An Exploratory Study on Exception Handling Bugs in Java Programs

Felipe Ebert, Fernando Castor, Alexander Serebrenik

PII: S0164-1212(15)00086-2
DOI: 10.1016/j.jss.2015.04.066
Reference: JSS 9503

To appear in: *The Journal of Systems & Software*

Received date: 22 August 2014
Revised date: 29 March 2015
Accepted date: 20 April 2015

Please cite this article as: Felipe Ebert, Fernando Castor, Alexander Serebrenik, An Exploratory Study on Exception Handling Bugs in Java Programs, *The Journal of Systems & Software* (2015), doi: 10.1016/j.jss.2015.04.066

Highlights

- We study exception handling bugs from two real systems.
- We survey developers to understand their thoughts about exception handling bugs.
- Analysis of bug repositories shows small percentages of exception handling bugs.
- Exception handling bugs seems to be as hard to fix as other kind of bugs.
- We create an exception handling bug classification.

An Exploratory Study on Exception Handling Bugs in Java Programs

Felipe Ebert[a], Fernando Castor[a], Alexander Serebrenik[b]

[a]*Centro de Informática (CIn), Universidade Federal de Pernambuco (UFPE), Recife, Brazil*
[b]*Eindhoven University of Technology, Eindhoven, The Netherlands*

Abstract

Most mainstream programming languages provide constructs to throw and to handle exceptions. However, several studies argue that exception handling code is usually of poor quality and that it is commonly neglected by developers. Moreover, it is said to be the least understood, documented, and tested part of the implementation of a system. Nevertheless, there are very few studies that analyze the actual exception handling bugs that occur in real software systems or that attempt to understand developers' perceptions of these bugs. In this work we present an exploratory study on exception handling bugs that employs two complementary approaches: a survey of 154 developers and an analysis of 220 exception handling bugs from the repositories of Eclipse and Tomcat.

Only 27% of the respondents claimed that policies and standards for the implementation of error handling are part of the culture of their organizations. Moreover, in 70% of the organizations there are no specific tests for the exception handling code. Also, 61% of the respondents stated that *no* to *little* importance is given to the documentation of exception handling in the design phase of the projects with which they are involved. In addition, about 40% of the respondents consider the quality of exception handling code to be either *good* or *very good* and only 14% of the respondents consider it to be *bad* or *very bad*. Furthermore, the repository analysis has shown (with statistical significance) that exception handling bugs are ignored by developers less often than other bugs. We have also observed that while overly general `catch` blocks are a well-known bad smell related to exceptions, bugs stemming from these `catch` blocks are rare, even though many overly general `catch` blocks occur in the code. Furthermore, while devel-

Email addresses: `fe@cin.ufpe.br` (Felipe Ebert), `castor@cin.ufpe.br` (Fernando Castor), `a.serebrenik@tue.nl` (Alexander Serebrenik)

opers often mention empty `catch` blocks as causes of bugs they have fixed in the past, we found very few bug reports caused by them. On top of that, empty `catch` blocks are frequently used as part of bug fixes, including fixes for exception handling bugs.

Based on our findings, we propose a classification of exception handling bugs and their causes. The proposed classification can be used to assist in the design and implementation of test suites, to guide code inspections, or as a basis for static analysis tools.

Keywords: Exception Handling, Bugs, Surveys, Repository Mining.

1. Introduction

Modern software systems must include provisions to handle errors at runtime. An error is "part of the system internal state which is liable to lead to subsequent failure", while "a system *failure* occurs when the service delivered by the system deviates from what the system is aimed at" [23, p. 198]. Errors may stem from application logic-related erroneous conditions, *e.g.*, an invalid bank account number, undetected bugs, *e.g.*, null dereferences and arithmetic overflow, or environmentally triggered erroneous conditions, *e.g.*, impossibility to open a file or communicate via network. Exception handling mechanisms promote separation of concerns between the normal execution flow of an application and the execution flow in which errors are handled.

At the beginning, exceptions were handled just by returning error codes (success or failure) [12]. ML [36] was the first programming language that implemented typed exceptions, *i.e.*, it allowed developers to define a new type (within the language) for each different type of error. Prior to that, different types of errors were defined by values in the language. The use of types is important because it promotes static checking of exception usage. The throw-catch style of exception signaling and handling was introduced by LISP [34]. More recently, several modern programming languages, like Java, Ruby, C#, C++ and Scala, implement exception handling and a considerable part of the system source code is often dedicated to error detection and handling [12, 53]. Nonetheless, developers have a tendency to focus on the normal behavior of the applications, *i.e.*, what it should do when no errors occur, and deal with error handling only during the system implementation, in an *ad hoc* manner [18, 40].

Several studies [18, 40, 48] argue that the quality of exception handling code is usually poor and that this part of the code is commonly neglected by developers.

Moreover, the exception handling code is often hard to test due to both the numerous exceptional conditions that might occur in a non-trivial application and to the need to stimulate all possible causes for exceptions during testing [16]. Quality of exception handling code, lack of developers' attention, and testing-related challenges specific to exception handling code can therefore be expected to create a fertile ground for bugs. Nevertheless, very few studies analyze exception handling bugs (EH-bugs) occurring in real software systems and no study has attempted to understand developers' perceptions about these bugs. We consider an EH-bug to be a bug whose cause is related to exception handling. It may be a problem related to the definition, throwing, propagation, handling, or documentation of exceptions, to situations where an exception should be throw or handled but is not, and to the use of clean-up actions associated to regions of the code that may throw exceptions.

In this paper, we address this challenge by conducting an empirical study of EH-bugs, *i.e.*, bugs caused by the definition, throwing, propagation, handling, or documentation of exceptions. We aim to gain a better understanding of the causes of these bugs, their frequency, severity, and difficulty of fixing them. Such understanding can be beneficial not only for developers but also for tool designers aiming at supporting software developers in their daily tasks. We complement the objective information about EH-bugs with an investigation of the developers' perceptions about exception handing, in general, and EH-bugs, in particular. The combination of the empirical study of EH-bugs with the investigation of their perceptions allows us to triangulate our findings, such a triangulation being considered an important step in empirical software engineering research [42].

The study involved, therefore, analysis of two data sources: (i) 220 bug reports related to error handling from the Bugzilla repositories of two large systems, Tomcat and Eclipse; and (ii) 154 responses to a survey conducted with software developers from industry and academia. Furthermore, we have inspected the source code of patches attached to the aforementioned bug reports, when available. We have considered the following four research questions:

RQ1 Do organizations and developers take exception handling into account? Our findings indicate that developers do pay attention to exception handling, even though their organizations do not. Only 27% of the respondents claimed that policies and standards for the implementation of error handling are part of the culture of their organizations. In 70% of the organizations there are no specific tests for the exception handling code. Furthermore, 61% of the respondents stated that their organizations give little to no importance to the documentation of exception handling in the design phase. In contrast, 66% of the respondents claim that

they employ exception handling to create ways to tolerate faults and 63% do it to improve the system functionality (they could select multiple answers). Only 17% use exception handling mainly for debugging and 21% because of organizational policies. In addition, the repository analysis has shown that exception handling bugs are ignored by developers less often than other bugs. For example, for Eclipse, 96.65% of EH-bugs have the "Fixed" resolution whereas only 3.26% have "Wontfix" or "Worksforme" as resolutions.

RQ2 How common are EH-bugs? Developers seem to overestimate the frequency of occurrence of EH-bugs. On the average, they believe that 9.72% of the bugs in a system are EH-bugs. Analysis of the bug repositories of Eclipse ant Tomcat yielded much smaller percentages, 0.35 and 1.87%, respectively.

RQ3 Are EH-bugs harder to fix than other bugs? Following the suggestion of Fonseca *et al.* [21] we employed the bug fixing time and the number of discussion messages as proxies for the difficulty of fixing a bug. The analysis of the bug repositories of Tomcat revealed that there is no significant difference for both proxies. For Eclipse there was a statistically significant difference only for the number of discussion messages: it is greater for EH-bugs. This could be an evidence that EH-bugs are as hard to fix as any other bug but they might generate lengthier discussions.

RQ4 What are the main causes of EH-bugs? We discovered that bug reports describing bugs stemming from overly general `catch` blocks, a well-known bad smell in programs that use exceptions [12, 41], are rare, even though there are many opportunities for them to occur and developers report that they have encountered this kind of bug in the past. Empty `catch` blocks, another well-known bad smell, are not only prevalent, as previously reported in literature [12], but also commonly used as part of bug fixes, including fixes for EH-bugs. Moreover, developers often state in the code, by means of comments, that these `catch` blocks do not capture exceptions in practice. However, we found very few bug reports (only 2 among 220) whose causes are empty `catch` blocks, although developers often mention empty `catch` blocks as causes of bugs they have fixed in the past.

In addition to the aforementioned findings, we present a classification of EH-bugs, on Table 1, and their causes—reported during the survey or obtained by analyzing bug reports. The proposed classification can be used as a checklist to design test cases and to assist during code reviews, as well as a basis for static analysis tools for code defect detection, *e.g.*, similar to FindBugs [5].

The remainder of this paper is organized as follows. Section 2 presents the methodology used in this work and also the threats to the validity. Section 3 presents the results from the survey and the analysis of the bug repositories and

Table 1: Classification of EH-bugs.

Lack of a handler that should exist
Exception not thrown
Error in the handler
Error in the clean-up action
Exception caught at the wrong level
General `catch` block
Wrong exception thrown
Exception that should not have been thrown
Wrong encapsulation of exception cause
Lack of a `finally` block that should exist
Error in the exception assertion
Inconsistency between source code and API documentation
Empty `catch` block
Error in the definition of exception class
`catch` block where only a `finally` would be appropriate

also our proposed classification for causes of EH-bugs. Section 5 discusses related work and Section 6 summarizes the contributions and conclusions of this work and discusses future work. Finally, Appendix A presents a comprehensive explanation of the comparison between ours and Barbosa *et al.* [7] EH-bug classification.

2. Methodology

To explore the reality of both EH-bugs and developers' perceptions, we combined investigation of bug repositories with a survey of developers' opinions. Our study focuses on two large and mature open source applications: Tomcat[1] and Eclipse[2]. Both systems are written in Java (which comprise Java versions from 1 to 7, but our analysis did not make any distinction based on the Java version), the language that arguably popularized exception handling, and use Bugzilla as their bug reporting system, which has powerful search features, facilitating analysis of bug reports. Moreover, these systems were examined in a number of earlier empirical studies [31, 32, 43, 56]. Our survey targeted developers of Tomcat and Eclipse. To inquire whether opinions on EH-bugs of the Tomcat and Eclipse

[1] http://tomcat.apache.org
[2] http://www.eclipse.org

developers can be generalized to software developers in general, we have also administered the same survey questionnaire to a diverse group of Brazilian software developers.

This section is organized as follows. Section 2.1 starts out by discussing the notion of an "exception handling bug". Section 2.2 then proceeds to explain how we analyzed the repositories of Tomcat and Eclipse. Section 2.3 presents the survey conducted and in the Section 2.4 we explain how we conducted the data analysis. Finally, we discuss the threats to validity of this work on Section 2.5.

2.1. What is an Exception Handling Bug?

Exception handling is known to be complex; so complex that Black recommends developers to avoid it [10]. Notwithstanding, many modern languages include exception handling mechanisms and programmers do use them in practice, which means that EH-bugs are an expected phenomenon [12, 53]. In order to study EH-bugs, we need first of all to define when a bug is considered an "exception handling bug". Despite the fact that EH-bugs have been studied in the literature in the past (Section 5), no definition of what constitutes an EH-bug is widely accepted. We chose to focus on bug reports where *exception handling is the cause of the problem as opposed to where the problem manifests itself*: *e.g.*, an exception that is not thrown when it should have been thrown or a `catch` block that captures exceptions that it should not have captured are considered EH-bugs while division by zero resulting in an exception being thrown is not considered as an EH-bug. We stress that our definition of the EH-bug differs from the one used by Sawadpong and colleagues [45], which considered as an EH-bug any bug report mentioning the words "exception", "throw", and "threw" or their derivatives. Division by zero, for instance, would always be considered an EH-bug by this previous work but not by ours. Also, our definition extends the one by Barbosa *et al.* [7] by including the cases where the bug occurs in the clean-up action, when the exception should have been thrown or handled while it is not thrown or handled.

More precisely, we define EH-bugs as follows:

> *An **Exception Handling Bug** is a bug whose cause is related to exception handling. EH-bugs can occur when the exception is defined, thrown, propagated, handled or documented; in the clean-up action of a protected region where the exception is thrown; when the exception should have been thrown or handled while it is not thrown or handled.*

To illustrate the definition, consider code snippets pertaining to two real bugs from Eclipse. In Figure 1, the cause of the problem is that the exception has not been handled correctly. The patched snippet on the right-hand side shows the `catch` block to log additional information. We consider this bug to be an EH-bug.

```
        monitor.done();
        fUpdatedExistingClassButton= true;
    } catch (JavaModelException e) {
        JUnitPlugin.log(e);
    }
}
}
```

```
        monitor.done();
        fUpdatedExistingClassButton= true;
    } catch (JavaModelException e) {
        String title= WizardMessages.getString
            ("NewTestSuiteWizPage.error_tile"); //$NON-NLS-1$
        String message= WizardMessages.getString
            ("NewTestSuiteWizPage.error_message"); //$NON-NLS-1$
        ExceptionHandler.handle(e, getShell(), title, message);
    }
}
```

Figure 1: Real EH-bug from Eclipse—bug ID 21018.

Similarly, Figure 2 shows another code snippet from Eclipse (bug repot ID 81417). The problem reported in this case was a `NullPointerException` being thrown. The support team discovered that the problem was a `null` check not being done. In this case, we can see that the problem was not related to exception handling although the problem has manifested itself by means of an exception. We do not consider this bug to be an EH-bug.

2.2. Repository Analysis

To identify the EH-bugs we started our analysis of the Bugzilla repositories of Eclipse and Tomcat by performing a keyword search using the following query:

```
        return declaringType.getTypeParameter(typeVariableName);
    }
} else {
    // member or top level type
    ITypeBinding declaringTypeBinding = getDeclaringClass();
    if (declaringTypeBinding == null) {
}
```

```
        return declaringType.getTypeParameter(typeVariableName);
    }
} else {
    if (fileName == null) return null;
        // case of a WilCardBinding
        // that doesn't have a corresponding Java element
    // member or top level type
    ITypeBinding declaringTypeBinding = getDeclaringClass();
    if (declaringTypeBinding == null) {
```

Figure 2: Real non-EH-bug from Eclipse—bug ID 81417.

catch OR *caught* OR *handl* OR *exception* OR *throw* OR *finally* OR *rais* OR *signal*

These terms encompass related terms that might also be relevant, such as "catches", "raises", "thrown", etc., because Bugzilla considers each search word as a radical to query the database. We tried to cover most of the bug reports related to EH-bugs with that search string. We think that all those keywords are likely to be related to exception handling issues.

Even though Eclipse and Tomcat use Bugzilla, they can have some differences in their layout in search. For example, they can, and in fact they use different *status* and *resolution* categories. Moreover, search results can be restricted to specific *products* or *components*, or bugs having a certain *status*, *e.g.*, "new", "assigned" or "resolved", *resolution*, *e.g.*, "fixed" or "later", *priority* or *severity*.

For Tomcat we included all Tomcat products and all components of these products. We furthermore include bugs with any status except for "unconfirmed" since we do not consider a bug report to refer to an actual bug as long as it has not been "confirmed"; and any possible resolution except for "invalid" and "duplicate". The search in Tomcat's bug repository returned 740 bug reports out of a total of 6,855 (as of January 23, 2013). To identify EH-bugs we have manually analyzed the 740 bug reports and discovered 128 EH-bugs. The manual analysis was conducted by analyzing each bug report, reading all comments in the report and checking the patch code of the bug, when available. This analysis relied on the subjective judgement of the first and the second authors so as to identify only bugs whose cause could be clearly identified as being related to exception handling.

As opposed to Tomcat, Eclipse is a much larger project. To make the study manageable, we focused solely on the Java Development Tools (JDT) product, its Core (Java IDE headless infrastructure) and User Interface (Java IDE user interface) components. Those two components implement very different functionalities, are large, each comprising tens of thousands of lines of code, and, as of January 23, 2013, had 26,002 bugs associated with them. Similarly to Tomcat, the only status we exclude is "unconfirmed", while for resolutions we exclude "invalid", "duplicate" and "not eclipse". The search in Eclipse's bug repository returned 1,779 bug reports out of 26,002. After manual analysis, we found 92 EH-bugs in the Eclipse Bugzilla repository. Table 2 summarizes these numbers for both Eclipse and Tomcat.

	Eclipse		Tomcat	
Bugs resulting from the search	1,779	6.84%	740	10.80%
Exception handling bugs	92	0.35%	128	1.87%
Other kind of bugs	25,910	99.65%	6,727	98.13%
Total number of bugs	26,002	100%	6,855	100%

Table 2: Bugs in Eclipse and Tomcat

Bachmann *et al.* [6] provide some evidence that, in some projects, bugs are reported in mailing lists, instead of using a bug reporting system. Since at least one of the projects where this practice is in use is maintained by the Apache Foundation, similarly to Tomcat, we investigated whether Tomcat developers do the same. Therefore, we conducted a search for EH-bugs in two mailing lists of Tomcat: the tomcat-dev (development list) and the tomcat-announce (the list where security vulnerabilities are announced). The search string used was the same as the one used in the analysis of the Bugzilla repositories, with an additional item to exclude automatic emails related to Bugzilla entries. The search returned 8,573 emails from the development mailing list and only 8 emails from announce list. For the development list, after removing the responses and the svn automatic messages, 860 unique emails remained. We read all those email messages and in the end we found only 7 emails pertaining to EH-bugs. For other bugs, there were 60 emails. Furthermore, from those 860 emails, there are 35 emails where Tomcat experts explicitly say that a Bugzilla entry should always be created for patches and fixes, we present an example[3] on Table 3. From the announce list, 5 emails are announcements and 3 related to other bugs (security). We did not find EH-bugs within this list. Considering the small number of relevant messages (5.5% of the number of EH-bugs found by investigating the Bugzilla repository), we decided to remain focusing on the Bugzilla entries.

"As with any patch, open a bugzilla item and attach it there. Patches attached to mailing list messages tend to get lost."

Table 3: Tomcat developer's citation about Bugzilla usage.

[3]http://www.mail-archive.com/dev%40tomcat.apache.org/msg27325.html

2.3. Survey

The questionnaire used in this work[4] was designed according to the recommendations of Groves *et al.* [25] and Kitchenham and Pfleeger [30]. Firstly, we defined the topics for the questions which resulted in four groups of questions: experience, documentation, testing, and bugs. In addition to questions about exception handling and EH-bugs, we have asked how project design, documentation and testing are carried out in the respondents' organizations. We hypothesize that the software development processes and the policies of the organizations could have affected developers' perception of the importance of bugs, in general, and EH-bugs, in particular.

In the first topic we are interested in knowing the respondents' development experience and the languages they are familiar with. We also included documentation and testing as topics because of the well-known paper by Cristian [18], which states that the exception handling code of a system is in general the least documented, tested, and understood part. To obtain evidence pertaining to this claim, we posed questions that aimed to understand how the exception handling documentation is developed at the respondents' organizations. It contains seven questions. The questions in the testing topic aim to obtain insights about the testing procedures of the developers' organizations, with emphasis on testing of exception handling code. It is important to understand how their testing process is conducted because bugs are usually discovered in the testing phase. The last and largest topic (with eleven questions) is related to bugs themselves. It aims to measure how developers deal with and what are their insights about exception handling bugs and other kinds bugs.

Prior to deploying the questionnaire we have validated it by means of discussion with experts from the Software Productivity Group [5] as well as by applying the survey in two pilot studies. Based on the experts' feedback and results of the pilot studies, questions have been added, removed and revised. After the revisions, the questionnaire was composed of 24 questions ranging from *yes-no* questions through five-point Likert scale multiple choice questions (*never*, *rarely*, *sometimes*, *most of the times*, *always*) to open questions, designed to obtain deeper insights into developers' use of exception handling. Table 4 presents all the questions on the questionnaire.

Our target population consists of developers with practical experience in Java,

[4] http://goo.gl/RM8VR8

[5] http://twiki.cin.ufpe.br/twiki/bin/view/SPG/WebHome

Table 4: Summary of survey questions.

	Experience
1.	For how long have you been a Java developer?
2.	What is the approximate size of the project you are currently working on (LoC estimate)?
3.	Which programming languages have professionally worked with?
	Context
4.	In the design phase of your projects, what is the importance given to the documentation of exception handling?
	Documentation
5.	Are there any specifications, documented policies or standards that are part of your organization's culture related to the implementation of error handling?
5.1.	If you answered yes to the previous question, please describe the policies adopted by your organization
6.	How often are bugs reported at your organization?
7.	How often are bugs related to exception handling reported at your organization?
8.	Does your organization use any tool for reporting and keeping track of bugs?
8.1.	If you answered yes in the previous question, please describe these tools
	Testing
9.	Is there any testing process implemented in your organization?
9.1.	If you answered yes to the previous question, please describe this process
10.	Are there specific tests for the exception handling code in your organization?
	Bugs
11.	How often do you find bugs related to exception handling?
12.	How often do you find bugs that are not related to exception handling?
13.	Estimate the percentage of bugs related to exception handling code in your projects (estimate a value between 0 and 100%).
14.	Have you ever needed to fix bugs related to exception handling?
14.1.	If you answered yes to the previous question, please describe some of these situations.
15.	Select the main causes of bugs related to exception handling you have ever needed to fix, analyze or have found documented (you can select more than one answer).
16.	What is the average level of difficulty to fix bugs related to exception handling?
17.	What is the average level of difficulty to fix other bugs that are not related to exception handling?
18.	What is the average priority/severity of reported bugs related to exception handling code?
19.	Why do you use exception handling in your projects? (you can select more than one answer)
20.	What is your opinion about the quality of exception handling code in your projects compared to other parts of the code?

and specifically Eclipse and Tomcat developers. To contact the developers we have used email addresses found in Bugzilla issue trackers of these projects. To verify whether our findings can be generalized to Java developers beyond these projects, we have also contacted a number of Brazilian software developers and asked them to fill in the questionnaire translated to Portuguese. Translation has been carried out by the first author and verified by the second author. We sent the questionnaire to known points of contact and asked them to redistribute it within their organizations. Over a period of 2 months we sent more than 4,000 emails. In total we obtained 154 responses, 58 from developers of Eclipse and Tomcat, and 96 from the Brazilian developers.

2.4. Data Analysis

The bug reports from Bugzilla were downloaded as a set of XML files. Then, as explained in Section 2.2, we manually analyzed those bug reports to identify EH-bugs. The coding process [46] was used to classify each bug report. Each time we found an EH-bug, we tried to define a classification code for it. In the end, we had several classification codes which represent our EH-bug classification.

Then, we employed a Java program to prepare the data set for statistical analysis to be run in R (version 3.1.1) [1] on an Intel Core i7-2640M 2.80GHz with 6.GB of RAM and Windows 8.1 Pro. With this analysis we want to compare the distributions of EH-bugs and other kinds of bugs. First we check the distributions' normality with Shapiro-Wilk [49] test. If both distributions are normal, then we use the Student's t-test [37] to compare distributions; however if at least one of the distributions is not normal then we use the Mann-Whitney-Wilcoxon [54] test.

For the survey analysis, we also want to compare the distribution of the responses for pairs of questions, *i.e.*, we would like to understand whether an answer to the first question impacts an answer to the second question. To achieve that, the analysis was conducted in three phases. Firstly, we built contingency tables for all question pairs deemed *a priori* interesting. Secondly, following the guidelines of Agresti [2] we analyzed the dependence between the answers using Fisher's test [20] and determined correlation between the answers using Kendall's test [29]. Since we have performed multiple hypothesis tests, it was necessary to adjust the p-values with Benjamini-Hochberg [9] method to reduce the false positive p-values.

2.5. Threats to Validity

As any empirical study our work is subject to a number of threats to validity. We identified three kinds of threats to its validity: internal, external and construct, all of which are discussed below.

Internal Validity. One threat to internal validity is the search string that we employed for preliminary identification of EH-bugs. We tried to cover well-known terms that appear in exception handling literature [24, 17, 3]. Moreover, the search string included terms that are associated with the Java language, since we analyzed the bug repositories of two applications written in Java. As a consequence, the search string is more specific than that employed in a previous study [45].

We have performed manual analysis of the bug reports retrieved by the keyword search. There were several bug reports that did not contain enough information to identify whether they referred to EH-bugs. In some cases, we could mit-

igate the problem by studying the attached patches and by analyzing the source code and the documentation of the system. In general, however, bugs that did not contain enough information were classified as non-EH-bugs.

Unlike previous work [45], we did not rely solely on the search string to identify exception handing bugs. We have manually analyzed the more than 2,000 bug reports that the search produced as results. This manual inspection revealed that the search returns a large number of false positives, most of them mention exceptions but not EH-bugs. Because the inspection was manual, two kinds of mistake may have been committed: (i) a regular bug being classified as an EH-bug; and (ii) an EH-bug not being classified as such. To reduce the chance of this occurring, two of the authors examined many of the bug reports independently. Moreover, a third examiner also analyzed many of the bug reports, which were later reviewed by the authors.

External Validity. The threats to external validity are related to the generalizability of the study results. The first of these threats is that we have only analyzed bug reports referring to two applications, Eclipse and Tomcat. Moreover, for Eclipse, we only examined bug reports associated with two (large-scale) components: Core and UI. The conflicting results discussed in this section highlight this point: software development culture, community, and technical characteristics of each project have a strong impact on the results of a study such as this one. In a similar vein, since the two applications are written in the Java language, it would not make sense to extrapolate our findings to applications written in other languages. Further studies are necessary to establish whether some of the findings of our study, *e.g.*, that bugs stemming from overly general `catch` blocks are rarely reported, apply to Java development in general.

Our survey involved 154 respondents. This number is small and limits the generalizability of the results. Nonetheless, respondents of the survey came from different professional and cultural backgrounds. Furthermore, the largest study to date on the viewpoints of developers about exception handling [48] involved only 15 respondents (they were interviewed instead of responding to a survey). Therefore, we can say that our study is an improvement over the current state-of-the-art.

Another threat is the proportionally low number of EH-bugs found in each system. Even though we analyzed more than 200 bugs, this number is small in comparison to the overall number of bug reports in the repositories of Eclipse and Tomcat. We believe that this is not a fault of our study. Instead, as reinforced by the results of the survey, developers and organizations seem to pay less attention to EH-bugs: they document less and test the system less for their occurrence.

Moreover, as highlighted by previous work on exception handing [12, 14, 41], many EH-bugs are simply never identified by developers and testers.

Construct Validity. The threats to construct validity are related to how properly a measurement actually measures the concept being studied. One threat to the validity of our study is that our survey was conducted with an online, self-administered questionnaire. In the instructions section of this questionnaire we tried to explain the definition of EH-bug to the respondents. Nonetheless, it is possible that they may have misunderstood this definition and answered the questions based on a different understanding of the meaning of EH-bug. We tried to reduce the probability of occurrence by providing some simple examples in the instructions. Moreover, some of the questions include specific information that points out some of the kinds of bugs that we consider to be EH-bugs.

Additionally, our questionnaire might not have covered all questions that could have been asked of the respondents. Nonetheless, the final questionnaire was the result of several discussions between the authors (one of whom is a specialist in exception handling) and with a number of software developers and academics. Moreover, we ran at least two small pilot studies before finally making the questionnaire public. Moreover, respondents may have been influenced in the survey by the options offered in Questions 15 and 19. As the options from Question 15 were provided by the repository study and the options from Question 19 were based on the work of Shah *et al.* [48]. To a certain extent, questions 14 and 14.1 address this problem because the latter asks for spontaneous answers and it is placed before the one where possible answers are suggested (Question 15), thus partially avoiding a potential bias created by question 15. As pointed out elsewhere, the order of related questions may influence survey respondents [52].

There are also threats to the validity of the EH-bug classification. First, it might not cover all possible causes of EH-bugs. Also, we drew that classification from a relatively small amount of data, which means that we cannot statistically validate it. Finally, we might have misread some of the causes, based on the informal comments made by the survey respondents and the text of the bug reports. But we tried to minimize this threat by creating a classification based on both data sources: bug repository analysis and the survey. Moreover, the classification was reviewed by two of the authors.

Another threat to the validity of our work is how we calculate the fix time. Bugzilla does not record the status' history of a bug, so we calculated the fix time by calculating the difference between closing date and opening date of each bug. Hence, a bug could have been reopened and closed again during that time and we would not take that event into account. If that information were available, it could

be used by itself as a third proxy for the difficulty of fixing a bug

3. Study Results

In this section we present the results for both the repository analysis and the survey based on the four research questions stated in the Introduction.

For the survey we have obtained 154 responses. Table 5 summarizes the respondents' professional experience (Questions 1–3). On average, respondents had between 7 and 10 years of software development experience. More than 40% of them work on large projects with 100KLoC or more. Finally, almost all of them identified themselves as Java programmers (98.70%), with the second most popular language being Javascript (61.69%). Approximately 66% of the respondents have worked professionally with at least three programming languages (mean of 3.29).

We found a statistically significant difference between the Brazilian developers and Eclipse/Tomcat developers in terms of Java experience and the size of the current project. Fisher's test returned a p-value less than 0.0001 for Java experience and less than 0.01 for the size of the project. The former result shows that origin of the respondent impacts the experience, and the latter one shows that origin of the respondent also impacts the size of the project. Furthermore, the *odds ratio* [2] for Java experience was 0.062136 and for the size of the project was 0.352. These results indicate that open-source developers tend to have more experience and also work on larger projects than Brazilian respondents. Indeed, Eclipse and Tomcat themselves are huge projects, so Eclipse/Tomcat developers can be expected to indicate that they already have worked on big projects. For most of the questions, however, the answers of developers in the two groups were not statistically different. Hence, in the reminder of this section, except where otherwise noted, we treat the survey respondents as a single population.

Table 5: Professional experience of the survey participants

Question 1 Java experience (years)		**Question 2** Current project (LOC)		**Question 3** Professional experience with programming languages			
<2	7.14%	<20K	24.03%	Java	98.70%	PHP	22.73%
2–5	20.78%	20K–50K	20.13%	Javascript	61.69%	Python	15.58%
5–7	14.94%	50K–100K	13.64%	C++	38.31%	Perl	13.64%
7–10	20.13%	100K–200K	9.09%	C	32.47%	Ruby	9.74%
>10	37.01%	>200K	33.12%	C#	30.52%	Objective-C	6.49%

The main goal of the repository analysis is to discover causes of EH-bugs. As stated in Section 2.2, we found 92 EH-bugs in Eclipse and 128 in Tomcat.

The remainder of this section is organized as follows. In Sections 3.1–3.4 we address the four research questions first by analyzing the survey data and then bug reports. Section 4 contrasts the two data sources, discusses the findings and presents the classification of causes for EH-bugs derived from both data sources.

3.1. RQ1: Do organizations and developers take exception handling into account?

3.1.1. Survey

The survey included seven questions whose goal was to determine whether developers worry about exception handling when they are not directly implementing the system, *e.g.*, during system design or testing.

For the question "there are any specifications, policies or standards that are part of your organization's culture related to the implementation of error handling" (Question 5), we could not compare all answers together because we found a statistically significant difference between the Brazilian developers and Eclipse/-Tomcat developers. Fisher's test returned the p-value = 0.00843. This result shows that origin of the respondent impacts the existence or not of such specifications or policies in their organizations. So, for Brazilian respondents we found more than 80% saying that there are no such specifications or policies in their organizations. For Eclipse/Tomcat developers, the percentage drops to 60%.

An illustrative example of an organizational policy pertaining to exception handling has been provided by a Tomcat/Eclipse developer can be show in Table 6. Although developers say there should be some comments within the ignored catch block, we could find violations of that policy in the repository analysis of Eclipse and Tomcat bugs. For example, in the latest version of Tomcat, we observed that, among the 12 empty catch blocks in the system (110 catch blocks total), we identified 5 that did not include a comment, thus violating said policy.

"any exception should be either rethrown or logged, but not both to avoid duplicate logging of the same exception. If the exception will definitely not be thrown, the ignoring catch block should include a comment in a specific format that tells static code analysis tools that this situation is expected and no warning should be raised here."

Table 6: Developer's citation about organizational policy.

The organizations of 69% of the respondents have a testing processed implemented (Question 9). In this case, we could group together all the respondents. Nevertheless, when asked whether there are specific tests for exception handling code in their organizations (Question 10), the two groups of respondents provided different answers. Fisher's test yielded a p-value of 0.015. This result shows that origin of the respondent impacts the presence of specific tests for exception handling code. For Question 10, 79% of the Brazilian respondents claimed that there are no specific tests for exception handling code while that percentage dropped to 60% for the Eclipse/Tomcat developers.

Moreover, when discussing the importance given to the documentation of exception handling in the design phase (Question 4), 61% of the respondents indicated *no* to *little* importance, almost four times more than respondents indicating *much* to *very much* importance being given. These results suggest that organizations usually do not pay attention to the exception handling code.

About 40% of the respondents consider the quality of exception handling code to be either *good* or *very good* (Question 20) and only 14% of the respondents consider it to be *bad* or *very bad*. This suggests that while exception handling is treated at the implementation level rather than at the design level, the respondents are overall satisfied with the quality of the exception handling code.

Inspired by the work of Shah *et al.* [48], we asked the respondents to select the reasons why they use exception handling (Question 19). The options of Question 19 were created based on the work of Shah *et al.*. We created a list based on their findings while interviewing novice and expert developers. Our intention with this question is not to discover every possible cause for developers to use exception handling. Instead, we would like to elicit some of the strongest reasons and, at the same time, validate the findings of Shah *et al.* [48]. The respondents also had an opportunity to indicate additional reasons, but less than 4% of them have used this opportunity. Table 8 presents reasons most frequently selected by the respondents. Most of the respondents indicated that creating ways to tolerate faults and improving the quality of a functionality are the main reasons to use exception handling. One of the survey respondents provided a particularly interesting spontaneous answer shown in Table 7. It suggests he/she believes that using exception handling is akin to other common activities and practices in software development and not something special.

Only 17% of the respondents said that they use exception handling for debugging purposes. In contrast, in the work of Shah *et al.* [48], which interviewed a group of 8 novice developers and 7 experts, most of the novice developers claimed to use exception handling mostly for debugging and because of language require-

"exception handling is part of code flow—not using exception handling for some arbitrary reason would be like not using 'if' blocks, or not having your code compile."

Table 7: Developer's citation about the reasons to use exception handling.

ments. Expert developers on the other hand, claimed that they use exception handling mainly to convey understandable failure messages. This latter result agrees with our findings presented in Table 8, and indeed most of the respondents can be considered experts (cf. Table 5).

Furthermore, Table 8 indicates that 21% of the respondents use exception handling because of *organizational policies*, slightly lower than the 27% of the respondents indicating the presence of specifications, policies or standards pertaining to the implementation of error handling. It is interesting to note that 28 respondents who indicated the presence of specifications, policies or standards pertaining to the implementation of error handling in their organizations did not cite *organizational policies* as a cause to use exception handling. On the other hand, 19 respondents who claimed that their organizations do not have specifications, policies or standards pertaining to the implementation of error handling did cite *organizational policies* as a cause to use exception handling. In Question 19, the term "organizational policies" appears as one of the possible answers. At the same time, Question 5 cites "policies [..] that are part of your organization's culture". A possible interpretation for this apparent discrepancy is that developers use exception handling because of the policies of their organizations require them to, but these organizations do not provide directions or guidance in this regard.

Finally, it is worth noting that 2% of the respondents who claimed not to use exception handling have only worked professionally with languages that implement exception handling and all of them have worked professionally with Java.

Table 8: Why do developers use exception handling?

To create ways to tolerate faults	66%
To improve the quality of a functionality	63%
Importance of functionality	53%
Language requirement	43%
Organizational policies	21%
To debug a specific part of the code	17%
Does not use exception handling	2%

3.1.2. Repository Analysis

Previous studies [11, 12, 13, 45, 50, 53] have shown empirically that developers do pay attention to error handling. Assessing whether they also pay attention to EH-bugs is harder, since many EH-bugs are probably never uncovered due to insufficient or non-existent testing procedures. Nevertheless, we can use bug report information to analyze the bugs that do get reported. If EH-bugs and other bugs are reported in similar ways, then developers are likely to consider EH-bugs equally important to other bugs. Table 9 compares bug reports pertaining to EH-bugs and other bugs in terms of their priorities, severities, resolutions, status, and the presence of attachments (which usually are patches of the source code or the exception's stack trace) for Eclipse and Tomcat.

Table 9: Priorities, severities, resolutions, status, and the presence of attachments for EH-bug-reports and non-EH-bug-reports for Eclipse and Tomcat.

Category		Eclipse		Tomcat	
		EH-bugs	Other	EH-bugs	Other
Attachment	With	49 (53.26%)	6,799 (26.24%)	38 (29.69%)	1,901 (28.26%)
	Without	43 (46.74%)	19,111 (73.76%)	90 (70.31%)	4,826 (71.74%)
Priority	Blocker	1 (1.09%)	116 (0.45%)	2 (1.56%)	223 (3.31%)
	Critical	0 (0.00%)	537 (2.07%)	4 (3.13%)	392 (5.83%)
	Enhancement	2 (2.17%)	4,440 (17.14%)	10 (7.81%)	1,055 (15.68%)
	Major	8 (8.70%)	1,762 (6.80%)	30 (23.44%)	863 (12.83%)
	Minor	4 (4.35%)	1,463 (5.65%)	9 (7.03%)	548 (8.15%)
	Normal	74 (80.43%)	17,101 (66.00%)	73 (57.03%)	3,467 (51.54%)
	Regression	n/a		0 (0.00%)	58 (0.86%)
	Trivial	3 (3.26%)	491 (1.90%)	0 (0.00%)	121 (1.80%)
Resolution	– (open)	1 (1.09%)	3,251 (12.55%)	3 (2.34%)	219 (3.26%)
	Fixed	88 (96.65%)	15.597 (60.20%)	108 (84.38%)	4,429 (65.84%)
	Later	n/a		2 (1.56%)	141 (2.10%)
	Moved	n/a		0 (0.00%)	1 (0.01%)
	Remind	n/a		0 (0.00%)	21 (0.31%)
	Wontfix	2 (2.17%)	3.775 (14.57%)	10 (7.81%)	1,182 (17.57%)
	Worksforme	1 (1.09%)	3.287 (12.69%)	5 (3.91%)	734 (10.91%)
Severity	P1	1 (1.09%)	581 (2.24%)	7 (5.47%)	535 (7.95%)
	P2	5 (5.43%)	1,60 (6.21%)	52 (40.63%)	2,834 (42.13%)
	P3	85 (92.39%)	22,327 (86.17%)	66 (51.56%)	3,173 (47.17%)
	P4	1 (1.09%)	899 (3.47%)	1 (0.78%)	59 (0.88%)
	P5	0 (0.00%)	494 (1.91%)	2 (1.56%)	126 (1.87%)
Status	Assigned	0 (0.00%)	1,283 (4.95%)	0 (0.00%)	1 (0.01%)
	Closed	0 (0.00%)	444 (1.71%)	6 (4.69%)	237 (3.52%)
	Needinfo	n/a		1 (0.78%)	15 (0.22%)
	New	1 (1.09%)	1,881 (7.26%)	2 (1.56%)	181 (2.69%)
	Reopened	0 (0.00%)	87 (0.34%)	0 (0.00%)	22 (0.33%)
	Resolved	28 (30.43%)	13,349 (51.52%)	119 (92.97%)	6,268 (93,18%)
	Verified	63 (68.48%)	8,866 (34.22%)	0 (0.00%)	3 (0.04%)

In Eclipse, EH-bug reports carry attachments more often than other bug reports: the percentage is twice as high for EH-bugs (53.26% vs. 26.24%). In Tomcat, the percentage of bug reports that carry attachments is very close for exception handing bugs and other bugs (29.69% vs. 28.26%). The choice of adding or not an attachment is made by the developer who is working on the bug and there may be more than one attachment for each bug report. Furthermore, we found a interesting relationship for EH-bugs when comparing the bugs with attachments

and the number of discussion messages. EH-bugs with attachments have more discussion messages than EH-bugs without attachments for both Eclipse (p-value 3.322×10^{-6}) and Tomcat (p-value 4.177×10^{-7}).

Also, it is less common for an EH-bug to be an *enhancement* than for other bugs. We believe that this is a sign that developers do not consider an exception handling fix to be an enhancement, *i.e.*, they do not consider it as an improvement to the system. Moreover, the default *normal* priority is the most common in both systems for EH-bugs as well as non-EH-bugs, and the severities of EH-bugs and non-EH-bugs for both Eclipse and Tomcat are similar.

For the two analyzed systems, *wontfix* and *worksforme* are more common as resolutions for other bugs than for EH-bugs, *e.g.*, in Tomcat the percentage of *wontfix* bugs is more than twice higher. It is also interesting to note that, for both systems, proportionally more EH-bugs have the *fixed* resolution than other bugs, *e.g.*, 96.65% vs. 60.20% in Eclipse. When examined in conjunction, these two results suggest that reported EH-bugs are ignored less often than other bugs. To verify this observation we conducted the χ^2 test [2] of independence to check the relationship between ignored bugs (the ones marked as *worksfome* and *wontfix*) and non-ignored bugs (the ones marked as *fixed*). The results of the test were statistically significant for any commonly used confidence level (for Eclipse the p-value 1.903×10^{-8}, for Tomcat 2.425×10^{-5}) indicating that the bug's resolution depends on whether it is an EH-bug or non-EH-bug.

Finally, we have observed a difference in the usage of Bugzilla for Eclipse and Tomcat. Eclipse developers usually employ the status *verified* to mark a bug as *fixed* and properly working, as recommended by Bugzilla[6]. However, this does not seem to be the case for Tomcat developers who usually add a comment in the bug report itself stating that the bug has been fixed and the system operates as expected. That would explain why there are almost no bugs marked as *verified* for Tomcat *i.e.*, more than 90% of the Tomcat bugs are at the *resolved* state comparable to Eclipse, which has more than 98% EH-bugs (*resolved* + *verified*) and more than 85% for non-EH-bugs (*resolved* + *verified*). It is also interesting to note that 68.48% of the EH-bugs are labeled as "verified" in Eclipse, *i.e.*, supposedly one of the developers verified the patch after the bug was fixed. For non-EH-bugs, the percentage is much lower, 34.22%.

The remaining categories have similar values for the two kinds of bugs. Since there is no obvious difference, we believe that it is possible to say that develop-

[6] `https://bugzilla.mozilla.org/page.cgi?id=fields.html`

ers do pay attention to *documented* EH-bugs at least as much as they do to any other bug. This observation agrees with the comment of one of the Brazilian survey respondents (translated from Portuguese from Question 14.1) as showed in Table 10:

"Of course I already fixed errors inside exception handling blocks, just like I already fixed errors in all places of the source code. The exception handling block is just like any other piece of source code."

Table 10: Developer's citation about fixing EH-bugs.

3.2. RQ2: How common are EH-bugs?

We want to know how usual EH-bugs are. This is an area where there is much room for divergence between the two data sources.

3.2.1. Survey

To assess how common developers believe EH-bugs to be, we asked how often bugs (Question 6) and specifically EH-bugs (Question 7) are reported in their organisations, and how often are they found by the respondents themselves (Questions 11 and 12).

Responses to Questions 6 and 7 are summarized in Table 11. We put the responses to these questions on a numeric scale (from 1-*never* to 5-*always*). As we found a statistically significant difference between the responses of the Brazilian developers and the Eclipse/Tomcat developers for Question 7, we analyzed the responses for Questions 6 and 7 separately. Fisher's test comparing distributions of the answers to Questions 6 and 7 for Brazilian respondents resulted in the p-value 3.828×10^{-4}, implying that the answers to these questions are dependent: the frequency that Brazilian developers report bugs impacts the frequency they report EH-bugs. The same happened for Eclipse/Tomcat developers, Fisher's test comparing distributions of the answers to Questions 6 and 7 for those respondents resulted in the p-value 2.106×10^{-9}. It also implies that the answers to these questions are dependent: the frequency that Eclipse/Tomcat developers report bugs impacts the frequency they report EH-bugs

We used Fisher's exact test to analyze Questions 11 and 12 (responses are summarized in Table 12), *i.e.*, to check whether the kind of bug (EH-bugs or "other") affects perceived frequency of their finding (how often each kind of bug is found). The test indicated that the p-value is too small to be calculated exactly

Table 11: Frequency of bugs found and reported.

	Question 6 and 7 How often are bugs reported?			
Respondents	Brazilian	Eclipse/Tomcat	Brazilian	Eclipse/Tomcat
Finding frequency	EH-bugs	EH-bugs	Other	Other
Never	14.58%	3.45%	4.71%	1.72%
Rarely	15.63%	58.62%	8.33%	6.90%
Sometimes	30.21%	25.86%	20.83%	37.93%
Most of the time	17.71%	10.34%	28.13%	31.03%
Always	21.88%	1.72%	38.54%	22.41%

($p < 2.2 \times 10^{-16}$) meaning that the kind of the bug indeed affects the perceived frequency of their finding.

Table 12: Frequency of bugs found.

	Question 11 and 12 How often do you find bugs?	
Finding frequency	EH-bugs	Other
Never	1.30%	0.65%
Rarely	38.96%	4.55%
Sometimes	53.90%	29.22%
Most of the time	5.84%	59.74%
Always	0.00%	5.84%

Considering that developers rarely seem to find EH-bugs, we wondered whether this might be explained by few bugs being related to exception handling. Therefore, we asked the respondents to estimate the percentage of bugs related to exception handling code in their projects as a value between 0 and 100% (Question 13). Figure 3 shows the histogram and the kernel density estimator [51] of the responses to this question, a method used to estimate the density of the data according to a central value (the kernel). Two respondents declined to answer, reducing the total number of answers for this question to 152. We observe that the estimates range between 0% and 90% with the mean being 9.66%, and the median—5%. In the next subsection we compare these estimates provided by the survey respondents with the results of the repository analysis.

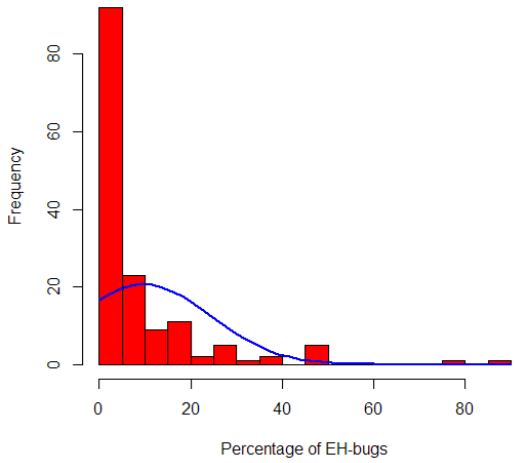

Figure 3: Estimates of the percentage of EH-bugs.

3.2.2. Repository Analysis

Reported EH-bugs are rare. As mentioned in Section 2, we obtained 92 and 128 EH-bugs for Eclipse and Tomcat, or 0.35% and 1.87% of all the bugs, respectively. In spite of this, according to previous studies [12, 53], between 3% and 7% of the lines of code of mature Java applications implement exception handling. This result suggests that either EH bugs are less likely to occur than other kinds of bugs or that they are underreported in the bug repositories.

Previous work has provided evidence that exception handling code is fertile ground for bugs that are difficult to detect [12, 41, 53]. Hence, we expect that EH-bugs are more common than one would assume by looking at the bug reports. Indeed, the survey respondents provided much higher estimates of the number of EH-bugs. The difference between the respondents' estimates and the results of the repository analysis might be attributed to some EH-bugs not being recognized as such either due to limitations of the keyword-based search or due to limitations of the manual analysis following the search. Indeed, many bug reports do not explicitly mention the cause, thus hindering identification of EH-bugs.

3.3. RQ3: Are exception handling bugs harder to fix than other bugs?

Although there is evidence that EH-bugs are hard to detect [12, 41, 53], little is known about how hard they are to fix. In this section, we attempt to address this issue by inquiring the two data sources about the difficulty of fixing reported EH-bugs.

Table 13: How difficult is it on average to fix a bug?

Difficulty	EH-bugs	Other
Very easy	9.09%	0.00%
Easy	34.42%	7.14%
Medium	46.10%	65.58%
Hard	10.39%	25.97%
Very hard	0.00%	1.30%

3.3.1. Survey

To understand whether fixing EH-bugs is more or less difficult than non-EH-bugs, we have asked the respondents to indicate the average level of difficulty of fixing bugs related to exception handling (Question 16) and bugs that are not related to exception handling (Question 17).

Based on the answers to Questions 16 and 17, we have conducted the Fisher's test that resulted in the p-value equal to 4.997×10^{-14}. This result suggests that the kind of the bug (whether EH-bugs or non-EH-bugs) influences the level of the difficulty to fix.

Table 13 shows the percentages of the responses for Questions 16 and 17. EH-bugs are easier to be fixed (*easy* or *very easy*), according to 43.51% of respondents. In sharp contrast, only 7% of the respondents say the same about other kinds of bugs.

Finally, in order to understand importance of EH-bugs to the project, we also asked the respondents about the priority/severity of EH-bugs (Question 18). Responses to this question are summarized in Table 14. Most of the respondents answered that the priority / severity of EH-bugs is *medium*. This result confirms the priority found for EH-bugs in Eclipse and Tomcat from Table 9, where the most often reported priority was *normal*.

Table 14: The average priority / severity of reported EH-bugs?

Very low	5.84%
Low	18.18%
Medium	45.45%
High	25.32%
Very high	5.19%

3.3.2. Repository Analysis

We complemented the survey by means of repository analysis. Since difficulty of bug fixing is not explicitly represented in the repository, we used two measurements as proxies for the difficulty to fix a bug: the number of discussion messages associated with the bug report, and the time to fix the bug, measured in days. Both proxies have been employed with this goal in previous studies [21]. It is important to stress that the fix time was calculated considering bug-repot closing date minus the opening date. Unfortunately Bugzilla does not record the history of the bug's status. Hence, there is no way to account for bugs that are closed and then reopened. Table 15 summarizes basic descriptive statistics for these two measurements. Figure 4 presents *vioplots* [26] for the fix time and the number of comments for Eclipse and Tomcat. Vioplots can be seen as combinations of box plots and kernel density plots.

Table 15: Fix time in days and number of discussion messages for EH-bugs and for other bugs.

		Fix time (days)		**Number of discussion messages**	
		EH-bugs	Non-EH-bugs	EH-bugs	Non-EH-bugs
Eclipse	Min.	0	0	2	1
	1st Qua.	13	6	5	3
	Median	36	38	7	4
	Average	184.3	344.6	7.902	6.285
	3rd Qua.	111.5	245	10	7
	Max.	2,690	4,021	36	206
	SD	466.3	671.3	5.1	6.5
Tomcat	Min.	0	0	1	1
	1st Qua.	45.5	39	2	2
	Median	586.5	512	3	3
	Average	588.8	637.5	4.07	4.534
	3rd Qua.	864	1,092	5	5
	Max.	3,257	4,406	15	97
	SD	601.4	637.0	2.7	4.4

We employed the Wilcoxon test to check whether the measurements differ significantly between EH-bugs and other bugs. In Eclipse, the number of discussion messages is significantly smaller for non-EH-bugs than for EH-bugs ($p \simeq 2.28 \times 10^{-8}$). However, no such difference could be established for the fix time ($p \simeq 0.899$). In Tomcat, for both measurements, the fix time and the number of discussion messages, no statistically significant difference has been found between EH-bugs and non-EH-bugs. For the fix time the p-value associated with

the Wilcoxon test was 0.5584 and for the number of discussion messages p-value was 0.5937.

Combining results of the survey with the results of the repository analysis we tend to answer RQ3 negatively, *i.e.*, there is not enough evidence to suggest that fixing EH-bugs is harder than fixing other bugs. Indeed, in Eclipse there was no significant difference in the bug fixing time while the number of discussion comments of EH-bugs is greater. At the same time, no statistically significant difference could be observed for Tomcat. Moreover, developers think that EH-bugs are easier to fix. In Section 4 we further revisit differences between the survey results and the repository analysis.

3.4. RQ4: What are the main causes of exception handling bugs?

Based on the two data sources, we have devised a classification for EH-bugs. In this section we analyze the results for the survey and the repository analysis independently.

3.4.1. Survey

To uncover the main causes of EH-bugs according to the survey respondents, we asked the developers whether they have ever needed to fix EH-bugs (Question 14) and if so, why (Questions 14.1). We also asked them to select the main causes of bugs related to exception handling they have ever needed to fix, analyze, or have found documented (Question 15). When answering Question 15, the respondents were allowed to select zero or more causes from a list that mentioned a number of causes but also could suggest additional ones. We started the repository analysis before designing the questionnaire because this latter was used to confirm the findings of the repository analysis. Hence, we created an initial classification of exception handling bugs by following a coding process [47], which is a technique for preparing qualitative data to be analyzed quantitatively. This initial classification was used as a basis for Question 15.

Table 16 summarizes these results. The top four causes were provided to respondents, and the remaining ones were suggested by the respondents themselves. The most commonly cited causes for EH-bugs were *lack of a handler that should exist*, *no exception thrown in a situation of a known error* and *programming error in the* `catch` *block*.

It is also interesting to note some relationships that were not statistically significantly different. For example, the size of the project developers worked on (Question 2) compared with i) the existence of organizational EH-bug policies (Question 5), ii) the importance of EH-bug documentation (Question 4) and iii)

the existence of EH-bug testing (Question 10). For all three comparisons we could not say that answers to Question 2 impacts any of the others questions.

Table 16: What are the main causes of EH-bugs?

Bug Classification	Quantity
Lack of a handler that should exist	108
No exception thrown in a situation of a known error	85
Programming error in the `catch` block	84
Programming error in the `finally` block	47
Exception caught at the wrong level	2
`catch` block where only a `finally` would be appropriate	1
Exception that should not have been thrown	1
Wrong encapsulation of exception cause	1
Wrong exception thrown	1
Lack of a `finally` block that should exist	1
Error in the exception assertion	1

To better understand developers' perception of the causes of EH-bugs, we also posed Questions 14 and 14.1. 83% of the respondents to Question 14 have had to fix an EH-bug at some point. 113 out of 154 survey respondents (73.38%) answered Question 14.1. The answers varied widely and many of them refer to specific technologies, frameworks and applications. Besides the causes highlighted by Table 16 (which are from Question 15), the responses to Question 14.1 cited various additional causes. For example, 16 responses mentioned exceptions caught at the wrong level (14.16% of the answers) and 19 responses mentioned empty catch blocks as common causes of EH-bugs (16.81% of the answers). Furthermore, three respondents cited both causes (*exceptions caught at the wrong level* and *empty* `catch` *blocks*) in their answers.

3.4.2. Repository Analysis

Table 17 presents the causes of bugs identified while examining the bug reports. For Eclipse, the three most common causes of EH-bugs are (i) *exception not handled* (34.78%), (ii) *error in the handler* (29.35%), and (iii) *exception that should not be thrown* (14.13%). For Tomcat, the most common causes are (i) *error in the handler* (37.50%), (ii) *exception not handled* (19.53%), and (iii) *exception not thrown* (16.41%).

Figure 5 shows an example of *exception not handled* and Figure 6 shows an example of *error in the handler*, both from Eclipse.

Table 17: EH-bug classification according to repository analysis.

Bug Classification	Tomcat		Eclipse	
Exception not handled	25	19.53%	32	34.78%
Exception not thrown	21	16.41%	6	6.52%
Exception that should not have been thrown	4	3.13%	13	14.13%
Wrong exception thrown	10	7.81%	5	5.43%
Error in the handler	48	37.50%	26	28.26%
Error in the finally block	1	0.78%	4	4.35%
General `catch` block	2	1.56%	1	1.09%
Inconsistency between source code and API	0	0.0%	3	3.26%
Empty `catch` block	1	0.78%	1	1.09%
Error in the definition of exception class	0	0.0%	1	1.09%
Invalid or non-existent root cause	16	12.50%	0	0.0%

Figure 7 shows an example of *exception that should not be thrown* and Figure 8 shows an example of *exception not thrown*, both from Tomcat.

It is interesting to note the differences between the two systems. On the one hand, *exception not thrown* is a common cause of EH-bugs in Tomcat but not in Eclipse. On the other hand, *exception that should not have been thrown*, the third most common cause of EH-bugs in Eclipse, does not rank among the top 5 most common causes in Tomcat. Furthermore, *invalid or non-existent root cause* is the originator of 12% of the EH-bugs in Tomcat but no bugs in Eclipse.

Another interesting point is the rareness of *empty* `catch` *blocks* as causes of bugs: only one for each application. *Empty* `catch` *blocks* are in widespread use in large-scale, mature applications [12, 40]. Notwithstanding, developers seem to believe that they create many problems [35, 38, 39, 40, 44] because they can make bugs subtler and hinder debugging. Indeed, empty `catch` blocks ignore exceptions, and therefore the problems that the ignored exceptions signalize can only be detected indirectly. Moreover, since there is no stack trace and, in fact, no exception, finding the root cause of the problem becomes particularly difficult. Finally, we have seen comments stating that empty `catch` blocks are used when developers are certain that a given exception cannot be thrown, indicating that the `catch` block can never be reached. However, as a consequence of software maintenance, the preconditions that guaranteed the impossibility of the exception being thrown can be violated, thus introducing bugs that are hard to detect. Therefore, we believe that *empty* `catch` *blocks* constitute a maintenance risk. It is surprising therefore that only few bug reports mention empty `catch` blocks as their cause.

A similar case can be made for `catch` clauses and generic exception types, such as `Throwable` or `Exception`. There is convincing evidence [14, 22, 41] that they are often sources of bugs. Despite of this, overall, we only found three bug reports with this cause. Even though *empty* `catch` *blocks* are a well-known bad smell, we found ten bugs whose patches use *empty* `catch` *blocks*. Figures 9 and 10 show one example from Eclipse and one from Tomcat, respectively. Even more patches for EH-bugs might use *empty* `catch` *blocks* as not every bug report is explicitly associated with the corresponding patch.

4. Discussion

Contrasting the actual bugs that we identified in the repositories with the bugs that developers have had to fix, we notice that a number of problems have rarely been documented in bug reports. Some of these problems have been mentioned by several respondents, *e.g.*, exceptions caught at the wrong place. Moreover, there is ample opportunity for problems stemming from inadvertently caught exceptions to manifest. For example, the trunk version of Tomcat 7.0 in April 24th 2013 had 280 `catch(Throwable...)` blocks and more than 520 `catch(Exception...)` blocks. Previous studies [14, 41] using static analysis tools have shown that general `catch` blocks do capture exceptions they were not intended to in practice.

Summarizing the preceding discussion we can say that (i) developers claim to have fixed bugs with causes that rarely appear in bug reports; (ii) bugs with these causes are known to be hard to find without proper testing, *e.g.*,*exceptions caught at the wrong place*; and (iii) exception handling code is rarely tested. These hard-to-find bugs manifest only indirectly and tracking the cause is difficult. One of the survey respondents summarized this situation when asked "have you ever needed to fix bugs related to exception handling? (if yes, please describe some of these situations)" (Question 14.1):

> *"Exceptions caught too early allowing the program to proceed with invalid data,* e.g.*, returning null from a method instead of throwing a meaningful exception. This usually causes another related exception soon, but in hairy cases may cause data corruption and other irregularities".*

There are diametrically different opinions on the subject of *empty* `catch` *blocks*. As discussed in Section 3.4, a number of developers seem to believe that *empty* `catch` *blocks* are problematic. Out of the 113 survey respondents who described

EH-bugs they had to fix in the past, 19 claimed to have fixed bugs where *empty* `catch` *blocks* were either potential or actual bug causes, exemplified by the following comment (translated from Portuguese, Question 14.1):

> *"Many developers consider that 'swallowing' exceptions is normal, so that the system does not show errors to the user. However, the system behavior becomes unpredictable. 'Swallowed' exceptions are the worst problem that I used to find in the systems."*

Nevertheless, examination of the bug reports for the two target applications revealed only two bugs due to empty `catch` clauses. In addition, some survey respondents seem to radically disagree, in spite of admitting that 'swallowed' exceptions hinder debugging:

> *"I'm also an Eclipse committer (on Platform/UI). On 4.2 we've changed how parts (*e.g.*, editors and views) are rendered. Our new system silently swallows otherwise-uncaught exceptions. Tracing what happens when an EditorPart or ViewPart throw an uncaught exception is a teensy bit annoying."*

```
catch(Throwable t) {
    if (t instanceof ThreadDeath) {
        throw (ThreadDeath) t;
    }
    if (t instanceof VirtualMachineError) {
        throw (VirtualMachineError) t;
    }
    // All other instances of Throwable
    // will be silently swallowed
}
```

Figure 11: The implementation of many handlers for `Throwable` in Tomcat.

The `catch` block in Figure 11 is not strictly speaking empty. Notwithstanding, in practice, any exception (and most errors) caught by a `catch` block with this implementation, including any instance of `Exception`, will be simply ignored, as if the `catch` block was empty. The comment at the end of the code snippet makes

Table 18: Merged classification terms.

Survey	Repository Analysis
Lack of a handler that should exist	Exception not handled
No exception thrown in a situation of a known error	Exception not thrown
Programming error in the `catch` block	Error in the handler
Programming error in the `finally` block	Error in the clean-up action
Wrong encapsulation of exception cause	Invalid or non-existent root cause

it clear that this behavior has been intended. This approach is used mostly in situations where it is difficult to know what to do with an exception, for example, in a `finally` block responsible for freeing resources. It is surprising, however, to see that the exception is not even logged.

The respondents of the survey and the bug reports also did not agree on the difficulty to fix EH-bugs. The former seem to believe that these bugs are easier to fix than other bugs. In spite of this, analysis of the repository data has shown that there is no obvious answer. EH-bugs seem to be as difficult to fix as other bugs. Whether this false sense of security has any impact on the overall system reliability is something to be discovered in future work.

According to the respondents of the survey, the median estimated percentage of EH-bugs is 5%. However, from the repository analysis we found only 1.87% of EH-bugs for Tomcat and 0.35% for Eclipse. Even considering conservative estimates for the amount of exception handling code in a system, *e.g.*, 3% of the LoC [12], the number of EH-bugs does not seem to be proportional to the amount of exception handling code. This discrepancy might be attributed to (i) exception handling code being less bug-prone; (ii) some detected EH-bugs not being reported; or (iii) some EH-bugs going undetected. As discussed earlier in this section, there is evidence that the latter is more probable.

4.1. Classification of EH-bugs

We followed the coding process [46] to create our EH-bug classification. We started the repository analysis with no classification in mind, and while checking all EH-bugs from Bugzilla, we started to code them into categories which originated the classification (which was used afterwards to build Question 15 of the survey). The source of information from Bugzilla was: the comments themselves and the source code attached or pasted in the comments. Then we got the classification from the survey, and merged it with the classification from the repository analysis. Bugs that did not contain enough information neither source code at-

tached were not considered to be related to exception handling, as we could not identify their causes. It is also important to emphasize that the classification of Barbosa *et al.* [7] did not influence ours.

Using the lists of causes for EH-bugs obtained from the survey (Section 3.4.1) and the repository analysis (Section 3.4.2), it is possible to derive a comprehensive classification of causes for EH-bugs. To compile this classification, we first analyzed the results from the two lists in order to identify different terms with the same meaning. We merged these terms as shown in Table 18. We then proceeded to include in the final classification terms appearing in both lists and terms appearing in only one of them.

Figure 19 presents the final classification. A general list of causes for EH-bugs can assist testers in devising thorough test suites and can be used as a checklist to guide code inspections. It can also serve as a basis for the construction of static analysis tools. Even though there already some static analysis tools [27, 41] that can identify some EH-bugs, they only cover a small subset of the items in Figure 19.

Table 19: Comprehensive classification of EH-bugs.

Lack of a handler that should exist
Exception not thrown
Error in the handler
Error in the clean-up action
Exception caught at the wrong level
General `catch` block
Wrong exception thrown
Exception that should not have been thrown
Wrong encapsulation of exception cause
Lack of a `finally` block that should exist
Error in the exception assertion
Inconsistency between source code and API documentation
Empty `catch` block
Error in the definition of exception class
`catch` block where only a `finally` would be appropriate

Most of the items in Figure 19 are self-explanatory, *e.g.*, lack of a handler that should exist. Nevertheless, some of them need additional clarification:

- **Exception caught at the wrong level**: An exception is caught unintentionally. The exception is being handled by handler that is not the one intended

by the developers of the system, either because it is handled too early or too late;

- **General `catch` block**: This is a subcase of **Exception caught at the wrong level**. In this case the handler catches a generic exception, such as `Exception` or `Throwable` rather than a specific one;
- **Exception that should not have been thrown**: An exception is thrown in a situation where it should not have been thrown. This kind of EH-bug usually means that either the program should do something other than throwing an exception when an error is detected, such as returning `null`; or that part of the program should not even try to detect the error, possibly because it will be detected somewhere else;
- **Wrong encapsulation of exception cause**: An exception is caught and, as a consequence, a second one is thrown. However, the second exception does not encapsulate the first one. Due to wrong encapsulation the root cause of the error is lost, hindering debugging;
- **Error in the exception assertion**: There is a bug within an assertion. This is an EH-bug because assertions are responsible for detecting errors;
- **Inconsistency between source code and API documentation**: This is the case where the documentation is not up-to-date with the software functionality;
- **Error in the definition of exception class**: This is the case where a definition of exception class is not done correctly, for example, an exception class is a subclass of `Error` when it should be a subclass of `RuntimeException`.

5. Related Work

Cristian [18] was the first to state that exception handling code is the least documented, tested, and understood part of source code of an application. Since his seminal work going back to 1989, a number of studies have reconsidered this statement and assessed its implications.

Marinescu [32] conducted an empirical study targeting three releases of Eclipse with the goal of analyzing the defect-proneness of classes that use exception handling. The study inspected both the source code and the bug repository for the

versions of Eclipse and associated the reported bugs with classes that they mention. The analysis revealed that indeed classes that throw or handle exceptions are more defect-prone than others classes that do not throw and do not handle exceptions. However, this study did not attempt to uncover the causes of these bugs, *i.e.*, they may be unrelated to exception handling. Also, it does not study developers' perceptions about the analyzed bugs. Therefore, this work is complementary to ours.

Recently, Marinescu has extended [33] the aforementioned work [32]. She has shown that classes using exceptions are more complex than those not using exceptions. Moreover, classes that handle exceptions in an improper manner show a higher probability of exhibiting defects than classes which handle them properly. Marinescu has taken a slightly different perspective on bugs and exception handling by considering complexity and defect-proneness of classes handling exceptions. Moreover, Marinescu considers "improper handling of exceptions" (*i.e.*, bug) to be general catch blocks (*e.g.*, catch Exception) and general throws clauses (*e.g.*, throws Exception). As opposed to this work [33] we did not have a preconceived notion of what constitutes an EH-bug prior to the repository analysis, which was conducted before the survey. While analyzing Bugzilla, we created the first EH-bug classification following the coding process. Then we conducted the survey with an initial EH-bug classification.

Sawadpong *et al.* [45] performed the first study on EH-bugs by looking at bug reports. Their study aimed to determine whether the usage of exception handling is relatively risky by analyzing the defect densities of exception handling code and the overall source code. The source code and bug repository of six Eclipse releases were analyzed. In the bug repository, the study looked specifically for EH-bugs, performing a search using the keywords "exception", "throw", and "threw". The main finding of the study is that the exception handling defect density of exception handling constructs is approximately three times higher than overall defect density. We believe that the definition of EH-bug ("defect" in their study) employed in this work is too coarse-grained. It assumes that every bug report returned by the repository search pertains to exception handling. The problem with this assumption is that it confuses bugs whose manifestation is an exception being thrown with bugs whose cause is associated in some way with exceptions (cf. Section 2). One would hardly, if ever, classify a typical division by zero or invalid cast as an EH-bug. As stated in the last section of their paper, "Our goal was to determine whether using exception handling is risky...". This indicates that this earlier study has goals different from ours, since we want to understand the characteristics of bugs whose causes are in some way related to the use of exception handling. Fur-

thermore, Sawadpong *et al.* [45] did not study developers' perception of EH-bugs.

A number of other studies have investigated developer habits pertaining to exception handling. To understand how exception handling is being perceived in the industry, Shah *et al.* [48] conducted a series of semi-structured interviews with 8 novices (2 years of development experience on average) and 7 experts (5+ years of professional software development). The results show that novice developers neglect exception handling until they are forced to address it by the programming language or until a system failure is noticed and needs to be fixed. Furthermore, most of the novices use exception handling only for debugging purposes and do not like it when exception handling is imposed by the programming language. As opposed to the novice developers, experienced developers consider exception handling to be a very important part of software development. We postpone a detailed comparison of these findings with our results to Section 3. It is important to stress that the work of Shah *et al.* did not focus on exception handling *bugs* and that their study involved only 15 respondents.

Cabral and Marques [12] examined 32 systems written in Java (including Eclipse and Tomcat) and C#. By manual examination of the exception handling code of those systems the authors aimed at understanding how developers use exception handling mechanisms. Cabral and Marques discovered that the total amount of exception handling code is less than expected, even in Java programs that force developers to handle checked exceptions. For example, Java Stand-Alone applications (including Eclipse) have only 3.11% of exception handling code. Server applications, such as Tomcat, reach 7% of exception handling code. Another interesting result is that most of the time the handlers are empty or exclusively dedicated to logging, re-throwing the caught exception, or exiting the method or program. In contrast, we did not focus on the source code. Instead, we examined bug report data from Eclipse and Tomcat, including associated patches, and conducted a survey.

Another study that investigated the use of exception mechanisms in Java applications was conducted by Reimer and Srinivasan [40]. The authors analyzed 7 applications and identified various antipatterns of exception handling usage. According to them, improper usage of exception handling reduces the maintainability of these systems. The antipatterns they found were: (i) exception being ignored with empty `catch` blocks; (ii) single `catch` block for multiple exceptions, *i.e.*, overly general `catch` block; (iii) exceptions not being handled at the appropriate level, and (iv) logging verbosity in `catch` blocks. In our work we found that some of these antipatterns are indeed causes of bugs. Counter-intuitively, we also found that some of these antipatterns are being used as bug patches (cf. Section 3).

Coelho *et al.* [15] created a bug pattern catalogue for exception handling in aspect-oriented programs. Based on a previous study [14] and analysis of three applications with both Java and AspectJ versions available, the bug pattern catalogue lists for each bug its symptoms, causes, a code example, cures, and prevention techniques. By analyzing source code the authors identified exceptions thrown and not caught, and exceptions caught at the wrong level as the most common causes of EH-bugs in aspect-oriented programs. These two patterns have also been reported by the developers surveyed in the current study. Since our study used a completely different methodology (analysis of bug reports and survey as opposed to analysis of the source code) and different systems, we can say that the work of Coelho *et al.* [15] complements our results.

Robillard and Murphy [41] focused on the control flow aspects of exceptions. They developed a static analysis tool that can show the paths that exceptions traverse from the methods that throw them to the ones the handle them, if any. They then employed the tool to identify bugs in three target systems. The problems identified by the tool stemmed from uncaught exceptions that should be captured and from exceptions caught accidentally, often as a consequence of `catch` blocks that are overly general, placed at the wrong level, or both.

Zhang and Elbaum [55] studied bug reports of five popular open source applications for the Android phone platform. By searching for "exception", "throw", and "catch" in the bug tracker and manually reviewing the results, the authors have identified 282 bug reports. Almost a third of the bugs that led to code fixes have been recognized as being caused by poor implementation of exception handling constructs. Furthermore, the authors also suggested an approach for amplifying existing tests to validate exception handling code associated with external resources.

More recently, Kechagia and Spinellis [28] examined crash stack traces from 1,800 Android applications. They were interested in finding Android API methods with undocumented exceptions that are part of application crashes. This work is complementary to ours as EH-bugs do not necessarily result in crashes, and while crashes studied originated from unchecked exceptions, *i.e.*, exceptions representing bugs [4], those are not necessarily EH-bugs.

Neither of the aforementioned studies analyzes issues such as whether EH-bugs are easier to fix than other kinds of bugs. Moreover, only the studies of Coelho *et al.* [14] and Robillard and Murphy [41] attempt to analyze the causes of EH-bugs. Nevertheless, since both employ static exception flow analysis tools, they are only able to identify bug causes related to exception control flow. These studies have also not accounted for the perceptions of developers about EH-bugs.

Barbosa *et al.* [7] conducted study similar to ours. They created a categorization of EH-bugs based on *Apache Tomcat*[7] and the *Hadoop*[8] framework. They first collected the full revision history of the target systems from their version control systems and then looked for the word "exception" in the comments of each revision history. Thus, they looked for the bug reports related to those revisions and finally they did a manual analysis of all those artifacts (code, comments, and bug reports). They found 28 EH-bugs for each system and created a categorization with 10 bug patterns. This work differs from ours for a number of reasons. It studies a much smaller number of bugs, does not conduct any statistical analysis, uses a slightly different definition of EH-bug, and does not attempt to analyze developers' perceptions about exception handling and EH-bugs. It does, however, present some categories that we have not encountered in our study. We compare our EH-bug classification to theirs in Table 12. For a comprehensive explanation of the comparison please see Appendix Appendix A.

In a previous paper [19], we have presented the results of the survey we conducted with developers. This work did not analyze bug reports nor patches and, hence, could not contrast developers perceptions with their actions. Furthermore, in the current submission we have performed a more rigorous statistical analysis of the data obtained and reconsidered some of the earlier observations in light of the new statistical results.

Additionally, our questionnaire might not have covered all questions that could have been asked of the respondents. Nonetheless, the final questionnaire was the result of several discussions between the authors (one of whom is a specialist in exception handling) and with a number of software developers and academics. Moreover, we ran at least two small pilot studies before finally making the questionnaire public. Moreover, respondents may have been influenced in the survey by the options offered in Questions 15 and 19. As the options from Question 15 were provided by the repository study and the options from Question 19 were based on the work of Shah et al. [48]. Finally, questions 14 and 14.1 alleviate this problem, since the latter asks for spontaneous answers.

6. Conclusion

In 1989, Flaviu Cristian [18] stated that "...since exceptions are expected to occur rarely, the exception handling code of a system is in general the least doc-

[7] http://tomcat.apache.org
[8] http://hadoop.apache.org/

umented, tested, and understood part". In the same paper, he claimed that "Most of the design faults existing in a system seem to be located in the code that handles exceptional situations". More than 20 years later, we can say that these two statements are debatable. Our study has shown that there are many contradictions pertaining to exception handling. We summarize these contradictions and the results of our study below.

Based on the responses to the survey, we can say that organizations usually do not take exception handling into account. Policies for exception handling are uncommon, as are tests and documentation for exception handling code. However, developers do employ exception handling in practice [13, 12, 45, 50, 53] and their motivations seem to go beyond issues such as language requirements: most of them them use it because they want to improve their programs (Section 3.1.1).

We could note that EH-bugs are less frequent than others kind of bugs according to the survey and much less frequent according to repository analysis. For Eclipse and Tomcat, only 0.35 and 1.84% of the bug reports pertain to EH-bugs, respectively. In contrast, they are ignored less often than other bugs, for both systems.

Many developers seem to think that *empty* `catch` *blocks* and general `catch` blocks cause EH-bugs. Nevertheless, they are often used, even in patches for EH-bugs. Also, bug reports describing bugs stemming from overly general `catch` blocks are rare, although there are many opportunities for them to occur.

We also presented a comprehensive classification of EH-bugs based on the study results. We verified from both repository analysis and the survey that the most common causes of EH-bugs are: *lack of a handler that should exist*, *no exception thrown in a situation of a known error*, *programming error in the catch block*, and *exception that should not have been thrown*.

The results of this study emphasize that the views of developers and organizations about EH-bugs are conflicting. To improve the quality of software systems these views must be reconciled so that exception handling code can receive more attention. The presented classification of EH-bugs can provide assistance in this task, *e.g.*, by working as a checklist for code inspections, a guide in the design of test cases, or a group of bugs that can be targeted by static analysis tools.

For future work we intend to expand this study by analyzing other kinds of data. For example, combining bug report data with the information stored in version control systems can help us to more precisely pinpoint the impact of a bug and its fix. We also plan to conduct interviews with developers since very useful information in our study came from spontaneous answers provided by the respondents of the survey. Through interviews we can get personal viewpoints that would be

hard to get from a survey. Furthermore, we plan to try to understand why developers from Eclipse and Tomcat state that there should be ignoring comments within *empty catch blocks* but in fact there are some case where there is not and also why there are some developers who answered the survey saying that they do not use exception handling in Java.

It is common for developers to employ empty `catch` blocks in circumstances where the implementation of the system guarantees that the exception will not be thrown. However, software maintenance can break those guarantees. Tools capable of assisting developers in identifying whether that has occurred would be useful. Furthermore, we need better support to help developers decide what to do in the presence of exceptions. If not, they will continue to use empty `catch` blocks as if they were a good solution. We believe that the development of recommendation systems capable of suggesting exception handling strategies based on the existing code base is a goal worth pursuing [8]. Finally, the empirical results presented in this work, in particular the list of causes of EH-bugs, can help future research in prediction and localization of EH-bugs.

References

[1] R, January 1993. http://www.r-project.org/.

[2] Alan Agresti. *Categorical Data Analysis*. Wiley Series in Probability and Statistics. Wiley-Interscience, 2nd edition, 2002.

[3] Thomas Anderson and Peter A. Lee. *Fault Tolerance: Principles and Practice*. Springer, 2nd edition, 1990.

[4] Ken Arnold, James Gosling, and David Holmes. *The Java Programming Language*. Addison Wesley Professional, New York, NY, USA, 4th edition, 2005.

[5] Nathaniel Ayewah, William Pugh, David Hovemeyer, J. David Morgenthaler, and John Penix. Using static analysis to find bugs. *Software, IEEE*, 25(5):22–29, Sept 2008.

[6] Adrian Bachmann, Christian Bird, Foyzur Rahman, Premkumar Devanbu, and Abraham Bernstein. The missing links: Bugs and bug-fix commits. In *Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE '10, pages 97–106, New York, NY, USA, 2010. ACM.

[7] Eiji Adachi Barbosa, Alessandro Garcia, and Simone D. J. Barbosa. Categorizing faults in exception handling: A study of open source projects. In *Proceedings of the 28th Brazilian Symposium on Software Engineering*, October 2014. To appear.

[8] Eiji Adachi Barbosa, Alessandro Garcia, and Mira Mezini. A recommendation system for exception handling code. In *Proceedings of ICSE'2012 Workshop on Exception Handling*, June 2012.

[9] Yoav Benjamini and Yosef Hochberg. Controlling the False Discovery Rate: A Practical and Powerful Approach to Multiple Testing. *Journal of the Royal Statistical Society. Series B (Methodological)*, 57(1):289–300, 1995.

[10] Andrew P. Black. *Exception Handling: The Case Against*. PhD thesis, University of Oxford, January 1982.

[11] Magiel Bruntink, Arie van Deursen, and Tom Tourwé. Discovering faults in idiom-based exception handling. In *Proceedings of the 28th International Conference on Software Engineering*, pages 242–251, May 2006.

[12] Bruno Cabral and Paulo Marques. Exception handling: a field study in java and .net. In *Proceedings of the 21st European conference on Object-Oriented Programming*, pages 151–175. Springer-Verlag, 2007.

[13] Fernando Castor, Nélio Cacho, Eduardo Figueiredo, Alessandro Garcia, Cecília M. F. Rubira, Jefferson Silva de Amorim, and Hítalo Oliveira da Silva. On the modularization and reuse of exception handling with aspects. *Softw., Pract. Exper.*, 39(17):1377–1417, 2009.

[14] Roberta Coelho, Awais Rashid, Alessandro Garcia, Fabiano Cutigi Ferrari, Nélio Cacho, Uirá Kulesza, Arndt von Staa, and Carlos José Pereira de Lucena. Assessing the impact of aspects on exception flows: An exploratory study. In *Proceedings of the 22nd European Conference Object-Oriented Programming*, pages 207–234, Paphos, Cyprus, Julho 2008.

[15] Roberta Coelho, Awais Rashid, Arndt von Staa, James Noble, Uirá Kulesza, and Carlos Lucena. A catalogue of bug patterns for exception handling in aspect-oriented programs. In *Proceedings of the 15th Conference on Pattern Languages of Programs*, PLoP '08, pages 23:1–23:13, 2008.

[16] Roberta Coelho, Arndt von Staa, Uirá Kulesza, Awais Rashid, and Carlos Lucena. Unveiling and taming liabilities of aspects in the presence of exceptions: A static analysis based approach. *Inf. Sci.*, 181(13):2700–2720, July 2011.

[17] Flaviu Cristian. A recovery mechanism for modular software. In *Proceedings of the 4th ICSE*, pages 42–51, 1979.

[18] Flaviu Cristian. Exception handling. In *Dependability of Resilient Computers*, pages 68–97. Blackwell Science, 1989.

[19] Felipe Ebert and Fernando Castor. A study on developers' perceptions about exception handling bugs. In *Proceedings of the 29th IEEE International Conference on Software Maintenance*, September 2013.

[20] Ronald A. Fisher. *Statistical methods for research workers*. Edinburgh Oliver & Boyd, 1925.

[21] Pedro Fonseca, Cheng Li, Vishal Singhal, and Rodrigo Rodrigues. A study of the internal and external effects of concurrency bugs. In *Proceedings of the 2010 IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 221–230, June/July 2010.

[22] Chen Fu and Barbara G. Ryder. Exception-chain analysis: Revealing exception handling architecture in java server applications. In *Proceedings of the 29th International Conference on Software Engineering*, pages 230–239, May 2007.

[23] Alessandro F. Garcia, Cecília M. F. Rubira, Alexander B. Romanovsky, and Jie Xu. A comparative study of exception handling mechanisms for building dependable object-oriented software. *Journal of Systems and Software*, 59(2):197–222, 2001.

[24] John B. Goodenough. Exception handling: Issues and a proposed notation. *Communications of the ACM*, 18(12):683–696, December 1975.

[25] Robert M Groves, Floyd J Fowler, Mick P Couper, James M Lepkowski, Eleanor Singer, and Roger Tourangeau. *Survey Methodology*. Wiley, 2nd edition, 2009.

[26] Jerry L. Hintze and Ray D. Nelson. Violin plots: A box plot-density trace synergism. *The American Statistician*, 52(2):181–184, May 1998.

[27] David Hovemeyer and William Pugh. Finding bugs is easy. *SIGPLAN Notices*, 39(12):92–106, December 2004.

[28] Maria Kechagia and Diomidis Spinellis. Undocumented and unchecked: Exceptions that spell trouble. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, MSR 2014, pages 312–315, New York, NY, USA, 2014. ACM.

[29] Maurice G. Kendall. A new measure of rank correlation. *Biometrika*, 30(1/2):81–93, 1938.

[30] Barbara Kitchenham and Shari L. Pfleeger. Personal opinion surveys. In Forrest Shull, Janice Singer, and Dag I. K. Sjoberg, editors, *Guide to Advanced Empirical Software Engineering*, pages 63–92, 2008.

[31] Wei Li and Raed Shatnawi. An empirical study of the bad smells and class error probability in the post-release object-oriented system evolution. *J. Syst. Softw.*, 80(7):1120–1128, July 2007.

[32] Cristina Marinescu. Are the classes that use exceptions defect prone? In *Proceedings of the 12th International Workshop on Principles of Software Evolution*, pages 56–60, September 2011.

[33] Cristina Marinescu. Should we beware the exceptions? an empirical study on the eclipse project. In *Symbolic and Numeric Algorithms for Scientific Computing (SYNASC), 2013 15th International Symposium on*, pages 250–257, Sept 2013.

[34] John McCarthy. History of lisp. *SIGPLAN Not.*, 13(8):217–223, August 1978.

[35] Tim McCune. Exception-handling antipatterns, 2006. Address: http://today.java.net/pub/a/today/2006/04/06/ exception-handling-antipatterns.html – Last access: May 10th 2013.

[36] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. MIT Press, Cambridge, MA, USA, 1990.

[37] Douglas C. Montgomery and George C. Runger. *Applied Statistics and Probability for Engineers*. Wiley, 2th edition, 2006.

[38] Andreas Muller and Geoffrey Simmons. Exception handling: Common problems and best practice with java 1.4. In *Proceedings of NetObject Days'2002*, October 2002.

[39] Ian Nelson. Empty catch blocks, June 2009. http://www.ianfnelson.com/blog/empty-catch-blocks.

[40] Darrel Reimer and Harini Srinivasan. Analysing exception usage in large Java applications. In *Proceedings of ECOOP Workshop on Exception Handling in Object-Oriented Systems*, pages 10–19, July 2003.

[41] M. Robillard and G. Murphy. Static analysis to support the evolution of exception structure in object-oriented systems. *ACM Transactions on Software Engineering and Methodology*, 12(2):191–221, April 2003.

[42] Per Runeson and Martin Hst. Guidelines for conducting and reporting case study research in software engineering. *Empirical Software Engineering*, 14(2):131–164, 2009.

[43] Swarup Kumar Sahoo, John Criswell, and Vikram Adve. An empirical study of reported bugs in server software with implications for automated bug diagnosis. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering*, pages 485–494, May 2010.

[44] Dustin Sallings. Empty catch blocks are always wrong, June 2007. http://www.rockstarprogrammer.org/post/2007/jun/15/empty-catch-blocks-are-always-wrong/.

[45] Puntitra Sawadpong, Edward B. Allen, and Byron J. Williams. Exception handling defects: An empirical study. *9th IEEE International Symposium on High-Assurance Systems Engineering*, pages 90–97, October 2012.

[46] Carolyn B. Seaman. Qualitative methods in empirical studies of software engineering. *IEEE Trans. Softw. Eng.*, 25(4):557–572, July 1999.

[47] Carolyn B. Seaman. Variable kernel density estimation. *IEEE Transactions on Software Engineering*, 25(4):557–572, 1999.

[48] Hina B. Shah, Carsten Gorg, and Mary Jean Harrold. Understanding exception handling: Viewpoints of novices and experts. *Software Engineering, IEEE Transactions on*, 36(2):150–161, 2010.

[49] Samuel Sanford Shapiro and Martin B. Wilk. An analysis of variance test for normality (complete samples). *Biometrika*, 52(3/4):591–611, Dec. 1965.

[50] Thiago B. L. Silva and Fernando Castor. New exception interfaces for java-like languages. In *Proceedings of the 28th Annual ACM Symposium on Applied Computing*, pages 1661–1666, 2013.

[51] George R. Terrell and David W. Scott. Variable kernel density estimation. *The Annals of Statistics*, 20(3):1236–1265, 09 1992.

[52] Emma Tosch and Emery D. Berger. Surveyman: Programming and automatically debugging surveys. *SIGPLAN Not.*, 49(10):197–211, October 2014.

[53] Westley Weimer and George C. Necula. Exceptional situations and program reliability. *ACM Trans. Program. Lang. Syst.*, 30(2):1–51, 2008.

[54] Frank Wilcoxon. Individual comparisons by ranking methods. *Biometrics Bulletin*, 1(6):80–83, 12 1945.

[55] Pingyu Zhang and S. Elbaum. Amplifying tests to validate exception handling code. In *In Proceedings of 34th International Conference on Software Engineering*, pages 595 –605, june 2012.

[56] Thomas Zimmermann, Rahul Premraj, and Andreas Zeller. Predicting defects for eclipse. In *Proceedings of the Third International Workshop on Predictor Models in Software Engineering*, 2007.

Appendix A. Comparison of EH-bugs Classification

In this appendix we provide a deep explanation of our EH-bug classification to the one created by Barbosa *et al.* [7]. They created a classification with ten categories of EH-bugs while analyzing *Hadoop* and *Tomcat* (versions 6.0.x and 7.0.x). They analyzed both their source code, their bug repositories, and patches corresponding to these bugs. For two categories they found some frequent patterns, so for those two they also created subcategories. Below we describe their categorization and compare it with our own.

- **Information Swallowed**: This category occurs because of lack of proper information provided with a given exception. It was divided into six subcategories:
  - **Uninformative or Wrong Error Message**: It occurs when exceptions are raised without proper information about the cause or context of their occurrence. This category is equivalent to *wrong encapsulation of exception cause* in our classification.
  - **Swallowed Exception**: It occurs when a `catch` block catches an exception and ignores its occurrence. In our classification, swallowed exceptions occur as a consequence of *exceptions caught at the wrong level*, *general* `catch` *blocks*, *empty* `catch` *blocks*, or combinations of these categories.
  - **Suppressed Exception**: It occurs when the original exception raised in the context of a method is suppressed by another exception raised in the same context. This category does not have an equivalent category in our classification.
  - **Missing Log**: It occurs when the `catch` block handled an exception, but did not register proper information in the log system. This category is an instance of the *error in the handler* category of our classification.
  - **Destructive Remapping**: It occurs when an exception is caught by a `catch` block, remapped to a different exception type and rethrown. This category is equivalent to *wrong encapsulation of exception cause* in our classification.
  - **Uninformative Generic Type Thrown**: It occurs when an exception with an overly generic type is thrown and, therefore, module clients cannot implement proper handling actions. This category is an instance of the *wrong exception thrown* category of our classification.
- **Improper Continuation of Execution**: This category occurs when the system continues its execution on an inconsistent state. It was divided into 4 subcategories:
  - **Missing Throwing Condition**: It occurs when an exception is not thrown because a specific condition (typically corner cases) was not checked in the source code. This category is equivalent to the *exception not thrown* category of our classification.

  - **Wrong Context Configuration**: It occurs when the system reaches some point at its normal execution flow where its context should be in an expected state but, for some unknown reason, it is not. This category does not have an equivalent category in our classification.
  - **Wrong Location of Execution Resumption**: It occurs when the statements after the `catch` block should not be executed if an exception occurs. This category is related to *exception caught at the wrong level* in our classification.
  - **Missing Termination Action**: It occurs when termination actions are not taken when exceptions occur. This category is equivalent to *lack of a finally block that should exist* in our classification.

- **Resource leak**: It occurs when previously allocated resources are not deallocated in the occurrence of exceptions. This category does not have an equivalence in our classification, although it is closely related to the *lack of a finally block that should exist*.

- **Uncaught Exception**: It occurs when an exception reaches the entry point of the program and is not handled by any handler. This category is equivalent to *lack of a handler that should exist* in our classification.

- **Overly-generic catch-block**: It occurs when a `catch` block has as argument an exception with an overly-generic exception type, inadvertently catches an exception by subsumption and leads the system to an unexpected state of error. This category is equivalent to *general catch block* in our classification.

- **Premature Termination**: It occurs when an exception handler captures an exception and terminates the execution of the method without retrying the execution of the failed action. This category is an instance of the *error in the handler* category of our classification.

- **Throwing wrong type**: It occurs when a method throws an exception that adheres to its exceptional interface, but do not adhere to other specification. This category is equivalent to *wrong exception thrown* in our classification.

- **Overly protective try-block**: It occurs when a `try` block is very long and protects the occurrence of many different exceptions, *i.e.*, when the scope of `try` block is too its associated `catch` blocks may capture exceptions that

should be caught by other handlers. This category is an instance of the *exception caught at the wrong level* category of our classification.

- **Exceptional loop-break**: It occurs when the block of a loop statement is not protected and an exception inadvertently breaks this loop, *e.g.*, when the `try` block should be inside the loop and it is outside. This category does not have an equivalent category in our classification, but it is related to some categories, *e.g.*, *lack of a handler that should exist* and *exception caught at the wrong level*.

- **Excessive Throwing Condition**: It occurs when an exception is thrown by a condition that is not actually an exceptional condition. This category is equivalent to *exception that should not have been thrown* in our classification.

As we can see, there are similarities and differences between the classifications. As a general trend, we can observe that the categories proposed by Barbosa *et al.* are more specific than the ones we propose. Only five categories were not found in ours, though three of them, *exceptional loop-break*, *wrong location of execution resumption* and *resource leak* are similar to categories that we identified. On the other hand, we identified four categories not pointed out in the work of Barbosa *et al.*: *error in the clean-up action* (which is related to *resource leak* and *suppressed exception* from the classification of Barbosa *et al.*), *inconsistency between source code and API documentation*, *error in the definition of exception class* and *catch block where only a finally would be appropriate*. Overall, we believe that developers should combine the two classifications in order to achieve the best results.

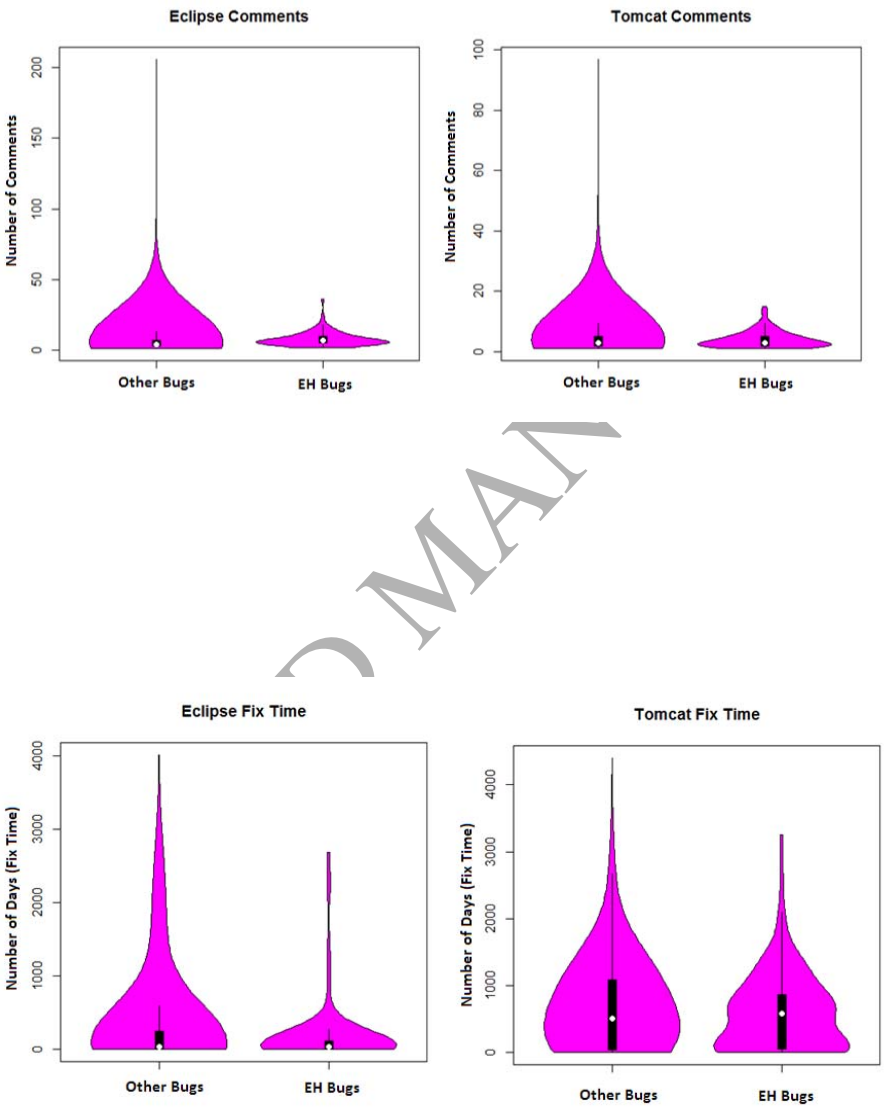

Figure 4: Vioplots of bug fixing time (top) and the number of discussion messages (bottom)

```
    classFile.completeCodeAttribute(codeAttributeOffset);
}
```

```
    try {
        classFile.completeCodeAttribute(codeAttributeOffset);
    } catch(NegativeArraySizeException e) {
        throw new AbortMethod(this.scope.
            referenceCompilationUnit().compilationResult, null);
    }
```

Figure 5: An example of "exception not handled" from Eclipse—bug ID 298250.

```
} catch (ClassFormatException e) {
        // ignore
        this.document.removeAllIndexEntries();
        Util.log(e, "ClassFormatException in " + this.document.
            getPath() + ". Please report this issue to JDT/Core
            including the problematic document"); //$NON-NLS-1$ //
            $NON-NLS-2$
} catch (RuntimeException e) {
        ...
        this.document.removeAllIndexEntries();
        Util.log(e, "Indexer crashed on document " + this.document.
            getPath() + ". Please report this issue to JDT/Core
            including the problematic document"); //$NON-NLS-1$ //
            $NON-NLS-2$
}
```

```
} catch (ClassFormatException e) {
        // ignore
        this.document.removeAllIndexEntries();

        if (JavaCore.getPlugin().isDebugging()) {
                Util.log(e, "ClassFormatException in " + this.
                    document.getPath() + ". Please report this issue
                    to JDT/Core including the problematic document");
                    //$NON-NLS-1$ //$NON-NLS-2$
        }
} catch (RuntimeException e) {
        ...
        this.document.removeAllIndexEntries();

        if (JavaCore.getPlugin().isDebugging()) {
                Util.log(e, "Indexer crashed on document " + this.
                    document.getPath() + ". Please report this issue
                    to JDT/Core including the problematic document");
                     //$NON-NLS-1$ //$NON-NLS-2$
        }
}
```

Figure 6: An example of "error in the handler" from Eclipse—bug ID 195823.

```
public Principal authenticate(String username,
    String credentials) {
...
// If not a "Socket closed." error then rethrow.
if (e.getMessage().indexOf("Socket closed") < 0)
    throw(e);
```

```
public Principal authenticate(String username,
    String credentials) {
// if code removed
```

Figure 7: An example of "exception that should not be thrown" from Tomcat—bug ID 18698.

```
public void include(String relativeUrlPath)
    throws ServletException, IOException {
```

```
public void include(String relativeUrlPath)
    throws ServletException, IOException {
...
    if (resourceStream == null) {
        throw new IllegalArgumentException
            (``Included resource not found: ''
                + relativeUrlPath);
    }
```

Figure 8: An example of "exception not thrown" from Tomcat—bug ID 8200.

```
javadocContents = extractJavadoc(declaringType,
    javadocContents);
```

```
try {
 javadocContents = extractJavadoc(declaringType,
     javadocContents);
} catch(JavaModelException e) {
 // ignore
}
```

Figure 9: A patch containing an empty `catch` block from Eclipse, bug ID 139160.

```
session.expire();
```

```
try {
        session.expire();
} catch (Throwable t) {
        ;
}
```

Figure 10: A patch containing an empty `catch` block from Tomcat, bug ID 24368.

Figure 12: EH-bugs classification comparison.

Our Classification	**Barbosa's *et al.* Classification**
Lack of a handler that should exist	Uncaught Exception
Exception not thrown	Missing Throwing Condition
Error in the handler	Missing Log Premature Termination
Error in the clean-up action	—
Exception caught at the wrong level	Wrong Location of Execution Resumption Overly protective try-block
General `catch` block	Overly-generic catch-block
Wrong exception thrown	Uninformative Generic Type Thrown Throwing wrong type
Exception that should not have been thrown	Excessive Throwing Condition
Wrong encapsulation of exception cause	Uninformative or Wrong Error Message Destructive Remapping
Lack of a `finally` block that should exist	Missing Termination Action
Error in the exception assertion	—
Inconsistency between source code and API documentation	—
Empty `catch` block	Wrong Context Configuration
Error in the definition of exception class	—
`catch` block where only a `finally` would be appropriate	—

Biography

Felipe Ebert is a Ph. D. student at the Informatics Center (CIn), Federal University of Pernambuco (UFPE), where I also completed my Masters degree there. My advisor is prof. Fernando Castor.

Fernando Castor is a tenured assistant professor at the Informatics Center, Federal University of Pernambuco. Prior to this, I held positions as assistant professor at the Department of Computing and Systems of the University of Pernambuco, and postdoctoral researcher at the Department of Computer Science of the University of Sao Paulo.

Alexander Serebrenik is a part-time visiting researcher at the Software Analysis and Transformation group of CWI, lead by Jurgen J. Vinju. From September till December 2012 he was on sabbatical at the Software Engineering lab of University of Mons, headed by Prof. Tom Mens.