What Can Stack Traces Tell us about the Use of Exceptions in Java?

Roberta Coelho*, Georgios Gousios†, Arie van Deursen†, Lucas Almeida*
*Federal University of Rio Grande do Norte
Natal, Brazil
Email: roberta@dimap.ufrn.br,lucas.almeida@ppgsc.ufrn.br
†Delft University of Technology
Delft, The Netherlands
Email: {g.gousios,arie.vandeursen}@tudelft.nl

***Abstract*—In this work, we perform the first large scale analysis of Java stack traces and pinpoint how they can reveal bad programming practices. We mined and analyzed the stack traces embedded in all issues of Java projects available GitHub. Overall, our research set includes 356,057 issues defined at 16,836 projects, from which 28,800 stack traces were extracted. From this set, the stack traces of 482 Android projects were investigated in more detail, in combination with source code and bytecode analysis. In this study some patterns of exception (mis)-use were consistently detected such as: unexpected wrappings (e.g., Errors being wrapped in checked exceptions), revealing that Java hybrid exception model is not fully used according to its purpose; undocumented runtime exceptions signaled by third party code, which is a serious threat to program dependability; and a high prevalence of java.lang exceptions reported on issues, causing approximately 50% of the analyzed stack traces.**

I. Introduction

Modern applications have to cope with an increasing number of abnormal computation states that arise as a consequence of faults in the application itself (e.g., access of null references), noisy user inputs or faults in underlying middleware or hardware. The exception handling mechanism [1] is one of the most used schemes for detecting and recovering from such exceptional conditions. Although the exception handling mechanism have been embedded in several programming languages (e.g. Java, C++, C#) and the target of several studies (e.g. [2], [3], [4], [5], [6], [7], [8]), the exception handling code is often generally poorly understood and the least tested part of software systems.

Often some exception handling constructions may lead the developers into believing that by just re-throwing the exceptions they can forget about the exceptional situations during the development of the "happy path". This "ignore-for-now" approach may turn the exception handling into a generalized "goto" mechanism [9] making the program more complex and even less reliable. This behaviour may lead to the well known uncaught exceptions problem [10], one of the main causes of application crashes.

Moreover, some languages use exceptions to represent programming mistakes (out-of-bounds array index, division-by-zero, access to a null). In these cases exceptions are implicitly signaled by the runtime environment when such condition happens. One of such implicit exceptions (caused by data conversion from a 64-bit floating point to a 16-bit signed integer) was the responsible for the famous failure on Ariane 5's first test flight [11] — leading to the rocket self-destructing 37 seconds after launch and a loss of 500 million dollars. Besides Ariane 5, several applications crash everyday due to uncaught exceptions [10]. The consequences of not careful exception usage may have the opposite effect of its initial intention of improving system robustness.

In Java, when the program fails due to an uncaught exception, it automatically terminates, while the system prints a stack trace to the console. A stack trace includes a detail message and the execution stack frame when the exception occurred. A typical Java stack trace consists of an ordered list of methods that were active on the call stack before the exception has occurred. Stack traces are a useful source of information about system crashes, and is often used to support developers in debugging [12]. Moreover, information from stack traces combined with crash and bug reports can be used to support bug classification and clustering [13], [14], [15], fault-proneness prediction models [16] and even automated bug fixing [17] tools.

However, all these works focus on the analysis of a single system at a time. In our current context, on which we have plenty access to the information available on open-source repositories like GitHub initial questions arise: What is the prevalence of exception stack traces on reported issues? What are the common characteristics of such stack traces?

To answer these questions, we conduct an exploratory study of stack traces in the wild. Using a custom tool called ExceptionMiner, that was specifically developed to for this study, we mined stack traces from issues across all Java projects on GitHub. Overall, we analyzed 356,057 issues from 16,836 projects, from which 28,800 stack traces were extracted and used as a source of information for our analysis. The stack trace analysis was augmented by additional information extracted using byte code and source code analyses done on a carefully selected subset of our initial sample (482 Android projects).

To guide this exploration we compiled general guidelines on how to use exceptions proposed by Gosling [18], Wirfs-Brock [19] and Bloch [20] and then we investigate what stack traces can tell us about the adherence to such practices in Java programs.

This study is built on the following assumption: the stack traces reported on issues contain relevant information related to system crashes. Several studies and techniques have also based on this assumption (e.g., [17] [15] [16]) and we build on it as well. Some outcomes were consistently detected through this large scale analysis of exception stack traces in the wild, such as:

- A multitude of programming mistakes as the main causes of stack traces.
- Undocumented runtime exceptions signaled by third party code (i.e., libraries and Android platform).
- Odd exception chains (e.g., Error wrapping checked and runtime exceptions and vice-versa). As an indication that the purpose of Java hybrid exception model may not have been adequately used.
- Stack traces caused by uncaught checked exceptions.

Hence, the contributions of this study are as follows:

- It performs the first large scale analysis of Java stack traces and how they can reveal bad-practices on the use of exceptions in Java.
- It briefly introduces ExceptionMiner, a tool developed to support the analysis.

The contributions of this work allow for developers of robust Java applications to familiarize with the most common reported exceptions mis-uses which is the first step to help developers to avoid making them; (ii) designers of languages to consider ways of reducing the abundance of NullPointerExceptions, (iii) and tool designers to consider developing tools that enable a cross-project defect analysis. We strongly believe that, specially in the open-source environment, faults are not to be hidden in a private bug issue. Faults should be share and discussed, so that a developer can learn from other projects mistakes. Currently, the search facilities of repositories are very limited, and the ExceptionMiner tool is a contribution in this direction.

The remainder of this paper is organized as follows. Section 2 presents a background on exception handling mechanism. Section 3 presents the study design. Section 4 reports study findings. Section 5 provides further discussions and insights. Section 6 presents the threats to validity associated to this study. Finally Section 7 describes the related work, and Section 8 presents our conclusions and directions for future work.

II. Background

A. Error Handling in Programming Languages

Techniques for error detection and handling are not an optional add-on but a fundamental part of the system [21]. Several approaches exists for detecting errors and responding to them, such as the return-code idiom (very popular in C programs and operating systems), and the exception handling mechanism, embedded in many main stream programming languages.

The main disadvantage of return codes is the fact that the programmer can just ignore the return code and let the program execute in inconsistent state. The exception mechanism [1] was proposed as a way of preventing errors from being ignored, and was embedded in many mainstream programming languages. Among some similarities they mainly differ in the way exceptions are represented and handled in each language [6]:

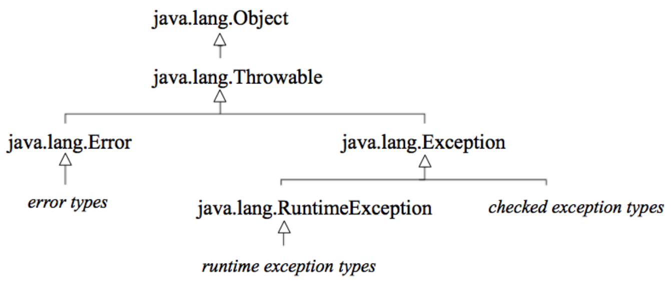


Fig. 1: Exception Hierarchy in Java

Checked Approach. In languages such Modula-3, Guide, and Extended Ada all exceptions are checked, which means that the compiler statically checks if appropriate handlers are provided within the system. Accordingly, the programmer must explicitly specify every exception that can be signaled to support the compile time checking. This approach is known to be costly to maintain (Dooren and Steegmans, 2005) since exception updates are cascading; if a new exception type is added to a method signature then all other methods directly using this method must either change their signatures to include the new exception type or handle the new exception.

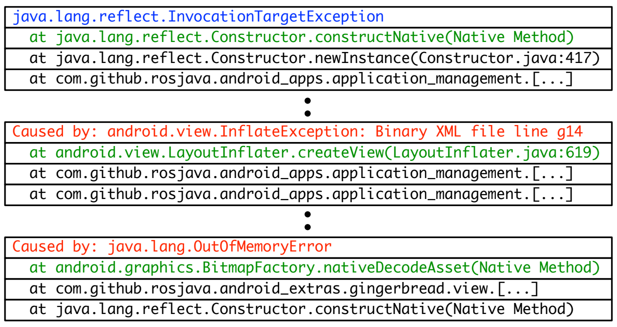
Unchecked Approach. In languages such as Lore, C++, and Arche all exceptions are unchecked but the developer can optionally list exception types on method signatures. However, the developer is not warned by the compiler if an unchecked exception is not handled inside the application. Such an approach may hinder robustness because the client of a method cannot easily know which unchecked exceptions may be thrown, unless the code of the method and the methods called from it is inspected, which can be a very time consuming or infeasible task.

Hybrid Approach. In Java a method can throw either checked or unchecked exceptions. Java is the only language that allows this hybrid approach.

B. Exception Handling in Java

In Java, an exception is represented according to the class hierarchy shown in Figure 1. According to it every exception is an instance of the Throwable class, and can be of three kinds: checked exceptions (extends Exception), run-time exceptions (extends RuntimeException), and errors (extends Error) [18].

Neither run-time exceptions nor errors need to be specified on method signatures, and hence both are referred as unchecked exceptions. The Java Virtual Machine represents as runtime exceptions invalid operations detected in the program (out-of-bounds array index, divide-by-zero error, null pointer references) from which most of the programs are not expected to recover. By convention an Error represents an unrecoverable condition which usually results from failures detected by the Java Virtual Machine, such as OutOfMemoryError and normally should not be handled inside the application.



```
java.lang.reflect.InvocationTargetException
  at java.lang.reflect.Constructor.constructNative(Native Method)
  at java.lang.reflect.Constructor.newInstance(Constructor.java:417)
  at com.github.rosjava.android_apps.application_management.[...]
                              :
Caused by: android.view.InflateException: Binary XML file line g14
  at android.view.LayoutInflater.createView(LayoutInflater.java:619)
  at com.github.rosjava.android_apps.application_management.[...]
  at com.github.rosjava.android_apps.application_management.[...]
                              :
Caused by: java.lang.OutOfMemoryError
  at android.graphics.BitmapFactory.nativeDecodeAsset(Native Method)
  at com.github.rosjava.android_extras.gingerbread.view.[...]
  at java.lang.reflect.Constructor.constructNative(Native Method)
```

Fig. 2: Example of an Exception stack trace in Java.

The Java specification [18] suggests that user-defined exceptions should be checked exceptions. The main reason is because doing so the callers of a method will know about the exceptions that a method can throw, and so they can decide what to do about them. However, there is a long-lasting debate about the use of checked and unchecked exceptions in Java [22], [23], [24][1], since there are pros and cons associated to each of them.

In Java an exception can be thrown in one of the following circumstances [18]: (i) explicitly thrown when a throw statement is executed; (ii) implicitly thrown by the JVM when the evaluation of an expression violates the normal semantics of language (e.g., out-of-bounds array index, division-by-zero, access to a null reference); or (iii) implicitly thrown by the JVM due to an internal error or resource limitation (e.g., OutofMemoryError).

Exception chaining is a common way of propagating exceptions in Java programs, which allows one exception to be wrapped in another exception and re-thrown. Figure 2 presents a stack trace associated which illustrates an exception chaining. The bottom part of the stack trace is the *root cause*, which indicates the first reason for the error to be thrown (in this case, the computer run out of memory). The top part of the stack trace indicates the location of the exception manifestation (which will call *top level exception* along this paper). The execution flow between the root cause and the exception manifestation may include other intermediate exception wrappings. In all levels, the exception *signaler*, is the method that threw the exception, represented on the stack trace as the first method call below the exception declaration.

C. Exception Handling Best Practices in Java

Several general guidelines have been proposed on how to use Java exceptions [9], [18], [19], [20]. A compiled list of the guidelines related to exception types and exception propagation in Java the following: [2]

I-Use checked exceptions to represent recoverable conditions ([9], [18], [19], [20]) The developer should use checked exceptions for conditions from which the caller is expected to recover. By confronting the API user with a checked exception, the API designer is telling the client to handle the exceptional condition. The client can explicitly ignore the exception (swallowing, or converting it to other type) at the expense of the program's robustness [18].

II-Error represents an unrecoverable condition detected by the JVM which should not he handled ([18]). It results from failures detected by the Java Virtual Machine which indicate resource deficiencies, invariant failures or other conditions that make impossible the program to recover.

III-A method should throw exceptions that precisely define the exceptional condition ([18], [20]). To do so, developers should either try to reuse the exception types already defined in the JVM or they should create a specific exception. Throwing general types such as a pure Exception or a RuntimeException is hence considered a bad practice.

IV-Document all exceptions thrown by a method ([9], [18], [19], [20]). The exceptions thrown by a method are an important part of methods interface, and is required to use the method properly. The checked exceptions are already part of the methods signature, and the method caller is aware of the checked exceptions being thrown by it. According to [20], it is also wise to document the explicitly signaled runtime exceptions[3] as carefully as checked exceptions. Doing so, the clients of a method will be aware of the exceptions the method can throw. If the developer fails to do follow this practice (specially when developing library code) it will be difficult or even impossible for the caller to make effective use of such method [19], [20].

These best practices guided our exploratory study described next. Based on information extracted from stack traces (complemented by bytecode and source-code analysis), we investigate whether stack characteristics can reveal whether these practices have been obeyed.

III. Study Design

This work describes an exploratory study which was based on a sequential mixed-methods approach [25] for collecting, analyzing, and integrating both quantitative and qualitative analysis along the stages of the research process. The main goal of this study was to gain a better understanding of the use of exceptions in Java programs based mainly on the information available on exception stack traces. More specifically, we aimed at answering the following research questions:

RQ1 What is the prevalence of stack traces on reported issues?
RQ2 What are the common characteristics of such stack traces?
RQ3 What patterns of exception (mis)-use emerge from the stack trace analysis?

For RQ1 and RQ2, we explore the domain quantitatively, and use theirs answers as a basis to investigate the third research question. RQ3 aimed at finding whether of exception (mis)-uses could be revealed through stack trace analysis. To support this investigation, source code and bytecode analysis

[1] 152 questions in Stackoverlow are related to this debate
[2] We also compiled guidelines related to exception handling but they are out of the scope of this paper.

[3] This excludes the implicitly signaled JVM runtime exceptions related to programming mistakes

was used to leverage the understanding of stack traces. For RQ3 we investigate quantitatively and highlight interesting cases by exploring cases qualitatively. Figure 3 presents an overview of the mixed-methods approach conducted to answer these questions.

We begun our investigation using the dataset provided by the GHTorrent project [26], an off-line mirror of the data offered through the Github API. The Github API data come in two forms; a streaming data flow lists events, such as reporting an issue, happening on repositories in real time, while a static view contains the current state of entities. The data is stored in unprocessed format, in a MongoDB database, while metadata is extracted and stored in a relational database. The GHTorrent dataset covers a broad range of development activities on Github, including issues. The project has been collecting data since February 2012. Up to Dec 2013, when we queried it to retrieve our bootstrapping sample, it included 356.057 issues from 16.836 Java projects (step 1).[4]

After obtaining the bootstrap issue sample from GHTorrent, each issue was processed by ExceptionMiner tool developed in this work which extracted every exception defined on issues or on issue comments (step 2). Since this initial set of Java project is too diverse (it includes libraries, web-based or stand-alone applications), in the second phase of this study we restrict the analysis domain to Android projects (step 3). This decision was driven by the fact that most open source Android applications are known for dealing with several sources of exceptions due to multi-threaded execution and interaction with input/output resources; moreover they share a common well-defined platform and a limited set of libraries.

To support a deeper investigation of the stack traces of Android apps, every exception defined on a stack trace was then classified according to its type (e.g. Error, Exception or RuntimeException) (step 4). The exception classifier was initialized with all the exceptions defined on the Java SDK version 7, the exceptions of the Android (Platform and OS) (level 19), and all the exceptions of the 482 Android projects selected for this study, extracted in the previous step. Moreover, all the packages associated to exceptions and signalers were also extracted to support further analysis (step 5). The signaler classifier is initialized with information about all packages found on Java SDK version 7, and Android Platform and OS (level 19). Finally, additional meta-data is combined to each stack trace (step 6) which used together with manual inspections (step 7) enabled to provide answers to RQ3.

A. Github Data

Kinds of Analyzed Issues. Issues on Github are different from issues on dedicated bug tracking tools such as Bugzilla and Jira. The most important difference is that there are no predefined fields for bug reporters to select from (e.g. severity and priority); Github uses instead a more open ended tagging system. Repositories are offered a pre-defined set of labels, but repository owners can modify them at will. An issue can have an arbitrary set of labels attached. Hence, this study does not restrict the analysis of stack traces defined on defect-issues, based on the assumption that: regardless the type of issue where its is defined the stack trace information contain relevant information concerning the exception use. Moreover, issue and pull requests are dual on Github; all pull requests have a corresponding "backing" issue which are automatically generated. Therefore, we needed to exclude pull request based issues from our analysis. Finally, the Github API allows the automated generation of issues for specific repositories, which automated tools often use to report crashes. In some cases, this led to high number of issues that included stack traces. We identified and filtered those cases out (e.g., the pullwifi project was responsible for almost 50% of all issues created on GitHub, it was removed from our analysis and also from GitHub).

[4]A Java project is specified in GitHub with a specific label.

B. Android Apps Selection

To identify Android projects, we performed a case insensitive search for the term android in the repository's names and short descriptions. The heuristic filtered 2.542 repositories, from which 589 apps had at least one issue containing a stack trace.

Then we performed further cleanup, inspecting the site of every Android reporting at least one stack trace, to make sure that they represented real mobile apps. During this clean up 106 apps were removed because they were either example projects (i.e., toy projects) or tools to support Android development (e.g. selendroid, roboeletric — tools to support the testing of Android apps). The filtered set consisted of 482 apps, from which approximately 50% are also hosted in Google Market Place. For each one of the analyzed projects, we downloaded the source code and using custom scripts, we extracted their package hierarchy (recursive list of of Java packages) and their custom exception types as described next.

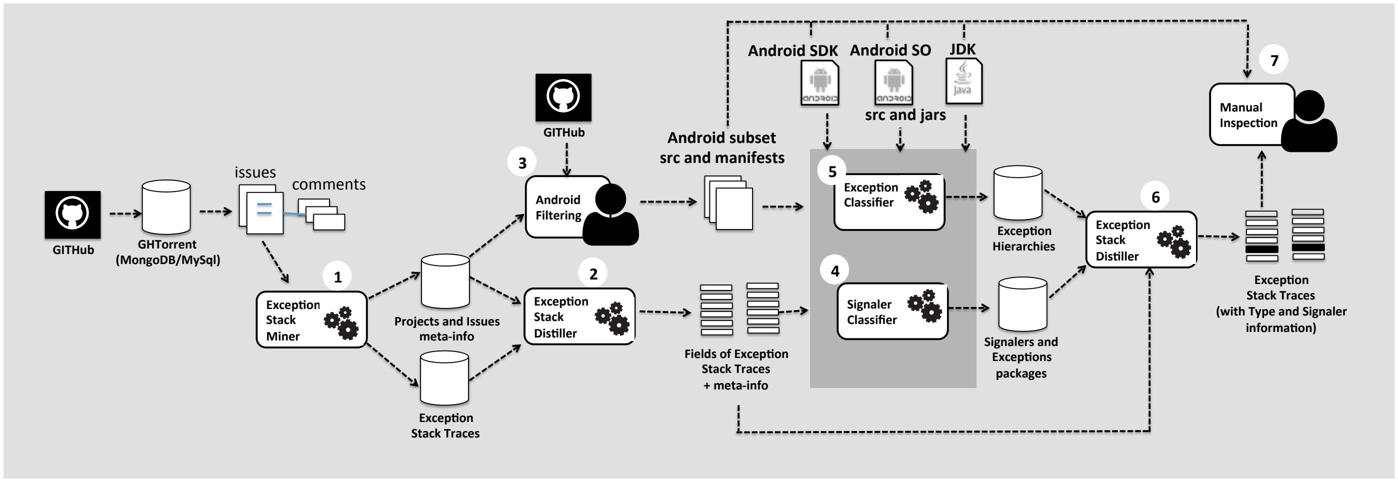
C. The ExceptionMiner Tool

To extract exceptions from issues, we implemented ExceptionMiner, a modular mining tool able to connect to various repositories (such as Google Code, GHTorrent or even directly to Bugzilla), extract issues, mine stack traces from them and classify them in predefined categories. The main components of ExceptionMiner are as follows:

StackTraceMiner and Distiller The first step in the process is mining references to exceptions and stack traces embedded in issues, distilling the information that composes a stack trace and storing the results in a relational database. The extracted stack trace information the exception signaler, the root cause, the exception wrappers. The tool is based on a combination of a regular expression based parser with heuristics able to identify and filter exception names and stack traces inline with text. In contrast to existing issue parsing solutions such as Infozilla, the parser created in this work can extract all causes related to an exception on a stack, and stack traces embedded in logs files.

Exception Classifier The next step in the ExceptionMiner process is classifying exceptions as either checked exceptions, runtime exceptions or an errors. The exception classifier is based on bytecode analysis (based on Design Wizard [27]) that walks up the type hierarchy of a Java exception until it reaches a base exception type.

Exception Signaler Classifier After its initialization it classifies each signaler according to its origin (i.e. Android Plaform,



GITHub
GHTorrent (MongoDB/MySql)
issues
comments
1
Exception Stack Miner
Projects and Issues meta-info
Exception Stack Traces
GITHub
3
Android Filtering
2
Exception Stack Distiller
Android subset src and manifests
Fields of Exception Stack Traces + meta-info
Android SDK
Android SO
JDK
src and jars
5
Exception Classifier
4
Signaler Classifier
Exception Hierarchies
Signalers and Exceptions packages
6
Exception Stack Distiller
7
Manual Inspection
Exception Stack Traces (with Type and Signaler information)

Fig. 3: Overview of the study based on mixed approaches.

Java repository-level metrics	
Repositories	16,836
incl. exception on 1+ issues	4,776 (28,37%)
incl. stack trace on 1+ issues	3,758 (22,32%)
incl. exception on 1+ defect issues	2,019 (41,44%)
incl. stack trace on 1+ defect issues	1,698 (34,84%)

TABLE II: Projects x Stack Traces

Issue-level metrics	Java	Android Subset
Repositories	16,836	482
Issues	356,057	32,582
incl. exceptions	30,236 (8,5%)	3,956 (12,1%)
incl. stack traces	21,013 (5,9%)	3,101 (7,3%)
Defect Issues	55,226	2,376
incl. exception	6,741 (12,2%)	1,557 (65,6%)
incl. stack trace	5,196 (9,4%)	1,393 (58,7%)
Exception Stack Traces		
on issues	28,800	4,208
on issues labeled as defect	6,529	1,699

TABLE III: Quantitative overview of the analyzed dataset

Aplication, Library, Libcore, or Java) based on pattern matching between the signaler name and the packages associated to each category.

Next section presents the results for each of the study phases, providing both a quantitative and qualitative analysis of the outcomes.

IV. RQ1: What is the prevalence of stack traces on reported issues?

Prevalence or prevalence proportion, a term often used in epidemiology, represents the proportion of a population found to have a particular condition (e.g. a disease) over the total number of the studied population. In our work, we used this term to evaluate a condition (i.e., presence of a stack trace) over issues reported on GitHub repositories.

The use of the Github issue tracker in repositories hosted on Github is optional and not widespread. As of March 2014, from the 332,864 non-fork Java repositories registered in the GHTorrent dataset, only 44,323 have received an issue report in their lifetime while from those only 16,837 are being watched by external users. To ensure that our work targets projects that are openly used by users other than their developers, we focused our analysis on just those projects. Using the heuristics presented in Section III-B, we selected 482 repositories featuring Android projects. Table II and Table III presents two views of the extracted dataset, namely grouped by repository and grouped by issue.

From Table III, we can see that repositories that include at least one exception or stack trace on an issue are fairly common: 28% and 22% respectively. Considering only the exceptions and stack traces reported on defect issues we find out that 41% of the repositories reported at least one exception in a defect issue, and approximately 35% of the projects mentioned one stack trace in at least one defect issue.

At the individual issue level, 8,5% and 5,9% of all issues reported in Java repositories contain the full name of an exception or a stack trace respectively. The numbers are similar if we isolate Android projects from the general population (12,1% and 7,3%). Considering only the defect-issues however, in the Android repositories as almost 58,7% of defect-issues include a stack trace. This is not true for the general case however.

V. RQ2: What are the common characteristics of such stack traces?

Based on the assumption that regardless the issue type, every exception stack trace contains relevant information concerning the exception structure of projects analyzed we opted for not restricting the analysis only on defect issues.[5]

A. Top 10 most common exceptions in Java Projects

Table IV illustrates the top 10 most common root causes which represents more than 50% of the analyzed set. The table illustrates i) the number of times each exception appeared as

[5]We conducted the same analysis on the defect issues and the top exceptions are similar to the ones mentioned on defect issues. Due to space limitation we limit to present the general analysis here. More detailed analysis on the defect issues can be found at: www.dimap.ufrn.br/~roberta/icsme2014

Feature	Description	quant_5	mean	median	quant_95	histogram
Java						
all	All issues recorded for the project.	1.00	23.37	3.00	80.00	
with_exception	All issues featuring an exception.	0.00	3.21	0.00	6.00	
with_stack	All issues featuring a stack trace.	0.00	2.67	0.00	4.00	
defect_issues	All issues labeled as defect.	0.00	5.51	0.00	12.00	
defect_with_exception	All issues labeled labeled as defect and including an exception.	0.00	1.82	0.00	1.00	
defect_with_stack	All issues labeled labeled as defect and including a stacktrace	0.00	1.73	0.00	1.00	
Android						
all	All issues recorded for the project.	2.00	67.70	20.00	248.00	
with_exception	All issues featuring an exception.	1.00	8.16	2.00	34.00	
with_stack	All issues featuring a stack trace.	1.00	6.44	2.00	27.00	
defect_issues	All issues labeled as defect.	0.00	13.28	1.00	52.00	
defect_with_exception	All issues labeled labeled as defect and including an exception.	0.00	3.25	0.00	13.00	
defect_with_stack	All issues labeled labeled as defect and including a stacktrace	0.00	2.92	0.00	10.00	

TABLE I: Descriptive statistics for the analyzed issue dataset. Historgrams are in log scale.

Exception	Occurrences		Projects	
	#	%	#	%
java.lang.NullPointerException	7578	26,31%	1671	44,47
java.lang.IllegalArgumentException	1573	5,46%	670	17,83%
java.lang.ClassNotFoundException	1203	4,18%	514	13,68%
java.lang.RuntimeException	1093	3,80%	411	10,94%
java.lang.IllegalStateException	964	3,35%	431	11,47%
java.lang.ClassCastException	886	3,08%	408	10,86%
java.lang.ArrayIndexOutOfBoundsException	834	2,90%	345	9,18%
java.lang.NoSuchMethodError	822	2,85%	369	9,82%
java.io.IOException	656	2,28%	337	8,97%
java.lang.OutOfMemoryError	601	2,09%	235	6,25%
All	23956	100%	3758	100%

TABLE IV: Top 10 most popular exceptions on issues in Java projects

Signaler	Description
android	If the exception is thrown by a method defined in Android Platform or OS, or in a JDK library used by them.
app	If the exception is thrown by an application method or in a DK library used by it.
libcore	If the exception is thrown by one of the core libraries reused by Android (i.e., org.apache.harmony, org.w3c.dom, sun.misc, org.apache.http, org.json, org.xml).
lib	If the exception is thrown by a method that was not defined by any of the elements above.
java	If all methods on the stack trace are JDK library methods.

TABLE V: Sources of exceptions in Android

the root cause of a stack trace (#occurrences), ii) the number of distinct projects reporting such causes on stack traces, and iii) the exception popularity among projects.

We can observe that the NullPointerException was the one with the higher number of occurrences (26,3% of all stacks were caused by NullPointerExceptions) and the one which had the highest popularity across projects. Actually, most of the exceptions in the top 10 list (except java.lang.IllegalArgumentException, java.io.IOException and java.lang.RuntimeException) are exceptions implicitly thrown by Java Virtual Machine, when an expression violates the normal semantics of the Java programming language (e.g. ArrayIndexOutOfBounds) or due to an internal error or resource limitation (OutOfMemoryError).

The high prevalence of NullPointerExceptions, and the other implicitly-thrown exceptions is aligned with the findings other works [16], [28], [29]. For instance, Sunghun et al. [16] showed that in Eclipse bug report system 38% the bugs related to exception handling were caused by NullPointerException; other works on robustness testing [30], [29] showed that most of the automatically detected bugs were due to NullPointerExceptions and implicitly-signaled of Java environment (as the ones found in this study).

B. Top 10 most common exceptions in Android Projects

As detailed in Section III-B, 482 Android projects were carefully selected to enable a deeper investigation of stack trace information, by leveraging stack trace information with the source code and bytecode analysis. For this subset of Android project, we could extract every package related to each exception signaler, and based on this information we could classify every stack trace based on its root signaler as presented in Table V.

For each exception we could discover the number of times it occurred on stacks, the number of distinct projects such stacks were defined (and consequently its popularity) and the of times it was signaled by different signaler types. We could calculate the popularity of each root cause, considering the number of distinct projects they were found and the whole set of projects analyzed with at least one stack trace (482). Table VI presents the mined data.

Programming mistakes are the most common root causes for most types of signalers. We could observe that the NullPointerException is still the exception with higher number of occurrences. The NullPoiterExceptions are mainly signaled inside Android platform and inside application code, although we also find NullPointerExceptions being signaled from third-party libraries. Regarding reusable code, there is no consensus if it is a good or bad practice to re-throw NullPointerException. Some prefer to encapsulate such an exception on IllegalArgumentException, while others [20] argue that the NullPointerException makes the cause of the problem explicit and hence should not be wrapped.

We could observe in this study that the NullPointerException and other other implicitly signaled exceptions represented most of the exceptions reported on issues in both Java and Android subset. For such exceptions, which represent programming bugs or resource limitations, there is usually no proper handling besides presenting an error message to the user and restarting the application — only high fault tolerant systems

Root Exception	Occurrences		Projects		android	libcore	app	lib	java
	#	%	#	%					
java.lang.NullPointerException	1225	30,08%	254	52,70%	473	18	595	137	2
java.lang.IllegalStateException	234	5,75%	99	20,54%	165	12	36	20	1
java.lang.IllegalArgumentException	255	6,26%	94	19,50%	146	6	64	39	0
java.lang.RuntimeException	232	5,70%	91	18,88%	167	1	47	17	0
java.lang.OutOfMemoryError	180	4,42%	56	11,62%	121	15	17	23	4
java.lang.NoClassDefFoundError	73	1,79%	52	10,79%	9	0	37	26	1
java.lang.ClassCastException	94	2,31%	49	10,17%	43	0	40	11	0
java.lang.IndexOutOfBoundsException	127	3,12%	47	9,75%	47	0	71	8	1
java.lang.NoSuchMethodError	57	1,40%	40	8,30%	9	0	39	9	0
java.util.ConcurrentModificationException	54	1,33%	38	7,88%	5	0	43	6	0

TABLE VI: Root Exceptions occurrences and popularity.

Category	Java	Android Subset
Programming logic (java.lang and util)	12625 (41,10%)	2235 (55,64%)
Resources (IO)	12440 (40,50%)	727 (18,10%)
Security	220 (0,72%)	165 (4,11%)
Concurrency	633 (2,06%)	116 (2,89%)
Backward compatibility	1580 (5,14%)	219 (5,45%)
Reflection	1413 (4,60%)	91 (2,27%)
Specific (GUI,FRAMEWORK)	340 (1,11%)	197 (4,90%)
General (Error, Exception, Runtime)	1468 (4,78%)	267 (6,65%)

TABLE VII: Characterization of the 100 root causes.

Type	Android	Libcore	App	Lib	Java	All
Runtime	1075	56	1446	374	10	2961
Error	144	34	168	105	9	460
Checked	110	230	139	145	12	636
Throwable	0	0	1	0	0	1
Undefined	1	0	6	8	0	15
All	1330	320	1760	632	31	4073

TABLE VIII: Types of root exceptions.

need to provide solutions to handle them.

C. Classification of the main causes of top exceptions

To better understand the main causes of such exceptions, we evaluated semantics related to the top 100 root causes reported on stack-issues of both Java dataset and Android subset (which correspond to approximately 77% of all exceptions reported reported on Java stack-issues and 95% of all exceptions reported on Android stack-issues), and classified them according to the the categories presented in Table VII.

Programming errors are the causes of most of the exception stack traces reported on issues. In the Android subset we could find more issues related to Backward compatibility and security issues than on the general data set.

VI. RQ3. What kinds of exception (mis)-use emerge from stack trace analysis?

To answer this question we analyzed general characteristics of reported stack traces and, in the case of stack traces reported on Android repositories, we combine stack trace information with source code and byte code analyses to enable a more detailed analysis. Doing so, we mined some characteristics of stacks that points to best practices violations.

A. Direct instances of RuntimeExceptions being thrown.

From Tables IV and VI we can observe that, in both Java and Android related issues, direct instances java.lang.RuntimeException were thrown in 10,94% of Java repositories and 18,88% of Android repositories respectively. In the Android repositories, most of such exceptions were thrown by the Android platform/OS (167 out of 232).

Throwing general exceptions, such as direct instances of RuntimeException, is considered a bad practice, because the exception type does not carry enough information to identify the cause of the exceptional behaviour, and as a consequence developers need to relying on the exception message which can be neither complete nor precise [18].

B. Undocumented runtime exceptions being thrown.

To better investigate the extent of this problem we classified each stack trace according to the type of the root exception. Table VIII presents the types of root causes of all stack traces reported on the Android repositories. As we can see from Table reftyperoottab, most of the exceptions signaled by third-party code (e.g., Android and libs) were runtime exceptions. After filtering all the exceptions implicitly signaled by JVM (due to programming mistakes) and inspecting the signaler methods of such exceptions (i.e., direct an inderect instances of java.lang.RuntimeException). We could observe observe that only 1 method out of 118 inspected signalers (i.e., 0,8included the runtime exception (reported on the stack) as part of the exception interface of the method (i.e., using throws clause on method signature). This result is aligned with the results of other study conducted by from Sacramento et al [31] which observed that the runtime exceptions in .Net programs are most often not documented.

Such undocumented runtime exceptions represent a serious threat to system robustness, specially when such exceptions are thrown then the third party code is invoked inside the application (e.g. libraries, or framework utility code). In such cases the client usualy do not have access to the source code, and in the absense of the exception documentation it is very difficult or even impossible for the client of such thrid party code to protect the application against undocumented runtime exceptions. As a consequence, the undocumented runtime exception may remain uncaught and lead to system crashes.

On the other hand, we could observe that most of the exceptions signaled by the set of libraries reused by Android platform (i.e., org.apache.harmony, org.w3c.dom, sun.misc, org.apache.http, org.json, org.xml, javax – referred in this work as libcore) were checked exceptions. It is pointed as a good practice for libraries (see Section II-C) because by using checked exceptions libraries can define a precise exception

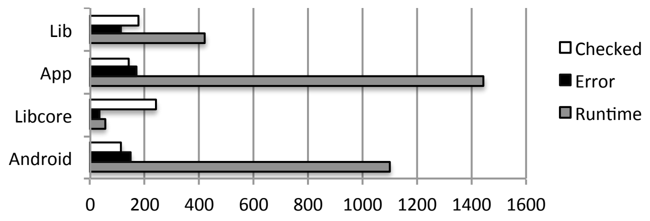


Fig. 4: Distribution of Checked and Unchedked exceptions.

Root Cause	Wrapper Type	Occurrences	Projects
Runtime	-	2360	359
Runtime	Runtime	560	182
Checked	-	422	117
Error	-	381	141
Checked	Runtime	109	70
Checked	Checked	98	42
Error	Runtime	55	38
Runtime	Checked	22	12
Error	Error	15	14
Undefined	-	15	10
Runtime	Error	13	6
Error	Checked	8	7
Checked	Error	7	4
Runtime	Throwable	3	1
Runtime	Undefined	3	3
Throwable	-	1	1
Error	Undefined	1	1

TABLE IX: Kinds of wrappings found on the stacks of 482 ANDROID projects found on Github

interface to its clients. Such libraries are heavily used in several projects, and such precise exception interface may be related to the libraries maturity.

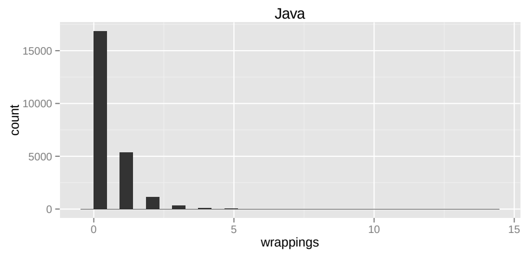
C. Uncaught Checked Exceptions

Although most of the reported exceptions were runtime, we could also find checked exceptions as the root causes of stack traces reported on issues. A checked exception can remain uncaught in two circumstances: if all methods on the execution trace on which this exception flows explicitly specifies the exception, or if the checked exception is wrapped in a runtime and than re-thrown. Since checked exceptions should be used to represent recoverable conditions, ignoring a checked exception, is considered a bad practice. As mentioned before, when the developer is confronted with a checked exception, the designer of the API is telling him to handle the exceptional condition. To better understand what was causing checked exceptions to scape from being handled we investigated the kinds of wrappings that happened on stacks the stacks. Next sections presents the wrappings found and discuss about them.

D. Odd Wrappings

Table IX presents the wrappings found in this study for all stack traces found in Android repositories. As we can see, some of the checked exceptions where indeed wrapped in runtime exceptions or even errors, while most of them were not wrapped along the stack trace.

This analysis also revealed unexpected wrappings such as: Error wrapping a Checked exception; a Runtime wrapping and Error; and an Error wrapping a Runtime. Java is the only language that provides a hybrid exception handling mechanism which offers different types to represent different exception behaviors (i.e., error, runtime and checked) (see Section II-C) . According to Java specification Errors should not be handled inside the system since they usually represent unrecoverable conditions detected by the JVM such as OutOfMemoryError. Checked exceptions should represent recoverable conditions and Runtime exceptions represent are usually related to programming mistakes from which the developer is not expected to recover from.



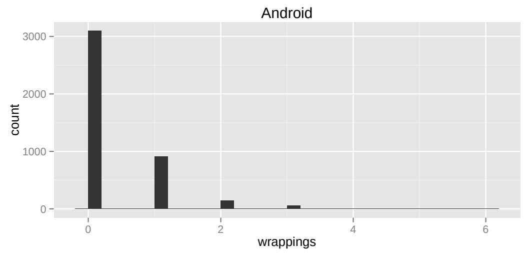


Fig. 5: Number of stack traces wrappings for Java and Android repositories.

We could also observe that in 205 stacks found in Android repository the exception was wrapped more then once. Figure 5 illustrates the number of wrappings per stack, both in Java and Android studied repositories. One interesting multiple-wrapping found in Android repo was the following: Runtime-Checked-Runtime-Checked-Runtime-Checked-Runtime.

Since there is no way of enforcing Java exception type conventions during program development, we could observe the stack trace analysis combined with types these three types of exceptions were used interchangeably. This interchangeability makes the behaviour of the exception handling code more complex and less reliable. For instance, considering a situation were an instance of OutOfMemoryError (a situation that should not be handled) was wrapped in a checked exception (a situation that must he handled), in this case this wrapping is telling to the caller that it should handled.

VII. Discussions

The exception (mis-)use patterns that emerged in this study sheds light on problems that may be reducing the robustness of current Java programs in the face of exceptional conditions. *The undocumented runtime exceptions* thrown by library code - documenting runtime exceptions is a tedious and error prone task, to help developers mitigating this problem tools should be developed to automate document runtime exceptions scaping from library code, few solutions in this directions have been proposed so far [32]. *The odd wrappings* found in this study can lead to what we call an *exception handling confusion problem* which can lead the program to an unpredictable state in the presence of exceptions. To illustrate this problem, we can use one of the examples found in this study: when the developer is confronted with a checked exception, the designer of the API is telling him to handle the exceptional condition, however we could observe that in some cases the checked exception wrapped an OutOfMemoryError, which represent resource deficiency detected by the JVM which make impossible the program to recover. Further investigation is needed on finding ways to help developers in deadling with such problem, either preventing odd wrappings or enabling the developer to better dealing with them. Last but not least, the high number of null pointer exceptions lead us to think on other pertinent research direction: whether language designs which avoid null pointers, such as Monads citeWalde95 (i.e., used in functional languages for values that may not be available or computations that may fail) could improve the robustness of Java programs.

VIII. Threats to Validity

Internal Validity We used a heuristics-based parser to mine exceptions from issues. Our parsing strategy was conservative by default; for example, we only considered exception names using a fully qualified class name as valid exception identifiers, while, in many cases, developers use the exception name in issue description. Conservative parsing may minimize false positives, which was our initial target, but also tends to increase false negatives, which means that some cases may have not been identified as exceptions or stack traces. Our limited manual inspection did not reveal such cases.

External Validity The results presented here were based on mining issues on Github through the GHTorrent dataset. While comprehensive and extensive, GHTorrent is not an exact replica of Github, so several issues might be left out. Due to the way data collection works with GHTorrent, for projects that are relatively inactive, GHTorrent might have not collected a significant proportion of their issues. We did not investigate the extend of this threat on our sample.

Construct Validity Parts of our analysis are based on the availability of stack traces on issues on Github. In using this dataset, we make an underlying assumption: the stack traces reported on issues are representative of real crashes in the applications. The assumption is impossible to mitigate without access to the full set of crash data per application. Services exist to collect all crash data from applications, so access to such a dataset will allow for a thorough replication of our analysis, perhaps at the individual application level.

IX. Related Work

In this section, we present works that are related to our own, distributed in four categories: i) works that use the information available on stack traces; ii) empirical studies on the usage of Java exceptions and its fault proneness; and iii) tools to extract stack traces information from natural language artifacts (i.e., issues and emails).

Analysis of Java Exception Usage. Since the manual analysis of the Java exception flow can easily become infeasible, Robillard and Murphy [3] employed dataflow analysis to find the propagation paths of checked and unchecked exception types. Modeled after Robillard and Murphy's work, other tools have been proposed to support the static analysis of exception flows [33]. The main limitation off all static analysis tools is the number of false positives inherent to static analysis solutions, which can lead to a high number of exception flows, specially if considering Java Environment exceptions and exceptions signaled from libraries. Additionally, Cabral and Marques [7] analyzed the exception handling code of 32 open-source systems, both for Java and .NET. They observed that the action handlers were very simple (e.g., logging and present a message to the user). Reimer and Srinivasan [34] listed a set of bad practices on exception handling that hinder software maintainability and robustness, based on their own experience with Java enterprise applications. Our work differs from those two works as we tried to identify bad practices from the combined use of stack traces extracted from issues and bytecode and source code analysis.

Analysis and Use of Stack Trace Information. Several works have investigated the use of stack trace information to support bug classification and clustering [13], [14], [15], fault-proneness prediction models [16] and even automated bug fixing tools [17]. Kim et al. [14] use an aggregated form of multiple stack traces available in crash reports to detect duplicate crash reports and to predict if a given crash will be fixed. Dhaliwal et al. [15] proposed a crash grouping approach that can reduce bug fixing time in approximately 5%. Wang et al. [13] propose an approach to identify correlated crash types and describe a fault localization method to locate and rank files related to the bug described on a stack trace. Schroter et al. [12] conducted an empirical study on the usefulness of stack traces for bug fixing and showed that developers fixed the bugs faster when failing stack traces were included on bug issues. In a similar study, Bettenburg et al. [35] identify stack traces as the second most stack trace feature for developers. Sinha et al. [17] proposed an approach that uses stack traces to guide a dataflow analysis for locating and repairing faults that are caused by the JVM implicitly signaled exceptions. Kim at al. [16] proposed an approach to predict the crash-proneness of methods based information extracted from stack traces and methods' bytecode operations. They observed that most of the stack traces were related to NullPointerException and other JMV implicitly thrown exceptions had the higher prevalence on the analyzed set of stacks.

Extracting Stack Traces from natural language artifacts. Apart from bug reports, stack traces can be embedded in other forms of communication between developers, such as discussion logs and emails. Being intermixed with text makes the accurate extraction of stacktraces an involved process. Infozilla [36] is based on a set of regular expressions that extract a set of frames related to a stack trace. The main limitation of this solution is that it is not able to extract stack traces embedded on verbose log files (i.e., on which we can find log text mixed with exception frames). Bacchelli and Lanza [37] propose a solution to recognize stack trace frames from development emails and relate it to code artifacts (i.e. classes) mentioned on the stack trace. In addition to those tools, ExceptionMiner is able to both extract stack traces from natural language artifacts and to classify them in a set of predefined categories.

X. Conclusion

In this work we present an exploratory study on which we mined the stack traces embedded in all issues of Java projects available GitHub, and analyzed in mode detail thr stack traces reported on a subset of 482 Android projects. In this study, the information extracted from stack traces was used in combination with source code and bytecode analysis to pin-point patterns of exceptions (mis-)use in Java. And some patterns of exception (mis)-use were consistently detected such as: unexpected wrappings (e.g., Errors being wrapped in checked exceptions), revealing that Java hybrid exception model is not fully used according to its purpose; undocumented runtime exceptions signaled by third party code (which makes almost impossible for library clients to protect against such exceptions); and a high prevalence of java.lang exceptions reported on issues (representing approximately 50% of the analyzed issues). Such (mis)uses detected in out study can negatively affect the robustness of current Java software systems. Such results shed light to needs for developing tools support to help developers dealing with such problems and improving the design of exception handling mechanism in Java.

Acknowledgment

This work is partially supported by: CNPq – Proc. 484209/2013-2 and the NWO TestRoots project (639.022.314).

References

[1] J. B. Goodenough, "Exception handling: issues and a proposed notation," *CACM*, vol. 18, no. 12, pp. 683–696, 1975.

[2] R. Miller and A. Tripathi, "Issues with exception handling in object-oriented systems," in *Proceedings of ECOOP'97*. Springer, 1997, pp. 85–103.

[3] M. P. Robillard and G. C. Murphy, "Designing robust Java programs with exceptions," in *Proceedings of FSE 2000*, pp. 2–10.

[4] H. B. Shah, C. Gorg, and M. J. Harrold, "Understanding exception handling: Viewpoints of novices and experts," *IEEE Trans. Soft. Eng.*, vol. 36, no. 2, pp. 150–161, 2010.

[5] A. Garcia, C. Rubira *et al.*, "Extracting error handling to aspects: A cookbook," in *Proceedings of ICSM 2007*. IEEE, pp. 134–143.

[6] A. F. Garcia, C. M. Rubira, A. Romanovsky, and J. Xu, "A comparative study of exception handling mechanisms for building dependable object-oriented software," *Journal of systems and software*, vol. 59, no. 2, pp. 197–222, 2001.

[7] B. Cabral and P. Marques, "Exception handling: A field study in Java and .Net," in *Proceedings of ECOOP 2007*. Springer, pp. 151–175.

[8] R. Coelho, A. von Staa, U. Kulesza, A. Rashid, and C. Lucena, "Unveiling and taming liabilities of aspects in the presence of exceptions: a static analysis based approach," *Information Sciences*, vol. 181, no. 13, pp. 2700–2720, 2011.

[9] D. Mandrioli and B. Meyer, *Advances in object-oriented software engineering*. Prentice-Hall, Inc., 1992.

[10] J.-W. Jo, B.-M. Chang, K. Yi, and K.-M. Choe, "An uncaught exception analysis for Java," *Journal of systems and software*, vol. 72, no. 1, pp. 59–69, 2004.

[11] J.-L. Lions *et al.*, "Ariane 5 flight 501 failure," 1996. [Online]. Available: http://www.cs.berkeley.edu/\~demmel/ma221/ariane5rep.html

[12] A. Schroter, N. Bettenburg, and R. Premraj, "Do stack traces help developers fix bugs?" in *Proceedings of MSR 2010*, pp. 118–121.

[13] S. Wang, F. Khomh, and Y. Zou, "Improving bug localization using correlations in crash reports," in *Proceedings of MSR 2013*, pp. 247–256.

[14] S. Kim, T. Zimmermann, and N. Nagappan, "Crash graphs: An aggregated view of multiple crashes to improve crash triage," in *Proceedings of DSN 2011*, pp. 486–493.

[15] T. Dhaliwal, F. Khomh, and Y. Zou, "Classifying field crash reports for fixing bugs: A case study of mozilla firefox," in *Proceedings of ICSM 2011*, pp. 333–342.

[16] S. Kim, T. Zimmermann, R. Premraj, N. Bettenburg, and S. Shivaji, "Predicting method crashes with bytecode operations," in *Proceedings of the 6th India Software Engineering Conference*, pp. 3–12.

[17] S. Sinha, H. Shah, C. Görg, S. Jiang, M. Kim, and M. J. Harrold, "Fault localization and repair for Java runtime exceptions," in *Proceedings of ISSTA 2009*, pp. 153–164.

[18] J. Gosling, *The Java language specification*. Addison-Wesley Professional, 2000.

[19] R. J. Wirfs-Brock, "Toward exception-handling best practices and patterns," *Software, IEEE*, vol. 23, no. 5, pp. 11–13, 2006.

[20] J. Bloch, *Effective java*. Pearson Education India, 2008.

[21] M. Bruntink, A. Van Deursen, and T. Tourwé, "Discovering faults in idiom-based exception handling," in *Proceedings ICSE 2006*, pp. 242–251.

[22] "The Java tutorial. Unchecked exceptions: The controversy," http://docs.oracle.com/javase/tutorial/essential/exceptions/runtime.html, Mar 2014, online.

[23] "Java: checked vs unchecked exception explanation," http://stackoverflow.com/questions/6115896/java-checked-vs-unchecked-exception-explanation, Mar 2014, online.

[24] "Checked or unchecked exceptions?" http://tutorials.jenkov.com/java-exception-handling/checked-or-unchecked-exceptions.html, Oct 2013, online.

[25] N. V. Ivankova, J. W. Creswell, and S. L. Stick, "Using mixed-methods sequential explanatory design: From theory to practice," *Field Methods*, vol. 18, no. 1, pp. 3–20, 2006.

[26] G. Gousios, "The GHTorrent dataset and tool suite," in *Proceedings of MSR 2013*, pp. 233–236.

[27] J. Brunet, D. Guerrero, and J. Figueiredo, "Design tests: An approach to programmatically check your code against design rules," in *Proceedings of ICSE/NIER 2009*, pp. 255–258.

[28] G. Fraser and A. Arcuri, "1600 faults in 100 projects: automatically finding faults while achieving high coverage with evosuite," *Empirical Software Engineering*, pp. 1–29, 2013.

[29] C. Csallner and Y. Smaragdakis, "Jcrasher: an automatic robustness tester for Java," *Software: Practice and Experience*, vol. 34, no. 11, pp. 1025–1050, 2004.

[30] A. K. Maji, F. A. Arshad, S. Bagchi, and J. S. Rellermeyer, "An empirical study of the robustness of inter-component communication in Android," in *Proceedings or DSN 2012*, pp. 1–12.

[31] P. Sacramento, B. Cabral, and P. Marques, "Unchecked exceptions: can the programmer be trusted to document exceptions," in *International Conference on Innovative Views of .NET Technologies*, 2006.

[32] M. Van Dooren and E. Steegmans, "Combining the robustness of checked exceptions with the flexibility of unchecked exceptions using anchored exception declarations," in *ACM SIGPLAN Notices*, vol. 40, no. 10, 2005, pp. 455–471.

[33] R. Coelho, A. Rashid, A. Garcia, F. Ferrari, N. Cacho, U. Kulesza, A. von Staa, and C. Lucena, "Assessing the impact of aspects on exception flows: An exploratory study," in *Proceedings of ECOOP 2008*, pp. 207–234.

[34] D. Reimer and H. Srinivasan, "Analyzing exception usage in large Java applications," in *Proceedings of ECOOP 2003*, 2003.

[35] N. Bettenburg, S. Just, A. Schröter, C. Weiss, R. Premraj, and T. Zimmermann, "What makes a good bug report?" in *Proceedings of FSE 2008*, pp. 308–318.

[36] N. Bettenburg, R. Premraj, T. Zimmermann, and S. Kim, "Extracting structural information from bug reports," in *Proceedings of MSR 2008*. ACM, 2008, pp. 27–30.

[37] A. Bacchelli, T. Dal Sasso, M. D'Ambros, and M. Lanza, "Content classification of development emails," in *Proceedings of ICSE 2012*, pp. 375–385.