UNIVERSIDADE FEDERAL DO CEARÁ

CENTRO DE CIÊNCIAS

DEPARTAMENTO DE COMPUTAÇÃO

PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

VLADYMIR DE LIMA BEZERRA

AN EMPIRICAL STUDY ON INTER-COMPONENT EXCEPTION NOTIFICATION IN ANDROID PLATFORM

FORTALEZA

2017

VLADYMIR DE LIMA BEZERRA

AN EMPIRICAL STUDY ON INTER-COMPONENT EXCEPTION NOTIFICATION IN ANDROID PLATFORM

Dissertação apresentada ao Curso de do Programa de Pós-Graduação em Ciência da Computação do Centro de Ciências da Universidade Federal do Ceará, como requisito parcial à obtenção do título de mestre em Ciências da Computação. Área de Concentração: Engenharia de Software

Orientador: Prof. Dr. Fernando Antonio Mota Trinta

Orientador: Prof. Dr. Lincoln Souza Rocha

FORTALEZA

2017

Dados Internacionais de Catalogação na Publicação
Universidade Federal do Ceará
Biblioteca Universitária
Gerada automaticamente pelo módulo Catalog, mediante os dados fornecidos pelo(a) autor(a)

B469a Bezerra, Vladymir de Lima.
An Empirical Study on Inter-Component Exception Notification in Android Platform / Vladymir de Lima Bezerra. – 2017.
70 f. : il. color.

Dissertação (mestrado) – Universidade Federal do Ceará, Centro de Ciências, Programa de Pós-Graduação em Ciência da Computação, Fortaleza, 2017.
Orientação: Prof. Dr. Fernando Antonio Mota Trinta.
Coorientação: Prof. Dr. Lincoln Souza Rocha.

1. Exception Notification. 2. Android Inter-Component Communication. 3. Mining Software Repository. I. Título.

CDD 005

VLADYMIR DE LIMA BEZERRA

AN EMPIRICAL STUDY ON INTER-COMPONENT EXCEPTION NOTIFICATION IN ANDROID PLATFORM

Dissertação apresentada ao Curso de do Programa de Pós-Graduação em Ciência da Computação do Centro de Ciências da Universidade Federal do Ceará, como requisito parcial à obtenção do título de mestre em Ciências da Computação. Área de Concentração: Engenharia de Software

Aprovada em: 30 de Novembro de 2017

BANCA EXAMINADORA

Prof. Dr. Fernando Antonio Mota Trinta (Orientador)
Universidade Federal do Ceará (UFC)

Prof. Dr. Lincoln Souza Rocha (Orientador)
Universidade Federal do Ceará (UFC)

Prof(a). Dra. Roberta de Souza Coelho
Universidade Federal do Rio Grande do Norte (UFRN)

Prof. Dr. João Bosco Ferreira Filho
Universidade Federal do Ceará (UFC)

Aos meus pais, que sempre me apoiaram e me deram a liberdade de escolher meu caminho. À minha esposa por ter suportado e me apoiado incondicionalmente em todos os momentos difíceis.

ACKNOWLEDGEMENTS

First, I am very grateful to Professor Lincoln Rocha for guiding and trusting me. Thank you for sharing your wisdom with me and giving me support in difficult times.

I also thank Professor Fernando Trinta for making this dissertation possible, your advice was fundamental. I thank Professor Rossana Andrade, chair of the Computer Networks and Software Engineering (GREaT), where this dissertation was produced.

I am very grateful to Professor João Bosco Ferreira Filho for giving structure to this work, for your advices, contribution and support.

I am very grateful to the friends who shared lessons, presentations, coffees and studies during this journey. Among them are Italo Linhares, Rainara Carvalho, Paulo Artur, Anderson Almada and Lana Mesquita.

I would like to thank all the teachers and staff of MDCC and GREat, especially Janaína, Darilu and Hudson, Gláucia Mota and Jonatas Martins.

I thank my parents, Manuel Bezerra and Vera Bezerra, and my wife Cristina Amorim, who gave me support in the emotional aspect. I am also grateful to my sisters Tatiana Bezerra and, specially, Marilia Bezerra for being a safe haven.

I am very grateful to Cristina for your love, for all the patience and for standing by my side even in the hardest times.

I thank my colleagues from Federal Institute of Ceará, specially to Professor Alexandre Oliveira, Vanessa Almeida, Danielle Queiroz and Lourdes Neta, who supported me in their own ways.

This study was financed in part by the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - Finance Code 001.

"There is only one corner of the universe you can be certain of improving, and that's your own self."

(Aldous Huxley)

RESUMO

Desenvolvedores de aplicações Android usam tratamento de exceções para aumentarem a robustez de suas aplicações. A arquitetura do Android, juntamente com o paradigma orientado a objetos aumentam a complexidade do tratamento de exceções; diferentes componentes se comunicam entre si e exceções podem ser levantadas em partes do código que não são responsáveis por tratar tais erros. Uma solução direta é enviar uma notificação de exceção para um tratador responsável. Entretanto, nós não sabemos em que medida os desenvolvedores Android estão enviando tais notificações de exceções entre os componentes. Investigar e analisar o envio de notificações de exceções no Android, no seu estado da prática, nos possibilitará identificar padrões e falhas em aplicações reais; ao desenhar este panorama, estaremos ajudando os desenvolvedores Android a construírem aplicações mais confiáveis, modulares e manuteníveis. Com esse propósito, nós conduzimos um estudo empírico com 66,099 projetos Android e respondemos: (i) se os projetos estão enviando notificações de exceção; (ii) como essa notificação é realizada. Nós encontramos cerca de 1,327 aplicações que lançam mão dessa estratégia e que seguem diferentes padrões: 12 para notificação e 3 para tratamento de exceções. Nosso estudo abre caminho para a construção de melhores mecanismos para a comunicação de notificações de exceções na plataforma Android.

Palavras-chave: Notificação de Exceção. Comunicação Intra-Componente do Android. Mineração de Repositórios de Software.

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 patterns: 12 for sending notifications and 3 for handling the exceptions. Our study paves the way for constructing better mechanisms for communicating exception notifications in Android.

Keywords: Exception Notification. Android Inter-Component Communication. Mining Software Repositories.

FIGURES LIST

Figure 1 – Activity Lifecycle. 20
Figure 2 – Service Lifecycle. 21
Figure 3 – The method call stack and the search for the exception handler. 23
Figure 4 – Example of `try-catch-finally`. 24
Figure 5 – Stages of the study. 35
Figure 6 – Exception notification between Sender and Handler. 38
Figure 7 – Example of a graph in Neo4J. 41
Figure 8 – Exception notification graph. 41
Figure 9 – Distribution of method invocation over the categories 46
Figure 10 – `sendBroadcast` notification patterns 49
Figure 11 – `startActivity` notification patterns 50
Figure 12 – One `Activity` to handle all Notifications 54
Figure 13 – `startActivity` retry 57

LIST OF TABLES

Table 1 – A short list of Android Intent-sending methods 23
Table 2 – Projects that do exception notification 43
Table 3 – Number of Projects 46
Table 4 – Components from `Other` category 47
Table 5 – Common exceptions caught in Exception Notification 47
Table 6 – Senders doing Exception Notification via `sendBroadcast` 49
Table 7 – Components doing Exception Notification via `startActivity` 50
Table 8 – Exceptions caught in `sendBroadcast` EN 52

LIST OF ACRONYMS

API	*Application Programming Interface*
EHM	*Exception Handling Mechanism*
EN	*Exception Notification*
GPS	*Global Positioning System*
IAC	*Inter-Application Communication*
LOC	*Lines of Code*
PDA	*Personal Digital Assistant*
UI	*User Interface*
WAP	*Wireless Application Protocol*

CONTENTS

1 INTRODUCTION 14
1.1 Context and Motivation 14
1.2 Problem Statement 15
1.3 Goal and Research Questions 16
1.4 Contributions 17
1.5 Structure of the Dissertation 18
2 BACKGROUND 19
2.1 Android Architecture 19
2.1.1 Android Components 20
2.1.2 Intents 22
2.2 Java Exception Handling 22
2.3 Other Approach for Error Handling 25
2.4 Boa Language and Infrastructure 26
2.5 Final Considerations 27
3 RELATED WORK 28
3.1 General Studies on Mobile Platform 28
3.2 Studying the Android Communication Model 29
3.3 Studying the Exception Handling Mechanism in Android 30
3.4 Final Considerations 33
4 EMPIRICAL STUDY 35
4.1 Research Design and Methodology 35
4.2 Definitions 36
4.3 Goal and Research Questions 37
4.4 Data Extraction 38
4.5 Data Preparation 40
4.6 Manual Inspection 42
4.7 Study and Catalogue 45
4.8 Results 45
4.9 RQ1: Do developers use Intents to send Exception Notifications to components of Android applications? 45
4.10 RQ2: If so, how is the Exception Notification implemented/designed? . . 48

4.10.1 ***RQ2a: How are the projects structured?*** . . . 48
4.10.2 ***RQ2b: how is the code used to send exception notifications?*** . . . 51
4.10.2.1 *SendBroadcast* . . . *51*
4.10.2.2 *StartActivity* . . . *52*
4.10.3 ***RQ2c: How is the code used to handle exceptions?*** . . . 53
4.11 **Discussion** . . . 54
4.11.1 ***Insights*** . . . 55
4.11.2 ***Bad Practices*** . . . 59
4.11.3 ***Solving the Problem*** . . . 60
4.12 **Threats to Validity** . . . 61
4.13 **Final Considerations** . . . 62
5 **CONCLUSIONS AND FUTURE WORK** . . . 64
5.1 **Overview** . . . 64
5.2 **Limitations** . . . 65
5.3 **Future Works** . . . 65
REFERENCES . . . 67

1 INTRODUCTION

This master dissertation reports an empirical study regarding the exception notification between components in Android platform. Our findings reveal the way developers are dealing with inter-component exception notification in Android platform and how it impacts on the source code structure and behavior. In this Chapter, we first introduce the context and motivation for this study (Section 1.1) and the problem statement (Section 1.2). Next, we states the goal and research questions (Section 1.3) and main contributions (Section 1.4). Finally, Section 1.5 presents the structure of this document.

1.1 Context and Motivation

The Android platform became the most popular mobile platform in the market and the number of 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 applications using Java programming language. These applications are made of components that have different natures (i.e., `Activity`, `Service`, `BroadcastReceiver`, and `ContentProvider`). Such components 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 (SHAHROKNI; FELDT, 2013). Most of mainstream programming languages (e.g., Java, C++, and C#) provide built-in facilities to implement exception handling features (CACHO *et al.*, 2014). 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 (JENKOV, 2013). 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 (BUHR; MOK, 2000; GARCIA *et al.*, 2001). 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 notification in Android apps, furthermore enabling us to design better notification mechanisms.

On one hand, some previous studies on Android exception handling have been focused on ensuring that programmers do not throw any unexpected exception, mining stack traces to reveal bug hazards, search undocumented exceptions of the Android *Application Programming Interface* (API) that could decrease app robustness, and validate exception handling code with tests and static code analysis (PAYET; SPOTO, 2012; BAVOTA *et al.*, 2015a; OCTEAU *et al.*, 2013; KECHAGIA; SPINELLIS, 2014; COELHO *et al.*, 2016; ZHANG; ELBAUM, 2014). On the other hand, researchers who have studied the message passing mechanism on Android platform have been focused on security risks in applications components, robustness of applications, and detection of vulnerabilities (CHIN *et al.*, 2011; MAJI *et al.*, 2012; YE *et al.*, 2013; HAY *et al.*, 2015). 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 exceptional behaviour notification in Android platform.

1.2 Problem Statement

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 exception notifications.

The hypotesis driving this study is that, if developers want to send exception notifica-

tion 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 evidences: a `sendBroadcast` inside a `catch` block at line 11 of the very MMS (Multimedia Messaging Service) of the Android plattform[1].

Listing 1 – MMS Exception Notification Code Snippet extracted from <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 " + msgUri + ",
        caught ", e);
6   mSending = false;
7   messageFailedToSend(msgUri, SmsManager.RESULT_ERROR_GENERIC_FAILURE);
8   success = false;
9   // Sending current message fails. Try to send more pending messages
10  //  if there is any.
11  sendBroadcast(new Intent(SmsReceiverService.ACTION_SEND_MESSAGE, null, this
        , SmsReceiver.class));
12 }
```

1.3 Goal and Research Questions

The main goal of this dissertation is to know if the usage of inter-component communication inside exception handling code to send exceptional behaviour 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 research questions:

- **RQ1**. *RQ1: Do developers use Intents to send Exception Notifications to components of Android applications?*?
- **RQ2**. *If so, how is the Exception Notification designed and implemented*?

This study is divided into two stages. First, a tool to perform the repository mining was used and selected only the Android projects stored in the GitHub. Next, we choose a subset of the selected applications to conduct a manual inspection. Our analysis is based on the results

[1] <http://bit.ly/2p4LPzf>

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 exception notification between components using the Android message passing mechanism (RQ1). In fact, developers are marshaling 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 exception notifications 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 *User Interface* (UI) instead of background components (RQ2). Moreover, our study shows that opening external apps to perform specific tasks is a major source of exception notification (RQ2). We summarize the structure and behavior of the most common ways to notify exceptions found in the study.

1.4 Contributions

The main contributions of this dissertation are:

- **CT1**. *The study report showing the state of practice on how Android developers use the message passing mechanism to send exceptional information* - we found several applications sending exceptional information to another components;
- **CT2**. *The study reveals and classify the traits of applications' source code* - we manually inspected the code of dozens programs and analyzed the exception notification/handling code, identifying its characteristics and flaws;
- **CT3**. *The study shed light on the design decisions adopted by Android developers to implement the exceptional notification* - applications sending exception notification presented a set of patterns in their structure, i.e., what are the components and how they communicate. More than that, we analyzed what programming language constructs are used to implement the solution.
- **CT4**. *We pave the way to design a new exception notification mechanism* - we give an in-depth report on Android applications that use the message passing mechanism to send exceptional events. We describe the structure of apps in depth and we provide a set of insights that can guide developers and researchers.

Our findings suggest that the way developers sending exceptional information brings well-known exception code 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) (BUHR; MOK, 2000; CHEN *et al.*, 2009). Thus, our findings can be useful to design and implement a proper and intentional exception notification mechanism to support developers and avoid such problems.

1.5 Structure of the Dissertation

The remainder of the dissertation is organized as follows. Chapter 2 presents the background and motivations behind this study. In Chapter 3 we describe relevant studies related to this dissertation. Chapter 4 provides the research methodology in details. Besides that, it shows the results we have found and describes a set of insights and discussion about the results. Chapter 5 concludes this report and present future works.

2 BACKGROUND

In this chapter, we present the background and motivations that serves as the inspiration for this study. In Section 2.1 we present the Android Architecture in details. In Section 2.2 we describe the Java exception handling mechanism and, in Section 2.3, we discuss other approaches to handling errors. Section 2.4 we discuss the tools we used to perform this study.

2.1 Android Architecture

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, a subset of Java™ programming language and the application layer.

The applications, therefore, are written in Java programming language. The code of an application is compiled, combining code and resources (images, sounds, etc.), into a single file with `apk` extension, which can be installed. Each application in Android runs in an isolated environment, they are executed by their own VM, and for every application, a new user is created and assigned to it. Moreover, each application runs in their own Linux process. The Android platform also can kill processes to release memory.

Android applications can share data between them despite the isolated environment. One application interested can ask for permission to access resources and data of another app such as contacts, SMS messages, etc.

Android Components are the building blocks of applications. They are entry points for the application itself, they are independent from each other, and they have their own lifecycles. In the next section, we present the Components.

Furthermore, every application demands a file called `AndroidManifest.xml`, where components are declared. This file is also responsible for:

- Identify permissions required to execute the app, e.g., internet access;
- Inform the API level that application runs;
- Declare hardware and software resources needed by the application.

2.1.1 Android Components

An Android app is built using one or more of Android's four core application components:

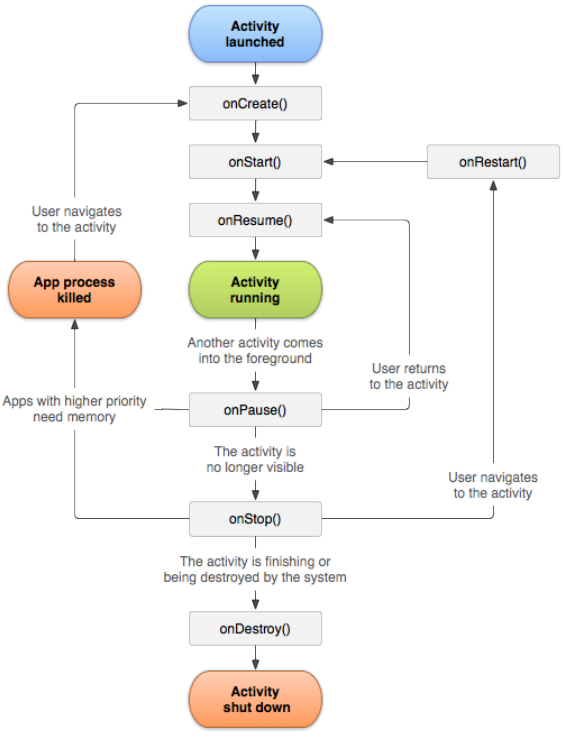
Activity: It's the main component of an Android application. They provide a screen the user can interact. Each Activity receives a window that often fills the entire screen. However, they can be smaller and can stand above other windows. Activities are managed by the *activity stack*. When an Activity starts, it is placed on the top of the stack, becoming the actual running screen. The old Activities will remain in the stack, and when a user clicks on the *Back* button, the running Activity is popped from the *activity stack*.

As mentioned before, each component has lifecycles. The possible states of an Activity are:

- *Running or Active*: if an Activity is on the foreground (on top of the stack)
- *Paused*: when an Activity lost focus but still visible
- *Stopped*: the Activity is completely obscured by another Activity but still retains state.

Activities in *Paused* or *Stopped* state can be killed by the Android system to release memory. Figure 1 show the details of Activity states and events.

Figure 1 – Activity Lifecycle.



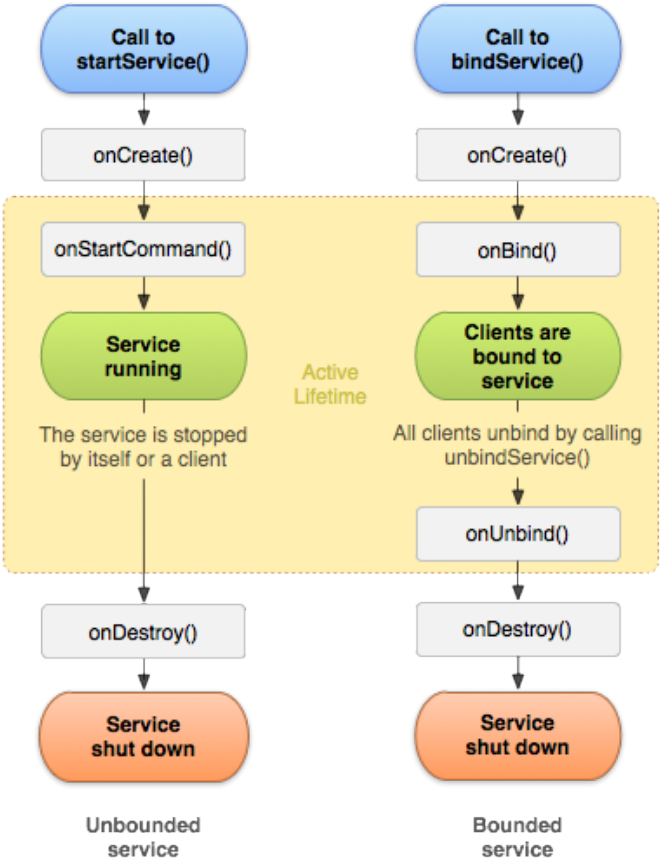
Source: <https://developer.android.com/images/activity_lifecycle.png>

The application Layout is declared using an XML file. Moreover, the controls of the interface, i.e., buttons, labels, etc. are maintained in this file. This XML file, therefore, is injected into the Activity object using the `setContentView` method.

Service: Service is the application component that can run tasks in the background. The service component have no User Interface and can be started from other components. Services keeps running even when the application is not visible. In general, services are used to play music or download files from the internet, and load files from file system. Services have their state machine, and have two forms: bounded and unbounded.Their lifecycles can be seen in Figure 2:

- **Started**: this Service is started by calling `startService()` method. When started, this Service runs endlessly even if the calling component is destroyed.
- **Bounded**: a Service is bounded when the component interacting with it calls `bindService()` method. This creates an client/server interface and remains alive as long as they are bounded to another component.

Figure 2 – Service Lifecycle.



Source: <https://developer.android.com/images/service_lifecycle.png>

BroadcastReceiver: This component is used to receive broadcast messages, called `Intents`, sent from other components, applications or the Android platform itself. The Android system sends broadcasts when significant events occur, such as when the battery is low or when the system boots up. The `BroadcastReceiver` follows to the publish-subscribe design pattern, 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: this component is responsible for grouping and sharing data between applications. They encapsulate data and provides a way to access it. The Android platform comes with built-in providers that manages audio, video, images and contacts that can be accessed by any application.

2.1.2 Intents

The Android platform provides a message passing system that is used to link applications and components. The Intent is a message object that can carry additional data. An Intent **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 are used for both internal and external communication, by internal we mean that message object do not cross the application boundary, and by external, we mean the messages sent to another application.

Also, Intents can be explicit, in which the recipient is known at compile time, or they can be implicit, in which the recipient is known only at runtime. Implicit (broadcast) Intents are delivered to any application that supports the desired operation. Moreover, the Android operating system sends notification of events using implicit Intents, such events can be “battery is low” or “wifi is down”.

Intents can be sent between three of four components (`Activity`, `Service`, and `BroadcastReceiver`) using some of the methods shown in Table 1.

2.2 Java Exception Handling

When an error occurs in the Java language, an exception is raised (GALLARDO *et al.*, 2014) 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

Table 1 – A short list of Android Intent-sending methods

Name	Description
`startActivity(Intent)`	Starts an `Activity` execution.
`startActivityForResult(Intent)`	Starts an `Activity` and return a result when it finishes.
`startService(Intent)`	Starts a `Service` execution.
`bindService(Intent)`	Binds a `Service` to a caller component.
`stopService(Intent)`	Stops a `Service` execution.
`sendBroadcast(Intent)`	Sends a broadcast message to a `BroadcastReceiver`.

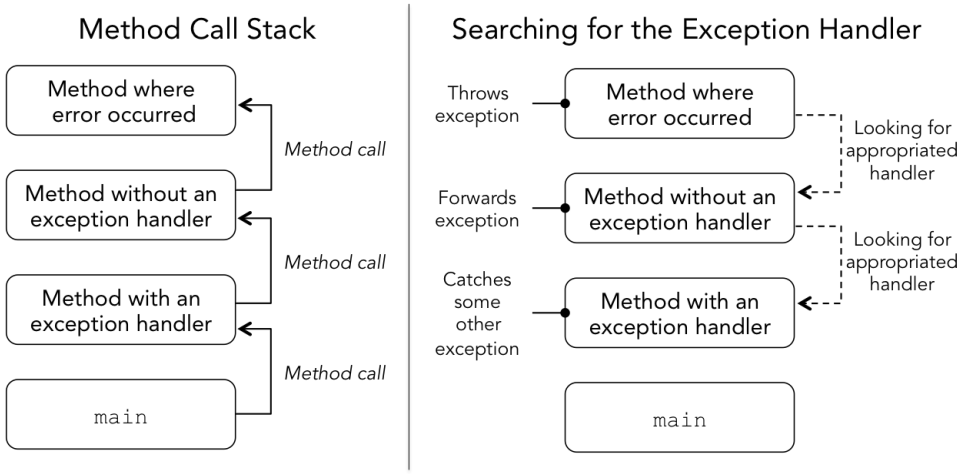
Source: Produced by the author.

this hierarchy: **checked** and **unchecked exceptions** (JENKOV, 2013). 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 error is handled. In Java, exceptions can be raised using the `throw` statement, propagated using the `throws` statement in method signatures and handled in the `try-catch-finally` block. We must first understand the method call stack of Java in order to fully understand the exception handling mechanism.

In Java, every time a method is invoked a new stack frame is created. A frame is used to store data (e.g., local variables of a method) as well as to return values from a method or to dispatch an exception. In Figure 3 (left side), we can see a call stack where three methods were called sequentially. The frame in the bottom is the `main` method and the frame on top of the stack is the one where the error occurs. When an error occurs an exception is raised and the

Figure 3 – The method call stack and the search for the exception handler.



Source: Produced by the author

Figure 4 – Example of `try-catch-finally`.

(a) Method handling an Exception.

```
1 void methodA() {
2   try {
3     methodB();
4   } catch (Exception e) {
5     // handle exception
6   } finally {
7     // always execute
8   }
9 }
```

Source: Produced by the author.

(b) Method propagating an Exception.

```
1 void methodB() throws Exception {
2   (...)
3   methodC();
4   (...)
5 }
6
```

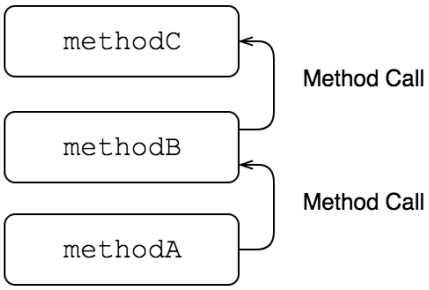
Source: Produced by the author.

(c) Method throwing and Exception

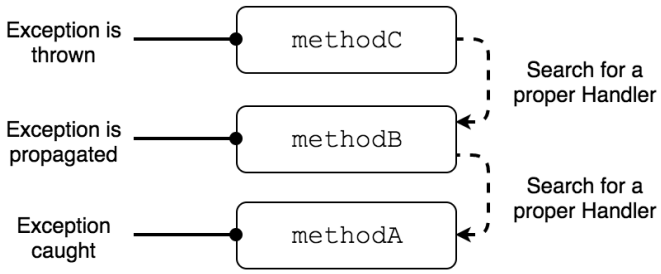
```
1 void methodC() throws Exception {
2   if (/* condition */) {
3     // do something
4   } else {
5     throw new Exception();
6   }
7 }
```

Source: Produced by the author.

(d) Call Stack.



(e) Search for a Handler.



Source: Produced by the author.

Source: Produced by the author.

exception handling mechanism starts the search for a proper handler to handle that exception. This search begins at the top of the stack and goes down until a proper catch block captures it. Figure 3 (right side) show the steps.

The Figure 4 show the use of the constructs `throw` (Figure 4c), `throws` (Figure 4b) and `try-catch-finally` (Figure 4a). In the Figure 4c an exception of type `Exception` is thrown in `methodC`, line 5. In the `methodC` we can see the `throws` in the method signature. This construct indicates that this method can throw this type of exception. In Figure 4b, the `methodB` calls the `methodC`. In this case, as the `methodB` have no exception handling code for `Exception`, it must have in its signature the `throws` construct (Figure 4b, line 1). In Figure 4a, the `methodA`, which calls the `methodB`, have a `catch` block that captures the exception of type `Exception`. The call of the `methodB` is surrounded by a `try` block, indicating where the exception is thrown. In `methodA`, on line 6, we can see the `finally` block. This `finally` is always executed, even if the exception is not thrown.

2.3 Other Approach for Error Handling

In procedural languages such as C, the error handling strategy applied are return codes. Return codes, as the name suggests, are values returned by functions. The codes returned are from primitive types, and they indicate the actual result of a function call.

Listing 2 – Return code (produced by the author)

```
1  int FunctionA() {
2      // do something
3      if (condition)
4          return 0
5      //...
6      else
7          return -1
8  }
```

Listing 3 – Error handling (produced by the author)

```
1  void FunctionB(){
2      int res = FunctionA();
3      if (res == -1) {
4          // Handle Error
5      } else if(res == 0) {
6          // Happy path
7      } else if(res == 2) {
8          // Other status
9      }
10     // keep checking
11     // returning codes
12 }
```

The code above shows an example of the returning code style for reporting errors in a C-like programming language. Listing 2 shows one function (FunctionA) that returns the status code that depends on some condition, in line 4 and 7. In this example, the result can be `0`, which could mean success and `-1` to indicate some error in the execution. One problem in this approach is when the number of possible returning codes increases. Error codes have no contextual information about the error. Unlike exceptions, error codes can be ignored/muted by the callers of the function. The caller of `FunctionA` must check every possible returning case, using if or switch statements. This checking can be ignored, as said above, once they do not interrupt the flow of execution. We can see the use of `if` statement handling the returning codes in lines 3, 5 and 7 of Listing 3. We discuss more drawbacks of this strategy in Section 4.11.2.

2.4 Boa Language and Infrastructure

The Boa Platform, used in this study, consists of a backend infrastructure and a programming language dedicated to mining huge software repositories datasets. Boa contains projects from Github and Sourceforge, the latest Github snapshot is from September 2015, and the Sourceforge dataset is from September 2013.

The repositories are broken into three categories: small, medium and full. The small dataset represents 0.1% of the entire dataset, and the medium one contains 10% of the full dataset. The full dataset contains exactly 7,830,023 projects, and the number of unique files in the full Github repository is 146,398,339.[1].

The Boa infrastructure translates queries into Hadoop/MapReduce programs that parallelize the execution. This particular property guarantees the Boa platform the speed necessary to run queries on time, even when using the full data set. Queries can be submitted using the web page of the tool or via an Eclipse IDE plugin.

The Boa programming language, on the other hand, provides a set of commands and types that abstracts the mining, also the language is based on the Visitor Pattern, which permits to walk over the *Abstract Syntax Tree*(AST) in flexible ways. The types available in Boa are divided into domain-specific and built-in types. Domain-specific types can be split into two categories, metadata, and content itself. For metadata, we have types like `Project` and `Repository`. For the content itself, we have the AST node, which represents the Abstract Syntax Tree of the source files. Inside it, there is `Declaration, Expression, Statement` types, etc., very suited for mining. The language also has built-in types, ranging from basic data types (ints, strings, etc.) to data structures. The language also offers a variety of functions, divided into domain-specific and built-in. Moreover, the user can define their own functions, which gives more flexibility to the language.

In Listing 4 we see an example of Boa code. This function gets a `ChangedFile` as parameter and return `true` if there is a call of `sendBroadcast` inside `catch` blocks.

Listing 4 – Find Components sending Exception Notifications (produced by the author)

```
1 send_broadcast_inside_catch := function(c: ChangedFile): bool {
2   send := false;
3   visit(c,visitor {
4     before s: Statement -> {
```

[1] http://boa.cs.iastate.edu/stats/index.php

```
5       if(s.kind == StatementKind.CATCH) {
6         visit(s, visitor{
7           before e: Expression -> {
8             if(e.kind == ExpressionKind.METHODCALL &&
9                match("sendBroadcast", e.method)) {
10                if(haskey(broad,lowercase(c.name))) {
11                  stop;
12              }
13              send = true;
14            }
15          }
16        });
17      }
18    }
19  });
20  return send;
21 };
```

The Boa platform, however, has limitations. One of them is the fact that the language does not support the `finally` block, of the `try-catch-finally` triad, of the Java programming language. The plugin for the Eclipse IDE works but lacks a local syntax checker. We also have found bugs in the type system when writing functions, e.g., wrapping a function that returns an array can reveal an inner type called `ProtoList`.

2.5 Final Considerations

This chapter presented the background that supports this dissertation. We showed in details the Android Architecture, The Java Exception Handling Mechanism, and the Boa platform. We also provide the motivations for this work. In Section 2.1, we give an overview of the Android platform and its features. In this section, we also introduced the central concepts concerning an Android application: Components and Intents, representing the structure and the flow of information respectively. In Section 2.2, we explored in details the exception handling mechanism in Java programming language, its characteristics and how they are used. In Section 2.3, we present the usage of the error handling strategy based on return codes (a.k.a, error codes). In Section 2.4, we talked about the Boa platform, a tool used to mine software repositories. We show how the platform works and we give some examples of the Boa programming languages. In the next chapter, we will give the main studies related to this dissertation. We show studies related to mobile computing in general and studies related to Android exception handling.

3 RELATED WORK

This Chapter describes previous work on studying the communication model and the Exception Handling problems in Android platform. In Section 3.1 we describe general studies on mobile platform. In Section 3.2 we describe studies on Android communication model. In Section 3.3 we present what have been done on Android exception mechanism.

3.1 General Studies on Mobile Platform

Minelli and Lanza (MINELLI; LANZA, 2013) focused on understand apps from a *structural and historical perspective*. They focus on three factors: source code, usage of third-party APIs, and historical data. To understand these factors, they must answer questions like how different apps are from traditional systems, how often apps rely on third-party libraries, and are the code smells the same? To support their analysis, they build a tool called SAMOA. This tool mines software repositories, extract metrics and produce rich visualization graphics. Analyzing the F-Droid corpus of apps, they present a set of insights concerning maintenance and evolution-related issues. *Apps are smaller than traditional software systems.* Many apps have few functionalities and, therefore, are built from a small number of classes. As the number of Activities and Services grow and the number of third-party libraries increases, the complexity of the app increase as well. They found that approximately 2/3 of all method calls were to external APIs. For this reason, the heavy use of external API calls difficult the maintainability and the comprehension of the app. Other studies have shown that the exception handling becomes troublesome when there's interaction with external APIs. Another insight says that apps almost do not use inheritance, though they are Object-Oriented systems. Some apps contain the entire source code of external libraries. Such fact affects the results of automatic metric extraction, leading to wrong conclusions. The history of a project is an excellent source of information, provided by the version code system at hand. However, some developers use versioning systems only at late stages of development. In some cases, they found that applications were made solely by the core components of the platform. In other words, developers are putting too much business logic inside the Android components, mainly in Activities (View). This paper is relevant to this work because it discusses the main characteristics of the Android apps and also provides an essential set of insights.

Picco et al. (PICCO *et al.*, 2014) gave an overview of mobile computing before and

after 2000. They revisited a paper written in 2000, observing how the mobile computing has changed. In the past, the state-of-the-art device was the *Personal Digital Assistant* (PDA). The PDAs was very limited regarding computing power and connectivity. For mobile internet, they had *Wireless Application Protocol* (WAP), suited for small screen devices with low bandwidth. Wi-fi technology existed, but the cards were large and not suited for mobile devices. The challenges presented in their original paper was divided into two categories: Theory and Systems. In Theory category, they focused on Models and Algorithms, and for Systems category, they concentrated in Applications and Middleware. It is worth mention that the models in the past were focused on mobility, location, and context. In the Algorithms, they addressed recurring challenges like location changes, frequent disconnections, power limitations, communication constraints, among others. For Systems challenges, they focused on Applications and Middleware, two elements that are still relevant nowadays. At some point in history some *disruptive events that dramatically changed both the perspective and direction of software engineering for mobile computing*. One of the *Game Changers* is the **omnipresence of smart devices**, resulting in an explosion in software tailored to these devices. **Sensors became cheap and small**, being embedded into smartphones. **Ubiquitous Connection** was another game changer according to the paper, enabling applications to still "always on". **Social Networks** also play a significant role in the new era of mobile computing, changing the way the users interact on the internet. When peering at the future, the author, suggest the following trends in mobile computing: **Mobile Sensing; Mining Mobile Big Data; Off-loading; and Wearables**. Several challenges are discussed in the paper, including mobile privacy and the disappearance of computers, as Mark Weiser (WEISER, 1991) predicted. The mobile computing has a profound impact on software engineering because mobility, ubiquity, and sensing now must be considered. This article is relevant because it shows the challenges and the lessons learned, and it also introduces trends for the future of mobile computing as well.

3.2 Studying the Android Communication Model

Studies on Android communication model focus on discovering and mitigating the vulnerabilities related to the Intent model, such as application hijacking or information leaking (CHIN *et al.*, 2011; HAY *et al.*, 2015; OCTEAU *et al.*, 2013). Other studies focused on the testing the robustness of the applications (MAJI *et al.*, 2012; SASNAUSKAS; REGEHR, 2014). Our study focuses on the uses of the component communication model to propagate

exceptions. No previous work have explored this trend.

Chin et al. (CHIN *et al.*, 2011) 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. (MAJI *et al.*, 2012) studied the robustness of Android applications with respect to malformed Intents. They created a tool that sends randomly generated Intents (lacking mandatory data fields for example) to applications and found that, in general, applications are vulnerable to this attack. However, in spite of interest in application robustness, this work focused in crashing applications using the *Inter-Application Communication* (IAC) model. Our study, on the other hand, focused on mining Android applications to identify if developers are using the communication model to propagate exceptions between components.

Hay et al. (HAY *et al.*, 2015) demonstrate that Inter-Application Communication of Android, which enables reuse of functionality across apps, can be used as an attack surface. They found 8 concrete vulnerability types that can potentially arise due to handling of incoming IAC messages. This study focused on using the IAC as an attack surface while ours focused on finding if developers are using the IAC as a channel to propagate exceptions.

Payet and Spoto (PAYET; SPOTO, 2014) define an operational semantics for a large part of Android platform, not only the Dalvik virtual machine as previous works. They formalize the Android communication model in order to analyze applications made up of several components, i.e. many Activity screens.They also simplified the Dalvik bytecode and created an operational semantics for it. This work sets bases for formal analysis of Android applications.

3.3 Studying the Exception Handling Mechanism in Android

There are a number of studies focusing on exception handling in Android. Kechagia and Spinellis (KECHAGIA; SPINELLIS, 2014) studied undocumented runtime exceptions thrown by the Android platform and third party libraries. They mined 4,900 different stack traces from 1,800 apps looking for undocumented API methods with undocumented exceptions participating in the crashes. Coelho et al. (COELHO *et al.*, 2016) also mined stack traces from issues on GitHub and Google Code looking for *bug harzards* 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. These studies focused on identifying methods on Android platform and third-

party libraries with undocumented exceptions and hazards that may cause bugs in applications. However, none of these studies explore the exception propagation across components using the Android communication model.

Android programs can be very vulnerable to exceptions, as shown in this section. For example, Choi and Chang t(CHOI; CHANG, 2015) 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. To make Android apps more robust is necessary to handle uncaught exceptions. They, therefore, propose a mechanism for component-level exception providing a new component API that extends the existing components. Developers can use the component-level exception handling mechanism by writing programs with this newly extended API. The new mechanism provides a catch method for intra-component exceptions, i.e., it catches uncaught exceptions thrown within the component, and, also they give a way to propagate uncaught exceptions to its caller along the Activity stack. The library they produced offers an inter-component try-catch construct; override Android interface methods and; inter-component throw construct. Also, they give a set of conversion rules to transform an ordinary Android program to a more robust version using the library. A formal semantics was also provided with robust properties. Finally, they run an experiment with nine Android programs, rewriting them with the library proposed. During the experiment, they caught 30 exceptions which would otherwise lead the original program to crash, showing the robustness gain.

Android apps deals 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, *Global Positioning System* (GPS), and sensors in general. Zhang and Elbaum(ZHANG; ELBAUM, 2014) developed an automated approach to support the detection of faults in exception handling code that deal with external resources. In their proposal, the first step is instrument the target program so the result of calls to external resources can be mocked at will to throw exceptions. The existing test cases are amplified by re-executing them under several mocked patterns to explore exception code. They select a collection of Android applications to expand its tests. For one app they demonstrate a gain of 62% of coverage. In general, they show that their approach can validate exception code automatically.

Bavota et al. (BAVOTA *et al.*, 2015b) 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. They aim to understand what extent API fault- and change-proneness affect the user rating of an app, what extent Android developers experience problems using APIs, and how much they feel these issues can cause bad reviews. They conducted a software repository mining and a survey with developers. The questionnaire aims to answer the following big questions: (i) developer's background; (ii) factors that negatively impact apps' user rating; (iii) experience with used APIs and; (iv) the impact of faulty APIs on bad reviews. For the repository mining stage, they found that there is statically evidence that fault- and change-proneness decreases the app's rating. Developers declared the factors that negatively impact the user ratings are: the app contains bugs and crashes, features not useful, the poor usability of apps and, better apps available. The most *frequent perceived cause of bugs/crashes* are Java programming errors (wrong implementation), fault-proneness of third-party APIs, fault-proneness of Android APIs followed by change-proneness of Android and third-party APIs. To summarize, change- and fault-proneness of APIs threatens the proper functioning of a program; developers are more concerned about the bugs in APIs than the changes performed in new releases of APIs. Finally, developers are more concerned about the changes in the Android API than the third-party ones.

Oliveira and colleagues (OLIVEIRA *et al.*, 2016) performed an exploratory study of exception handling behavior in Android and Java applications. They were interested in how the exception code evolves, and how the robustness evolves between versions. They extracted metrics that capture changes in code and metrics that measure the robustness of the program. The metrics divided into size, robustness, and change. For the Size, they use the traditional *Lines of Code* (LOC) metric. For Robustness, they count the number of exceptional flows and the relative number of exceptional flows between version. Finally, for change metrics, they used the typical change impact metrics, i.e., elements like classes or methods and try/catch blocks added, changed or removed. They also did the manual inspection to compute the metrics, textually describe the change scenario and assess the impact of the changes in robustness. Their results show that "Android developers not only worry less about exception handling but also about exception interfaces.". Normal code changes imply changes in the robustness of Android applications. In Android, 67% of all exceptions are from the unchecked type; these exceptions come from external libraries and Android platform methods. They conclude that the excessive use of unchecked exceptions decreases the Android applications robustness.

Queiroz and Coelho (QUEIROZ; COELHO, 2016) did an exploratory study that

analyzed 15 Android applications. This study aims at answer questions about the quality of exception handling code. More than that, they surveyed Android developers to assess the main obstacles to implementing exception handling code and what developers do to overcome these constraints. The application selection was based on the following criteria: the application should be public hosted (e.g., Github); the application should be on Google Play Store; the application should be popular, i.e., they should have, at least, one million of downloads. They characterize eleven kinds of handling: Log, Specific, Throw/Rethrow, Empty, Continue, Return, Close Resource, Message and Other. The most relevant categories were: Log with 44%, Empty with 12%, Return and Other with 11% and Close Resource with 9%. They found that 41% of all caught exceptions were from `Exception` type. Generic handlers, i.e., catch blocks capturing `Exception`, `Throwable` or `RuntimeException` corresponds to 47% of all handling. The Log category is deeply related to generic exception handling, while 25% of Empty handlers captures unchecked exceptions. The classes that appear the most in exception code are the ones from the Android platform (e.g., `Activity, Fragment, Service`) and Thread related classes (e.g., `Thread, Handler` and `Runnable`). In the survey the developers cited as "bad practices" in exception handling code: to silence exceptions (i.e., empty catch blocks), catch generic types, poor or vague error messages, not use an external server for logging and the excessive use of try/catch. 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 advice such as: avoid heavy use of Fragments; manage the multitask environment carefully; make use of error reporting tools; fail fast and; only capture an exception if can handle.

3.4 Final Considerations

In this chapter, we presented the studies related to this dissertation. We focused on three aspects: General Studies on Mobile Platform, Studies about the Android communication model, and the Android exception handling mechanism. In Section 3.1, we present the perspective of the mobile computing scenario, showing the devices and challenges pre and post 2000's. We also discuss one work that gives an overview and several insights concerning Android applications implementation. In Section 3.2, we focused on studies associated with the communication model of the Android platform, exposing their characteristics and flaws. Finally, in Section 3.3, we

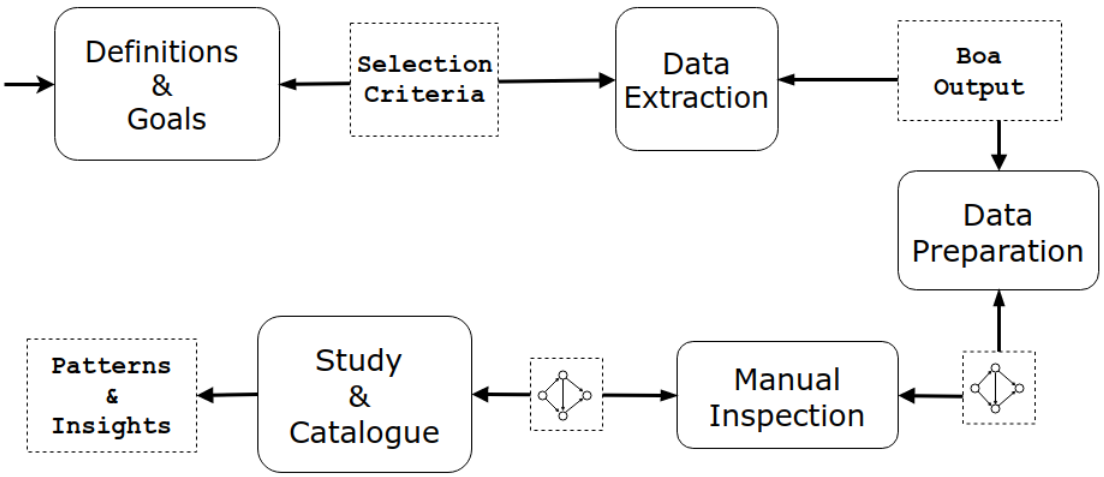
discuss a set of works related to the Android Exception Handling Mechanisms. In the next chapter will be presented the design and methodology of this dissertation, the results we found, a set of insights related to our findings and the threats to validity.

4 EMPIRICAL STUDY

In this Chapter, we present the precise methodology followed by this study. In Section 4.1, we give an overview of the steps taken to produce this dissertation. We also describe the definitions used throughout this work in Section 4.2. Moreover, we describe the Research Goals and Questions that guided this study. In Section 4.4 we describe the process of extracting data, and in Section 4.5 we present the steps taken to prepare the data for next stage of the study. In Section 4.6 we describe the process of analysis of the applications, we give the list of apps selected and the methodology adopted. In Section 4.7 we define how we catalog these apps. Section 4.8 presents the results for the 4 Research Questions of this dissertation, giving details behind the findings. In Section 4.11, we offer a set of insights discovered by this research, and we provide an overview of the bad practices made by developers concerning the exception propagation. Finally, we present the threats to validity in Section 4.12, and the countermeasures are taken to mitigate them.

4.1 Research Design and Methodology

Figure 5 – Stages of the study.



Source: Produced by the author.

In Figure 5 we can see the steps taken in this study, the output of each step and the information flow. The rounded box represents the step or stage itself and, the dotted boxes are the output of that particular stage. In the first stage, Definitions and Goals, as the name suggests, we define the concept of "Exception Notification". At the same time, we ran the first data extractions in the dataset 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 run the queries, based on the selection criteria defined in the two first stages. As output, this step produced several log files in ASCII format.

At this point, in the Data Preparation stage, we decided to use a graph database to persist and model the elements of the study. The graph generated at this stage is not complete yet, lacking the Handler component that will be identified by the manual analysis. The graph database assisted us in two different ways: (i) modeling the problem naturally, and; (ii) the DBMS provides a visualization tool that helped us to understand the structure of the projects. In the Manual Inspection stage, we selected a set of relevant projects to do the manual investigation. As stated on section Definitions, an Exception Notification involves two parts, Sender and Handler, however, in the data extraction stage we found only the Senders. The manual inspection, then, ought to identify the Handlers of the Exception Notification in the source code and complete the graph, by inserting these components and linking them to their Sender.

Once the graph is complete, we could understand the project's structure, and we could identify 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 works.

The next sections of the methodology are organized as follows, in Section 4.2 we present the definitions the guided this study. In Section 4.4 we give details of the data extraction. Section 4.5 shows the steps taken to prepare data for analysis. In Section 4.6 we present the manual exploratory inspection. The next Section 4.7 presents the classification method we use to catalogue the projects.

4.2 Definitions

In this Section, we present definitions that are essential to grasp this research. These definitions cover three central aspects of this study: (i) the concept of Component; (ii) the definition of Exception Notification, and; (iii) the classification of these components.

The definitions presented here will guide this study and are central for its correctness. Every definition comes with a small description and context, further, they are essential to design the study. Definitions 1, 2 and 3 are used throughout this study, Definitions 4 and 5 are used in section 4.4 and, in section 4.6, Definition 5 is employed as well.

Definition 1 (Component) *A component is a Java class of an Android application.*

Definition 2 (Internal Component) *Components we have access to the source code.*

Definition 3 (External Component) *Components we do not have access to the source code, i.e., components from other applications.*

Definition 4 (Exception Notification) *Any call of* `startActivity` *or* `sendBroadcast` *methods* ***inside a catch block*** *of any component represents an intention to notify the caught exception to another component.*

As Java documentation[1] stated, "Each catch block is an exception handler that handles the type of exception indicated by its argument." Hence, when method calls from Android communication APIs happen inside catch blocks, it means that a component is handling an exception and, this exception handler involves sending a message to another component.

As we said in Section 2.1, the Android platform uses the term Component for specific classes of its framework used to implement applications. However, from now on, we will use Component as defined in Definition 3.

Definition 5 (Sender and Handler) *There are two Components in exception notification:*

- ***Sender****: a component that starts the notification, i.e., catch an exception and, inside the catch block, call a method from the Android communication API.*
- ***Handler****: a component that handle notification, i.e., they receive the exceptional events sent by Senders.*

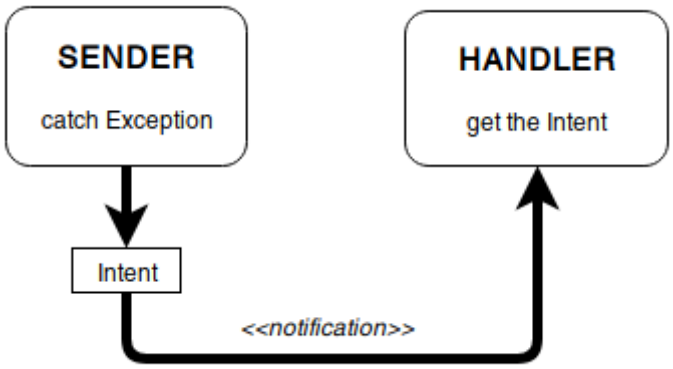
The Exception Notification scenario requires two Components. The one who is responsible for catching a given exception, encapsulating it into an `Intent` object, and then broadcasting it using the inter-component communication API. The second Component is the one that handles the sent message. Figure 6 illustrates the this scenario.

4.3 Goal and Research Questions

The primary goal of this dissertation 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 applications? Which patterns of notification can arise in an environment without a proper

[1] https://docs.oracle.com/javase/tutorial/essential/exceptions/catch.html

Figure 6 – Exception notification between Sender and Handler.



Source: Produced by the author.

mechanism to deal with it? Specifically, we are aiming at answering the following research questions:

- **RQ1. Do developers send Exception Notifications in Android applications?**
- **RQ2. If so, how is notification implemented/designed?**
  - RQ2a: How are the projects structured?
  - RQ2b: How is the code used to send Exception Notifications?
  - RQ2c: How is the code used to handle notifications?

4.4 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 dataset (DYER *et al.*, 2013) 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:

- The project have Senders?
- How many of them have Senders in their code?
- If so:

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

- Who are the Senders?
- What are the Exceptions being encapsulated?
- What are the types of Components?

We can see an example of mining in Listing 5. This function counts the most caught exceptions and ranks the top 20.

Listing 5 – Counting the most caught exceptions (produced by the author)

```
top_exceptions: output top(20) of string weight int;
exceptions_in_propagation:= function(c: ChangedFile) {
  visit(c, visitor {
    before s: Statement -> {
      if(s.kind == StatementKind.CATCH) {
        propagate := false;
        visit(s,visitor{
          before e: Expression -> {
            if(e.kind == ExpressionKind.METHODCALL
              && (match(" startActivity", e.method) ||
                match(" startActivityForResult", e.method) ||
                match("sendBroadcast", e.method)
              )) {
              propagate = true;
            }
          }
        });
        if(propagate) {
          top_exceptions << s.variable_declaration.variable_type.name weight
              1;
        }
      }
    }
  });
};
```

One example of a Boa output can be seen in Listing 6.

Listing 6 – Boa output - produced by (DYER *et al.*, 2013)

```
top_exceptions = ActivityNotFoundException, 817.0
top_exceptions = Exception, 472.0
top_exceptions = android.content.ActivityNotFoundException, 172.0
top_exceptions = UserRecoverableAuthException, 113.0
top_exceptions = IOException, 74.0
top_exceptions = UserRecoverableAuthIOException, 71.0
```

```
top_exceptions = JSONException, 60.0
top_exceptions = Throwable, 40.0
top_exceptions = NoMediaMountException, 35.0
top_exceptions = MmsException, 28.0
top_exceptions = NullPointerException, 27.0
top_exceptions = NativeDaemonConnectorException, 25.0
top_exceptions = FrameAnimationException, 23.0
top_exceptions = TweenAnimationException, 23.0
top_exceptions = HttpClientErrorException, 14.0
top_exceptions = PackageManager.NameNotFoundException, 14.0
top_exceptions = XMLRPCException, 14.0
top_exceptions = FileNotFoundException, 13.0
top_exceptions = XMPPException, 11.0
top_exceptions = NameNotFoundException, 10.0
```

4.5 Data Preparation

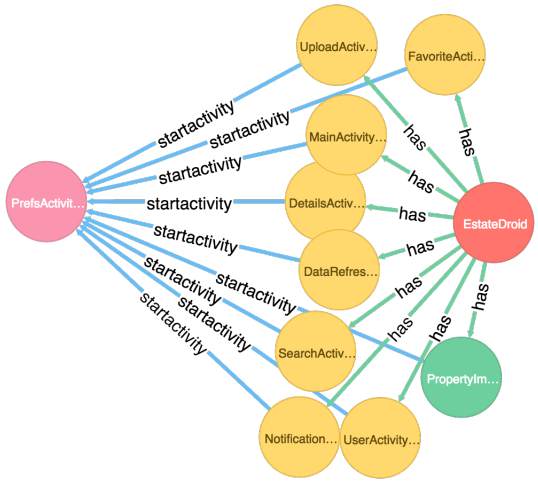
To perform the analysis, we carried out a data preparation over the results collected so far. The product of the Data Extraction was a collection of output files generated by the queries we ran on the Boa platform. This platform offers a flexible output system, supporting the extraction of fine-grained information about projects, including metadata and source code artifacts. On the other hand, the Boa output can generate thousands of lines of entangled data, and, for this reason, it is not suitable for human reasoning. We decided to transform this unreadable output to a graph format, for this reason, we picked a Graph DBMS that offers a graphical representation and a query language. We choose the graph representation and a Graph DBMS for several reasons: (i) the core interest of this dissertation is to find relationships between components of an Android project and graph theory fits the problem; (ii) we would like to have a visual representation of the resulting projects and components; (iii) we'd like to persist our findings; (iv) we'd like to navigate through the projects, components and relationships using a declarative language.

Neo4J (NEO4J, 2012) is a graph database written in Java, and it falls into the No-SQL family of databases. It has an internal web server that supports visualization where one can write queries using Cypher (PANZARINO, 2014) and navigate on the resulting graph. In Figure 7 we ca see one example of visualization.

To model this problem into a graph, we had to follow a set of transformation rules:

- Every Project is a Node in the graph;

Figure 7 – Example of a graph in Neo4J.

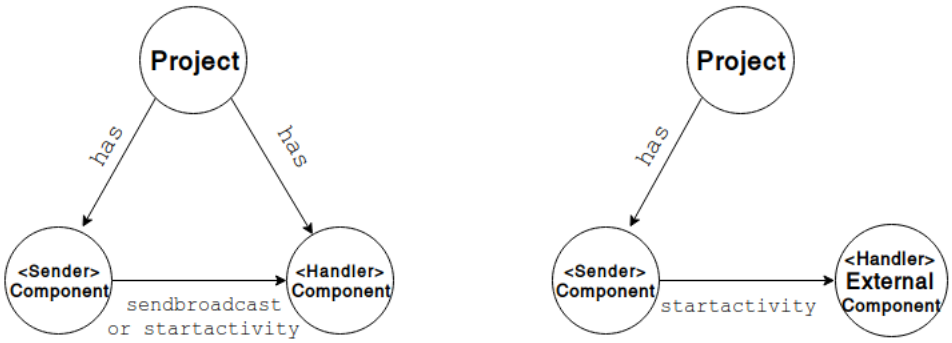


Source: Produced by (NEO4J, 2012).

- Every Component is also a Node;
- Every Internal Component must belong to a project;
- There are three types of edges: `has`, `startActivity` and `sendbroadcast`;
- All edges have direction;
- `startActivity` and `sendbroadcast` are the edges used to indicate the direction of the notification.

Nodes and Edges in such database can also have extra labels, to enrich the semantics of our dataset. Figure 8 shows the graph that models the exception notification.

Figure 8 – Exception notification graph.



Source: Produced by the author.

The top 100 projects that notify the most were selected and loaded into the Neo4J database, excluding discontinued and “toy” projects from this study.

4.6 Manual Inspection

In this stage, we describe the manual and exploratory inspection we performed in the dataset collected in previous stages. Here we explain why this step was needed and how we executed the manual analysis. In previous stages, we extracted data using a source code mining tool and, after that, we selected the top 100 projects using the number of *Exception Notification* (EN) per project as criteria. If, on the one hand, the automatic extraction was essential to discover projects doing EN, on the other hand, the mining tool was not able to locate Handlers. For this reason, we employ a thorough examination of the apps source code in order to identify the remaining components. Finally, we fill the gaps in our dataset by introducing new nodes and edges in the graph. However, before we comprehend the apps' code, we selected, randomly, 30 applications, from the 100 previously elected, to be inspected. We take this step because the manual analysis is very cumbersome and we lack resources to do an extensive exploration. The chosen app must satisfy at least one of these requisites:

- The app is available in Play Store, or
- It is a stock app from Google, or
- The app is relevant by the number of commits, contributors and, stars.

So, the question that remains is how we found the Handler components? To answer this question we create the following protocol:

- Locate the Sender file where the EN happens
- In this file, locate the exact catch blocks where EN occurs
- Find the specific Intent used in this EN based on `ACTION and DATA`
- Discover the Handler for that particular Intent

Once we find the specific Handler, we take two actions: (i) create a new Node in the graph dataset; (ii) associate the Sender with its Handler, creating an edge between them.

We had to analyze two cases in the manual inspection, whether a `startActivity` or `sendBroadcast` starts a notification. For `startActivity` method, we also have two cases: the handling `Activity` might be internal or external. To find the handlers of an internal `Activity`, we just need to observe the method call inside the catch block and read the name of the target class, as in Listing 7[3]. In this example the handler class is easily spotted on line 9 when the `Intent intent` is created with the `ErrorScreen.class` parameter and, in line 11 the `startActivity` is called with this `Intent`. After this, we just have to use GitHub search tool

[3] http://bit.ly/2fDiZCr

Table 2 – Projects that do exception notification

Name	URL
fqrouter	https://github.com/fqrouter/fqrouter
Yaaic	https://github.com/pocmo/Yaaic
PocketSafe	https://github.com/rusmonster/PocketSafe
docnext	https://github.com/archilogic/docnext
SOCIETIES-Platform	https://github.com/societies/SOCIETIES-Platform
EstateDroid	https://github.com/gorog/EstateDroid
SOCIETIES-SCE-Services	https://github.com/societies/SOCIETIES-SCE-Services
AboveGameServer	https://github.com/SvenAke/AboveGameServer
FlipDroid	https://github.com/helianbobo/FlipDroid
droiddice	https://github.com/urvaius/droiddice
PhotoNoter	https://github.com/lnanek/PhotoNoter
TuCanMobile	https://github.com/Tyde/TuCanMobile
Continuous-Shooting	https://github.com/gengen/Continuous-Shooting
myfeedle	https://github.com/avantgarde280/myfeedle
Samarth-TheCleanerApp	https://github.com/jsankalp/Samarth-TheCleanerApp
MMS	https://android.googlesource.com/platform/packages/apps/Mms
Camera	https://android.googlesource.com/platform/packages/apps/Camera
Gallery2	https://android.googlesource.com/platform/packages/apps/Gallery2
web_install	https://github.com/j-pac/web_install
ZooTypers	https://github.com/ZooTypers/ZooTypers
owncloud-1.4.0	https://github.com/owncloud/android/tree/oc-android-1-4-0
CafeteriaTUE	https://github.com/manuelVo/CafeteriaTUE
BrainwaveStudio	https://github.com/HeesangLee/BrainwaveStudio
aNexter	https://github.com/Tody-Guo/aNexter
android-xbmcremote	https://github.com/freezy/android-xbmcremote
mirakel-android	https://github.com/azapps/mirakel-android
ZipInstaller	https://github.com/beerbong/com_beerbong_zipinst
MultiWii_EZ_GUI	https://github.com/eziosoft/MultiWii_EZ_GUI
Crydev.net_Reader	https://github.com/Geo-Piskas/Crydev.net_Reader
serenity-android	https://github.com/NineWorlds/serenity-android

Source: Produced by the author.

to find this class, so, for each internal notification, we discover their handlers and load them as nodes in the graph database and a new relationship is created between the `Sender` and `Handler`.

Listing 7 – Notification to an internal Activity (extracted from <http://bit.ly/2fDiZCr>)

```
1 public final void loginButton(final View view) {
2     // Try to login
3     String usernameString;
4     try {
5       usernameString = lp.loginButton();
6     } catch (InternetConnectionException e) {
7       e.fillInStackTrace();
8       Log.i("ZooTypers", "triggering internet connection error screen");
9       Intent intent = new Intent(this, ErrorScreen.class);
```

```
10        intent.putExtra("error", R.layout.Activity_connection_error);
11         startActivity(intent);
12        return;
13      }
14      // If login was successful, go to the multiplayer game
15      if (!usernameString.equals("")) {
16        multiIntent.putExtra("username", usernameString);
17        startActivity(multiIntent);
18        Log.i("ZooTypers", "user has logged in");
19      }
20    }
```

The next step was to discover the `External Activities` that plays the `Handler` role. The Android platform supports the cooperation between apps, providing a set of "default" applications that can be open. Such apps cover the basic tasks of a smartphone like open a page in the browser, showing a map location, taking a picture, etc. So, to open these "external apps", we just have to call `startActivity` passing an `Intent` with only the correct `ACTION` parameter set. The `ACTION`, however, is a `String` provided by the vendor of the app or library. For example, to open the stock Camera from one application a developer must use an `Intent` with the `ACTION` parameter set to `MediaStore.ACTION_IMAGE_CAPTURE` constant, which is a `String` containing "`android.media.action.IMAGE_CAPTURE`". Using this approach, we identify, classify and insert external applications into the graph database.

We finally discover the Handlers for `sendBroadcast` notification. In the same sense of the external apps, `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 follow this protocol:

1. Identify the Constant that characterize the Intent object;
2. Search for usage of that Constant in the code;
3. Find the `BroadcastReceiver` responsible for handle that specific constant;
4. Find where this `BroadcastReceiver` is registered.

It is important to mention that while the external apps also relies 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 developer's shoulders.

4.7 Study and Catalogue

This part of this study was designed to identify, classify, and discuss the structure and implementation of the Exception Notification schema. 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 these two categories as:

- Structural Patterns
- Code Patterns

A Structural Pattern 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. One example of one structure can be viewed in Figure 8.

The Code Pattern, on the other hand, rises in the structure of the source code. In other words, we want to discover patterns about the `Exceptions` being caught, the methods called, the creation of Intent and so forth. More information about this topic in the next section.

4.8 Results

In this section, we present our findings. In Section 4.9 we present a general overview of the results. Section 4.10 describes the design and implementation of projects. Section 4.10 aims to describe the structure of projects and the patterns of exception notification, handling, and code. Specifically, in Section 4.10.1 we aim to describe the structure and patterns of the projects. Section 4.10.2 shows the patterns of notification while in section 4.10.3 we describe the patterns of handling.

4.9 RQ1: Do developers use Intents to send Exception Notifications to components of Android applications?

At least 1,300 apps confirmed our claim; therefore the answer to this question is **yes, they are doing Exception Notification between components**. The results in this section, by the way, came from the automatic extraction stage of the study.

The dataset chosen have at least seven million projects. From this number, around 7% are Java projects. Likewise, 7% of Java projects are Android projects. Table 3 describe in detail these numbers.

Table 3 – Number of Projects

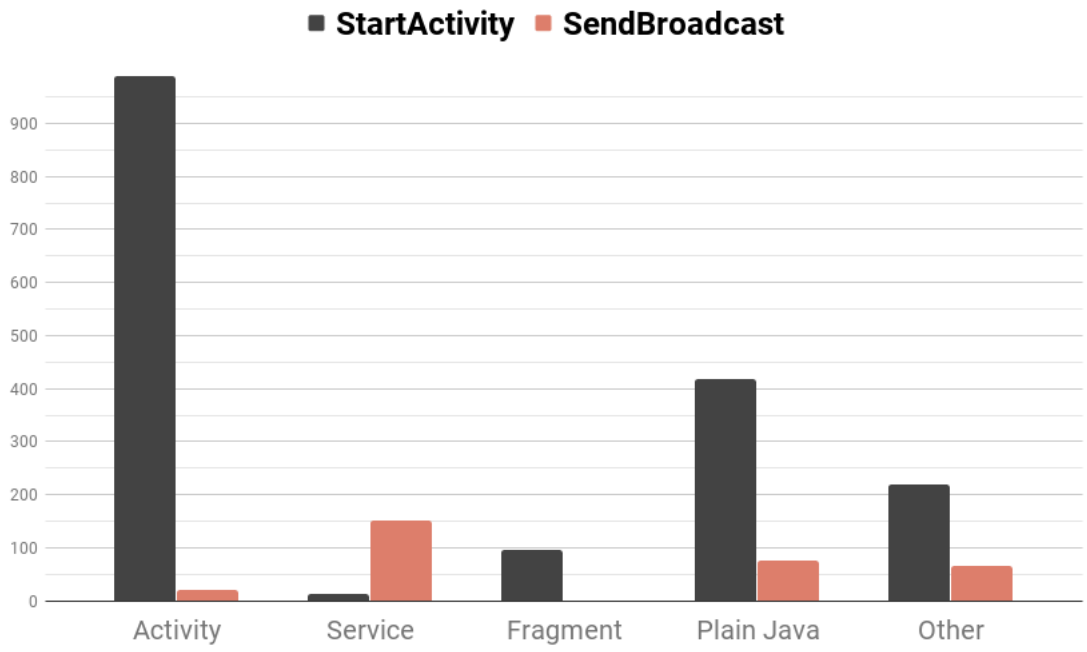
Total	Java	Android	Android apps Doing EN
7,830,23	554,864	66,093	1,327

Source: Produced by the author.

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 9.

Figure 9 – Distribution of method invocation over the categories



Source: Produced by the author.

From the `Other` components perspective, we observed that they are basically classes from Android API. In Table 4 we can see the types of classes from the `Other` category. As the

reader can note, we have classes related to Android UI API, and also classes associated with asynchronous tasks and concurrency.

Table 4 – 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%

Source: Produced by the author.

Table 5 – Common exceptions caught in Exception Notification

Ranking	Exception Type
1	`ActivityNotFoundException`
2	`Exception`
3	`UserRecoverableAuthException`
4	`UserRecoverableAuthIOException`
5	`IOException`
6	`JSONException`
7	`Throwable`
8	`NoMediaMountException`
9	`MMSException`
10	`NullPointerException`

Source: Produced by the author.

We mined Exceptions caught during the Exception Notification (EN) to provide a broader view of the problem. The most common Exception caught in this context was `ActivityNotFoundException`, that is thrown when the Android *runtime* is not able to find the target `Activity` from a `startActivity` call, i.e., when the app tries to open an `Activity` that cannot be reached by the system. We also identified direct references to `Exception` and `Throwable` in catch blocks, showing the reckless attitude of developers upon exception handling code. Previous studies have shown that catching these exceptions are often a source of exception handling bugs (EBERT *et al.*, 2015). Other exceptions ranged from `I/O` exceptions, malformed data formats (`JSONException`), user authentication related and exceptions from system services, like `MMSException`.

An interesting fact about the `ActivityNotFoundException` is that its massive

presence in result confirms the data presented in Figure 9. We see that the vast majority of exception propagation mined in this study comes from `startActivity` calls.

4.10 RQ2: If so, how is the Exception Notification implemented/designed?

To answer this question we analyzed three different aspects of projects: (i) structure: how the components are organized; (ii) notification: what are the code patterns used to send EN and; (iii) handling: how are the code patterns used to handle Exception Notification. So, to answer this question, it is necessary to break this generic question into three specific questions, detailing the aspects mentioned above.

4.10.1 RQ2a: How are the projects structured?

Apps are doing Exception Notification (EN) to both internal and external components.

From projects analyzed manually, we found that 26.6% are using `sendBroadcast` and 70% are using `startActivity` solely. Only 3.3% of the projects use both methods at the same time. We found several patterns in the structure of the projects, and such patterns are deeply related to the definition of Exception Notification, resulting in a clear separation between `sendBroadcast` and `startActivity` routines. This separation was needed because these two strategies of propagation lead to two different structural and code patterns. From now on we will always discriminate the kind of `Notification` we are referring to.

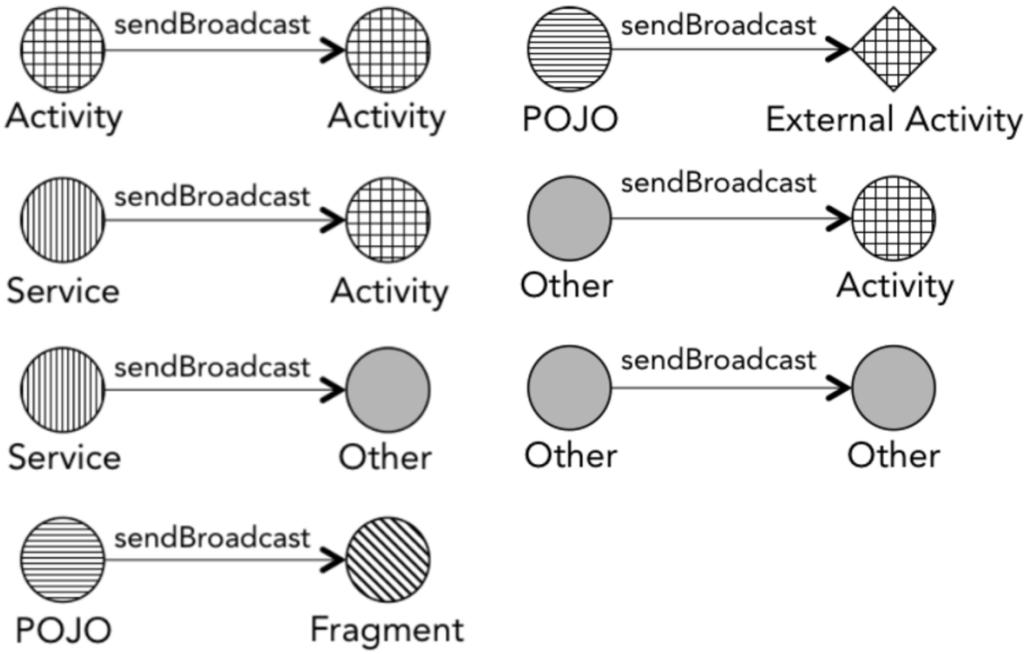
Listing 8 – Cypher queries to find components sending Exception Notifications (produced by the author)

```
1 match(p:Project)-[:has]-(comp)-[:sendbroadcast]->() with distinct comp as
      _comp return _comp.type,count(_comp) order by _comp.type;
```

Components that did EN via `sendBroadcast`are limited to Other, Service, `Activity` and `Plain Java`. There was no `Fragment` using `sendBroadcast` in our findings. Figure 10 shows the Exception Notification (EN) patterns, where the `Senders` are, as said, `Activity`, `Service`, `Plain Java` and Other, and the Handlers of this pattern are `Activity`, `External Activity`, `Other` and `Fragment`. We did not see, notwithstanding, in our findings, `Service`

doing EN to Service. In Listing 8 we present the Cypher query we used to find the components using sendBroadcast and Table 6 shows the distribution of Senders over components.

Figure 10 – sendBroadcast notification patterns



Source: Produced by the author.

Table 6 – Senders doing Exception Notification via sendBroadcast

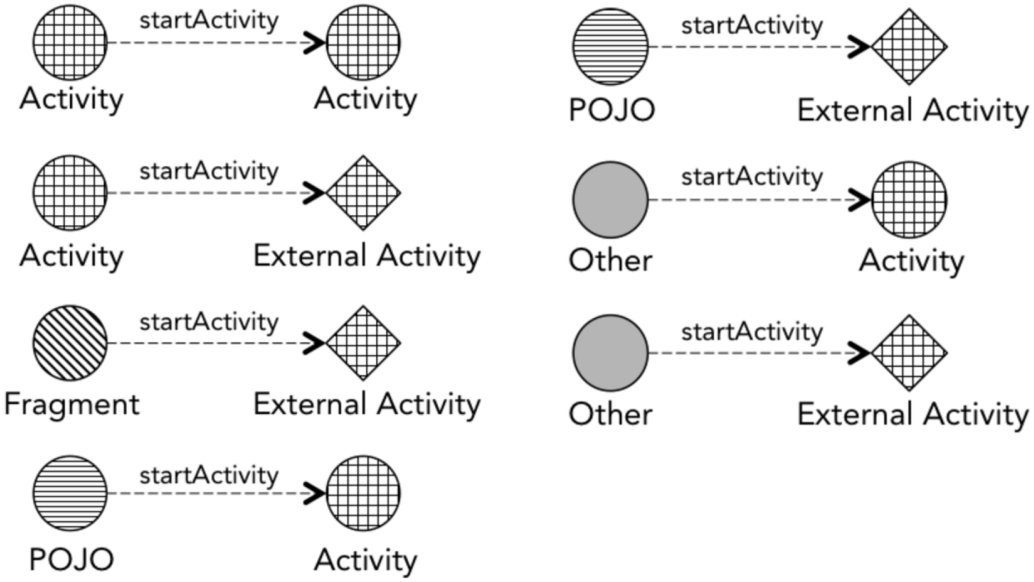
Class Type	**% of sendBroadcast occurrences**
Other	50%
Service	27%
Activity	11.5%
Plain Java	11.5%

Source: Produced by the author.

Components doing EN via startActivity are limited to Activity, Other, Plain Java, and Fragment. Figure 11 shows the patterns we have found when we restrict our context to startActivity. We could not find Service doing EN by startActivity, which is understandable due to background nature of this component. An interesting fact is that the Handlers of this pattern are just Activity, Internal and External. The reason for this fact is simple: the startActivity method is meant to open a new Activity, so that is why we only have this component as a Handler. In Listing 9 we can see the Cypher query used to get this information,

we look specifically for `:startactivity` relationships. `External Activity` classes are also present in this pattern. In Table 7 we can see the distribution of `Senders` components using the `startActivity`.

Figure 11 – `startActivity` notification patterns



Source: Produced by the author.

Listing 9 – Cypher queries to find components doing Exception Notification via `startActivity` (produced by the author)

```
1 match(p:Project)-[:has]-(comp)-[:startActivity]->() with distinct comp as
      _comp return _comp.type,count(_comp) order by _comp.type;
```

Table 7 – Components doing Exception Notification via `startActivity`

Class Type	% of occurrences
`Activity`	79.5%
`Other`	9%
`Plain Java`	6.4%
`Fragment`	5.1%

Source: Produced by the author.

4.10.2 RQ2b: how is the code used to send exception notifications?

`sendBroadcast` notification are directly related to `Service` components, and consequently, associated with background tasks.

In this section, we present our findings related to the actual code used to do the notification. First, we describe the notification based on `sendBroadcast` and, after that, we explain the code based on `startActivity`.

4.10.2.1 SendBroadcast

As said previously, the `sendBroadcast` type of notification is associated with background tasks, which are, in Android apps, mapped to the `Service` components. Here are just a few examples of `Services`: (i) a `Service` that plays music even when the app is running in the background; (ii) a `Service` that consumes data from a `REST API`, and; (iii) a `Service` that authenticates users. 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 10 we can see two examples of Exception Notification (EN) 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 chooses to encode the two `Exceptions (NoMediaMountException and JSONException)` into two different constants respectively: `BROADCAST_ERROR_NO_SD_CARD and BROADCAST_ERROR_BROKE_FILE`. Table 8 shows the exceptions caught in `sendBroadcast` notification, it is worth mentioning, however, that we computed this data in the manual inspection stage of the study. If we compare this result with the exceptions in Table 5, we can see that `NoMediaMountException and JSONException` are present in both lists.

Listing 10 – Exception Notification extracted from <https://bit.ly/2x4OdVQ>

```
1 try {
```

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

Table 8 – Exceptions caught in `sendBroadcast` EN

Exception Type
NoMediaMountException
JSONException
XMPPError
CommunicationException
SaxException
IOException
CommunicationException

Source: Produced by the author.

A seldom source of `sendBroadcast` notification, on the other hand, is the `Activity` component, and only two apps displayed this behaviour. Although, some of these calls originate in `Threads` instantiated within the `Activity`.

4.10.2.2 StartActivity

> The most caught exception was `ActivityNotFoundException` when we dealing with `startActivity` method call.

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 the `Activity` component. The most caught exception in this context was `ActivityNotFoundException`, that, according to Android documentation[4]:

> "This exception is thrown when a call to `Context.startActivity(Intent)` or one of its variants fails because an `Activity` cannot be found to execute the given `Intent`."

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.

4.10.3 RQ2c: How is the code used to handle exceptions?

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

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.

In the case of `sendBroadcast` the code used to handle exceptions are enclosed into a `BroadcastReceiver`, component responsible for handle the broadcast message. We were able to check that the `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 11 we can see an example of a real application that illustrates this problem. The method handling the error (line 10 of the Listing 11) 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.

Listing 11 – Exception handling extracted from <https://bit.ly/2U4T450>

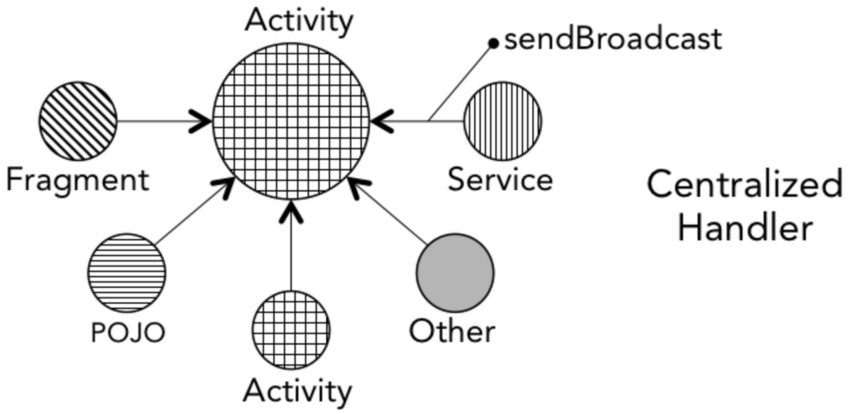
```
1 private BroadcastReceiver broadcastReceiver = new BroadcastReceiver()
2 {
3   @Override
```

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

```
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, one pattern we found interesting was the *One Handler for All Cases*. Usually this lonely handler is an `Activity` that provides a way to handle the error e.g. redirect user to Settings or restart the application in one specific screen e.g. the main screen. In Figure 12 we can see this case.

Figure 12 – One `Activity` to handle all Notifications



Source: Produced by the author.

4.11 Discussion

In this section, we discuss our findings based on the results of the empirical study. First, in Section 4.11.1, we introduce a set of insights with a short description for each one. In Section 4.11.2, we present the bad practices and decisions made by developers when coding with exception propagation in mind. And, finally, in Section 4.11.3 we offer a model to overcome the problem explained in this dissertation.

4.11.1 Insights

In this Section, we present a few insights that cover various aspects of Android Application development. We based our discussion on the metrics extracted by the repository mining tool and the careful observations taken in the manual inspection. The first insight describes a tight relationship between Components and the methods of notification. The next insight explains the flow of the EN between components. We also discuss some aspects of programming practices involved in Exception Notification. Finally, we have considered the consequences of the decisions taken by developers facing the EN challenge.

The `Activity` components prefer to send Exception Notifications to other `Activity` components, while `Service` components broadcast their Exception Notifications.

Description: During this study, we realize a deep connection between the `Activity` Component and the `startActivity` method of Exception Notification. We conclude that, in the presence of an error, an `Activity` would send the notification to another `Activity`. On the other hand, when the EN begins in `Service` components, they are not involved in user interaction, we see that they broadcast their internal errors using the `sendBroadcast` method. The `Activity/startActivity` pair is used to show errors to users and to recover from errors, by starting new `Activities`. Moreover, the `Service/sendBroadcast` pair is used to announce that an error occurred in the execution of the component.

94% of all propagations flows to Activities, internal or external to application.

Description: Almost all EN we manually inspected flows to `Activity` component. This fact suggests that developers when doing EN, are keeping the error handling as close as possible to the User Interface. It also indicates that Android developers reject traditional Object-Oriented (OO) practices, e.g., conventional OO systems are based on hierarchies of classes while in Android Apps, the application logic is embedded mostly in `Activities` (MINELLI; LANZA, 2013). In some cases, the app notifies the user using an `AlertDialog`, and in another instance, the app redirects the user to a Settings screen where he can fill the necessary configuration parameters.

Android developers prefer to encode Exceptions into primitive types rather than put in the exception instance into the `Intent` object.

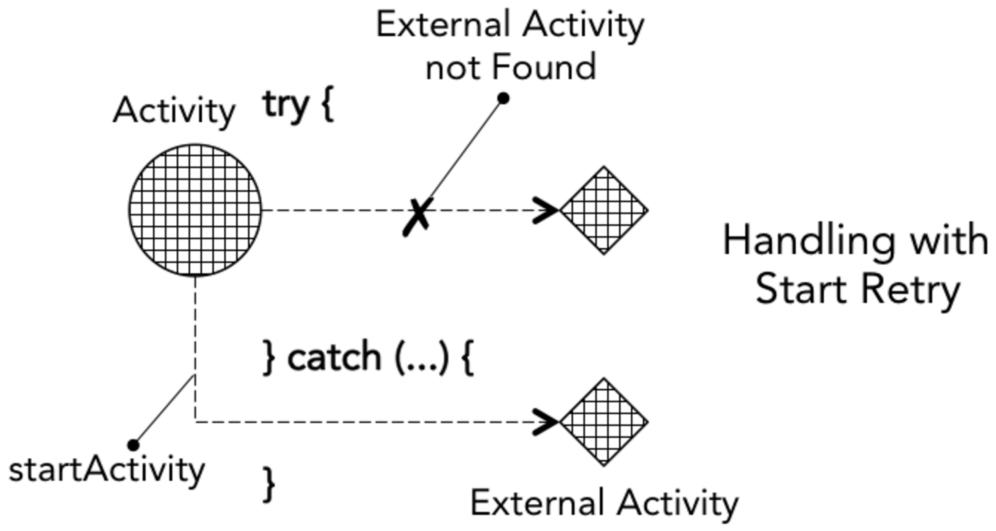
Description: `Intent` objects have a key/value store called `EXTRAS` which are used to transport data between components. This additional space can carry primitive types, arrays, and also `Serializable` and `Parcelable` instances. When the `Exception` instance is added to this `EXTRA` region and the `Intent` is transported by the Android system, it is no longer possible to distinguish the type of that specific Exception instance. Because of that, the component responsible for handling this `Intent` must extract the object from the `EXTRAS` and rethrow it inside a try/catch block. For each possible `Exception` type thrown, we need a different catch block. We could recognize this template in one application investigated. As a result, developers created a more straightforward strategy, that consists in convert the Exception type into constants. These constants are essentially `Strings` that need to be handled using if statements in another component, which is more comfortable than using the `Exception Handling Mechanism` (EHM) as a conditional statement.

The interaction between applications of Android platform can provoke an increase in the number of nested `try/catch` blocks.

Description: The Android platform encourages apps to communicate among themselves. Apps can expose their features and resources to other apps by the communication API using the `Intent` object so that they can provide a better user experience. Crafting specific `Intents`, a developer can start other applications that are present in the system, e.g., open the browser when the user clicks in a button, open the Map application to provide "location services" to your application. To achieve this functionality the Android platform relies on two pieces of information that the Intent carries, the `ACTION` (the action to be performed) and `DATA` (the data to operate on). With `ACTION` and `DATA` in hand, an app can launch a new application by setting up an `Intent` object and calling the `startActivity` method with that Intent. In addition to these attributes, there are secondary attributes that can be set, and they are category, type, component and, extras. However, this call can throw the `ActivityNotFoundException` and, therefore, must be surrounded by a `try/catch` block, given that the `Activity` might not be present in the system. Let's say an application needs to launch the Browser for some reason and, also, the

developer has a list of Browsers that he would try to open before he picks the stock Browser. The `try/catch` block nesting occurs when the scenario described above appears. In Figure 13 we can see this nesting schema.

Figure 13 – `startActivity` retry



Source: Produced by the author.

In Listing 12 we can see a concrete example of this pattern. The developer followed this pattern: first, he tries to open a specific File Manager app using the method `setClassName("com.oem.iFileManager", "com.oem.iFileManager.iFileManager")` from `Intent`, in line 6. In line 7 there is an invocation of the `startActivity` with that `Intent`. If this File Manager app is not present in the system, the `ActivityNotFoundException` is thrown, and the catch block in line 16 is activated. Next, the developer tries to open his second option using the same strategy. In line 19 he set the class name for the `File Manager`, which is different from the first one, and in line 20 he invokes the `startActivity` method. If this second attempt also fails, it means that this application is not installed and the nested catch block is activated in line 23. After these two attempts to open two different File Managers, if both of them fail, the app shows a `Toast` message warning this failure to the user, in line 24. We observed two patterns in this scenario: (i) in the last level of nesting the developer has two options: show a message to the user, as seen above, or open the stock app for that aim, e.g., open the stock Browser or File Manager; and (ii) if the app is not present in the system, the application is redirected to the App Store where the user can install it.

Listing 12 – Nested Try-Catch extracted from <https://bit.ly/2TYqy54>

```
1           try {
2             if (Environment.getExternalStorageState().equals(
3                             Environment.MEDIA_MOUNTED))
4             {
5               Intent i = new Intent();
6               i.setClassName("com.oem.iFileManager", "com.oem.iFileManager.
                    iFileManager");
7                startActivity(i);
8               bPass.setEnabled(true);
9               bFail.setEnabled(true);
10            } else {
11                Toast.makeText(getApplicationContext(),
12                      "No SD Card",
13                      Toast.LENGTH_SHORT)
14                      .show();
15            }
16          } catch(ActivityNotFoundException e) {
17            try {
18              Intent i = new Intent();
19              i.setClassName("com.fb.FileBrower", "com.fb.FileBrower.
                    FileBrower");
20               startActivity(i);
21                  bFail.setEnabled(true);
22                  bPass.setEnabled(true);
23            } catch(ActivityNotFoundException e1){
24              Toast.makeText(getApplicationContext(),
25                  "Open Filemanager Failed!",
26                  Toast.LENGTH_SHORT)
27                  .show();
28              return;
29            }
30          }
31          }
32        });
33 // we ommitted the rest of the code
34 }
```

The lack of an Exception Notification mechanism in Android platform results in bad smells in the application code.

Description: We show that the `Java Exception Handling Mechanism (EHM)` and the Android communication model are not compatible. The `JAVA EHM` is synchronous and stack based while the Android communication model is asynchronous and loosely coupled. We

have shown different patterns in structure and code that are not examples of good practices. On the contrary, we have seen so many bad practices like the use of nested `try/catch` blocks to chain `startActivity` calls, the use of "Returning Codes" as error recovering strategy instead of EHM. The `"Return Code"` strategy tends to mix regular code with the error handling code which is considered a bad practice (BUHR; MOK, 2000). For that reason, though, we need a new model that enables the `Exception` Notification between components without the problems pointed in this work.

4.11.2 Bad Practices

Based on our findings, Android programmers are currently using Android message-passing model as an extension for the Java exception handling mechanism in order to propagate exceptions between components. In summary, Android developers (i) create error codes in the signaling component to identify caught exceptions; (ii) put code inside Intent message; and (iii) send it to other components. Next, in the handling component, developers (iv) must extract the error code from incoming Intent; and (v) map it to a proper handler. However, this approach to propagate exception surfs on well-known exception handling drawbacks (BUHR; MOK, 2000; CHEN *et al.*, 2009) related to the usage of error codes (a.k.a., return codes) to identify exception occurrence during the program execution.

> The error code approach decreases code readability and programmability once the programmer needs to check error codes throughout the program and explicitly changes the program control flow. This approach also decreases the extensibility of exception handling code, turning more difficult to adding, changing, and removing exceptions.

Additionally, the Java's official documentation (GALLARDO *et al.*, 2014) presents three main advantages of adopting an exception handling mechanism instead of another error handling technique, such as error codes. These three advantages are: (i) separating error-handling code from "regular" code; (ii) propagating errors up the call stack; and (iii) grouping and differentiating error types. The way Android developers are propagating exception may leads to the opposite direction of such intended advantages.

First, the adoption of error code strategy creates a confusing tangled code by mixing regular and handling codes. This happens because the handling code is written in the same stands

of the regular code to process incoming Intents carrying or not error codes. Second, once it is not possible to employ the standard Java's exception handling mechanism to send Exception Notifications to Android components, the advantage (ii) cannot be achieved. Finally, the adoption of error code strategy makes difficult benefit from advantage (iii) because exceptions propagated are not Java exception objects, which cannot be grouped or categorized using an exception class hierarchy.

4.11.3 Solving the Problem

> To fully support exception propagation, we claim that the Android exception handling mechanism must be extended to be intentional, dedicated and modular, discouraging adoption of error code-like approaches.

Moreover, such extension must take into account the loosely coupled nature of Android interaction model. Furthermore, it's important provide means to make (i) the exception propagation intentional, using an explicit and dedicated propagating channel; and (ii) the structure of exception handling code more modular, separating it from the regular code and turning automatic the exception/handler matching process.

We recommend a framework that supports the notification of exceptional behavior from one component to another while avoiding the problems we mentioned. Instead of Constants like String and Integer, our system uses Java classes as the Exception Notification type, an advantage over primitive types, while constants are hidden in the code, classes are explicit in the type system. The framework we propose is an Event-Based system, components communicate by generating and receiving Exception notifications. These notifications, as said, are Java classes that represent an incident of interest, e.g., `DownloadFailNotification` might be a class representing the event `"Download Failed."` Our system relies on a *publish/subscribe middleware* that dispatches notifications from producers to consumers (Senders and Handlers, according to our definition). Components subscribe or register themselves as Handlers of specific Exception Notifications, and these events are properly delivered whenever they happen. This solution provides the isolation and modularity required to implement an EHM capable of handling the problems covered in this study.

For instance, RxAndroid[5] and EvenBus[6] solutions provide a reactive and event-based styles, respectively, which can be used as a basis for building such modular and loosely coupled extension for propagate exceptions in Android platform.

The Android platform adopted the Kotlin language as the official language to develop apps[7]. However, the *Exception Handling Mechanism* (EHM) used by Kotlin is similar to the Java EHM. Because of this, the problems described here will likely persist, once it also lacks an Exception Notification Mechanism built-in to the language.

4.12 Threats to Validity

In this section, we discuss the threats to validity associated with our investigation using the Shadish *et al.* (SHADISH *et al.*, 2002) classification (construct, internal, conclusion, and external validity).

Construct Validity. We constructed this study based on the concepts defined in Section 4.2. We assume as a premise that the calls of `startActivity` and `sendBroadcast` inside `catch` blocks reflect the developer's intention to send an exception notification, 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 propagating exceptions 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. As mentioned, libraries like RxAndroid and EventBus have their idiosyncrasies, influencing the way the exceptions are notified and handled. For this reason, 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, and the results kept the same. Our research design, therefore, addresses this menace.

Conclusion Validity. The reliability of the measures is a major concern in empirical

[5] <https://github.com/ReactiveX/RxAndroid>
[6] <https://github.com/greenrobot/EventBus>
[7] <https://developer.android.com/kotlin/index.html>

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 propagating). 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 applications, we have a threat. Instead, the applications came from several domains like native Camera app, apps published on Google PlayStore[8] with thousands of downloads, apps from secondary application stores, open source, and funded projects[9]. 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.

4.13 Final Considerations

This chapter presented in detail the steps conducted in this dissertation. We divided it into four big topics: Overview and Design, Methodology, Results, and Discussion.

In Section 4.1, we gave the design of the study, showing an overview of each step. In Section Definitions, we presented a set of definitions concerning the exception propagation in Android platform. We also introduced the goals and the research questions that guided this study. Two main questions were proposed and the second research question was divided into three.

Regarding the methodology, in Section 4.4, we detailed the how we extract data, using a mining software repository tool called Boa, and what kind of information we were interested, e.g., the types of exceptions caught. In Section 4.5, we presented the procedure and the tools used to transform the output of the mining into a graph data model. We used the Neo4J graph database to help in the organization and visualization of the application's structure. In Section 4.6 we described the manual inspection of the dataset. Section 4.7 described the classification of structural and code patterns found in this dissertation. This methodology was tailored to answer the questions proposed in Chapter 1.

[8] http://play.google.com
[9] http://www.ict-societies.eu/

Next, in Section 4.8, we presented the results found in this study. We were able to confirm our claim that developers were propagating exceptions between components. The results of this chapter prove the relevance and importance of this dissertation, once that no previous study focused on the propagation itself. We discovered and catalog a set of propagation patterns, divided into structural and code. In Section 4.11, we presented a discussion of the findings of this dissertation. In Section 4.11.1, we revealed the insights that emerged from the analysis of the patterns found in previous sections. In Section 4.11.2, we showed how the propagation of exceptions between components could lead to a series of bad practices in code. In Section 4.11.3, we consider an extension of the Android platform to support the propagation of exceptions between components. In Section 4.12, we introduced the threats to this dissertation validity and how we tackled these problems.

5 CONCLUSIONS AND FUTURE WORK

In this chapter, we give the final considerations of this dissertation. The Section 5.1 gives an overview of the work. In Section 5.2 present the limitations that appeared. In Section 5.3, future works are presented.

5.1 Overview

In this dissertation, we addressed the problem of sending exception notifications between Android components and what are the results of this fact in the applications' projects. This problem origins from the non-compatibility between the Java exception handling mechanism and the Android message passing style of communication. We declared 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 actually communicating these Exception Notifications (see Definition 4) to other components. If that is the case, then we investigated the design and implementation of 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 (DYER *et al.*, 2013). We designed this study 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. Initially, we set the Definitions used throughout this research, we defined, for example, the Exception Notification concept. To establish the selection criteria of this study, we used the Definitions as guidelines, and later, using the Mining Software Repository tool, we were able to query hundreds of projects that matched the selection criteria. Next, we translated the output files of the mining tool into a graph of Components and their relationships using the Neo4J DBMS as support. This tool was useful because it provides a web server where we run queries and a visualization tool to exhibit the results, as in Figure **??**. 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 exception notifications between components of applications. Furthermore, we showed that this fact has a significant impact on the source code of applications. Our results indicate that at least 1000

projects meet the selection criteria, 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. Thus, we had answered the Research Questions proposed in Chapter 1. Our contributions can be summarized in: (i) we observed and described a hidden phenomenon that relates the Java Exception Handling Mechanism(EHM) to the Android communication API; (ii) we demonstrated that the Exception Notification pattern could drive developers to adopt a style that decreases the quality of the application code; (iii) another contribution is presenting "real world" applications that send Exception Notifications, including apps from Android codebase such as the MMS; and (iv) we discuss a framework to solve the problem stated in this dissertation.

That said, we can confirm we have achieved the primary aim of this research, once we could found hundreds of examples of real projects doing EN. The following goals depended on the primary objective, and they are related to the organization of the components within the project and, further, how the EN pattern is designed and implemented by applications. We also fulfill these goals as we describe the results and discussions around this problem.

5.2 Limitations

The most significant limitation of this dissertation is related to the definition of Exception Notification that we translated into scripts that mined the repositories. Perhaps, other kinds of exception propagation exist and were not covered in this study or the definition is not complete.

Another limitation was the number of apps manually inspected. The process of manual inspection is very time consuming and inconvenient. If more apps were analyzed, more patterns would be discovered.

5.3 Future Works

The next step to be taken is to survey the developers of the apps used in this work to understand their motivations concerning the Exception Notification. Another future work could be the development of a tool that elegantly solves the EN problem; this tool must provide constructs that express the intention of sending an Exception Notification. Once implemented,

the framework must be evaluated and tested by real developers observing the benefits of its adoption.

Finally, this dissertation paves the way for Android application and library developers to build and use a proper mechanism to send Exception Notifications between components. This work aims to produce a better and saner exception handling/notification mechanism that is more clean, easy to use, test and change.

REFERENCES

BAVOTA, G.; LINARES-VÁSQUEZ, M.; BERNAL-CÁRDENAS, C. E.; PENTA, M. D.; OLIVETO, R.; POSHYVANYK, D. The impact of api change- and fault-proneness on the user ratings of android apps. **IEEE Transactions on Software Engineering**, v. 41, n. 4, p. 384–407, April 2015. ISSN 0098-5589.

BAVOTA, G.; LINARES-VÁSQUEZ, M.; BERNAL-CÁRDENAS, C. E.; PENTA, M. D.; OLIVETO, R.; POSHYVANYK, D. The impact of api change- and fault-proneness on the user ratings of android apps. **IEEE Transactions on Software Engineering**, v. 41, n. 4, p. 384–407, April 2015. ISSN 0098-5589.

BUHR, P. A.; MOK, W. Y. R. Advanced exception handling mechanisms. **IEEE Transactions on Software Engineering**, IEEE Press, Piscataway, NJ, USA, v. 26, p. 820–836, September 2000. ISSN 0098-5589.

CACHO, N.; CÉSAR, T.; FILIPE, T.; SOARES, E.; CASSIO, A.; SOUZA, R.; GARCIA, I.; BARBOSA, E. A.; GARCIA, A. Trading robustness for maintainability: An empirical study of evolving c# programs. In: **Proceedings of the 36th International Conference on Software Engineering**. [S.l.: s.n.], 2014. (ICSE 2014), p. 584–595. ISBN 978-1-4503-2756-5.

CHEN, C.-T.; CHENG, Y. C.; HSIEH, C.-Y.; WU, I.-L. Exception handling refactorings: Directed by goals and driven by bug fixing. **Journal of Systems and Software**, v. 82, n. 2, p. 333–345, 2009. ISSN 0164-1212.

CHIN, E.; FELT, A. P.; GREENWOOD, K.; WAGNER, D. Analyzing inter-application communication in android. In: **Proceedings of the 9th International Conference on Mobile Systems, Applications, and Services**. New York, NY, USA: ACM, 2011. (MobiSys '11), p. 239–252. ISBN 978-1-4503-0643-0.

CHOI, K.; CHANG, B.-M. A lightweight approach to component-level exception mechanism for robust android apps. **Comput. Lang. Syst. Struct.**, Elsevier Science Publishers B. V., Amsterdam, The Netherlands, The Netherlands, v. 44, n. PC, p. 283–298, dez. 2015. ISSN 1477-8424.

COELHO, R.; ALMEIDA, L.; GOUSIOS, G.; DEURSEN, A. v.; TREUDE, C. Exception handling bug hazards in android. **Empirical Software Engineering**, p. 1–41, 2016. ISSN 1573-7616. Disponível em: http://dx.doi.org/10.1007/s10664-016-9443-7. Acesso em: 01 nov. 2017.

DYER, R.; NGUYEN, H. A.; RAJAN, H.; NGUYEN, T. N. Boa: A language and infrastructure for analyzing ultra-large-scale software repositories. In: **Proceedings of the 35th International Conference on Software Engineering**. [S.l.: s.n.], 2013. (ICSE'13), p. 422–431.

EBERT, F.; CASTOR, F.; SEREBRENIK, A. An exploratory study on exception handling bugs in java programs. **J. Syst. Softw.**, Elsevier Science Inc., New York, NY, USA, v. 106, n. C, p. 82–101, ago. 2015. ISSN 0164-1212. Disponível em: http://dx.doi.org/10.1016/j.jss.2015.04.066. Acesso em: 01 nov. 2017.

GALLARDO, R.; HOMMEL, S.; KANNAN, S.; GORDON, J.; ZAKHOUR, S. B. **The Java Tutorial: A Short Course on the Basics**. 6th. ed. [S.l.]: Addison-Wesley Professional, 2014. 864 p. (Java Series). ISBN 0134034082.

GARCIA, A. F.; RUBIRA, C. M.; ROMANOVSKY, A.; XU, J. A comparative study of exception handling mechanisms for building dependable object-oriented software. **Journal of Systems and Software**, v. 59, n. 2, p. 197–222, 2001. ISSN 0164-1212.

HAY, R.; TRIPP, O.; PISTOIA, M. Dynamic detection of inter-application communication vulnerabilities in android. In: **Proceedings of the 2015 International Symposium on Software Testing and Analysis**. New York, NY, USA: ACM, 2015. (ISSTA 2015), p. 118–128. ISBN 978-1-4503-3620-8. Disponível em: http://doi.acm.org/10.1145/2771783.2771800. Acesso em: 01 nov. 2017.

JENKOV, J. **Java Exception Handling**. 1nd. ed. [S.l.]: Amazon Kindle, 2013.

KECHAGIA, M.; SPINELLIS, D. Undocumented and unchecked: Exceptions that spell trouble. In: **Proceedings of the 11th Working Conference on Mining Software Repositories**. New York, NY, USA: ACM, 2014. (MSR 2014), p. 312–315. ISBN 978-1-4503-2863-0.

MAJI, A. K.; ARSHAD, F. A.; BAGCHI, S.; RELLERMEYER, J. S. 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)**. Washington, DC, USA: IEEE Computer Society, 2012. (DSN '12), p. 1–12. ISBN 978-1-4673-1624-8.

MINELLI, R.; LANZA, M. Software analytics for mobile applications–insights & lessons learned. In: **Proceedings of the 2013 17th European Conference on Software Maintenance and Reengineering**. Washington, DC, USA: IEEE, 2013. (CSMR '13), p. 144–153. ISBN 978-0-7695-4948-4.

NEO4J. **Neo4j - The World's Leading Graph Database**. 2012. Disponível em: http://neo4j.org. Acesso em: 01 nov. 2017.

OCTEAU, D.; MCDANIEL, P.; JHA, S.; BARTEL, A.; BODDEN, E.; KLEIN, J.; TRAON, Y. L. 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**. Berkeley, CA, USA: USENIX Association, 2013. (SEC'13), p. 543–558. ISBN 978-1-931971-03-4.

OLIVEIRA, J.; CACHO, N.; BORGES, D.; SILVA, T.; CASTOR, F. An exploratory study of exception handling behavior in evolving android and java applications. In: **Proceedings of the 30th Brazilian Symposium on Software Engineering**. New York, NY, USA: ACM, 2016. (SBES '16), p. 23–32. ISBN 978-1-4503-4201-8. Disponível em: http://doi.acm.org/10.1145/2973839.2973843. Acesso em: 01 nov. 2017.

PANZARINO, O. **Learning Cypher**. [S.l.]: Packt Publishing, 2014. ISBN 1783287756, 9781783287758.

PAYET, E.; SPOTO, F. An operational semantics for android activities. In: **Proceedings of the ACM SIGPLAN 2014 Workshop on Partial Evaluation and Program Manipulation**. New York, NY, USA: ACM, 2014. (PEPM '14), p. 121–132. ISBN 978-1-4503-2619-3.

PAYET Étienne; SPOTO, F. Static analysis of android programs. **Information and Software Technology**, v. 54, n. 11, p. 1192–1201, 2012. ISSN 0950-5849.

PICCO, G. P.; JULIEN, C.; MURPHY, A. L.; MUSOLESI, M.; ROMAN, G.-C. Software engineering for mobility: Reflecting on the past, peering into the future. In: **Proceedings of the on Future of Software Engineering**. New York, NY, USA: ACM, 2014. (FOSE 2014), p. 13–28. ISBN 978-1-4503-2865-4.

QUEIROZ, F. D.; COELHO, R. 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**. [S.l.: s.n.], 2016. p. 131–140. Disponível em: https://doi.org/10.1109/SBCARS.2016.25. Acesso em: 01 nov. 2017.

SASNAUSKAS, R.; REGEHR, J. Intent fuzzer: Crafting intents of death. In: **Proceedings of the 2014 Joint International Workshop on Dynamic Analysis (WODA) and Software and System Performance Testing, Debugging, and Analytics (PERTEA)**. New York, NY, USA: ACM, 2014. (WODA+PERTEA 2014), p. 1–5. ISBN 978-1-4503-2934-7. Disponível em: http://doi.acm.org/10.1145/2632168.2632169. Acesso em: 01 nov. 2017.

SHADISH, W. R.; COOK, T. D.; CAMPBELL, D. T. **Experimental and Quasi-experimental Designs for Generalized Causal Inference**. [S.l.]: Houghton Mifflin Co, 2002. ISBN 0395615569.

SHAHROKNI, A.; FELDT, R. A systematic review of software robustness. **Information and Software Technology**, Butterworth-Heinemann, Newton, MA, USA, v. 55, n. 1, p. 1–17, jan. 2013. ISSN 0950-5849.

WEISER, M. The computer for the 21 st century. **Scientific american**, JSTOR, v. 265, n. 3, p. 94–105, 1991.

YE, H.; CHENG, S.; ZHANG, L.; JIANG, F. Droidfuzzer: Fuzzing the android apps with intent-filter tag. In: **Proceedings of International Conference on Advances in Mobile Computing & Multimedia**. New York, NY, USA: ACM, 2013. (MoMM'13), p. 68:68–68:74. ISBN 978-1-4503-2106-8.

ZHANG, P.; ELBAUM, S. Amplifying tests to validate exception handling code: An extended study in the mobile application domain. **ACM Trans. Softw. Eng. Methodol.**, ACM, New York, NY, USA, v. 23, n. 4, p. 32:1–32:28, set. 2014. ISSN 1049-331X.