. Finally, we read the files produced by the doclet and organized the methods in a dictionary based on their names.

[2]http://docs.oracle.com/javase/7/docs/api/java/lang/Throwable.html
[3]https://www.bugsense.com/
[4]http://www.splunk.com/
[5]http://developer.android.com/reference/classes.html
[6]http://docs.oracle.com/javase/6/docs/technotes-/guides/javadoc/doclet/overview.html

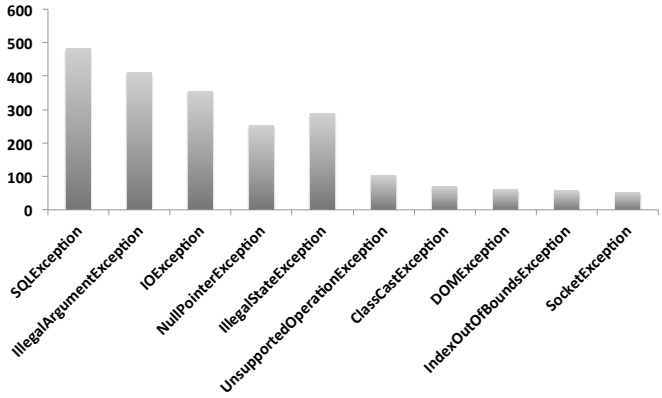

Figure 3: Top 10 exceptions in Android

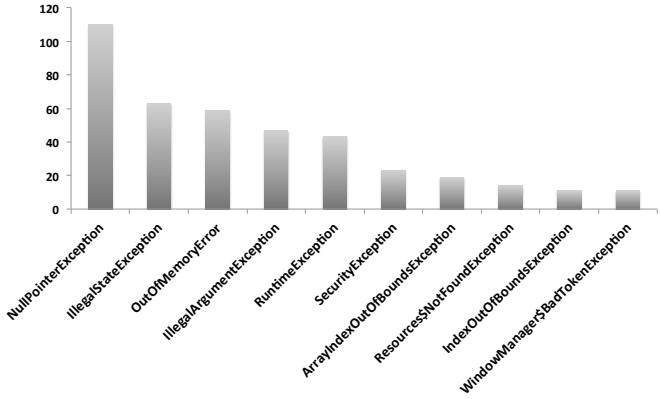

Figure 4: Top 10 exceptions in the stack traces

4. RESULTS

From the analysis of the Android source code, we found that only 18% of distinct non-private API methods (grouped by name) had documented exceptions (56% checked and 44% unchecked). We also investigated 186 methods from Android packages (`android.*`) involved in the crashes and we noticed that 128 of these methods (69%) had undocumented exceptions; all the involved exceptions were unchecked—uncaught on compile time. The found undocumented methods came from 944 stack traces. Then, if these methods had documented exceptions, even 19% of the examined crashes could have been avoided. In the rest section, we present descriptive statistics regarding the unchecked and undocumented exceptions of the Android API.

4.1 Unchecked Exceptions

Figure 3 illustrates the ten top exceptions in the Android API source code. We found these exceptions in `@throws` comments and declared exceptions in method signatures. It is notable that 40% of methods with documented exceptions had specific unchecked exceptions (`IllegalArgumentException`, `NullPointerException`, `IllegalStateException`) in their description.

4.2 Undocumented Exceptions

Figure 4 shows the ten top thrown exceptions from the stack traces. It seems that most crash causes came from programming errors (`NullPointerException`), race conditions (`IllegalStateException`) and out of memory problems (`OutOfMemoryError`). In addition, Table 1 presents the ten most common methods from the stack traces that had undocumented exceptions in the source code. These findings concern the version 15 of the Android API, according to the API level of our stack traces.

5. DISCUSSION

Our findings show that a significant number (44%) of API methods in the Android API reference has documentation for unchecked exceptions. We also found that `IllegalArgumentException` and `IllegalStateException` are among the most common documented unchecked exceptions in the Android API and among the most common reasons for an application to crash. This means that these exceptions can be included in the API documentation plentifully. In addition, we found in our stack traces many methods (24%) with generic exceptions (`RuntimeException` and `NullPointerException`) that have not got documented exceptions in their interfaces. This can be considered as an API design problem. Such methods could have more specific exceptions in their documentation based on possible crashes. For instance, the `open`[7] method for the camera could be associated with exceptions regarding permissions and concurrency issues. However, as it is not always feasible for API designers to include in the documentation exceptions about crashes related to resource management (`OutOfMemoryError`), they could provide examples for the safe implementation of crucial methods such as `createBitmap`.[8] Finally, they could include prevention mechanisms in the interfaces—for the previous case, an autoresize interface for costly bitmaps.

6. THREATS TO VALIDITY

Internal Validity For the identification of the risky API methods we used heuristics based on regular expressions. Then, we could have missed methods from third-party APIs (i.e. methods started with `com.*` in the stack traces.)

External Validity Even though we used data from real crashes and found thrown exceptions that are not in the documentation of the involved methods, we need further empirical evidence to strengthen our results. Our findings are associated with one system, Android, and a specific programming language, Java. To argue about the kinds of the exceptions that should be in the documentation, we need to examine other data sets such as from the iOS. In addition, we examined the methods of a limited sample of stack traces. We would like to investigate a larger sample and argue about specific kinds of methods that should have documented unchecked exceptions. We also aim to compare the stack traces from different Android API versions to validate our results. For instance, we found that the `createScaledBitmap` method had not got the `IllegalArgumentException` in its documentation for the version 15 of the Android API, whereas this exception exists in version 19. Finally, we found that 19% of our sample's crashes could have been avoided if the thrown exceptions in the stack traces where also in the documentation. However, this percentage is an upper bound,

[7]http://developer.android.com/reference/android/-hardware/Camera.html
[8]http://developer.android.com/reference/android/-graphics/Bitmap.html

Table 1: Top 10 methods involved in crashes

Methods	Exceptions
dismiss	IllegalArgumentException, NullPointerException
show	WindowsManager.BadTokenException, IllegalStateException, InflateException
setContentView	InflateException
createScaledBitmap	IllegalArgumentException, NullPointerException
onKeyDown	IllegalStateException
isPlaying	IllegalStateException
unregisterReceiver	IllegalArgumentException
onBackPressed	IllegalStateException
showDialog	WindowManager.BadTokenException
create	Resources.NotFoundException

because it not always feasible for API designers to include exceptions such as `OutOfMemoryError` in the documentation.

7. RELATED WORK

A crash report can include a stack trace and metadata: application name, operating system, date and time of the crash. These data provide valuable information for crash cause understanding and software reliability. Ganapathi et al. [4] analyzed crash reports from the Windows XP kernel and identify basic crash cause types. Kim et al. [7] conducted an empirical investigation on the Firefox and Thunderbird crash report databases to predict "top crashes" for new software releases. In addition, stack traces are useful for the mining and quality evaluation of crash reports [10, 2], as well as for bug localization [11]. Contrary to the existing works, we analyzed stack traces from crash reports and identified Android API methods with undocumented exceptions.

Although there are practical guidelines for well-written APIs [1, 6], there is still ground for empirical studies that distinguish good from bad API paradigms. Current studies focus on API learnability [9, 8]. Here, we argued about exceptions that could have been included in the documentation of API methods.

8. CONCLUSIONS

We analyzed stack traces from Android application crashes to identify methods from the Android API that have missing documentation concerning exceptional cases. We found that 18% of non-private methods in the Android API had undocumented exceptions. From the methods with documented exceptions, 56% were checked and 44% unchecked. Also, we found that 69% of the methods—from Android packages—in our stack traces had undocumented exceptions in the Android API. Then, 19% of our crashes could have been caused by insufficient documentation.

9. ACKNOWLEDGMENTS

We would like to thank BugSense and more specifically its founders Panos Papadopoulos and John Vlachogiannis for the data and information they provided us. In addition, we would like to thank Dimitris Mitropoulos for his ideas and internal reviews.

This research has been co-financed by the European Union (European Social Fund—ESF) and Greek national funds through the Operational Program "Education and Lifelong Learning" of the National Strategic Reference Framework (NSRF)—Research Funding Program: Thalis—Athens University of Economics and Business—Software Engineering Research Platform.

10. REFERENCES

[1] J. Bloch. How to design a good API and why it matters. In *Companion to the 21st ACM SIGPLAN symposium on Object–Oriented Programming Systems, Languages, and Applications*, OOPSLA '06, pages 506–507, New York, NY, USA, 2006. ACM.

[2] Y. Dang, R. Wu, H. Zhang, D. Zhang, and P. Nobel. ReBucket: a method for clustering duplicate crash reports based on call stack similarity. In *Proceedings of the 2012 International Conference on Software Engineering*, ICSE 2012, pages 1084–1093, Piscataway, NJ, USA, 2012. IEEE Press.

[3] J. E. Forrester and B. P. Miller. An empirical study of the robustness of windows NT applications using random testing. In *Proceedings of the 4th USENIX Windows System Symposium*, pages 59–68, 2000.

[4] A. Ganapathi, V. Ganapathi, and D. A. Patterson. Windows XP kernel crash analysis. In *LISA*, volume 6, pages 49–159, 2006.

[5] S. Garfinkel, D. Weise, and S. Strassmann, editors. *The UNIX Hater's Handbook*. IDG Books Worldwide, Inc., San Mateo, CA, USA, 1994.

[6] M. Henning. API design matters. *Commun. ACM*, 52(5):46–56, May 2009.

[7] D. Kim, X. Wang, S. Kim, A. Zeller, S. Cheung, and S. Park. Which crashes should I fix first?: Predicting top crashes at an early stage to prioritize debugging efforts. *Software Engineering, IEEE Transactions on*, 37(3):430–447, 2011.

[8] W. Maalej and M. P. Robillard. Patterns of knowledge in API reference documentation. *IEEE Transactions on Software Engineering*, 99(PrePrints):1, 2013.

[9] M. Robillard and R. DeLine. A field study of API learning obstacles. *Empirical Software Engineering*, 16(6):703–732, 2011.

[10] A. Schroter, N. Bettenburg, and R. Premraj. Do stack traces help developers fix bugs? In *Mining Software Repositories (MSR), 2010 7th IEEE Working Conference on*, pages 118–121, May 2010.

[11] S. Wang, F. Khomh, and Y. Zou. Improving bug localization using correlations in crash reports. In *Proceedings of the Tenth International Workshop on Mining Software Repositories*, MSR '13, pages 247–256, Piscataway, NJ, USA, 2013. IEEE Press.