An Empirical Study on Inter-Component Exception Notification in Android Platform

Vladymir L. Bezerra
Federal Institute of Ceará (IFCE)
Canindé, Ceará, Brazil
vladymir.bezerra@ifce.edu.br

Lincoln S. Rocha, João Bosco F. Filho, and Fernando A. M. Trinta
Group of Computer Networks, Software Engineering, and Systems (GREat)
Federal University of Ceará (UFC)
Fortaleza, Ceará, Brazil
{lincoln,bosco,fernando.trinta}@great.ufc.br

ABSTRACT

Android developers extensively use exception handling to improve robustness of mobile applications. The Android architecture and the object-oriented paradigm impose complexity to the way applications handle exceptions; many different components communicate among themselves and exceptions may be raised in parts that are not responsible for handling the error. A straightforward solution is to send the exception notification to its concerning handler. However, we do not know to which extent developers are sending exception notifications between Android components. Studying and analyzing the state of the practice of exception notification in Android will allow us to identify patterns and flaws in real-world applications; drawing this panorama can help developers to construct more reliable, modular and maintainable solutions. For this purpose, we conduct an empirical study that takes 66,099 Android projects and answers: (i) if the project uses exception notification; and (ii) how notification is performed (how signaling and handling code is implemented). We found that 1,327 applications use exception notification, following different practices: 2 for sending notifications and 2 for handling the exceptions. Our study paves the way for constructing better mechanisms for communicating exception notifications in Java-based Android applications.

KEYWORDS

Exception Notification, Android Inter-Component Communication, Mining Software Repositories

ACM Reference Format:
Vladymir L. Bezerra and Lincoln S. Rocha, João Bosco F. Filho, and Fernando A. M. Trinta. 2019. An Empirical Study on Inter-Component Exception Notification in Android Platform. In *XXXIII Brazilian Symposium on Software Engineering (SBES 2019), September 23–27, 2019, Salvador, Brazil.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3350768.3350784

SBES 2019, September 23–27, 2019, Salvador, Brazil

ACM ISBN 978-1-4503-7651-8/19/09…$15.00
https://doi.org/10.1145/3350768.3350784

1 INTRODUCTION

The Android platform became the most popular mobile platform in the market and the number of applications (apps) in its store is high. The platform provides an operating system, a collection of standard apps and an SDK (Software Development Kit) for developers to build new apps using Java or Kotlin programming languages. These apps are made of components that have different natures (i.e., `Activity`, `Service`, `BroadcastReceiver`, and `ContentProvider`), which communicate by passing messages (intents) to each other. These messages are used to invoke remote procedures on different components and carry required parameters to perform a task.

Exception handling is a well-known error recovery approach used to improve software robustness [27]. Most of mainstream programming languages (e.g., Java, C++, and C#) provide built-in facilities to implement exception handling features [4]. The Android platform inherits Java's exception handling model, which provides constructs to structure exception handling code and to propagate exceptions between methods in the reverse order in which the methods are called [14]. Exception handling propagation is an important feature of an exception handling mechanism, which is responsible to redirect the program control flow to a proper handler that will deal with the exceptional situations [3, 12]. In summary, the exception propagation mechanism is responsible to connect the program parts that signals exceptions to program parts devoted to handle those exceptions in an automatic way at runtime.

However, the loosely coupled and asynchronous communication model between Android components is far different from the Java's synchronous method call. Thus, once the Java's exception propagation relies on the method call stack concept, it is not possible to use it to send exceptions between Android components. Based on such limitation, we raise the following question: *Do developers send exceptional events using the message passing mechanism*? If the answer is yes, how do they actually do it? Is it possible to find patterns in the way they design their exception advertising? Which mechanisms do they use? Answering these questions will allow us to master the current status of coding inter-component Exception Notification (EN) in Android apps, furthermore enabling us to design better EN mechanisms to cope with this concern.

On one hand, some previous studies on Android exception handling have focused on ensuring that programmers do not throw any unexpected exception, mining stack traces to reveal bug hazards, search undocumented exceptions of the Android API that could decrease app robustness, and validate exception handling code with tests and static code analysis [1, 8, 15, 19, 30, 31]. On

the other hand, researchers who have studied the message passing mechanism on Android platform has been focused on security risks concerning apps components, the robustness of apps, and the detection of different kind of vulnerabilities [6, 13, 16, 29]. To the best of our knowledge, no previous studies have investigated the combination of Android's message passing and Java's exception handling mechanism to make feasible EN in Android platform.

Developers of medium to large Android applications have to deal with a graph of components that can interact in complex ways. Most importantly, the Java exception handling mechanism is based on call/return model, while inter-component communication model of the Android platform is based on message passing and asynchronous events; *therefore, it is not possible to send exceptional events to components of an android app using only the standard Java exception handling mechanism.* Nevertheless, sometimes, a component in a presence of an exceptional situation must signal this situation to its neighbors. In face of this challenge, developers have to find a way for gluing these two different worlds together, the Java exception handling mechanism and the Android inter-component communication model, in order to send ENs.

The hypotesis driving this study is that, if developers want to send EN to `Components` using only the native constructs available in Java + Android, they need to use *inter-component communication inside exception handling code (*`catch` *blocks)*. We verify that this hypotesis actually happens in practice by observing existing code in major Android projects, as Listing 1 reveals: a `sendBroadcast` inside a `catch` block at line 11 of Android's MMS (Multimedia Messaging Service) native app.

Listing 1: Exception Notification code snippet extracted from MMS native Android App (Source: http://bit.ly/2p4LPzf)

```
1  try {
2    sender.sendMessage(SendingProgressTokenManager.NO_TOKEN);
3    mSending = true;
4  } catch (MmsException e) {
5    Log.e(TAG, "sendFirstQueuedMessage:␣failed␣to␣send␣message␣" +
6          msgUri + ",␣caught␣", e);
7    mSending = false;
8    messageFailedToSend(msgUri,
9          SmsManager.RESULT_ERROR_GENERIC_FAILURE);
10   success = false;
11   // Sending current message fails. Try to send more
12   // pending messages if there is any.
13   sendBroadcast(new Intent(SmsReceiverService.ACTION_SEND_MESSAGE,
14         null, this, SmsReceiver.class));
15 }
```

The main goal of this paper is to know if the usage of inter-component communication inside exception handling code to send EN repeats in the large and, if so, how developers implement their solutions and what kind of patterns emerge from their code. To do so, we conducted an empirical study performed on Android apps stored at the GitHub repository to answer the following questions: **RQ1**. *Do developers use Intents to send Exception Notifications to components of Android applications*?; and **RQ2**. *If so, how is the Exception Notification designed and implemented*?

This study is divided into two stages. First, we used a tool to perform a repository mining on GitHub, and selected only the Android projects stored on it. Next, we chose a subset of the selected applications to conduct a manual inspection. Our analysis is based on the results of the mining and the manual inspection stages. We tracked the method calls responsible for communication between components to get the required evidences to answer RQ1. We found that developers really do EN between components using the Android message passing mechanism (RQ1). In fact, developers are encoding exception objects into primitive types like integers and strings. As expected, our study reveals that the Android platform does not provide an intentional way to send EN between components. The approach taken by the developers to surpass this weakness can decrease the source code maintainability (RQ2). We found that the notification happens more often in classes related to the UI instead of background components (RQ2). Moreover, our study shows that opening external apps to perform specific tasks is a major source of EN (RQ2). We summarize the structure and behavior of the most common ways to notify exceptions found in the study.

Our findings suggest that the way developers sending EN brings well-known error-handling problems to Android apps, such as exception handling code smells and bad practices (e.g., use of error code instead of typed exceptions and use of exception handling mechanism as part of the normal control flow logic) [3, 5]. Thus, our findings can be useful to design and implement a proper and intentional EN mechanism to support developers in this task, avoiding aforementioned problems.

2 BACKGROUND

2.1 Android Overview

Android is an open source platform for mobile devices (e.g., smartphones and tablets) based on Linux operating system. The system consists of a set of native libraries in C/C++, a runtime (Dalvik or Art VM), a framework to build applications (apps), a subset of Java™ or Kotlin programming language and the application layer.

The apps, therefore, are written in Java or Kotlin programming languages. Android apps can share data between them despite the isolated environment. One app interested can ask for permission to access resources and data of another app (e.g., contacts and SMS messages). An Android app is built using building blocks called Android Components. They are entry points for the app itself, they are independent from each other, and they have their own lifecycles. The Android's four core components are described in the following.
Activity: It is the main component of an Android app. They provide a screen through the user can interact. When an `Activity` starts, it is placed on the top of the *activity stack*, becoming the actual running screen. The possible states of an `Activity` are: (i) `Running` or `Active`: if an `Activity` is on the foreground (on top of the stack); (ii) `Paused`: when an Activity loses focus but it is still visible; and (iii) `Stopped`: the `Activity` is completely obscured by another `Activity` but still retains state. Activities in `Paused` or `Stopped` states can be killed by the Android system to release memory.
Service: This component can perform long-running operations in the background. It doesn't provide a user interface and can be started from other components. Services keeps running even when the app is not visible. Services have two forms: bounded and unbounded. One one hand, when *unbounded*, the service runs endlessly even if the calling component is destroyed. On the other hand, when *bounded*, the service creates a client/server interface and remains alive as long as they are bounded to another component.

BroadcastReceiver: It is used to receive broadcast messages, called `Intents`, sent from other components, apps or the Android system itself. The `BroadcastReceiver` follows to the publish-subscribe model, where apps can subscribe to specific broadcasts and publish broadcasts via a `sendBroadcast` method call. The Android system is responsible for routing the message delivering.
ContentProvider: It is responsible for grouping and sharing data between apps, encapsulating data and providing a way to access it.

The `AndroidManifest.xml` file describes essential information about an app to the Android build tools, the Android operating system, and Google Play. This file describes components, packaging name, permissions, API level, and hardware capabilities.

The Android platform provides a message passing system that is used to link apps and components. The `Intent` is a message object that **encapsulates action**, data, component, category, and extra data. An Intent can also be seen as a self-contained object that specifies a remote procedure to apply along with its arguments. Intents can be explicit, when the destination is known at compile time, and implicit, when the recipient is known at *runtime*.

2.2 Java Exception Handling in a Nutshell

When an error occurs in the Java, an exception is raised [11] and the raising of an exception is called *throwing*. In Java, exceptions are represented by objects based on specific classes with their own hierarchy. There are two categories in this hierarchy: checked and unchecked exceptions [14]. Checked exceptions are those which extends, directly or indirectly, the `Exception` class. Unchecked exceptions are those which extends from `RuntimeException` and it's handling is optional. On the other hand, checked exceptions must be handled and they are raised by using the `throw` statement.

When an exception is raised, the execution flow is interrupted and deviated to a specific point where the exceptional condition is handled. In Java, exceptions can be raised using the `throw` statement, signalled using the `throws` statement, and handled in the `try-catch-finally` blocks. The "`throw new E()`" statement is an example of *throwing* the exception `E`. The "`public void m() throws E,Z`" shows how `throws` statement is used in the method declaration to indicate the signalling of exceptions `E` and `Z`.

The `try` block is used to enclose the method calls that might throw an exception. If an exception occurs within the `try` block, that exception is handled by an exception handler associated with it. Handlers are associated to a `try` block by putting a `catch` block after it. A `try` block can be associated with multiples `catch` blocks. Each `catch` block catches a specific exception type and encloses the exception handler code. The `finally` block is optional, but whether declared, it will always execute when the `try` block finishes, even if an exception occurs. The coding of cleanup actions within the `finally` block is recognised as a good practice.

3 RELATED WORK

Minelli and Lanza's [17] study centered on understanding mobile apps from structural and historical perspectives. They focused on three aspects: source code, third-party libs, and historical data to understand the differences between traditional systems and apps concerning libs and code smells. They found that apps tend to be smaller than traditional systems, approximately 2/3 of all method calls were to external APIs; apps almost do not use inheritance; some developers used versioning systems at late stages of development.

Pico et al. [24] gave an overview of mobile computing before and after the year 2000. Before this date devices had low computing power, limited bandwidth, and small screens. Smart devices, on the other hand, resulted in an explosion of software made to them; sensors became cheap and small and ended up embedded into *smartphones*; ubiquitous connection enabled apps to still "always on" and social networks changed the way users interact on the internet. Challenges include mobile privacy and the disappearance of devices, as Mark Weiser [28] predicted. Mobile computing had, therefore, a huge impact on software engineering.

Chin et al. [6] first focused on identifying the security risks behind the Intent communication model of Android. They provided a tool that detects application communication vulnerabilities. Later Maji et al. [16] studied the robustness of Android apps with respect to malformed Intents and found that, in general, apps are vulnerable to this attack. Hay et al. [13] demonstrate that Inter-Application Communication of Android, which enables reuse of functionality across apps, can be used as an attack surface. Payet and Spoto [23] defined an operational semantics for a large part of the Android platform, not only the Dalvik virtual machine as previous works did. This work sets bases for formal analysis of Android apps.

Kechagia and Spinellis [15] studied undocumented runtime exceptions thrown by the Android platform and third-party libraries. Coelho et al. [8] also mined stack traces from issues on GitHub and Google Code looking for bug hazards related to the exception handling code. They detected four bug hazards: (i) "cross-type exception wrapping", (ii) undocumented unchecked exceptions raised by the Android platform and third-party libraries, (iii) Undocumented check exceptions signaled by native C code and (iv) Programming mistakes made by developers. However, none of these studies explore the exception propagation across components using the Android communication model.

Android programs can be very vulnerable to exceptions, Choi and Chang [7] inspected nine programs and found that 51% of Activities have no exception handlers, i.e., they do not have any try-catch block, and this, by the way, results in crashes. They, therefore, propose a mechanism for component-level exception providing a new component API that extends the existing components. Android apps deal with external resources all the time because these resources can be noisy and unreliable. Validating exception handling code is complicated, especially when dealing with external resources, such as wireless connection, GPS, and sensors in general [30].

Bavota et al. [2] study the impact of change- and error-proneness of APIs on user ratings of Android apps. For them, APIs breaking backward compatibility, for example, are harder to use, and therefore, brings instability to the application. Oliveira and colleagues [21] performed an exploratory study of exception handling behavior in Android and Java applications. They conclude that the excessive use of unchecked exceptions decreases the robustness of the Android application.

Queiroz and Coelho [25] performed an exploratory study that analyzed 15 Android apps. This study aims to answer questions about the quality of exception handling code. More than that, they surveyed Android developers to assess the main obstacles to implement exception handling code and what developers do to overcome

these barriers. The main obstacles concerning the implementation of exception handling code cited by the experts were: (i) the incompatibility between the Exception Handling Mechanism and the life cycle of Android components; (ii) communication between tasks (e.g., the impossibility of running I/O tasks on UI thread is a primary source of concern). To deal with this, the developers listed some advices such as: avoid heavy use of Fragments; manage the multitasking environment carefully; make use of error reporting tools; fail fast and; only capture an exception if it can be handled.

4 RESEARCH DESIGN AND METHODOLOGY

In Figure 1 we can see the steps taken in this study, the output of each step and the information flow. The rounded boxes represent the step or stage itself and, the dashed boxes are the output of that particular stage. In the first stage, Definitions and Goals, we define the concept of "Exception Notification". At the same time, we ran the first data extractions with two primary purposes: (i) validate the selection criteria and; (ii) refine the definitions proposed in (i) in a loop. So, in the second stage, called Data Extraction, we ran the queries, based on the selection criteria defined in the two first stages. As output, this step produced several log files in text format.

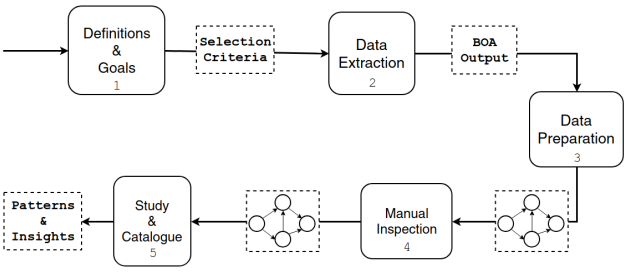

Figure 1: Stages of the study.

In Data Preparation, we decided to use a graph database to model and persist the projects and components found. With the output of automatic extraction, we were able to introduce Projects and Senders to the graph. The tool helped us to visualize and understand the structure of the projects. In the Manual Inspection stage, we could find the Handlers by analyzing the source code of the selected applications, and, finally, complete the graph by inserting and linking these components to their Senders.

Once the graph was completed, we could understand the project's structure, and we identified the emerging patterns. The final output of this study is a collection of structural and code patterns besides a set of insights that can guide future studies.

4.1 Definitions

The definitions presented here will guide this study and are central for its correctness. Definition 1 states what a component[1] means in this study. Definition 2 precisely establishes the notion of exception notification. Components involved in an exception notification can be classified into senders and receivers (Definition 3).

Definition 1 (Component). *A component is a Java class of an Android application, witch can be classified into* ***internal*** *(defined inside the app) and* ***external*** *(defined outside the app).*

Definition 2 (Exception Notification). *Any call of methods* `startActivity` *or* `sendBroadcast` *inside a catch block of any component represents an intention to notify the caught exception to another component.*

Definition 3 (Sender and Handler). *There are two Components in exception notification: sender and handler. The former starts the notification, i.e., catch an exception and, inside the catch block, call a method from the Android communication API. The later handle notification, i.e., it receives the exceptional events sent by a sender.*

4.2 Goal and Research Questions

The primary goal of this article is to assess if and how developers are using Android inter-component communication mechanism as a way to broadcast exception notification between components. By doing so, is it possible to identify this behavior among Android apps? Which patterns of notification can arise in an environment without a proper mechanism to deal with it? Specifically, we are aiming at answering the following research questions:

RQ1. *Do developers use Intents to send Exception Notifications to components of Android apps?*

RQ2. *If so, how is the Exception Notification designed and implemented?* **RQ2.1**. *How are the projects structured?* **RQ2.2**. *How is the code used to send Exception Notifications?* **RQ2.3**. *How is the code used to handle notifications?*

4.3 Data Extraction

Choosing the right data set is important for the success of the study. The data used in our analysis was collected via repository mining. We performed repository mining over Boa [9] dataset, updated on September 2015, and platform because it has a large number of projects, with more than 7 million projects. Moreover, Boa is extensively used by researchers [2] and provides a domain-specific script language for mining source files.

A set of specific scripts (queries) were implemented to extract the right information from the dataset. These scripts were executed several times during this research, and the results were checked manually to guarantee the correctness of the queries. The queries we performed can be summarized in: (i) project have Senders?; (ii) who are the Senders?; (iii) what are the exceptions being encapsulated? and; (iv) what are the types of Components?

4.4 Data Preparation

In this stage of the study, we collected the output files generated by the last phase and performed a set of transformations on it. Although the Boa platform is flexible and allowed us to extract fine-grained information about the projects, the data extracted are not easy to understand. Thus, we transformed the output text files to a

[1]The Android platform uses the term Component for specific classes of its framework used to implement apps. However, from now on, we will use Component as defined in Definition 1.

[2]http://boa.cs.iastate.edu/papers/index.php

graph of components, suited for human reasoning. We choose the graph representation because (i) we wanted to find relationships and patterns between components of Android apps; (ii) we would like to have a graphical description of such projects and; (iii) we wanted to navigate throughout the projects and it's relationships. For this purpose, we choose the Neo4J DBMS [18], once it offers the tools we needed: a way to navigate projects, a persistent storage, and a query language [22]. We followed this set of rules in order to convert between the two formats: every project became a node, every component became a node, and every component must belong to a project. The links, or edges, between the nodes, are: `has` - every project has components, `startActivity` - from one component to another, and `sendBroadcast` - from one component to another.

We selected the top 100 projects ranked by the number of EN to perform the transformation. Once the dataset was created, we selected 30 apps to be inspected manually. This process of manual inspection is cumbersome and we lack resources to do an extensive exploration. The chosen apps satisfied at least one of these characteristics: (i) the app is available in Play Store; (ii) it is a stock app from Google, and; (iii) the app is relevant by the number of commits, contributors and stars.

It is worth mentioning that the projects selected to be manually inspected, as mentioned above, share some unique properties and are pertinent to the research community. First, there are no 'toy projects' among these projects; second, most of the selected projects are or were available in the Google Play Store. Moreover, we found that three apps came from the Android platform itself, including the Camera, the MMS application and the Gallery. Two projects were from the SOCIETIES[3], a project funded by the European Commission that has more than nine thousand commits and forty-two contributors.

4.5 Manual Inspection

In this step of the study, we employed a thorough examination of the source code of the 30 projects previously selected. Even though the mining tool was powerful and flexible, it was not capable of discovering Handlers, i.e., the Components in charge of receiving the Exceptional Notification, due to the loosely coupled nature of the platform. Thus, we had to manually fill the gaps in the graph dataset by introducing new Nodes and edges. The methodological question we answered in this section was: how do we find the Handlers? In short terms we had to follow these steps: (i) locate the Sender file where the EN happens; (ii) in this file, locate the exact catch blocks where EN occurs; (iii) find the specific Intent used in this EN based on ACTION and DATA, and; (iv) discover the Handler for that particular Intent. Once we find the specific Handler, we create a new Node in the graph dataset, and associate the Sender with its Handler by creating an edge between them. Because we have two ways of sending EN to Components, via StartActivity or SendBroadcast, we analyzed them separately.

4.5.1 `startActivity`. For the this method, we also have two cases: the handling Activity might be internal or external. To find the handlers of an internal Activity, we need to recognize the startActivity method call inside the catch block and read the name of the target class. To discover External Activities, on the other hand, we had to be more careful. The Android platform supports the cooperation between apps, providing a set of external apps that can be open. Such apps cover the basic tasks, such as launching the browser, showing a map location, taking a picture, and so on. So, to open these "external apps", we have to call startActivity passing an Intent with the correct ACTION and DATA parameter set. The former, however, is a constant, usually a String type, provided by the vendor of the app or library. So, to find the Handler component for this action, we had to identify which application is responsible for handling these ACTION constants. For example: to open the stock camera we should pass the ACTION `MediaStore.ACTION_IMAGE_CAPTURE`, which is the string `"android.media.action.IMAGE_CAPTURE"`. Using this approach, we identified, classified, and inserted external apps into our dataset.

4.5.2 `sendBroadcast`. In the same sense of the external apps, the method `sendBroadcast` is based only on the `Intent` and its parameters. The Senders and Handlers, therefore, are unaware of the existence of each other. To find these Handlers we had to: (i) identify the Constant that characterize the Intent object; (ii) search for the usage of that Constant in the code; (iii) find the BroadcastReceiver responsible for handling that specific constant, and; (iv) find where this BroadcastReceiver is registered. It is essential to mention that while the external apps also rely on constant Strings to be activated, these constants tend to be embedded into APIs, which makes them stable. On the other hand, the constants used by sendBroadcast are meant to be used only by the application, so the burden of setting up such Strings fell in the developer's shoulders.

4.6 Study and Catalogue

This part of this study was designed to identify, classify, and discuss the structure and implementation of the Exception Notification practice. By observing the Project-Component and Component-Component relations in the resulting graph, we could understand the structure of the EN and, by examining the classes and methods involved in the EN, we could explain the idiosyncrasies related to the source code. We catalog patterns in two categories: (i) Structural Patterns, and; (ii) Code Patterns. Structural Patterns appears when two or more projects exhibit the same structure, regarding Project-Component and Component-Component relations. We observed the project's structure and grouped them by their similarities. The Code Patterns, on the other hand, rises in the structure of the source code. In other words, we aim at discovering similarities the code used to do the Exception Notification, such as caught Exceptions, methods called and Intent creation.

5 RESULTS

5.1 RQ1. Do developers use Intents to send Exception Notifications to components of Android apps?

The results in this subsection came from the automatic extraction stage of the study, i.e., we are considering all projects.

[3] http://www.ict-societies.eu

At least 1,300 apps confirmed our claim; therefore the answer to this question is **yes, developers are doing Exception Notification between components**.

The dataset chosen have at least seven million projects. From this number, around 7% are Java projects. Likewise, 12% of Java projects are Android projects, and 2% of all Android projects presented EN behavior.

We found 2,050 occurrences of Exception Notification (EN) in 1300 projects. From this number, 1,737 are from `startActivity`, which represents 84.73%, and 313 EN are from `sendBroadcast` which corresponds to 15.27% of the total.

From the Components standpoint, we found that 50% of Exception Notification originate in `Activity`. On the other hand, `Services`, another main component in Android apps, corresponds to 8% of notifications. We have identified components such as `Fragments`, with 4%, `Plain Java` classes, with 24% and `Other`, with 14% of the occurrences. We can see the distribution of Component versus EN method in Figure 2.

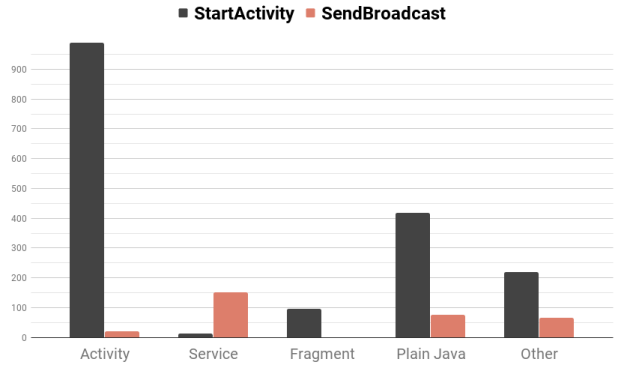

Figure 2: Distribution of method invocation over the categories

From the standpoint of the `Other` components, we observed that they are classes from Android API. In Table 1 we can see the types of classes from the this category. As the reader can note, we have classes related to Android UI API, and also classes associated with asynchronous tasks and concurrency.

Table 1: Components from `Other` category

Other Class	%
`AsyncTask`	20.5%
`OnClickListener`	19.2%
`ViewGroup`	15.4%
`AbstractGetNameTask`	13.4%
`AlertDialog`	12.2%
`java.lang.Thread.UncaughtExceptionHandler`	8.3%
`SurfaceHolder.Callback`	5.7%
`BaseCamera`	5.1%

In the mining stage of this study, in addition to identifying occurrences of EN, we looked for the Exceptions caught in such events

Table 2: Top 20 exceptions caught in Exception Notification

Exception Type	%
`ActivityNotFoundException`	48.10%
`Exception`	22.96%
`UserRecoverableAuthException`	5.50%
`IOException`	3.60%
`UserRecoverableAuthIOException`	3.45%
`JSONException`	2.92%
`Throwable`	1.95%
`NoMediaMountException`	1.70%
`MmsException`	1.36%
`NullPointerException`	1.31%
`NativeDaemonConnectorException`	1.22%
`NameNotFoundException`	1.17%
`FrameAnimationException`	1.12%
`TweenAnimationException`	1.12%
`HttpClientErrorException`	0.68%
`XMLRPCException`	0.68%
`FileNotFoundException`	0.63%
`XMPPException`	0.54%

to get the full picture of the problem. The most common Exceptions caught during EN can be found in Table 2. We can see that `ActivityNotFoundException` appeared more often than any other type. We also noted that instances of `Exception` and `Throwable` were widespread, revealing a careless attitude upon exception handling code [20]. Other exceptions ranged from I/O exceptions, malformed data formats (`JSONException`), user authentication related and exceptions from system services, like `MMSException`.

5.2 RQ2. If so, how is the Exception Notification designed and implemented?

To answer this question, we analyzed three different aspects of projects: (i) structure: how the components are organized; (ii) notification: how the code doing EN is implemented; and (iii) handling: what are the practices used by programmers to receive and handle EN. Thus, we break this generic question into three specific questions, detailing the aspects mentioned above.

5.2.1 RQ2.1: How are the projects structured? From projects manually analyzed, we found that 26.7% are using `sendBroadcast` and 70% are using `startActivity` exclusively, and only 3.3% of the projects use both methods at the same time. We found many similarities in the structure of projects and also identified a clear separation, in terms of structure and code, between `sendBroadcast` and `startActivity` EN.

Android apps are doing Exception Notification to both internal and external components.

Components that did EN via `sendBroadcast` are limited to `Other`, `Service`, `Activity`, and `Plain Java`. There was no `Fragment` using `sendBroadcast` in our findings. Table 3 shows the EN relationships, where the `Senders` are, as said, `Activity`, `Service`, `Plain Java`, and `Other`, and the `Handlers` of this type of notification are $\texttt{Activity}_i$ (internal), $\texttt{Activity}_e$ (external), `Other` and `Fragment`. We did not see, notwithstanding, in our findings, `Service` doing EN to `Service`.

Table 3: Summary of sendBroadcast notification.

From/To	$Activity_i$	$Activity_e$	Fragment	Other
Activity	14.80% ↗	–	–	–
Service	22.22% ↗	–	–	3.70% ↗
Plain Java	3.70% ↗	3.70% ↗	3.70% ↗	–
Other	37.00% ↗	–	–	11.11% ↗

Table 4: Senders doing notification via sendBroadcast

Component Type	% of sendBroadcast occurrences
Other	50%
Service	27%
Activity	11.5%
Plain Java	11.5%

Components sending ENs via startActivity are limited to Activity, Other, Plain Java, and Fragment. Table 5 shows the relationships we found regarding startActivity notifications. We could not find any Service doing EN by this method, which is understandable due to the background nature of this component. An interesting fact is that the Handlers of this type of notification are just $Activity_i$ (internal) and $Activity_e$(external). The reason is that the startActivity method is meant to open a new Activity, so that is why we only have this component as a Handler. $Activity_e$ classes are also present in this context. In Table 6 we can see the distribution of Senders components using the startActivity.

Table 5: Summary of startActivity Exceptioin Notification.

From/To	$Activity_i$	$Activity_e$
Activity	41.38% ↗	34.48% ↗
Fragment	–	4.6% ↗
Plain Java	6.9% ↗	4.6% ↗
Other	1.14% ↗	6.9% ↗

Table 6: Senders doing notification via startActivity

Component Type	% of startActivity occurrences
Activity	79.5%
Other	9%
Plain Java	6.4%
Fragment	5.1%

5.2.2 RQ2.2. How is the code used to send exception notifications? To answer this question, we took into account exception notifications started by calling sendBroadcast and sendBroadcast methods of Android inter-component comunication API.

sendBroadcast notification are directly related to Service components, and hence, associated with background tasks. This pattern is called **Background Exception Notification**.

As said previously, the sendBroadcast type of notification is associated with background tasks, which are, in Android apps, mapped into Service components. The nature of this component is to be decoupled from the component which started it. For this reason, they are often used for "fire and forget" tasks, like playing music, or they can be used for asynchronous tasks. In Listing 2 we can see two examples of Exception Notification in the same code snippet, this code, though, is used to download a file from the internet. Inside the try block, there is an attempt to download a file, and, the first catch block is used to recover from a NoMediaMountException, i.e., the system is not able to save the file to the SD card. The second catch block is responsible for handling the case when the file not in a valid JSON format. We can see the actual Exception Notification in lines 7 and 12, and the intention is to warn another component about the presence of these faults. In this case, the developer chose to encode the two Exceptions (NoMediaMountException and JSONException) into two different constants respectively: BROADCAST_ERROR_NO_SD_CARD and BROADCAST_ERROR_BROKE_FILE. Table 7 shows the exceptions being caught in sendBroadcast notification. If we compare this result with the exceptions in Table 2, we can see that JSONException and NoMediaMountException are present in both lists.

Listing 2: Exception Notification extracted from https://bit.ly/2x4OdVQ

```
1  try {
2      if (!Kernel.getLocalProvider().
3          isImageInitDownloaded( _info.localDir)) {
4          Kernel.getLocalProvider().setDownloadInfo(null);
5      }
6  } catch (final NoMediaMountException e) {
7      e.printStackTrace();
8      sendBroadcast(new Intent(CoreView\texttt{Activity}.
9          BROADCAST_ERROR_NO_SD_CARD));
10     stop();
11     return;
12 } catch (final JSONException e) {
13     e.printStackTrace();
14     sendBroadcast(new Intent(CoreView\texttt{Activity}.
15         BROADCAST_ERROR_BROKEN_FILE));
16     stop();
17     return;
18 }
```

Table 7: Exceptions caught in sendBroadcast notifications

Exception Type
NoMediaMountException
JSONException
XMPPError
CommunicationException
SaxException
IOException
CommunicationException

Activity component, on the other hand, is a rare source of sendBroadcast notification. Only two projects presented this behavior. Although some of these calls originate in Threads instantiated within the Activity.

When the exception notification is performed by the startActivity method, the results are entirely different. As mentioned earlier,

the only possible `Handler` for this type of notification is another `Activity`. `ActivityNotFoundException` was the most often caught exception in this context, that, according to Android documentation[4], the exception is thrown when the call of `startActivity` or one of its variants fails because an Activity cannot be found.

`ActivityNotFoundException` was the most often caught exception when dealing with `startActivity` method call.

Particularly, this `Exception` is thrown when an application is trying to open another app, e.g., one app that needs to open a picture from the file system will try with the stock `Gallery` application from Android. In other words, the developer cannot be sure that his request to open a third party app will be fulfilled.

The interaction between Android apps can lead to a **pattern** called **nested `try-catch` blocks**.

By crafting specific Intents, developers can start other apps present in the system. However, if the system cannot find that specific Activity, an ActivityNotFoundException is thrown, forcing developers to surround these calls by a try-catch block, as mentioned above. We found that 43% of the projects analyzed presented this programming practice. For this reason, we considered this practice a **code pattern**.

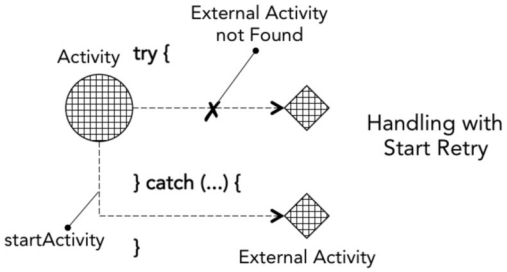

Figure 3: `startActivity` retry.

5.2.3 RQ2.3. How is the code used to handle exceptions? We observe that `Activities` (internal or external) are *the* handler of almost all EN in this study. They are accountable for handling exceptions in 94.74% of the time. Furthermore, for `startActivity` notification, they represent 100% of handling.

The central exception handling component are `Activities` (internal or external), responsible for 94.74% of all handlers.

In the case of `sendBroadcast`, the code used to handle exceptions must be enclosed into a `BroadcastReceiver`, component responsible for handle the broadcast messages (Intents). We could verify that `BroadcastReceivers` are registered, usually, inside `Activity` components. In a general form, the code inside `onReceive` must handle a variety of cases, dealing with all the possible messages using `if-elseif-else` statements. In Listing 3 we can see an example of a real application that illustrates this problem. The method handling the error (line 10 of the Listing 3) is responsible for show the error to user, using an `AlertDialog`. Other common options are: doing nothing, log the error or release resources in background. This is a common practice and will be considered a **handling code pattern**.

Listing 3: Exception handling extracted from https://bit.ly/2U4T450

```
1  private BroadcastReceiver bcastReceiver = new BroadcastReceiver()
2  {
3    @Override
4    public void onReceive(Context, Intent intent)
5    {
6      String action = intent.getAction();
7      if (action.equals(REFRESH_MENU_SCREEN_ACTION))
8        onRefreshMenuScreen(intent);
9      else if (action.equals(SHOW_ERROR_MESSAGE_ACTION))
10       onShowErrorMEssage(intent);
11     else if (action.equals(SWITCH_CAFETERIA_ACTION))
12       onSwitchCafeteria(intent);
13     else if (action.equals(SHOW_CAFETERIA_LIST_ACTION))
14       onShowCafeteriaListAction();
15   }
16 };
```

Concerning structure, something we found interesting was the *Centralized Handler* (see Figure 4). Usually, this lonely handler is an `Activity` that provides a way to deal with ENs (e.g., redirecting user to settings or restart the app in one specific screen). We found that this relationship is very common and should be considered a **structural pattern**.

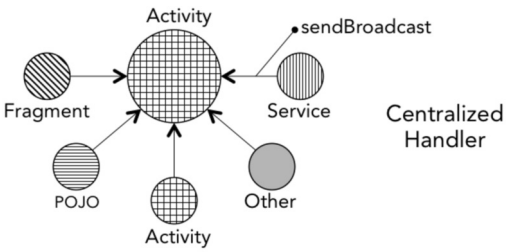

Figure 4: One `Activity` to handle all Notifications.

6 DISCUSSION

In this section, we discuss our findings and how they affect the development of Android applications. The definition of EN itself is interesting because it involves two different concepts, the Exception Handling mechanism, based on try-catch blocks, and inter-component communication, based on pub/sub, specific to the Android platform. If on the one hand, catch blocks are used to handle faulty behavior by calling methods that deal with these errors locally, on the other hand, the calling of event handler methods as a consequence of a catch block execution means that Handlers are spread throughout the application, due to the loosely coupled architecture of Android. When these two mechanisms are tied together, though, the Exception is handled in a different context. The combination of these two strategies hinders readability and modifiability because part of the exception handler is not always explicit in the code.

For example, in this project[5], a document reader, we can see a component responsible for downloading some content from the

[4] https://developer.android.com/reference/android/content/ActivityNotFoundException

[5] https://bit.ly/2x4OdVQ

Internet. It is using `sendBroadcast` to publish different types of events, and it uses `String` values to encode information. In this one[6], an IRC client, we see a background component that handles the client's connectivity with the server. The Exception Notification, in this case, occurs from within a thread, but it works in the same way as the previous example. Another example of this pattern can be seen in the MMS[7] app from Android. This component is in charge of sending messages in the background, and it uses the same constructs from previous examples. All the examples above share the same properties: (i) the Senders are Service components in charge of some background task; (ii) in the presence of some errors, these components publish events about exceptional information, and; (iii) they encode exceptions to `String` values. Other examples of this pattern can be seen here.

Background tasks are widespread in mobile development, and they are often used to execute pieces of code asynchronously. Tasks like playing audio, downloading contents from the Internet, and read or write a file, are in general, given to `Service` components. For this reason, they play a vital role in the development of a robust Android application. As they are usually dealing with tasks involving `I/O`, they have to deal with potential errors. Further, as they communicate by publishing events, to "propagate" these errors, they tend to encode exceptional information into constant values. These constants are scattered throughout the code base, and this reduces the ability to reason about it. An interesting fact is that Exception instances can be embedded into Intent messages in any case.

Although not present in the final data set, we could find, in the preliminary exploratory analysis, some projects that embed Exception instances into `Intents`. In this case, the `Sender`[8], between lines 171 and 180, embed two different exceptions objects (`IOException` and `JSONException`). On the other side, the `Handler`[9], extract the `Exception` from `Intent` at line 65 and show a Toast[10] message with information about the error to the user. In this example[11], at line 207, an instance of `Throwable` is embedded into the `Intent`, and, in the `Handler`, between lines 639 to 653, the same object is extracted and thrown again. We see that at least two different techniques are employed in order to communicate and handle internal errors, the first one is using constant values based on `Strings`, which is a weak manner to ensure correct handling, and the second way is to embed the caught `Exception` into the message as metadata. In the latter case, the `Exception` must be thrown manually, which is cumbersome.

Another finding interesting to mention is related to the Activity component. First, because it is the component that interacts with the user, and second because it is the Handler of EN in 94% of the time, implying that developers keep the error handling as close as possible to the UI. Also, our findings suggest that interaction between apps, i.e., starting an Activity in another application, produces what we called the nested try-catch pattern. This pattern happens very often, and the reason is that application interaction is essential to provide a good user experience. Moreover, this pattern shows that this interaction occurs in a very loosely coupled way by design, because it is based only on Intent extra data. The external apps we found in our analysis was: PlayStore, Browsers, File Managers, Settings, Video players, and Maps. Examples can be seen here[12][13][14].

The work presented here can help Android developers, that have to deal with the limitations of the platform, to build robust apps, informing them about the problems involving Exception Notification. Also, researchers can benefit from the results presented here, either by the patterns founded by this work or to provide solutions to the emerging problems. Finally, the developers of Android operating system can use this work to understand the limitations of the platform, and they can build solutions to overcome the problems in order to provide a better experience for apps developers.

7 THREATS TO VALIDITY

In this section, we discuss the threats to validity associated with our investigation using the Shadish *et al.* [26] classification (construct, internal, conclusion, and external validity).

Construct Validity. We constructed this study based on the concepts defined in Section 4.1. We assume as a premise that the calls of `startActivity` and `sendBroadcast` inside `catch` blocks reflect the developer's intention to send an EN, the caught one, for instance. However, if this assumption is false, the construct validity is threatened, putting at risk the validity of our findings. We tackle this risk by inspecting the repositories manually, by doing that we were able to show that the developers embodied information about exceptions into `Intent` message objects and sending them to other components, this confirms our central claim and reduce this threat.

Internal Validity. We stated that the lack of a native mechanism for exception notification could lead to a set of patterns in structure and code. However, one can question in which extent these patterns are resultant from the absence of an embedded propagation mechanism. Thus, we believe that the patterns presented in this study essentially originate from mentioned deficiency. The passage of time does not influence the outcome of the research, to decrease this threat we performed the same queries at different times. Our research design, therefore, addresses this threat.

Conclusion Validity. The reliability of the measures is a major concern in empirical studies based on software repository mining. We, on the other hand, developed this study based on objective measures (e.g., the name of exception classes and the number of projects doing EN). This procedure, in some sense, deals with the data reliability. We did not see random irrelevancies that could disturb the result. Furthermore, there is no evidence that the heterogeneity of population harms the contributions. On the contrary, the more diverse is the population, more patterns will result.

External Validity. There may be a concern regarding the generalizability of this study. If the selection criteria cannot capture a representative group of apps, we have a threat. Instead, the apps came from several domains like native Camera app, apps published on Google PlayStore[15] with thousands of downloads, apps from

[6] https://bit.ly/2YQHpcU
[7] https://bit.ly/32sDGVl
[8] https://bit.ly/2YTRfuG
[9] https://bit.ly/2Y9JnYG
[10] https://developer.android.com/reference/kotlin/android/widget/Toast?hl=en
[11] https://bit.ly/2Jxiq9o
[12] https://bit.ly/2LnUk2V
[13] https://bit.ly/2XTbxmP
[14] https://bit.ly/2LnUlUx
[15] http://play.google.com

secondary application stores, open source, and funded projects[16]. Another hazard is the outdated dataset, which can turn this study weaker. However, we do not limit ourselves only to the mining output and this, somewhat, alleviate this threat. We demonstrate that this study is realistic, and thus, this threat is mitigated.

8 FINAL REMARKS

In this paper, we addressed the problem of sending Exception Notifications (EN) between Android components and what are the results apps' code. This problem origins from the contrast between the Java exception handling mechanism and the Android message passing style of communication. We demonstrated that the combination of these two different programming practices could be used as a way to send EN. The central goal of this research is to determine if Android's apps are communicating these EN to other components, and how they design and implement their solutions.

The methodology used to support this research was empirical by nature. We performed an exploratory study on a dataset of projects we mined from the Boa Dataset [10], in order to identify projects employing the EN pattern, moreover, to see if this behavior happens in the "real world", and to recognize and describe the patterns in design and code. Then, we selected a subset of these projects to be manually analyzed in order to comprehend the decisions made by developers when facing the EN problem, both in design and implementation. Lastly, we described the way we catalog the patterns that appeared during this study.

This research showed that, indeed, Android developers are sending ENs between components of apps. Furthermore, we showed that this fact has a significant impact on the source code of apps. Our results indicate that at least 1000 projects presented this characteristic, the majority of notifications happened in UI related components, such as `Activity` and `Fragment` but background components like Services are very often used to do EN. We discovered several relationships in the graph of Components that later became patterns and programming practices that reduce the quality of the code. Our contributions are: (i) we shed light on a phenomenon that relates the Java exception handling mechanism to the Android communication API; (ii) we demonstrated that the EN pattern could drive the adoption of a style that decreases the quality of the app code; (iii) we presented "real world" apps that send ENs, including apps from Android itself; and (iv) we discuss a framework to solve this problem. Finnaly, we believe this paper paves the way for Android apps and library developers to build and use a proper mechanism to send EN between components.

REFERENCES

[1] G. Bavota, M. Linares-VÃąsquez, C. E. Bernal-CÃąrdenas, M. D. Penta, R. Oliveto, and D. Poshyvanyk. 2015. The Impact of API Change- and Fault-Proneness on the User Ratings of Android Apps. *IEEE Transactions on Software Engineering* 41, 4 (April 2015), 384–407.

[2] G. Bavota, M. Linares-VÃąsquez, C. E. Bernal-CÃąrdenas, M. D. Penta, R. Oliveto, and D. Poshyvanyk. 2015. The Impact of API Change- and Fault-Proneness on the User Ratings of Android Apps. *IEEE Transactions on Software Engineering* 41, 4 (April 2015), 384–407.

[3] Peter A. Buhr and W. Y. Russell Mok. 2000. Advanced Exception Handling Mechanisms. *IEEE Transactions on Software Engineering* 26 (September 2000), 820–836. Issue 9.

[4] Nélio Cacho, Thiago César, Thomas Filipe, Eliezio Soares, Arthur Cassio, Rafael Souza, Israel Garcia, Eiji Adachi Barbosa, and Alessandro Garcia. 2014. Trading Robustness for Maintainability: An Empirical Study of Evolving C# Programs. In *Proceedings of the 36th International Conference on Software Engineering (ICSE 2014)*. 584–595.

[5] Chien-Tsun Chen, Yu Chin Cheng, Chin-Yun Hsieh, and I-Lang Wu. 2009. Exception Handling Refactorings: Directed by Goals and Driven by Bug Fixing. *Journal of Systems and Software* 82, 2 (2009), 333–345.

[6] Erika Chin, Adrienne Porter Felt, Kate Greenwood, and David Wagner. 2011. Analyzing Inter-application Communication in Android. In *Proceedings of the 9th International Conference on Mobile Systems, Applications, and Services (MobiSys '11)*. ACM, New York, NY, USA, 239–252.

[7] Kwanghoon Choi and Byeong-Mo Chang. 2015. A Lightweight Approach to Component-level Exception Mechanism for Robust Android Apps. *Comput. Lang. Syst. Struct.* 44, PC (Dec. 2015), 283–298.

[8] Roberta Coelho, Lucas Almeida, Georgios Gousios, Arie van Deursen, and Christoph Treude. 2016. Exception handling bug hazards in Android. *Empirical Software Engineering* (2016), 1–41. https://doi.org/10.1007/s10664-016-9443-7 DisponÃıvel em: http://dx.doi.org/10.1007/s10664-016-9443-7. Acesso em: 01 nov. 2017.

[9] Robert Dyer, Hoan Anh Nguyen, Hridesh Rajan, and Tien N. Nguyen. 2013. Boa: A Language and Infrastructure for Analyzing Ultra-Large-Scale Software Repositories. In *Proceedings of the 35th International Conference on Software Engineering (ICSE'13)*. 422–431.

[10] Robert Dyer, Hridesh Rajan, and Tien N. Nguyen. 2013. Declarative Visitors to Ease Fine-grained Source Code Mining with Full History on Billions of AST Nodes. In *Proceedings of the 12th International Conference on Generative Programming: Concepts & Experiences (GPCE)*. 23–32.

[11] Raymond Gallardo, Scott Hommel, Sowmya Kannan, Joni Gordon, and Sharon Biocca Zakhour. 2014. *The Java Tutorial: A Short Course on the Basics* (6th ed.). Addison-Wesley Professional. 864 pages.

[12] Alessandro F Garcia, Cecília M.F Rubira, Alexander Romanovsky, and Jie Xu. 2001. A Comparative Study of Exception Handling Mechanisms for Building Dependable Object-Oriented Software. *Journal of Systems and Software* 59, 2 (2001), 197–222.

[13] Roee Hay, Omer Tripp, and Marco Pistoia. 2015. Dynamic Detection of Inter-application Communication Vulnerabilities in Android. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis (ISSTA 2015)*. ACM, New York, NY, USA, 118–128. https://doi.org/10.1145/2771783.2771800 DisponÃıvel em: http://doi.acm.org/10.1145/2771783.2771800. Acesso em: 01 nov. 2017.

[14] Jakob Jenkov. 2013. *Java Exception Handling* (1nd ed.). Amazon Kindle.

[15] Maria Kechagia and Diomidis Spinellis. 2014. Undocumented and Unchecked: Exceptions That Spell Trouble. In *Proceedings of the 11th Working Conference on Mining Software Repositories (MSR 2014)*. ACM, New York, NY, USA, 312–315.

[16] Amiya K. Maji, Fahad A. Arshad, Saurabh Bagchi, and Jan S. Rellermeyer. 2012. An Empirical Study of the Robustness of Inter-component Communication in Android. In *Proceedings of the 2012 42Nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN) (DSN '12)*. IEEE Computer Society, Washington, DC, USA, 1–12.

[17] Roberto Minelli and Michele Lanza. 2013. Software Analytics for Mobile Applications–Insights & Lessons Learned. In *Proceedings of the 2013 17th European Conference on Software Maintenance and Reengineering (CSMR '13)*. IEEE, Washington, DC, USA, 144–153.

[18] Neo4j. 2012. Neo4j - The WorldâĂŹs Leading Graph Database. DisponÃıvel em: http://neo4j.org. Acesso em: 01 nov. 2017.

[19] Damien Octeau, Patrick McDaniel, Somesh Jha, Alexandre Bartel, Eric Bodden, Jacques Klein, and Yves Le Traon. 2013. Effective Inter-component Communication Mapping in Android with Epicc: An Essential Step Towards Holistic Security Analysis. In *Proceedings of the 22Nd USENIX Conference on Security (SEC'13)*. USENIX Association, Berkeley, CA, USA, 543–558.

[20] Juliana Oliveira, Deise Borges, Thaisa Silva, Nelio Cacho, and Fernando Castor. 2018. Do android developers neglect error handling? a maintenance-Centric study on the relationship between android abstractions and uncaught exceptions. *Journal of Systems and Software* 136 (2018), 1 – 18. https://doi.org/10.1016/j.jss.2017.10.032

[21] Juliana Oliveira, Nelio Cacho, Deise Borges, Thaisa Silva, and Fernando Castor. 2016. An Exploratory Study of Exception Handling Behavior in Evolving Android and Java Applications. In *Proceedings of the 30th Brazilian Symposium on Software Engineering (SBES '16)*. ACM, New York, NY, USA, 23–32. https://doi.org/10.1145/2973839.2973843 DisponÃıvel em: http://doi.acm.org/10.1145/2973839.2973843. Acesso em: 01 nov. 2017.

[22] Onofrio Panzarino. 2014. *Learning Cypher*. Packt Publishing.

[23] Etienne Payet and Fausto Spoto. 2014. An Operational Semantics for Android Activities. In *Proceedings of the ACM SIGPLAN 2014 Workshop on Partial Evaluation and Program Manipulation (PEPM '14)*. ACM, New York, NY, USA, 121–132.

[24] Gian Pietro Picco, Christine Julien, Amy L. Murphy, Mirco Musolesi, and Gruia-Catalin Roman. 2014. Software Engineering for Mobility: Reflecting on the Past,

[16] http://www.ict-societies.eu/

Peering into the Future. In *Proceedings of the on Future of Software Engineering (FOSE 2014)*. ACM, New York, NY, USA, 13–28.

[25] Francisco Diogo Queiroz and Roberta Coelho. 2016. Characterizing the Exception Handling Code of Android Apps. In *2016 X Brazilian Symposium on Software Components, Architectures and Reuse, SBCARS 2016, Maringá, Brazil, September 19-20, 2016*. 131–140. https://doi.org/10.1109/SBCARS.2016.25 DisponÃŋvel em: https://doi.org/10.1109/SBCARS.2016.25. Acesso em: 01 nov. 2017.

[26] William R. Shadish, Thomas D. Cook, and Donald T. Campbell. 2002. *Experimental and Quasi-experimental Designs for Generalized Causal Inference*. Houghton Mifflin Co.

[27] Ali Shahrokni and Robert Feldt. 2013. A Systematic Review of Software Robustness. *Information and Software Technology* 55, 1 (Jan. 2013), 1–17.

[28] Mark Weiser. 1991. The Computer for the 21 st Century. *Scientific american* 265, 3 (1991), 94–105.

[29] Hui Ye, Shaoyin Cheng, Lanbo Zhang, and Fan Jiang. 2013. DroidFuzzer: Fuzzing the Android Apps with Intent-Filter Tag. In *Proceedings of International Conference on Advances in Mobile Computing & Multimedia (MoMM'13)*. ACM, New York, NY, USA, Article 68, 7 pages.

[30] Pingyu Zhang and Sebastian Elbaum. 2014. Amplifying Tests to Validate Exception Handling Code: An Extended Study in the Mobile Application Domain. *ACM Trans. Softw. Eng. Methodol.* 23, 4, Article 32 (Sept. 2014), 28 pages.

[31] ÃĽtienne Payet and Fausto Spoto. 2012. Static analysis of Android programs. *Information and Software Technology* 54, 11 (2012), 1192–1201.