



Pós-Graduação em Ciência da Computação

# “An Exploratory Study on Exception Handling Bugs in Java Programs”

By

**Felipe Ebert**

M.Sc. Dissertation



Federal University of Pernambuco  
posgraduacao@cin.ufpe.br  
[www.cin.ufpe.br/~posgraduacao](http://www.cin.ufpe.br/~posgraduacao)

RECIFE, AUGUST/2013



Federal University of Pernambuco  
Informatics Center  
Graduate in Computer Science

Felipe Ebert

## **“An Exploratory Study on Exception Handling Bugs in Java Programs”**

*A M.Sc. Dissertation presented to the Informatics Center  
of Federal University of Pernambuco in partial fulfillment  
of the requirements for the degree of Master of Science in  
Computer Science.*

*Advisor: Fernando Castor*

RECIFE, AUGUST/2013

*Dedico essa dissertação à minha família.*

# Acknowledgements

Agradeço primeiramente a Deus. Aos meus pais que sempre me apoiaram, educaram e me ensinaram os valores da vida. Ao meu irmão por sempre me dar conselhos nos momentos difíceis.

À Camila que sempre me apoiou durante todo este período e me incentivou para que eu terminasse o Mestrado.

Ao meu orientador, o professor Fernando Castor, pela sua paciência, ajuda e excelente orientação durante todo o Mestrado.

Agradeço também aos membros do SPG que forneceram enorme feedback no meu projeto. Ele foi de grande importância para o amadurecimento desta pesquisa.

Agradeço também aos meus amigos do "Pior o Meu Padasto", sempre compartilhando momentos únicos.

Finalmente, agradeço ao CIn, a UFPE, a FACEPE e ao CNPq por financiarem esta pesquisa.

*"Milagres acontecem quando a gente vai à luta!"*

—O TEATRO MÁGICO - FERNANDO ANITELLI

# Resumo

Vários estudos afirmam que o código de tratamento de exceções em geral tem baixa qualidade e que é geralmente negligenciado por desenvolvedores. Além disso, acredita-se que essa parte da implementação de um sistema é a menos compreendida, documentada e testada. Apesar desse cenário, existem poucos estudos que analisam bugs de tratamento de exceções que ocorrem em sistemas de software reais e nenhum estudo que tente entender a percepção dos desenvolvedores sobre esses bugs.

Neste trabalho, apresentamos um estudo exploratório sobre bugs de tratamento de exceções baseado em duas abordagens complementares: uma pesquisa com 154 desenvolvedores e uma análise de 220 bugs dos repositórios do Eclipse e Tomcat. Os desenvolvedores de nossa pesquisa acreditam que bugs de tratamento de exceções são mais facilmente corrigidos do que outros tipos de bugs. Há também uma diferença significativa na opinião dos desenvolvedores sobre a qualidade do código de tratamento de exceções: os desenvolvedores mais experientes tendem a acreditar que é pior.

A análise dos repositórios do Eclipse e Tomcat revelou resultados conflitantes. O tempo de correção dos bugs de tratamento de exceções do Eclipse é significativamente menor do que o de outros tipos de bugs. Entretanto, os bugs de tratamento de exceções têm um número significativamente maior de comentários do que os bugs que não são de tratamento de exceções. Por outro lado, para o Tomcat, não conseguimos achar uma diferença significativa para o tempo de correção dos bugs e os bugs de tratamento de exceções tem um número significativamente menor de comentários do que os outros tipos de bugs.

Além disso, descobrimos que os bugs decorrentes de blocos `catch` genéricos, um defeito bem conhecido em programas que usam exceções, são raros, embora existam várias oportunidades para que eles ocorram. Descobrimos também que blocos `catch` vazios não são só prevalentes, como previamente relatado na literatura, mas também geralmente usados como correções dos bugs, inclusive para bugs de tratamento de exceções. Também achamos poucos bugs reportados em que as causas deles são blocos `catch` vazios, embora desenvolvedores frequentemente mencionem eles como causas de bugs que já corrigiram no passado. E por fim, apresentamos uma proposta de classificação dos bugs de tratamento de exceções.

**Palavras-chave:** Tratamento de Exceções; Bugs; Questionário; Mineração de Repositórios.

# Abstract

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. In spite of this scenario, there are very few studies that analyze the actual exception handling bugs that occur in real software systems and no study that attempts to understand developers' perceptions about 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 bugs from the repositories of Eclipse and Tomcat. Respondents of our survey believe that exception handling bugs are more easily fixed than other kinds of bugs. There is also a significant difference in the opinion of the respondents pertaining to the quality of the exception handling code: more experienced developers tend to believe that it is worse.

Analysis of the repositories of Eclipse and Tomcat revealed conflicting results. The fix time for exception handling bugs in Eclipse is significantly shorter than for other bugs. However, exception handling bugs have a significantly greater number of discussion messages than non-exception handling bugs. On the other hand, for Tomcat, we could not find a significant difference for fix time and exception handling bugs have significantly less discussion messages than other bugs.

Moreover, we discovered that bug reports describing bugs stemming from overly general `catch` blocks, a well-known bad smell in programs that use exceptions, are rare, even though there are many opportunities for them to occur. In addition, empty `catch` blocks are not only prevalent, as previously reported in literature, but they are also commonly used as part of bug fixes, which includes fixes for exception handling bugs. Furthermore, we found very few bug reports whose causes are empty `catch` blocks, although developers often mention them as causes of bugs they have fixed in the past. And lastly, we present a proposal of the classification of exception handling bugs based on the data we collected.

**Keywords:** Exception Handling; Bugs; Survey; Repository Mining.

# Contents

<b>List of Figures</b>	<b>x</b>
<b>List of Tables</b>	<b>xi</b>
<b>1 Introduction</b>	<b>1</b>
1.1 Problem . . . . .	2
1.2 Objective . . . . .	2
1.3 Contribution . . . . .	3
1.4 Outline . . . . .	4
<b>2 Background</b>	<b>5</b>
2.1 Exception Handling . . . . .	5
2.2 Exception Handling in Java . . . . .	7
2.3 Exception Handling Antipatterns in Java . . . . .	9
2.3.1 Swallowed Exception . . . . .	9
2.3.2 Exception Not Handled at Appropriate Level . . . . .	10
2.3.3 Log and <code>throw</code> . . . . .	10
2.3.4 Throwing <code>Exception</code> . . . . .	10
2.3.5 Throwing The Kitchen Sink . . . . .	11
2.3.6 Catching <code>Exception</code> . . . . .	11
2.3.7 Destructive Wrapping . . . . .	12
2.3.8 Log and <code>return null</code> . . . . .	12
2.3.9 <code>catch</code> and Ignore . . . . .	13
2.3.10 <code>throw</code> from Within <code>finally</code> . . . . .	13
2.3.11 Multi-Line Log Messages . . . . .	13
2.3.12 Relying on <code>getCause()</code> . . . . .	14
<b>3 Methodology</b>	<b>15</b>
3.1 What is an Exception Handling Bug? . . . . .	16
3.2 Repository Analysis . . . . .	19
3.3 Survey . . . . .	22
<b>4 Study Results</b>	<b>24</b>
4.1 RQ1: Do organizations and developers take exception handling into account? . . . . .	25



---

4.1.1	Survey . . . . .	26
4.1.2	Repository Analysis . . . . .	28
4.2	RQ2: How commonplace are exception handling bugs? . . . . .	30
4.2.1	Survey . . . . .	30
4.2.2	Repository Analysis . . . . .	32
4.3	RQ3: Are exception handling bugs harder to fix than other bugs? . . . . .	32
4.3.1	Survey . . . . .	33
4.3.2	Repository Analysis . . . . .	34
4.4	RQ4: What are the main causes of exception handling bugs? . . . . .	37
4.4.1	Survey . . . . .	37
4.4.2	Repository Analysis . . . . .	38
4.5	Discussion . . . . .	42
4.5.1	Exception Handling Bug Classification . . . . .	45
4.6	Threats to Validity . . . . .	47
<b>5</b>	<b>Conclusion</b>	<b>50</b>
5.1	Related Work . . . . .	51
5.2	Future Work . . . . .	54
	<b>Bibliography</b>	<b>55</b>
	<b>Appendix</b>	<b>59</b>
<b>A</b>	<b>Survey</b>	<b>60</b>
<b>B</b>	<b>Exception Handling Bugs</b>	<b>67</b>
B.1	Exception Handling Bugs of Eclipse . . . . .	67
B.2	Exception Handling Bugs of Tomcat . . . . .	70
<b>C</b>	<b>R Scripts</b>	<b>75</b>
C.1	Eclipse R Script . . . . .	75
C.2	Tomcat R Script . . . . .	80

---

# List of Figures

2.1	Exception handling mechanism [GRRX01]. . . . .	7
2.2	Exception handling in Java [htt13b] . . . . .	8
2.3	A basic example of exception handling in Java. . . . .	9
2.4	Swallowed exception antipattern. . . . .	10
2.5	Log and <code>throw</code> antipattern. . . . .	10
2.6	Throwing <code>Exception</code> antipattern. . . . .	11
2.7	Throwing the kitchen sink antipattern. . . . .	11
2.8	Catching <code>Exception</code> antipattern. . . . .	12
2.9	Destructive wrapping antipattern. . . . .	12
2.10	Log and <code>return null</code> antipattern. . . . .	12
2.11	<code>catch</code> and <code>ignore</code> antipattern. . . . .	13
2.12	<code>throw</code> from within <code>finally</code> antipattern. . . . .	13
2.13	Multi-line log messages antipattern. . . . .	14
2.14	Relying on <code>getCause()</code> antipattern. . . . .	14
3.1	Real exception handling bug from Eclipse - bug ID 21018. . . . .	18
3.2	Real non-exception handling bug from Eclipse - bug ID 81417. . . . .	18
3.3	Life cycle of a bug in Bugzilla [htt13a]. . . . .	20
4.1	Graphic of the density of fix time. . . . .	36
4.2	Graphic of the density of comments. . . . .	36
4.3	An example of "exception not handled" from Eclipse - bug ID 298250. . . . .	39
4.4	An example of "error in the handler" from Eclipse - bug ID 170237. . . . .	40
4.5	An example of "exception that should not be thrown" from Tomcat - bug ID 18698. . . . .	40
4.6	An example of "exception not thrown" from Tomcat - bug ID 8200. . . . .	41
4.7	A empty catch block patch for Eclipse, bug ID 139160. . . . .	42
4.8	A empty catch block patch for Tomcat, bug ID 24368. . . . .	42
4.9	The implementation of many handlers for <code>Throwable</code> in Tomcat. . . . .	44

# List of Tables

3.1	Summary of Eclipse bugs. . . . .	21
3.2	Summary of Tomcat bugs. . . . .	21
3.3	Summary of survey questions. . . . .	23
4.1	For how long have you been a Java developer? . . . . .	24
4.2	What is the approximate size of the project you are currently working on? . . . . .	25
4.3	Which programming languages have you professionally worked with? . . . . .	25
4.4	Why do developers use exception handling? . . . . .	27
4.5	Priorities, severities, resolutions, status, and the presence of attachments for exception handling and non-exception handling bug reports for Eclipse (a) and Tomcat (b). . . . .	29
4.6	How often do you find bugs related to exception handling? . . . . .	31
4.7	How often do you find bugs that are not related to exception handling? . . . . .	31
4.8	How often are bugs reported at your organization? . . . . .	32
4.9	How often are bugs related to exception handling reported at your orga- nization? . . . . .	32
4.10	What is the average level of difficulty to fix bugs related to exception handling? . . . . .	33
4.11	What is the average level of difficulty to fix other bugs that are not related to exception handling? . . . . .	33
4.12	What is the average priority / severity of reported bugs related to excep- tion handling code? . . . . .	34
4.13	Fix time in days and number of discussion messages for exception han- dling bugs and for other bugs. . . . .	35
4.14	What are the main causes of exception handling bugs? . . . . .	38
4.15	Exception handling bug classification according to repository analysis. . . . .	39
4.16	Merged classification terms. . . . .	45
4.17	Comprehensive classification of exception handling bugs. . . . .	46
A.1	Questionnaire about exception handling bugs . . . . .	60
B.1	Exception handling bugs of Eclipse . . . . .	67
B.2	Exception handling bugs of Tomcat . . . . .	70

# 1

## Introduction

Owing to the complexity and pervasiveness of modern software systems, many of these systems must include provisions to handle errors at runtime, both because of undetected bugs and because of environmentally-triggered erroneous conditions (e.g., failure to open a file, connect to a database or to communicate via network). It is inevitable that these developed systems still contain faults, or bugs, arising from a variety of causes [[XRR<sup>+</sup>](#), [Som10](#)]. Moreover, sometimes software systems are faced with situations where they cannot proceed with the execution. Therefore, it is necessary to include ways to deal with the manifestation of these bugs at runtime in such systems, or, in a broader sense, situations where the system is not able to proceed according to the specification. As such, mechanisms and techniques such as exception handling [[Goo75](#)] are needed to deal with errors in the systems.

Software systems basically work by doing two basic operations: receiving user requests and producing answers for the users, or sometimes just processing their requests. Either way, if a system cannot process a request, it should return an exception [[GRRX01](#)]. In Software Engineering, exceptions are thrown when an error is identified and they point to some problem in the system or in the environment with which it interacts.

There are two types of response for those systems: normal, which occurs when a software component (in a broader sense: a method, a function, a module, or a whole system) proceeds in processing the request correctly, and exceptional responses, which occur when the system is not able to process the request and produce a normal response. In this case, the system should raise an exception that should be handled. In other words, the exception handler (which is the entity that handles exceptions) responsible for the execution should be triggered and should execute the actions to solve the problem. Such handlers implement the exceptional behavior. Systems capable of handling errors during

execution time to bring the system back to a consistent state are called fault-tolerant.

Exception handling helps to improve fault tolerance in software systems by separating the main logical execution flow from the flow which handles the exceptional responses of the system. Exceptional conditions can be triggered by bugs or situations which need specific treatment because they intrinsically can fail, for example, I/O operations or database operations.

## 1.1 Problem

Several modern object-oriented programming languages, like Java, Ruby, C#, and C++, implement exception handling. Moreover, it is often the case that a considerable part of the source code of a system is dedicated to error detection and handling [CM07, WN08]. Nevertheless, developers tend to focus on the normal behavior of the applications and deal with error handling only during the system implementation, in an ad hoc manner.

This practice creates a proper situation for the appearance of design faults or bugs. Several studies [Cri89, RS03, SGH10] argue that exception handling code is usually of poor quality and that it is commonly neglected by developers. There is another problem with the exception handling code: it is usually hard to test it because of the huge number of exceptional conditions there are in the code and also it is difficult to stimulate all possible exception causes during tests for complex systems [CvSK<sup>+</sup>11]. In spite of this scenario, there are very few studies that analyze the actual exception handling bugs that occur in real software systems and there is no study that attempts to understand developers' perceptions about these bugs. A more in-depth comprehension about exception handling bugs can help developers to avoid them. Moreover, it can point out to researchers and tool designers some of the problems that they could be addressing.

## 1.2 Objective

Our goal is to analyze bugs reported in software projects and to find out if it is common that those bugs are caused by the use of exception handling. Additionally, we are interested in developers' perception about exception handling and exception handling bugs because these perceptions do not necessarily mirror the state-of-the-practice concerning exception handling bugs.

The main questions that we are interested in answering in this work are the following:

- Do organizations and developers take exception handling into account?

- How commonplace are exception handling bugs?
- Are exception handling bugs harder to fix than other bugs?
- What are the main causes of exception handling bugs?

From the developers' perception, we are interested in obtaining answers such as: what their opinion about the quality of code that handles exceptions is; how frequent they find exception handling bugs; how complex they are; what the most common causes are; and whether they are used to documenting those kinds of bugs.

We aim to gather all the above mentioned information from both bug reports and from developers in order to understand exception handling bugs: what developers do about exception handling bugs and what developers say about them. We also compare our results with some existing studies in this area [[RS03](#), [ZE12](#), [CM07](#), [SGH10](#)] and analyze how exception handling bugs manifest in the real world. We employ two complementary approaches to achieve this: a survey of developers and an analysis of the bug repositories of Tomcat and Eclipse.

## 1.3 Contribution

The main findings of this work are:

- Most of respondents of our survey believe that exception handling bugs are more easily fixed than other kinds of bugs;
- There is a significant difference in the opinion of the respondents pertaining to the quality of the exception handling code: more experienced developers tend to believe that it is worse;
- Less experienced developers think that the priority / severity of exception handling bugs is lower than other kinds of bugs, even though they estimate that the percentage of exception handling bugs is bigger than non-exception handling bugs;
- Analysis of the repositories of Eclipse and Tomcat revealed conflicting results. The fix time for exception handling bugs in Eclipse is significantly shorter than for other bugs. However, exception handling bugs have a significantly greater number of discussion messages than non-exception handling bugs. On the other hand, for Tomcat, we could not find a significant difference for fix time and exception handling bugs have significantly less discussion messages than other bugs;

- We discovered that bug reports describing bugs stemming from overly general `catch` blocks, a well-known bad smell in programs that use exceptions, are rare, even though there are many opportunities for them to occur and developers say that they have encountered bugs with these characteristics in the past;
- Empty `catch` blocks, another well-known bad smell, are not only prevalent, as previously reported in literature, but also commonly used as part of bug fixes, including fixes for exception handling bugs. Moreover, developers often state in the code, by means of comments, that these `catch` blocks do not capture exceptions in practice;
- We found very few bug reports whose causes are empty `catch` blocks, although developers often mention them as causes of bugs they have fixed in the past;
- We present a proposal of the classification of exception handling bugs based on the data we collected;
- We can also consider the data set obtained from this research as a contribution for future works.

## 1.4 Outline

The remainder of this dissertation is organized as follows:

- Chapter 2 presents the main concepts required to understand this work. We introduce exception handling definition and concepts;
- Chapter 3 presents the methodology used in this work;
- Chapter 4 presents the results from the survey and the analysis of the bug repositories. We also propose an exception handling bug classification;
- Chapter 5 summarizes the contributions and conclusions of this work. It also discusses related work and future work.

And finally, the appendices are presented as follows:

- Appendix A presents the complete questionnaire used in this work;
- Appendix B presents the complete list of exception handling bugs found in the bug repositories analysis.

# 2

## Background

*"Everybody hates thinking about exceptions,  
because they're not supposed to happen"*

—BRIAN FOOTE

In this chapter we discuss some of the most important topics regarding exception handling and exception handling bugs. Firstly, we introduce exception handling in Section 2.1. In Section 2.2 we discuss Java exception handling. Lastly, in Section 2.3 we present some of the most common exception handling antipatterns.

### 2.1 Exception Handling

Programming languages should provide primitives to help developers to deal with abnormal situations in systems and to recover from them. An exception handling [Goo75] mechanism aims to improve modularity and robustness of programs in two ways: explicit separation of the code that handles the exceptional behavior from the normal code and declaration of exceptional interfaces [CvSK<sup>+</sup>11]. Moreover, it is exception handling's responsibility to change the normal control flow of the program to the exceptional control flow. Exception handling usually provides constructs to signal the occurrence of an error (to *throw* an exception) and to create a way to recover from the error (to *catch* and *handle* the exception).

Before explaining exception handling, we should introduce some general concepts [GRRX01]:

- **Failure:** Is a deviation of the program's specification;
- **Error:** Is an intermediate state which comes from the fault and can result in a failure later on. An error corresponds to an inconsistency in the state of the program;



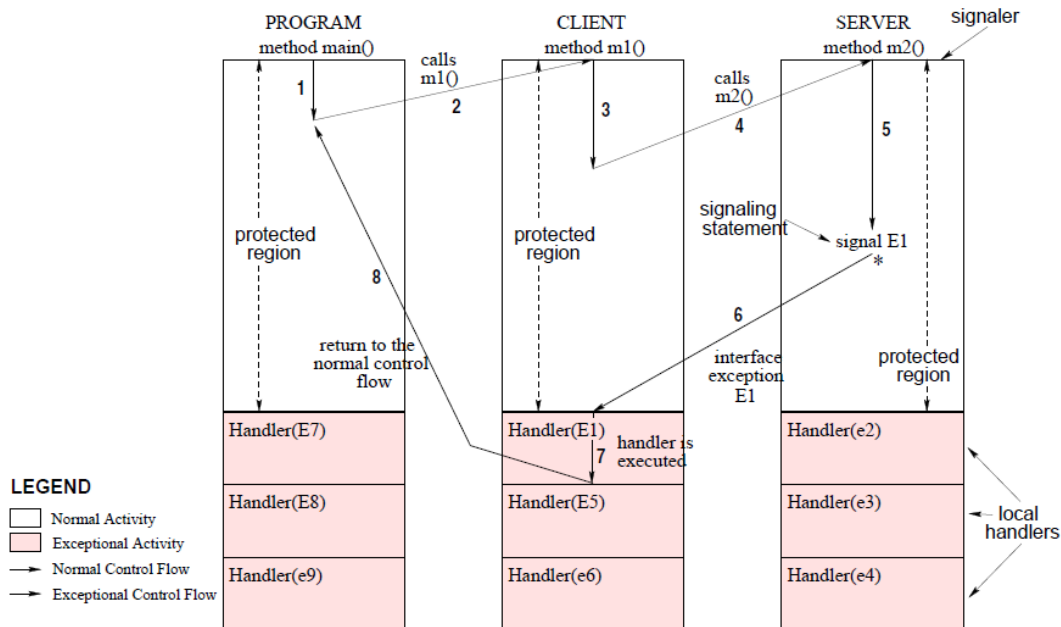
- **Fault:** Is the cause of the error. Usually classified as: (i) physical faults: those originating from software components, and (ii) human faults: those originating from software projects. The term bug is a synonym for human fault;
- **Exception:** Indicates a problem during the execution of a program. Exceptions are thrown (or signaled) when errors are identified and, may or may not make the program present a failure.

When discussing exception handling, it is also important to discuss the concepts of protected region, exception handler and clean-up action [GRRX01]:

- **Protected region:** Is a delimited part of the code liable to raise exceptions. Each protected region can have a set of associated handlers;
- **Handler:** Is the part of the code that handles the exception and is attached to the protected region;
- **Clean-up action:** A protected region may or may not have a clean-up action. It is the part of the code that performs actions to keep the program state consistent irrespective of an exception being thrown. A clean-up action is always executed after the execution of the protected region, regardless of whether an exception was thrown or not.

Exceptions can be divided into two basic types [GRRX01]: (i) internal exceptions, which are those handled by handlers associated with the protected region where the exception was thrown and (ii) external exceptions, which are those captured by handlers associated with other protected regions. This is because the component which raised the exception does not contain a handler for that exception in its protected region.

Figure 2.1 presents a high level view of the way in which exception handling works, extracted from the work of Garcia et al [GRRX01]. The program is executing the normal control flow when an exception  $E1$  is detected in method  $m2$  (arrows 1 to 5 in Figure 2.1). This method then signals the exception. The code block that raised the exception is the protected region of the exception. Now the exception mechanism needs to find an exception handler to deal with the exception (arrow 6). At this point, the normal control flow is stopped and the exceptional control flow starts. As we can see in Figure 2.1, the exceptions  $e2$ ,  $e3$  and  $e4$  are internal for  $m2$ , whereas  $E1$  and  $E5$  are external. Since  $m2$  has no handler for exception  $E1$ , it propagates that exception to its caller. The runtime of the language is then responsible for seeking a handler in the caller of  $m2$ , method  $m1$ .



**Figure 2.1** Exception handling mechanism [GRRX01].

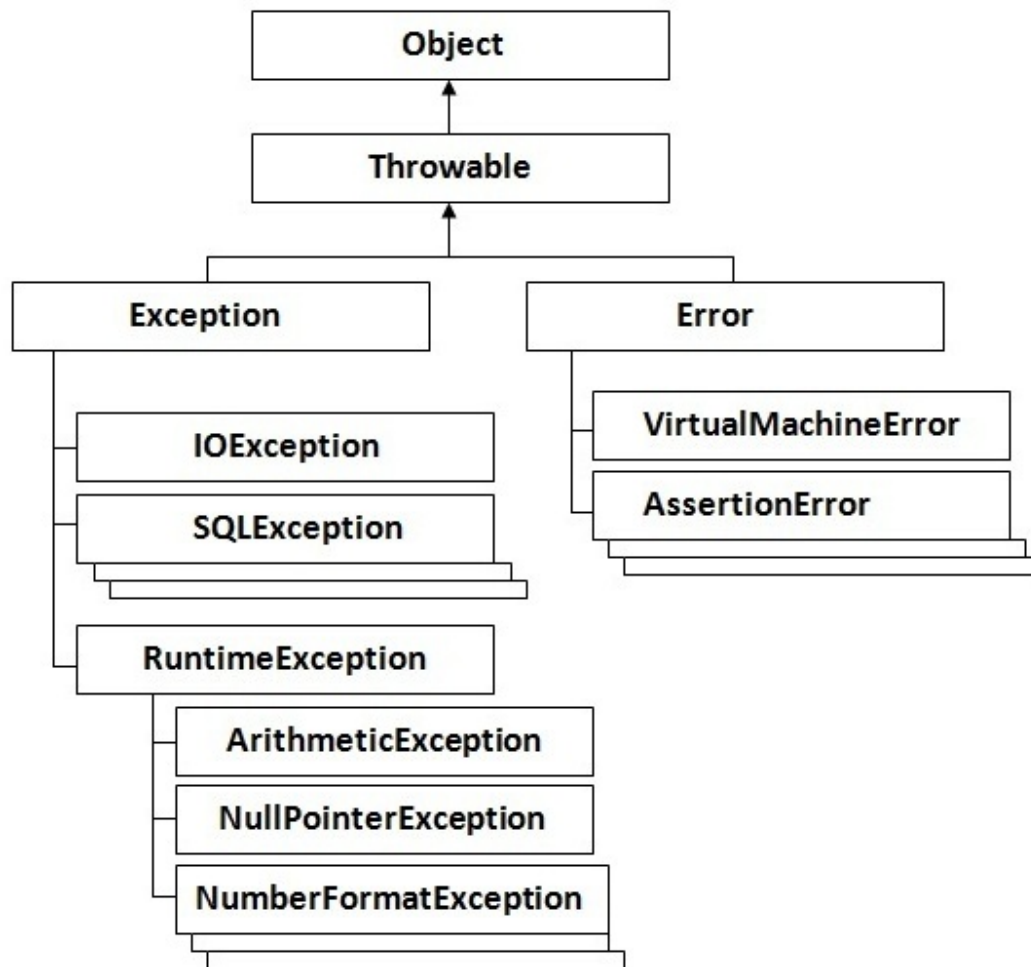
The handler for exception `E1` is in method `m1` (arrow 7). After handling the exception, the exception mechanism returns the system to the normal control flow (arrow 8).

## 2.2 Exception Handling in Java

Exception handling in Java does not make any distinction between internal and external exceptions. They are treated in the same way. Protected regions are defined by `try` blocks. The handlers are defined by `catch` blocks and the clean-up actions are defined by `finally` blocks. The `throw` statement is used to signal an exception. Exception handling in Java is called safe because its compiler can do static checking of the code in an elaborated way. This is because the developer is required to declare all checked exceptions, as explained below.

As Java is an object-oriented language, exceptions are objects of classes that are subclasses of `Exception`. Hence, developers can create new exception types as subclasses of other exceptions. Exceptions are structured in an hierarchical way as we can see in Figure 2.2.

There are basically two types of exception in Java: checked and unchecked exceptions. Checked exceptions inherit from the `Exception` class or directly from `Throwable` class, and unchecked exception inherit from the `Error` class or `RuntimeException`



**Figure 2.2** Exception handling in Java [\[htt13b\]](#)

class. As an `Error` is considered an irrecoverable condition, Barbosa:2012:RSEOracle actually considers three types of exceptions:

- **Checked exceptions** are checked at compile-time and need to be explicitly handled by the program. Some examples of checked exceptions are: `IOException` and `SQLException` as we can see in Figure 2.2;
- **Unchecked exceptions** are not checked at compile-time. Rather they are checked only at runtime. They do not need to be explicitly handled in the program and also do not need to be specified in the method signature. Some examples of unchecked exceptions are `NullPointerException`, `ArithmeticException`, and `NumberFormatException` as we can see in Figure 2.2;

- **Error** is an unrecoverable condition, e.g., a virtual machine problem. Some examples of errors are: `VirtualMachineError`, `AssertionError`, and `OutOfMemoryError` as we can see in Figure 2.2.

Figure 2.3 shows a simple example of exception handling in Java. The method `m1`, or any method it calls, can throw the exception of type `SomeException`, or a subclass of `SomeException`. As we discussed before, `SomeException` is a checked exception because the compiler checks whether it is explicitly handled or explicitly appears in the `throws` clauses of methods that throw it. Therefore, when `SomeException` is thrown the exception handling mechanism transfers execution to the exceptional control flow. The exception handling mechanism starts the search for a handler for `SomeException`. As it is an internal exception, the handler is in the same class. Then the handler starts its execution. In this case, the handler only logs the occurrence of the exception. There is also a clean-up action for this protected region. The `finally` block will always be executed, whether the `SomeException` is thrown or not.

```
// some method
try() {
    m1();
} catch (SomeException e) {
    Logger.log(e.getMessage());
} finally {
    clear();
}
```

**Figure 2.3** A basic example of exception handling in Java.

## 2.3 Exception Handling Antipatterns in Java

In this section we present some well-known exception handling antipatterns in Java. This list is based on [RS03] and [McC06].

### 2.3.1 Swallowed Exception

The exception handler should never ignore exceptions. If it does not log, handle, or re-throw the exception we say that the exception is swallowed. In this case, if an exception occurs, there will be no record of it in the system and it will be much harder for the

support team to find the problem. Figure 2.4 shows a simple example where exception `SomeException` is swallowed:

```
try() {  
    m1();  
} catch (SomeException e) {  
}
```

**Figure 2.4** Swallowed exception antipattern.

### 2.3.2 Exception Not Handled at Appropriate Level

Exceptions should be handled as near as possible to the source of the problem. If the exception is handled a long way up the call chain, it will be more difficult to debug the code and the error message will be less meaningful.

### 2.3.3 Log and throw

The exception handler should either log or throw the exception, not do both. It is wrong to do both because it makes life hard for the support team to fix the bug when there are log messages spread all over the code. Since the logged exception will be rethrown the exception will be logged again whenever it is caught. McCune [McC06] considers this antipattern one of the most annoying. Figure 2.5 shows an example of the **Log and throw** antipattern:

```
try() {  
    m1();  
} catch (SomeException e) {  
    Logger.log(e.getMessage());  
    throw e;  
}
```

**Figure 2.5** Log and throw antipattern.

### 2.3.4 Throwing Exception

Methods should never declare `throws Exception` because it is the most generic exception type in Java. It defeats the goal of using checked exceptions. Instead, the

developer should use the most specific type of exception being thrown. Figure 2.6 shows an example of the **Throwing** `Exception` antipattern:

```
public void m1() throws Exception {
    int x = 2;
    m2(x);
}
public void m2(int i) {
    if (i < 5) {
        throw new SomeException();
    }
}
```

**Figure 2.6** Throwing `Exception` antipattern.

### 2.3.5 Throwing The Kitchen Sink

Methods should never throw multiple checked exceptions when they basically mean the same thing to the caller. Instead, they should be wrapped in a single checked exception. Figure 2.7 shows an example of this antipattern:

```
public void m(String s) throws SomeException,
    OtherException, AnotherException,
    SomeOtherException, YetAnotherException {

    String client = s;
    // do something...

    // SomeExcetion: string client is null
    // OtherException: string client is invalid
    // AnotherException: string client is empty
}
```

**Figure 2.7** Throwing the kitchen sink antipattern.

### 2.3.6 Catching `Exception`

McCune [McC06] says "This is generally wrong and sloppy". If the method's handler just catches `Exception`, or even worse `Throwable`, it might accidentally capture exceptions that should be handled elsewhere. Figure 2.8 shows an example of the **Catching** `Exception` antipattern:

```
try() {  
    m1();  
} catch (Exception e) {  
    Logger.log(e.getMessage());  
}
```

**Figure 2.8** Catching `Exception` antipattern.

### 2.3.7 Destructive Wrapping

The handler should never catch an exception and throw another exception losing the stack trace of the original exception. McCune [McC06] says "This destroys the stack trace of the original exception, and is always wrong". Figure 2.9 shows an example of the **Destructive Wrapping** antipattern:

```
try() {  
    m1();  
} catch (SomeException e) {  
    throw new OtherException("Exception:" + e.getMessage());  
}
```

**Figure 2.9** Destructive wrapping antipattern.

### 2.3.8 Log and return null

This antipattern is not always incorrect, but it is commonly wrong. The correct way of reporting errors is by throwing exceptions and letting the callers handle them. Returning `null` should be used only in a normal situation and not in an exceptional one. Figure 2.10 shows an example of the **Log and return null** antipattern:

```
try() {  
    m1();  
} catch (SomeException e) {  
    Logger.log(e.getMessage());  
    return null;  
}
```

**Figure 2.10** Log and return `null` antipattern.

### 2.3.9 `catch` and Ignore

The handler should never only `return null` because it will lose the exception information. It should handle or rethrow the exception instead. This problem is just a special case of **Swallowed Exception** 2.3.1 Figure 2.11 shows an example of the `catch` and **Ignore** antipattern:

```
try() {  
    m1();  
} catch (SomeException e) {  
    return null;  
}
```

**Figure 2.11** `catch` and ignore antipattern.

### 2.3.10 `throw from Within` `finally`

When there is a method call inside the `try` block that can throw an exception and there is a method call in the `finally` block that can throw another exception as well, this is a problematic situation. The first exception thrown by the `try` block will be lost if the exception from the `finally` block is thrown. A more appropriate way of dealing with exceptions within a `finally` block would be handling or logging the exception and not letting it bubble out. Figure 2.12 shows an example:

```
try() {  
    m1(); // throws SomeException...  
} finally {  
    clear(); // throws AnotherException...  
}
```

**Figure 2.12** `throw from within finally` antipattern.

### 2.3.11 Multi-Line Log Messages

Logging messages should be put together into as few calls as possible and also the log method should be called as infrequently as possible, ideally at most once per `catch` or `finally` block. The log file might end up comprising thousands of lines if there are too many log messages in the code or too many calls to the log method. Figure 2.13 shows an example code of **Multi-Line Log Messages**:

---



```
try() {  
    m1();  
} catch (SomeException e) {  
    Logger.log(`The exception was thrown...`);  
    Logger.log(`Please contact support...`);  
    Logger.log(`The problem is...`);  
}
```

**Figure 2.13** Multi-line log messages antipattern.

### 2.3.12 Relying on `getCause()`

This is not necessarily a problem. However, if the system implementation changes (for example, so that the exception cause is now wrapped in another one), it can break exception handling. A call to `getCause()` on the outermost exception object would produce an exception object that would not correspond to the original cause of the problem. In this case the code should rely on `e.getCause().getCause()`. The ideal approach should be relying on the `getRootCause()` method. Figure 2.14 shows an example of **Relying on `getCause()`**, in this case the program would fail because `getCause()` will not get the correct exception cause.

```
try() {  
    int temperature = 50;  
} catch (SomeException e) {  
    if (e.getCause() instanceof OtherException) {  
        temperature = 45;  
        throw new AnotherException();  
    }  
} catch (AnotherException ae) {  
    throw new YetAnotherException(ae);  
}
```

**Figure 2.14** Relying on `getCause()` antipattern.

# 3

## Methodology

In this chapter we describe the methodology of this study. It aims to answer four research questions:

- RQ1: Do organizations and developers take exception handling into account?
- RQ2: How commonplace are exception handling bugs?
- RQ3: Are exception handling bugs harder to fix than other bugs?
- RQ4: What are the main causes of exception handling bugs?

In addition, we aim to contrast the perceptions of developers regarding exception handling bugs and their actions considering the bugs that get reported. However, as the title of this work emphasizes, this study is exploratory. As such, we want to find questions as much as answers about exception handling bugs.

To gather information about developers' perceptions and intentions about exception handling bugs, a self-administered questionnaire was conducted to get both direct and discursive responses. We also examined the bug repositories of two target applications: Tomcat and Eclipse. We chose these two systems because they are mature and large systems, both written in Java, the language that arguably popularized exception handling. Additionally, they were examined in a number of empirical studies [[LS07](#), [ZPZ07](#), [Mar11](#), [SCA10](#)]. Furthermore they use Bugzilla as their bug reporting system, which has powerful search features.

This chapter is organized as follows: Section [3.1](#) starts out by discussing what "exception handling bug" means. Section [3.2](#) then proceeds to explain how we analyzed the repositories of Tomcat and Eclipse. Finally, Section [3.3](#) presents the questionnaire that we administered.

### 3.1 What is an Exception Handling Bug?

The first methodological issue that we must address in this work is to define what we mean by exception handling bugs. There is no widely accepted definition of exception handling bug. In this study we want to select bug reports where exception handling is associated with the cause of the problem. Bugs in exception definition, exception throwing, exception handling, exception propagation, exception documentation and clean-up actions (`finally` blocks) are all of interest. Therefore, an exception that is not thrown when it should be according to the expected behavior of the system is an exception handling bug. The same applies to a `catch` block that captures exceptions that it should not. For simplicity, other approaches for signaling and handling errors, such as return codes [BvDT06], are not covered by our study. As such, we can create a general definition of exception handling bug:

**Exception Handling Bug:** Is a bug whose cause is related to exception handling. There are several kinds of bugs regarding exception handling:

- In the definition of the exception;
- When the exception is thrown;
- When the exception is handled;
- In the propagation of the exception;
- In the documentation of the exception;
- In the clean-up action of a protected region where the exception is thrown;
- When the exception should be thrown and it is not;
- When the exception should be handled and it is not.

A previous study on the subject [SAW12] defines exception handling bugs ("defects") in terms of bug reports that include any mention of the words "exception", "throw", and "threw" or its derivatives, e.g., "exceptional", "exceptions", "thrown", words ending with "exception", etc. We believe that this liberal approach allows for a number of bugs that are not related to exceptions to be taken into consideration. For example, if a program performs a division by zero, that error will be trapped by the runtime system of the language (in the case of Java and C#, among others) and an exception will be thrown. The raising of the exception in this case is not the cause of the problem. Instead, it is

a mechanism for the runtime system to indicate that some problem occurred. From a bug fixing perspective, as soon as the bug is fixed, the part of the code where the bug manifested will not necessarily have a relationship with exception handling anymore. Therefore, we do not consider this to be an exception handling bug. Nevertheless, it would still be treated as such in the aforementioned study [SAW12]. On the other hand, if a `catch` block throws a `NullPointerException` when it should be throwing an `IOException`, even after the bug is fixed the locus of its manifestation will still be associated with exception handling (`catch` block) and throwing (because it is expected to throw `IOException`). Thus, we consider this to be an exception handling bug.

Clearly identifying exception handling bugs is important. Exception handling is a complex mechanism. Some consider that it is so complex that it should not be used at all [Bla82]. However, many modern languages include exception handling mechanisms and programmers do use them in practice [CM07, WN08]. By analyzing bugs that are directly related with these mechanisms, we want to better understand the impact of this complexity on software development. Taking into account every bug that mentions exceptions, even if its cause has nothing to do with exceptions, defeats this purpose.

Figures 3.1 and 3.2 present two real bugs, one related to exception handling and the other unrelated. Figure 3.1 shows a real exception handling bug (ID 21018) case from Eclipse's bug repository. The problem reported was that a core exception was being thrown: *"junit wizard: core exception not handled [JUnit]"*. After some discussion and analysis, the support team found out that the problem was in the exception handler. We can clearly see that the cause of the problem is that the exception was not being correctly handled, as the patch changes the `catch` code so that instead of only logging the exception, now the handler does something else.

Figure 3.2 shows a real non-exception handling-related bug (ID 81417) from Eclipse's bug repository. 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 clearly see that the problem was not related to exception handling because the cause of the bug was regarding to the application's business rules: some program field should not be `null` and the program did not check that. In the patch we can see that there is no code related to exception handling.

---

### 3.1. WHAT IS AN EXCEPTION HANDLING BUG?

---

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

////////// PATCH CODE //////////
    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 3.1** Real exception handling bug from Eclipse - bug ID 21018.

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

////////// PATCH CODE //////////
        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 3.2** Real non-exception handling bug from Eclipse - bug ID 81417.

---

## 3.2 Repository Analysis

Mining software repository [KCM07] was used in this work, we analyzed the metadata from the bug repository of Eclipse and Tomcat: the Bugzilla. It is a bug tracking system that maintains the history of the entire lifecycle of a bug. To search the Bugzilla repositories of the two target applications, we employed the following search string:

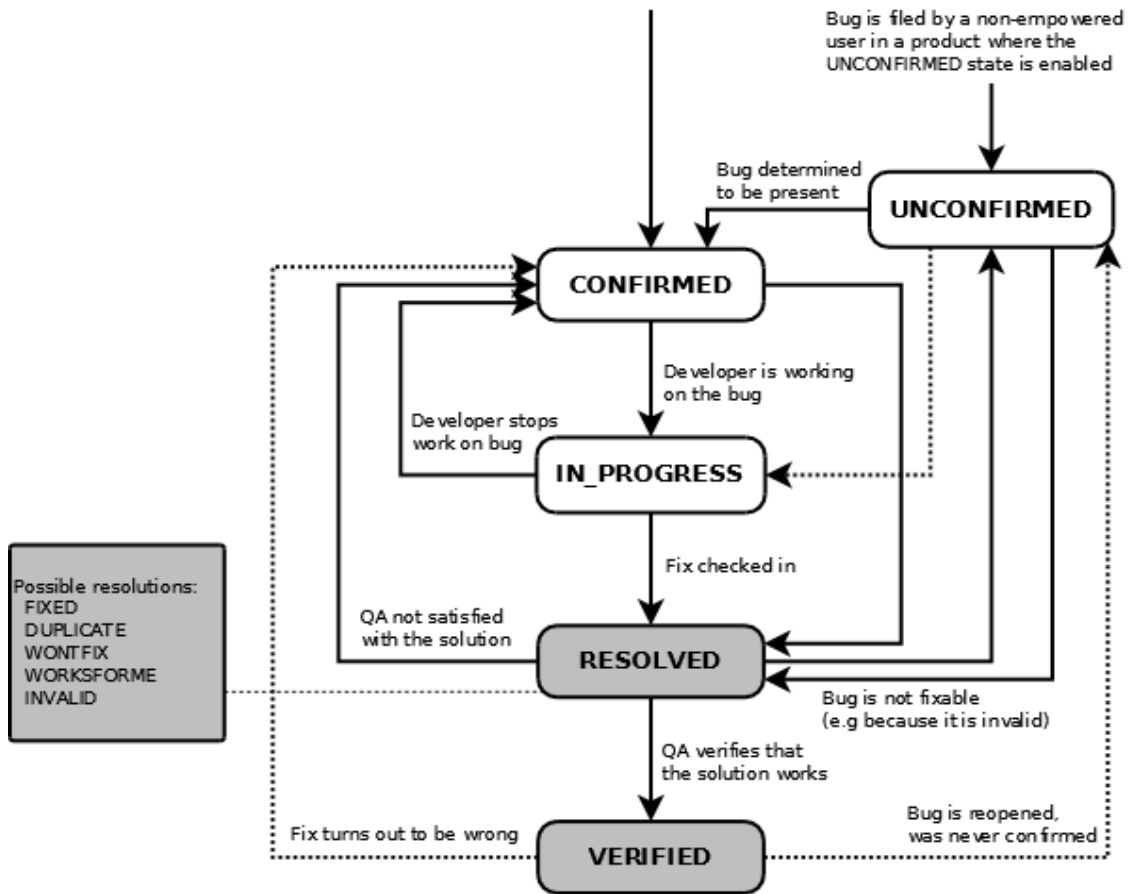
*catch OR caught OR handl OR exception OR throw OR finally OR raise  
OR signal*

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 exception handling bugs with that search string. We think that all those keywords have a high probability of being 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 do have, different *status* and *resolution* categories to be used in the search. Moreover, there are several fields we can select from the project in the search: *classification*, *product*, *component*, *status*, *resolution*, *priority*, *severity*, *version*, and others. Below we describe which fields were used to search the bug repositories of Eclipse and Tomcat.

Figure 3.3 shows the life cycle of a bug in Bugzilla system. There are two possibilities for the birth of the bug: it is created as *unconfirmed* and then someone confirms that it is a bug or it is created already as a *confirmed* bug, if it is possible to say that it is a bug in fact. When someone starts working on it, the *status* is changed to *in progress*. When the assigned developer finishes working on that bug, it is marked as *resolved* and there are several possible *resolutions*:

- **Fixed:** The bug is fixed;
- **Duplicate:** The bug is a duplicate of another existing bug;
- **Workforme:** Support team was not able to reproduce the bug and so there was no solution;
- **Won'tfix:** The issue reported is a bug which will never be fixed;
- **Invalid:** The issue reported is not a bug.



**Figure 3.3** Life cycle of a bug in Bugzilla [htt13a].

If the support team verifies that the solution is fine, the bug is marked as *verified*. If the solution is not fine the bug is marked again as *confirmed* and the process starts over.

As Eclipse is a huge project, it would be impossible to use all its components in the search. Hence, we selected only parts of it. Starting with the *classification*, we selected only the "Core" Eclipse IDE, and left out *components* such as Mylyn, BIRT, and so on. Under Eclipse, we selected only the *product* JDT (Java Development Tools) so we could analyze only *components* related to the Java IDE. Under JDT, we selected only the *components* Core (Java IDE headless infrastructure) and User Interface (Java IDE user interface). We selected those two components because we consider they are the most important of the Java IDE and they are coarse-grained components implementing different functionalities. Each comprises tens of thousands of lines of code and, more importantly, there are 26,002 bug reports associated with them<sup>1</sup>. In terms of bug *statuses*, we only exclude *unconfirmed*. This is because we do not consider a bug in fact if it has

<sup>1</sup>As of 01/23/2013.

not been *confirmed* yet. We included all the possible *resolutions* ("—" [still open], *fixed*, *wontfix*, *worksforme*, *moved*) in the search, except for *invalid*, *duplicate*, and *not\_eclipse*. We ignore duplicates because there is already another entry for the real bug. The search in Eclipse's bug repository returned 1,779 bug reports out of 26,002. Manual analysis of the 1,779 bug reports yielded 92 exception handling 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. We need to emphasize that this analysis relied on subjective judgement and we spent almost 3 months to finish all analysis for both tools (Eclipse and Tomcat).

For Tomcat, we included all Tomcat *products* in the search: *versions 3 to 8*, *Connectors*, *Modules*, and *Native*. We selected all *components* for all *products*. All *statuses* were included (*new*, *assigned*, *reopened*, *needinginfo*, *resolved*, *verified*, *closed*) except for *unconfirmed*. This is because *unconfirmed* bugs reports are not necessarily bugs. We included all the possible *resolutions* ("—" [still open], *fixed*, *wontfix*, *later*, *remind*, *worksforme*, *moved*) in the search, except for *invalid* and *duplicate*. We ignore duplicates because there is already another entry for the real bug. The search in Tomcat's bug repository returned 740 bug reports out of a total of 6,855<sup>2</sup>. In the end, we found 128 exception handling bugs in the Tomcat repository after the manual analysis. Tables 3.1 and 3.2 summarize these numbers for both Eclipse and Tomcat.

**Table 3.1** Summary of Eclipse bugs.

Exception handling bugs	92	0.35%
Bugs resulting from the search	1,779	6.84%
Other kind of bugs	25,910	99.65%
Total number of bugs	26,002	100%

**Table 3.2** Summary of Tomcat bugs.

Exception handling bugs	128	1.87%
Bugs resulting from the search	740	10.80%
Other kind of bugs	6,727	98.13%
Total number of bugs	6,855	100%

The bug reports were downloaded as a set of XML files and manually analyzed. After that, we employed a Java program to prepare the data set to be analyzed by an R [R93] script to process the exception handling bug results. With the R script we could

<sup>2</sup>As of 01/23/2013.



generate statistics and graphics of the results, Appendix C show all R scripts used in this work.

### 3.3 Survey

The questionnaire used in this work was designed according to the recommendations of [GFC<sup>+</sup>09, KP08], following the phases prescribed by the authors: planning, creating the questionnaire, defining the target audience, evaluating, conducting the survey, and analyzing the results. Firstly, we defined the topics for the questions. The topics are: respondents's experience, the languages they are familiar with, how the phases of project design, documentation and testing in their organizations are done, and lastly, we asked direct questions about exception handling and exception handling bugs. The questionnaire has 24 questions and it is structured to limit responses to multiple-choice, Likert scales (responses given in a scale), and also free-forms.

Our target population consists of developers with some experience in Java. After defining all the questions on the questionnaire, we obtained feedback iteratively and clarified and rephrased questions and explanations. This feedback was obtained from analysis and discussion with a group of specialists and also from two pilots of the survey. We also added new questions and removed existing ones based on this feedback. Table 3.3 presents the main questions on the questionnaire. The whole survey is available in Appendix A. The data we collected was analyzed with both descriptive and inferential statistics. For the former, for example, we employed the mean and median, and for the latter we employed Student's T-test [MR06] where appropriate. The tool used to process the data was R, the same as in the repository analysis. For some questions, we analyzed the results based on the experience of the respondents with Java development. We considered a developer to be novice if he/she claimed to have 5 or less years of experience in software development. We considered a developer to be experienced if he/she claimed to have more than 10 years of experience.

The questionnaire was sent to two distinct groups of people. The first one consisted mostly of Portuguese-speaking developers in Brazilian companies and universities. We sent the questionnaire to known points of contact and asked them to redistribute it within their organizations. The second group consisted of bug reporters and developers of Eclipse and Tomcat. Over a period of 2 months we sent more than 4,000 emails. In total we obtained 154 responses, 96 from Portuguese-speaking and 58 from developers of Eclipse and Tomcat.

**Table 3.3** 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? 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?
Testing	9. Are there specific tests for the exception handling code in your organization?
Bugs	10. How often do you find bugs related to exception handling? 11. How often do you find bugs that are not related to exception handling? 12. Estimate the percentage of bugs related to exception handling code in your projects (estimate a value between 0 and 100%). 13. Have you ever needed to fix bugs related to exception handling? 14. 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. Why do you use exception handling in your projects? (you can select more than one answer) 19. What is your opinion about the quality of exception handling code in your projects compared to other parts of the code? 20. What is the average priority/severity of reported bugs related to exception handling code?

# 4

## Study Results

In this chapter we present the results for both the repository analysis and the survey based on the four research questions of this work. The main goal of the survey we conducted is to understand developers' perceptions about exception handling bugs. It is known that non-trivial systems usually have bugs that are difficult to find, stemming from complex control flow and overly general `catch` blocks [RM03], and from I/O operations [ZE12]. However, even though the perceptions of developers about exception handling in general have been studied, at least on a small scale [SGH10], to the best of our knowledge there are no studies that attempt to understand how developers regard exception handling bugs.

We had 154 responses to the survey. Respondents had, on average, between 7 and 10 years of software development experience (question 1) as we can see in Table 4.1. Most are currently working on medium-sized projects ranging between 50 and 100KLoC (question 2), though more than 40% work on large projects with more than 100KLoC as we can see in Table 4.2. 98.70% of the survey respondents are Java programmers, 61.69% are JavaScript programmers, and 38.31% are C++ programmers (question 3). More than 66% of the respondents have worked professionally with at least three programming languages (mean of 3.29) as we can see in Table 4.3.

**Table 4.1** For how long have you been a Java developer?

Less than 2 years	7.14%
2 to 5 years	20.78%
5 to 7 years	14.94%
7 to 10 years	20.13%
More than 10 years	37.01%

The main goal of the repository analysis is to discover the main causes of real exception handling bugs. As presented in Section 3.2, we found only 92 exception handling

#### 4.1. RQ1: DO ORGANIZATIONS AND DEVELOPERS TAKE EXCEPTION HANDLING INTO ACCOUNT?

---

**Table 4.2** What is the approximate size of the project you are currently working on?

Less than 20K LOC	24.03%
20K to 50K LOC	20.13%
50K to 100K LOC	13.64%
100K to 200K LOC	9.09%
More than 200K LOC	33.12%

**Table 4.3** Which programming languages have you professionally worked with?

Java	98.70%
Javascript	61.69%
C++	38.31%
C	32.47%
C	30.52%
PHP	22.73%
Python	15.58%
Perl	13.64%
Ruby	9.74%
Objective-C	6.49%

bugs in Eclipse and 128 in Tomcat. The remainder of this chapter is organized as follows: in Section 4.1, we present the results for the RQ1: "Do organizations and developers take exception handling into account". Section 4.2 discuss about the RQ2: "How commonplace are exception handling bugs?", Section 4.3 about RQ3: "Are exception handling bugs harder to fix than other bugs?". And finally, Section 4.4 shows results for RQ4: "What are the main causes of exception handling bugs?". In Section 4.5 we make a discussion of all the results and in Section 4.6 we present the threats to the validity of this work.

### 4.1 RQ1: Do organizations and developers take exception handling into account?

With RQ1 we are interested in understanding how developers and organizations deal with exception handling. We want to know if they take it into account in all phases of the development process. In this section we present the results for RQ1. We also make considerations and compare the results from the survey and the repository analysis. In Section 4.1.1 we present the results from the survey and in Section 4.1.2 we present the results from the repository analysis.

### 4.1.1 Survey

The survey included five questions whose goal was to determine whether developers worry about exception handling when they are not directly implementing the system, e.g., designing, testing, etc. Three of the questions are listed below:

- **Question 4.** In the design phase of your projects, what is the importance given to the documentation of exception handling?
- **Question 5.** Are there any specifications, documented policies or standards that are part of your organization's culture related to the implementation of error handling?
- **Question 9.** Are there specific tests for the exception handling code in your organization?

When asked whether "there are any specifications, policies or standards that are part of your organization's culture related to the implementation of error handling" (question 5), only 27% of the developers answered *yes*. In a similar vein, 30% said that there are specific tests for exception handling code in their organizations (question 9). We also asked them "in the design phase of your projects, what is the importance given to the documentation of exception handling?" (question 4). 61% of the respondents said that *none* to *little* importance is given to the documentation of exception handling in the design phase. Moreover, only 16% said that *much* to *very much* importance is given to exception handling documentation in the design phase.

For the question 5, we left a text field so respondents could enter their own answers. One interesting spontaneous answer is presented below and it says the same of the antipattern "Log and Throw" from the Section [2.3.3](#):

*"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 excepted and no warning should be raised here."*

We also asked the questions:

- **Question 18.** Why do you use exception handling in your projects?

#### 4.1. RQ1: DO ORGANIZATIONS AND DEVELOPERS TAKE EXCEPTION HANDLING INTO ACCOUNT?

- **Question 19.** What is your opinion about the quality of the code that performs exception handling in your projects compared to the quality of other parts of the code?

About 40% of the respondents consider the quality of exception handling code ranges between *good* and *very good* (question 19). Surprisingly, only 14% of the respondents consider it to be *bad* or *very bad*. This result seems to vary with developer experience. We found that more experienced respondents tend to consider the quality of exception handling code to be worse. The answers differ significantly when we compare novice and experienced developers. Student's T-test produced a p-value of 0.02478.

Inspired by the work of Shah et al. [SGH10], we asked the respondents "why do you use exception handling in your projects?" (question 18). We provided them with a list of possible causes of exception handling bugs and gave them the opportunity to suggest additional ones. Table 4.4 presents the causes more frequently cited by the respondents.

**Table 4.4** 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%

Most of the respondents said that creating ways to tolerate faults and improving the quality of a functionality are the main reasons it is used. One of the survey respondents provided a particularly interesting spontaneous answer for this question:

*"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."*

Only 17% of the respondents said that they use exception handling for debugging purposes. In contrast, in the work of Shah et al. [SGH10], 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 requirements. Expert developers on the other hand, claimed that they use exception handling mainly to convey understandable failure messages. This latter result seems to be coherent with ours, where

#### 4.1. RQ1: DO ORGANIZATIONS AND DEVELOPERS TAKE EXCEPTION HANDLING INTO ACCOUNT?

---

most of the respondents claim that they use exception handling to create ways to tolerate faults or improve the quality of a functionality.

Based on previous work discussing the problems with checked exceptions in Java [CM07], we expected a larger percentage of respondents to mention *language requirement* as a reason for using exceptions. Moreover, only 21% use exception handling because of *organizational policies*. This result and the two previously discussed results pertaining to testing (question 9) and documentation (questions 4 and 5) suggest that organizations do not pay attention to exception handling, even though developers do. Finally, it is worth noting that the 3 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.

##### 4.1.2 Repository Analysis

There are already many studies [BvDT06, CCF<sup>+</sup>09, CM07, SAW12, WN08] which, based on factual data, show that developers do pay attention to error handling. Assessing whether they also pay attention to exception handling bugs is harder, since many exception handling bugs are probably never uncovered due to insufficient or non-existent testing procedures. However, we can use bug report information to analyze the bugs that do get reported. If exception handling bugs and other bugs have similar characteristics, that would suggest that developers take them in account, at least when they are reported.

Tables 4.5 (a) and 4.5 (b) present a comparison between bug reports pertaining to exception handling bugs and other bugs in terms of their priorities, severities, resolutions, status and the presence of attachments for Eclipse and Tomcat.

In Eclipse, exception handling bug reports carry attachments more often than other bug reports: the percentage is twice as high for exception handling bugs. In Tomcat, the percentage of bug reports that carry attachments is very close for exception handling bugs and other bugs. Somewhat intuitively, it is less common for an exception handling bug to be an *enhancement* than for other bugs. Besides this, *normal* is the most common priority in both systems for exception handling bugs and non-exception handling bugs.

For these two applications, *wontfix* and *worksforme* are more common as a resolution for other bugs than for exception handling bugs: the percentage is twice as high for non-exception handling bugs. It is also interesting to note that for both systems, proportionally more exception handling bugs have the *fixed* resolution than other bugs. When examined in conjunction, these two results suggest that reported exception handling bugs are ignored less often than other bugs. For severity category, as we can in

#### 4.1. RQ1: DO ORGANIZATIONS AND DEVELOPERS TAKE EXCEPTION HANDLING INTO ACCOUNT?

**Table 4.5** Priorities, severities, resolutions, status, and the presence of attachments for exception handling and non-exception handling bug reports for Eclipse (a) and Tomcat (b).

ECLIPSE	Category	Exception handling bugs		Other bugs	
Attachment	With	49	53.26%	6,799	26.24%
	Without	43	46.74%	19,111	73.76%
Priority	Blocker	1	1.09%	116	0.45%
	Critical	0	0.00%	537	2.07%
	Enhancement	2	2.17%	4,440	17.14%
	Major	8	8.70%	1,762	6.80%
	Minor	4	4.35%	1,463	5.65%
	Normal	74	80.43%	17,101	66.00%
	Trivial	3	3.26%	491	1.90%
Resolution	– (open)	1	1.09%	3,251	12.55%
	Fixed	88	96.65%	15,597	60.20%
	Wontfix	2	2.17%	3,775	14.57%
	Worksforme	1	1.09%	3,287	12.69%
Severity	P1	1	1.09%	581	2.24%
	P2	5	5.43%	1,60	6.21%
	P3	85	92.39%	22,327	86.17%
	P4	1	1.09%	899	3.47%
	P5	0	0.00%	494	1.91%
Status	Assigned	0	0.00%	1,283	4.95%
	Closed	0	0.00%	444	1.71%
	New	1	1.09%	1,881	7.26%
	Reopened	0	0.00%	87	0.34%
	Resolved	28	30.43%	13,349	51.52%
	Verified	63	68.48%	8,866	34.22%

(a)

TOMCAT	Category	Exception handling bugs		Other bugs	
Attachment	With	38	29.69%	1,901	28.26%
	Without	90	70.31%	4,826	71.74%
Priority	Blocker	2	1.56%	223	3.31%
	Critical	4	3.13%	392	5.83%
	Enhancement	10	7.81%	1,055	15.68%
	Major	30	23.44%	863	12.83%
	Minor	9	7.03%	548	8.15%
	Normal	73	57.03%	3,467	51.54%
	Regression	0	0.00%	58	0.86%
	Trivial	0	0.00%	121	1.80%
Resolution	– (open)	3	2.34%	219	3.26%
	Fixed	108	84.38%	4,429	65.84%
	Later	2	1.56%	141	2.10%
	Moved	0	0.00%	1	0.01%
	Remind	0	0.00%	21	0.31%
	Wontfix	10	7.81%	1,182	17.57%
	Worksforme	5	3.91%	734	10.91%
Severity	P1	7	5.47%	535	7.95%
	P2	52	40.63%	2,834	42.13%
	P3	66	51.56%	3,173	47.17%
	P4	1	0.78%	59	0.88%
	P5	2	1.56%	126	1.87%
Status	Assigned	0	0.00%	1	0.01%
	Closed	6	4.69%	237	3.52%
	Needinfo	1	0.78%	15	0.22%
	New	2	1.56%	181	2.69%
	Reopened	0	0.00%	22	0.33%
	Resolved	119	92.97%	6,268	93.18%
	Verified	0	0.00%	3	0.04%

(b)



Table 4.5, the results are very similar for exception handling and others bugs for both Eclipse and Tomcat. This would indicate that exception handling bugs usually have the same severity as others bugs.

The results for status in Tomcat are very similar for exception handling and other bugs. In Eclipse, there is a slight difference for the status *resolved* and *verified*. Proportionally, there are twice as many exception handling bugs *verified* as other bugs, i.e. someone did verify the bug after it was resolved.

We perceived a difference in the usage of Bugzilla for Eclipse and Tomcat. Developers usually used the status *verified* to mark a bug as *fixed* and properly working, as it is the normal life cycle of a bug in Bugzilla. But we did not see this for Tomcat developers. They usually do not use the status *verified*. Instead, they usually make comments in the bug report to say that the bug is fixed and properly working. That would explain why there are almost no bugs marked as *verified* for Tomcat.

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 developers do pay attention to *documented* exception handling bugs at least as much as they do to any other bug. We can see that in the comment from a respondent (translated from Portuguese from question 14):

*"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".*

## 4.2 RQ2: How commonplace are exception handling bugs?

With RQ2 we want to know how usual exception handling bugs are. In this section we present the results for RQ2. Additionally, we make considerations and compare the results from the survey and the repository analysis. In Section 4.2.1 we present the results from the survey and in Section 4.2.2 we present the results from the repository analysis.

### 4.2.1 Survey

To assess how commonplace developers believe exception handling bugs to be, we asked them:

- **Question 10.** How often do you find bugs related to exception handling?

---

## 4.2. RQ2: HOW COMMONPLACE ARE EXCEPTION HANDLING BUGS?

---

- **Question 11.** How often do you find bugs that are not related to exception handling?
- **Question 12.** Estimate the percentage of bugs related to exception handling code in your projects (estimate a value between 0 and 100%)

The answers are shown in Tables 4.6 and 4.7. We put the answers in a numeric scale and used Student's T-test to check whether the answers are significantly different. According to the respondents, exception handling bugs are less frequently found than other bugs (question 10 and 11). We obtained a p-value of less than 0.0001. We also asked them to estimate the percentage of exception handling bugs in their projects (question 12). The mean estimate was 9.72% and the median was 5%.

**Table 4.6** How often do you find bugs related to exception handling?

Never	1.30%
Rarely	38.96%
Sometimes	53.90%
Most of the time	5.84%
Always	0.00%

**Table 4.7** How often do you find bugs that are not related to exception handling?

Never	0.65%
Rarely	4.55%
Sometimes	29.22%
Most of the time	59.74%
Always	5.84%

Additionally, we presented the respondents with the questions:

- **Question 6.** How often are bugs reported at your organization?
- **Question 7.** How often are bugs related to exception handling reported at your organization?

Most of the respondents answered that exception handling bugs are reported *rarely* (question 7). Other bugs are reported *always* (question 6). The answers for the two questions differ significantly with a p-value of less than 0.0001, this means that in general, other kinds of bugs are reported more frequently than exception handling bugs. The answers for those two questions are shown in Tables 4.8 and 4.9.

### 4.3. RQ3: ARE EXCEPTION HANDLING BUGS HARDER TO FIX THAN OTHER BUGS?

---

**Table 4.8** How often are bugs reported at your organization?

Never	3.25%
Rarely	7.79%
Sometimes	27.27%
Most of the time	29.22%
Always	32.47%

**Table 4.9** How often are bugs related to exception handling reported at your organization?

Never	10.39%
Rarely	31.82%
Sometimes	28.57%
Most of the time	14.94%
Always	14.29%

#### 4.2.2 Repository Analysis

Reported exception handling bugs are rare. As mentioned in Chapter 3, we obtained 92 and 128 exception handling bugs for Eclipse and Tomcat, amounting respectively to 0.35% and 1.87% of all the bugs. However, according to previous studies [CM07, WN08], between 5% and 7% of the lines of code of mature Java applications implement exception handling. Assuming that exception handling code is not more likely to exhibit bugs than other parts of the code, one would expect the percentage of exception handling bugs to be proportional to the percentage of exception handling code.

In addition, previous work has provided evidence that exception handling code is fertile ground for bugs that are difficult to detect [CM07, RM03, WN08]. Hence, we believe that exception handling bugs are more commonplace than one would assume by looking at the bug reports. It is also important to mention that in the manual analysis of the bugs we found several bug reports that did not contain enough information about the cause of the bug. As such, for those bugs we could not classify as exception handling bugs, we marked them as non-exception handling. This could be a reason why we found a very small number of exception handling bugs.

### 4.3 RQ3: Are exception handling bugs harder to fix than other bugs?

With RQ3 we are interested in discovering how hard it is to fix exception handling bugs as compared to non-exception handling bugs. In this section we present the results for

---

### 4.3. RQ3: ARE EXCEPTION HANDLING BUGS HARDER TO FIX THAN OTHER BUGS?

---

RQ3. Additionally, we also make considerations and compare the results from the survey and the repository analysis. In Section 4.3.1 we present the results from the survey and in Section 4.3.2 we present the results from the repository analysis.

#### 4.3.1 Survey

Our study revealed that respondents consider exception handling bugs easier to correct than other types of bugs. We asked them the following three questions related to research question RQ3:

- **Question 16.** What is the average level of difficulty to fix bugs related to exception handling?
- **Question 17.** What is the average level of difficulty to fix other bugs that are not related to exception handling?
- **Question 20.** What is the average priority/severity of reported bugs related to exception handling code?

Tables 4.10 and 4.11 shows the percentages of the responses for questions 16 and 17.

**Table 4.10** What is the average level of difficulty to fix bugs related to exception handling?

Very easy	9.09%
Easy	34.42%
Medium	46.10%
Hard	10.39%
Very hard	0.00%

**Table 4.11** What is the average level of difficulty to fix other bugs that are not related to exception handling?

Very easy	0.00%
Easy	7.14%
Medium	65.58%
Hard	25.97%
Very hard	1.30%

43% of the respondents consider exception handling bugs to be *easy* or *very easy* to be fixed (question 16). In sharp contrast, only 7% of the respondents say the same about other kinds of bugs (question 17). We employed the T-test to analyze whether the

### 4.3. RQ3: ARE EXCEPTION HANDLING BUGS HARDER TO FIX THAN OTHER BUGS?

---

answers for the difficulty in fixing exception handling and other bugs are significantly different. We found a p-value of less than 0.0001, thus, according to the respondents, exception handling bugs are easier to fix than other bugs.

We also asked them the following "what is the average priority / severity of reported bugs related to exception handling code?" (question 20). The responses are shown in Table 4.12. Most of the respondents answered that the priority / severity of exception handling bugs is *medium*. It is interesting to note that novice developers think that the priority / severity of exception handling bugs is lower than what expert developers think it is (p-value of 0.0474). This result reinforces the results from [SGH10] which had the opinion of 15 developers. However our result is based on the opinion of a much larger number of developers.

**Table 4.12** What is the average priority / severity of reported bugs related to exception handling code?

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

#### 4.3.2 Repository Analysis

We used two measurements as proxies for the difficulty to fix a bug:

- The number of discussion messages associated with the bug report;
- The time to fix of the bug (measured in days);

Both have been employed with this goal in previous studies [FLSR10]. Table 4.13 present order statistics for these two measurements. We employed the T-test to check whether the measurements differ significantly between exception handling bugs and other bugs.

### 4.3. RQ3: ARE EXCEPTION HANDLING BUGS HARDER TO FIX THAN OTHER BUGS?

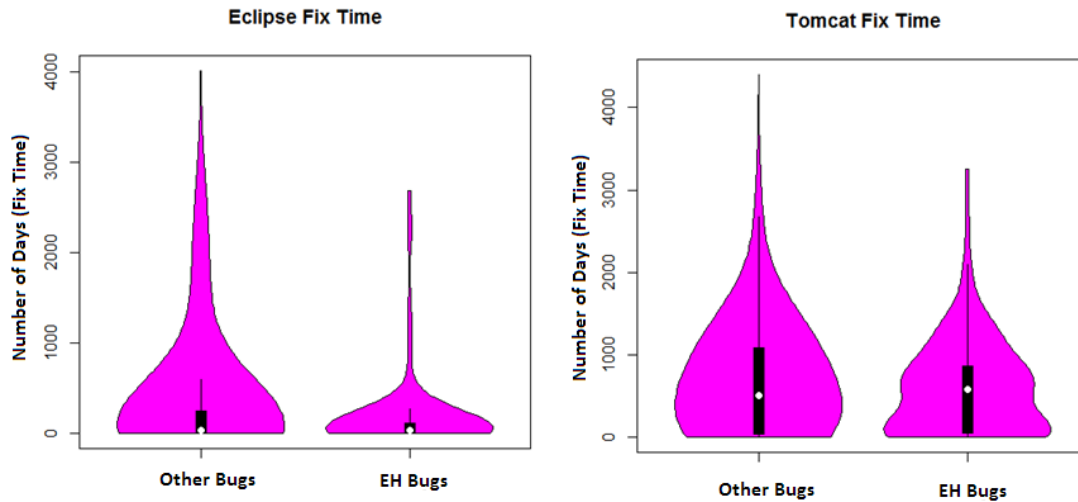
**Table 4.13** Fix time in days and number of discussion messages for exception handling bugs and for other bugs.

		Fix time		Number of discussion messages	
		Exception Handling Bugs	Other Bugs	Exception Handling Bugs	Other Bugs
<b>Eclipse</b>	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
	Var.	271,436.1	450,693.5	26.2	42.3
	SD	466.3	671.3	5.1	6.5
<b>Tomcat</b>	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
	Var.	361,748.1	405,785.0	7.78	20.2
	SD	601.4	637.0	2.7	4.4

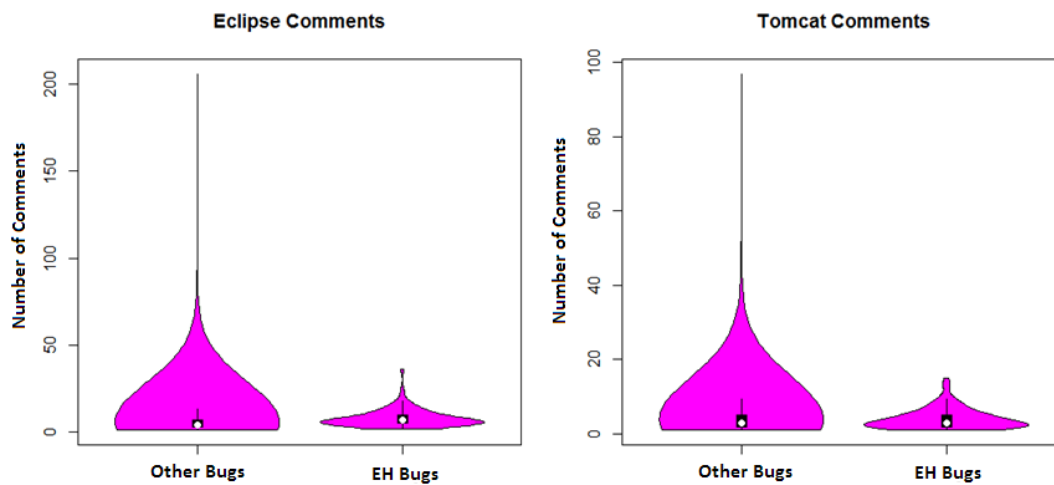
Figures 4.1 and 4.2 present the Vioplot [HN98] for the fix time and the number of comments for Eclipse and Tomcat. Violin plots are combinations of box plots and kernel density plots. It is similar to a box plot, but it also has a rotated kernel density plot to each side of the box plot. Kernel density plot is a method used to estimate the density of the data according to a central value (the kernel).

### 4.3. RQ3: ARE EXCEPTION HANDLING BUGS HARDER TO FIX THAN OTHER BUGS?

---



**Figure 4.1** Graphic of the density of fix time.



**Figure 4.2** Graphic of the density of comments.

In Eclipse, the fix time is significantly shorter ( $p\text{-value} < 0.001$ ) for exception handling bugs than for other bugs. However, the number of discussion messages for exception handling bugs is significantly greater ( $p\text{-value} < 0.001$ ).

In Tomcat, the fix time is not significantly different ( $p\text{-value} = 0.3667$ ). As for the number of discussion messages, it is smaller for exception handling bugs ( $p\text{-value} = 0.03441$ ).

---

#### 4.4. RQ4: WHAT ARE THE MAIN CAUSES OF EXCEPTION HANDLING BUGS?

---

Based on these two measurements we could not answer RQ3. In Eclipse we have shorter fix time for exception handling bugs but on the other hand, the number of discussion comments is greater. Furthermore, in Tomcat we have a significant difference in the number of comments. On average, exception handling bugs have fewer comments. These results lead us to believe that each application has different characteristics on the difficulty to fix bugs, if analyzing only the fix time and the number of discussion messages in the bug reports. It is also important to remember that those two proxies are relative to the level of experience of the developer's team and also to the amount of time spent working on the bugs. Finally, this result can be interpreted as evidence that exception handling bugs are as hard to fix as any other bug.

### 4.4 RQ4: What are the main causes of exception handling bugs?

In this section we present the results for the fourth research question. We also make considerations and compare the results of the survey and the repository analysis. In Section 4.4.1 we present the results from the survey and in Section 4.4.2 we present the results from the repository analysis.

#### 4.4.1 Survey

To uncover the main causes of exception handling bugs according to the survey respondents, we posed three questions:

- **Question 13.** Have you ever needed to fix bugs related to exception handling?
- **Question 14.** If you answered yes to question 13, please describe some of these situations:
- **Question 15.** Select below the main causes of bugs related to exception handling you have ever needed to fix, analyze or have found documented:

Question 15 directly asked them about the main causes. The respondents were allowed to select zero or more causes from a list and could also suggest additional ones. Table 4.14 summarizes these results. Causes 1 to 4 were given to respondents in the survey and the remaining ones (causes 5 to 11) were suggested by the respondents themselves. The most commonly cited causes for exception handling bugs were *lack*



#### 4.4. RQ4: WHAT ARE THE MAIN CAUSES OF EXCEPTION HANDLING BUGS?

*of a handler that should exist, no exception thrown in a situation of a known error and programming error in the catch block.*

**Table 4.14** What are the main causes of exception handling bugs?

Bug Classification	Quantity
1. Lack of a handler that should exist	108
2. No exception thrown in a situation of a known error	85
3. Programming error in the <code>catch</code> block	84
4. Programming error in the <code>finally</code> block	47
5. Exception caught at the wrong level	2
6. <code>catch</code> block where only a <code>finally</code> would be appropriate	1
7. Exception that should not have been thrown	1
8. Wrong encapsulation of exception cause	1
9. Wrong exception thrown	1
10. Lack of a <code>finally</code> block that should exist	1
11. Error in the exception assertion	1

To better understand developers' perceptions about the causes of exception handling bugs, we also asked "have you ever needed to fix bugs related to exception handling?" (question 13) and, pertaining to the this question, "if yes, please describe some of these situations" (question 14). 83% of the respondents have had to fix an exception handling bug at some point (question 13). It is also interesting to note a difference in the answers from novice and expert developers, with a p-value of less than 0.01, expert developers needed to fix more exception handling bugs than novices. This would be related to the experience of the expert developers and also their knowledge of the topic.

113 out of 154 survey respondents answered question 14. The answers varied widely and many of them refer to specific technologies, frameworks and applications. Besides the typical causes highlighted by Table 4.14, 16 of the respondents also cited *exceptions caught at the wrong level* (14.16% of the answers) and 19 of them cited *empty catch blocks* as common causes of exception handling bugs (16.81% of the answers). Furthermore, 3 of them cited both causes in their answers.

#### 4.4.2 Repository Analysis

Table 4.15 presents the causes of bugs that we identified while examining the bug reports. The table shows the number of bugs having each cause and the percentage of the number of exception handling bugs with that cause, for each target application. For Eclipse, the three most common causes of exception handling bugs are (i) *exception not handled* (34.78%), (ii) *error in the handler* (29.35%), and (iii) *exception that should not*

#### 4.4. RQ4: WHAT ARE THE MAIN CAUSES OF EXCEPTION HANDLING BUGS?

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%).

**Table 4.15** Exception handling bug classification according to repository analysis.

Bug Classification	Tomcat Bugs		Eclipse Bugs	
	Quantity	Percentage	Quantity	Percentage
Exception not handled	25	19.53%	32	34.78%
Exception not thrown	21	16.41%	5	5.43%
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%	27	29.35%
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 4.3 shows an example of *exception not handled* and Figure 4.4 shows an example of *error in the handler* of Eclipse. Figure 4.5 shows an example of *exception that should not be thrown* and Figure 4.6 shows an example of *exception not thrown* of Tomcat.

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

**Figure 4.3** An example of "exception not handled" from Eclipse - bug ID 298250.

#### 4.4. RQ4: WHAT ARE THE MAIN CAUSES OF EXCEPTION HANDLING BUGS?

---

```
////////// ORIGINAL CODE //////////  
    } catch (BadLocationException e) {  
        } catch (BadLocationException e) {  
    }  
////////// PATCH CODE //////////  
    } catch (BadLocationException e) {  
        throw Changes.asCoreException(e);  
    }
```

**Figure 4.4** An example of "error in the handler" from Eclipse - bug ID 170237.

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

**Figure 4.5** An example of "exception that should not be thrown" from Tomcat - bug ID 18698.

#### 4.4. RQ4: WHAT ARE THE MAIN CAUSES OF EXCEPTION HANDLING BUGS?

---

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

**Figure 4.6** An example of "exception not thrown" from Tomcat - bug ID 8200.

It is interesting to note the differences between the two systems. *Exception not thrown* is a common cause of exception handling bugs in Tomcat but not in Eclipse. On the other hand, *exception that should not have been thrown*, the third most common cause of exception handling 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 exception handling 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 [CM07, RS03]. However, developers seem to believe that they create many problems [McC06, MS02, Nel09, RS03, Sal07]. Firstly, because they might make bugs subtler. Since *empty catch blocks* ignore exceptions, the problems that the ignored exceptions signalize can only be detected by indirect means. Secondly, because they hinder debugging. Since there is no stack trace and, in fact, no exception, finding the root cause of the problem becomes particularly difficult. Thirdly, because they hurt maintainability. Developers often use *empty catch blocks* in situations where they are certain that a given exception cannot be thrown. Therefore, the *empty catch block* will never be reached. In our study we have seen comments in the source code stating precisely that. However, as a consequence of software maintenance, the preconditions that guaranteed the impossibility of the exception being thrown might be violated, thus introducing subtle bugs. The well-known problems that *empty catch blocks* can create are in stark contrast to the small number of bugs that have them as a cause.

A similar case can be made about `catch` clauses for general exception types, such as `Throwable` or `Exception`. There is convincing evidence [RM03, CRG<sup>+</sup>08, FR07] that they are often sources of bugs. However, overall, we only found 3 bug reports with this cause. In addition, one of them did not really describe an actual bug. It focused instead of highlighting the potential problems of overly general `catch` clauses.

Even though *empty catch blocks* are a well-known bad smell, we found 10 bugs whose patches use *empty catch blocks*. Figures 4.7 and 4.8 show one example from Eclipse and one from Tomcat, respectively. Not every bug points to its corresponding patch. Hence, it is possible that even more patches for exception handling bugs use *empty catch blocks*.

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

**Figure 4.7** A empty catch block patch for Eclipse, bug ID 139160.

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

**Figure 4.8** A empty catch block patch for Tomcat, bug ID 24368.

## 4.5 Discussion

Contrasting the actual bugs that we identified in the repositories with the bugs that developers have had to fix, according to their answers to one of the survey questions, we notice that a number of problems have rarely been documented in bug reports. Some of these problems appear in a number answers, such as exceptions caught at the wrong place. Moreover, there is ample opportunity for problems stemming from inadvertently caught exceptions to manifest. There are many `catch` blocks for general types, such as `Exception` and `Throwable`, in the two target applications of

---

the study. 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 [CRG<sup>+</sup>08, RM03] using static analysis tools have shown that general `catch` blocks do capture exceptions they were not intended to in practice.

In summary: (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):

*"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 4.4, a number of developers seem to believe that *empty catch blocks* are a bad thing. Out of the 113 survey respondents who described exception handling bugs they had to fix in the past, 19 claimed to have fixed bugs where *empty catch blocks* were either a potential or actual causes, as we can see in the comment of one respondent (translated from Portuguese from question 14):

*"Many developers consider it is normal to 'swallow' exceptions, 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."*

However, 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:

*"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."*

---

The citation above shows that the respondent does not want to know about problems in some parts of the system. Furthermore, somewhat surprisingly, examining the patches for the 87 bug reports that also include the corresponding patches as attachment, we discovered that 10 employed *empty catch blocks*. In addition, *empty catch blocks* are often used in conjunction with overly general *catch clauses*. For example, in Tomcat, there are 280 *catch clauses* that capture *Throwable*. Among them, 55 have the following implementation:

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

**Figure 4.9** The implementation of many handlers for *Throwable* in Tomcat.

This implementation means that, technically, the *catch block* is not empty. However, in practice, any exception (and most errors) caught by a *catch block* with this implementation will be simply ignored, as if it were empty. The comment at the end of the code snippet makes it clear that this behavior is intentional. 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 error is not even logged.

The respondents of the survey and the bug reports did not agree on the difficulty to fix exception handling bugs. The former believe that these bugs are easier to fix than other bugs. However, analysis of the repository data has shown that there is no obvious answer. Exception handling 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 average estimated percentage of exception handling bugs is 9.72%. But from the repository analysis we found only 1.87% of exception handling 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 [CM07], the number of exception handling bugs does not seem to be proportional to the amount of exception handling code. There are 3 potential causes for this, in the

context of the two target applications: (i) exception handling code is less bug-prone; (ii) exception handling bugs are not reported; or (iii) many exception handling bugs go undetected. As discussed earlier in this section, there is evidence that the latter is more probable.

### 4.5.1 Exception Handling Bug Classification

In this section we present the exception handling bug classification that we compiled from both results from the survey and the repository analysis. Firstly, we need to explain how we merged both classifications because there are some classifications which are the same. Table 4.16 shows the terms that mean the same classification. It presents which terms from the survey classification are equal to the classification from the repository analysis.

**Table 4.16** Merged classification terms.

Survey Classification	Classification from 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 <code>catch</code> block	Error in the handler
Programming error in the <code>finally</code> block	Error in the clean-up action
Wrong encapsulation of exception cause	Invalid or non-existent root cause

Section 4.4 provides a comprehensive list of causes for exception handling bugs. In summary, our final exception handling bug classification compiled the terms from the survey and the analysis of the bug repository. Table 4.17 presents a comprehensive classification. This list of causes for exception handling bugs can assist testers in devising thorough test suites and can also be used as a checklist to guide code inspections.



**Table 4.17** Comprehensive classification of exception handling 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 <code>catch</code> block
Wrong exception thrown
Exception that should not have been thrown
Wrong encapsulation of exception cause
Lack of a <code>finally</code> block that should exist
Error in the exception assertion
Inconsistency between source code and API
Empty <code>catch</code> block
Error in the definition of exception class
<code>catch</code> block where only a <code>finally</code> would be appropriate

---

Below we explain each exception handling bug classification.

- **Lack of a handler that should exist:** This is the case where an exception should be handled and it is not;
- **Exception not thrown:** This is the case where an exception should be thrown and it is not;
- **Error in the handler:** This is the case where there is an explicit error in the exception handler, in Java it is in the `catch` block;
- **Error in the clean-up action:** This is the case where there is an explicit error in the exception clean-up action, in Java it is in the `finally` block;
- **Exception caught at the wrong level:** This is the case where an exception is caught unintentionally. The exception is being handled by the wrong handler;
- **General `catch` block:** This is a subcase of **Exception caught at the wrong level**. It is the case where the handler catches a general exception, such as `Exception` or `Throwable`, instead of the specific one;
- **Wrong exception thrown:** This is the case where a wrong exception is thrown instead of the correct one;

- **Exception that should not have been thrown:** This is the case where an exception is thrown in a situation where it should not have been;
- **Wrong encapsulation of exception cause:** This is the case where an exception is not encapsulated correctly so it loses the original exception root cause before it is handled;
- **Lack of a `finally` block that should exist:** This is the case where there is no `finally` block in a situation where one should exist;
- **Error in the exception assertion:** This is the case where there is a problem with the exception assertion. It means there is an error in the checking for errors (assertion) in the exception handling code;
- **Inconsistency between source code and API:** This is the case where there is a discrepancy between the source code and the API: the source code is not following the API requirements or the API is not up-to-date with the software functionality;
- **Empty `catch` block:** This is the case where the handler is empty swallowing the exception;
- **Error in the definition of exception class:** This is the case where a definition of exception class is not done correctly. For example, in a situation where `"SomeException extends Error"` when the correct statement is `"SomeException extends RuntimeException"`;
- **`catch` block where only a `finally` would be appropriate:** This is the case where a `catch` block is used instead of a `finally` block when the latter should be more appropriate;

## 4.6 Threats to Validity

In this section we present the threats to the validity of this work. We identified three kinds of threats to the 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. We tried to cover well-known terms that appear in exception handling literature [Goo75, Cri79, AL90]. 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 [SAW12].

There were several bug reports that did not contain enough information to identify whether they referred to exception handling bugs. In some cases, we could mitigate 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-exception handling bugs.

Unlike previous work [SAW12], we did not rely solely on the search string to identify exception handling 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 exception but not exception handling bugs. Because the inspection was manual, two kinds of mistake may have been committed: (i) a regular bug being classified as an exception handling bug; and (ii) an exception handling bug not being classified as such. To reduce the chance of this occurring, the two 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 two 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 chapter 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 [SGH10] involved only 15 respondents (they were interviewed instead of responding to a survey). Therefore, from a comprehensiveness standpoint, we can say that our study is an improvement over the current state-of-the-art.

Another threat is the proportionally low number of exception handling 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 exception handling bugs: they document less and test the system less for their occurrence. Moreover, as highlighted by previous work on exception handling [CM07, CRG<sup>+</sup>08, RM03] and our own results presented in this chapter, many exception handling 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 exception handling 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 exception handling 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 exception handling bugs (see Appendix A).

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.

# 5

## Conclusion

In this work we presented an exploratory study on exception handling bugs based on two complementary approaches: an analysis of the bug repositories of Eclipse and Tomcat and a survey conducted with developers with some experience in Java. With the survey we could get personal perceptions from developers about exception handling and exception handling bugs and compare their answers to data about actual exception handling bugs. Our study has shown that there are many contradictions pertaining to exception handling. We summarize these contradictions and the results for the four research questions of our work below.

For RQ1: "do organizations and developers take exception handling into account?", we perceived from survey's answers that usually organizations do not take exception handling into account however developers seem to do, according to the data observed from the repository analysis. For RQ2: "how commonplace are exception handling bugs?", we could note that exception handling bugs are less frequent than others kind of bugs according to the survey and much less frequent according to repository analysis. For RQ3: "are exception handling bugs harder to fix than other bugs?", we verified that developers assume that fixing exception handling bugs is easier than fixing other bugs, however the repository data suggests that this is not the case. For RQ4: "what are the main causes of exception handling bugs?", we verified from both repository analysis and the survey that the most common causes of exception handling 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.*

Furthermore, many developers seem to think that *empty catch blocks* cause exception handling bugs. Nevertheless, they are often used, even in patches for exception handling bugs. Additionally, developers claim that they use exception handling mainly to improve the quality of the systems they produce. Notwithstanding, exception handling

code is seldom tested or documented and there are usually no organizational policies to guide its development. In addition to this, some causes for bugs that developers consider to be commonplace are rarely found in bug reports.

We also presented a comprehensive classification of exception handling bugs based on the study results. The results of this study emphasize that the views of developers and organizations about exception handling 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 exception handling bugs can provide assistance in this task, e.g., by working as a checklist for code inspections or a guide in the design of test cases.

## 5.1 Related Work

Cristian [Cri89] was the first to emphasize that exception handling code is the least documented, tested, and understood part of source code of an application. Since then, a number of researchers have conducted studies to investigate the extent to which this statement is true and to understand its implications.

To the best of our knowledge, the two studies that are most closely related to our own are the one by Marinescu [Mar11] and the one by Sawadpong et al. [SAW12].

Marinescu [Mar11] 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 take into account the causes of the bugs, and so there is no way to conclude the reason of the defects. It only says that classes that throw or handle exceptions have a higher probability of having defects than other classes. Her study did not attempt to distinguish exception handling and non-exception handling bugs. We looked for exception handling bugs documented by developers and we did not analyze the source code of Eclipse and Tomcat like Marinescu's work, but we complemented the repository analysis with the survey.

Sawadpong et al. [SAW12] performed the first study on exception handling 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 exception handling bugs, performing a search using the keywords "exception", "throw", and "threw". The main finding of the study is that exception handling defect density of exception handling constructs is approximately three times higher than overall defect density. We believe that the definition of exception handling bug ("defect" in their study) employed is not appropriate. 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 (as we discussed in Chapter 3). One would hardly, if ever, classify a typical division by zero or invalid cast as an exception handling bug. As stated in the last section of their paper, "Our goal was to determine whether using exception handling is risky...". However, to verify if using exception handling is risky, the study accounted for bugs that do not involve the use of exception handling.

A number of other studies have attempted to understand developer habits pertaining to exception handling. Shah et al. [SGH10] conducted a study with 8 novices (2 years of development experience on average) and 7 experts (5+ years of professional software development) from the software industry to understand their viewpoints on exception handling. They conducted semi-structured interviews with developers and the results show that novice developers neglect exception handling until there is an error or until they are forced to address due to language requirements. Furthermore, they do not like being forced by the language to use exception handling constructs and most of them use exception handling only for debugging purposes. In contrast, the experienced developers think that exception handling is a very important part of development. In our survey we found some interesting results as we discussed in Chapter 4. It is important to stress that the work of Shah et al. [SGH10] did not focus on exception handling bugs.

Cabral and Marques [CM07] examined 32 systems written in Java (including Eclipse and Tomcat) and C#. Their objective was to understand how developers use exception handling mechanisms by manual examination of the exception handling code of those systems. They 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 (this includes Eclipse) have only 3.11% of exception handling code. Server applications 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 pro-

gram. In contrast, we did not analyze the source code. Instead, we examined bug report data from Eclipse and Tomcat and conducted a survey.

Another previous study that investigated the use of exception mechanisms in Java applications was conducted by Reimer and Srinivasan [RS03]. They 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, (iii) exceptions not being handled at appropriate level, and (iv) logging verbosity in `catch` blocks. In our work we found some of these antipatterns as the causes of the bugs and additionally, to our surprise, as patches for some bugs as we discussed in Chapter 4.

Coelho and colleagues [CRvS<sup>+</sup>08] created a bug pattern catalog for exception handling based on a previous study [CRG<sup>+</sup>08] targeting three applications with both Java and AspectJ versions available. This bug pattern catalog focuses on aspect-oriented languages. Each bug pattern comprises the bug symptoms, causes, a code example, cures, and prevention. According to the authors the most common causes of exception handling bugs in aspect-oriented programs are exceptions thrown and not caught and exceptions caught at the wrong level. Differently from our work, Coelho's work focused on aspect-oriented programs. Moreover, they focused on identifying bugs in the source code, whereas our study analyzes bugs that developers have reported.

Robillard and Murphy [RM03] 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 that 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 overly general `catch` blocks.

More recently, Zhang and Elbaum [ZE12] studied bug reports of five popular open source applications for the Android phone platform. They used the keywords "exception", "throw", and "catch", in the search and obtained 282 bugs. They discovered that almost a third of the bugs that led to code fixes were caused by poor implementation of exception handling constructs. They also introduced an approach aimed at amplifying existing tests to validate exception handling code associated with external resources.

Neither of the two aforementioned studies analyze issues such as whether exception handling bugs are easier to fix. Moreover, only the studies of Coelho et al. [CRG<sup>+</sup>08] and Robillard and Murphy [RM03] attempt to analyze the causes of exception handling



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 exception handling bugs.

## 5.2 Future Work

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 information we cannot get from the survey.

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 [BGM12]. Finally, the empirical results presented in this work, in particular the list of causes of exception handling bugs, can help future research in prediction and localization of exception handling bugs.

# Bibliography

- [AL90] T. Anderson and P. A. Lee. *Fault Tolerance: Principles and Practice*. Springer, 2nd edition, 1990.
- [BGM12] 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.
- [Bla82] Andrew P. Black. *Exception Handling: The Case Against*. PhD thesis, University of Oxford, January 1982.
- [BvDT06] 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.
- [CCF<sup>+</sup>09] 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.
- [CM07] 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.
- [CRG<sup>+</sup>08] 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.
- [Cri79] Flaviu Cristian. A recovery mechanism for modular software. In *Proceedings of the 4th ICSE*, pages 42–51, 1979.
- [Cri89] Flaviu Cristian. Exception handling. In *Dependability of Resilient Computers*, pages 68–97. Blackwell Science, 1989.
- [CRvS<sup>+</sup>08] 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.
- [CvSK<sup>+</sup>11] 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.
- [FLSR10] 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.
- [FR07] 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.
- [GFC<sup>+</sup>09] Robert M Groves, Floyd J Fowler, Mick P Couper, James M Lepkowski, Eleanor Singer, and Roger Tourangeau. *Survey Methodology*. Wiley, 2nd edition, 2009.
- [Goo75] J. B. Goodenough. Exception handling: Issues and a proposed notation. *Communications of the ACM*, 18(12):683–696, December 1975.
- [GRRX01] 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.
- [HN98] Jerry L. Hintze and Ray D. Nelson. Violin plots: A box plot-density trace synergism. *The American Statistician*, 52(2):181–184, May 1998.
- [htt13a] <http://www.bugzilla.org>. The bugzilla guide - 4.2.6 release, May 2013. Address: <http://www.bugzilla.org/docs/4.2/en/html/lifecycle.html> – Last access: May 10th 2013.
-

- [htt13b] <http://www.javatpoint.com>. Exception handling in java, May 2013. Address: <http://www.javatpoint.com/exception-handling-and-checked-and-unchecked-exception> – Last access: May 10th 2013.
- [KCM07] Huzefa Kagdi, Michael L. Collard, and Jonathan I. Maletic. A survey and taxonomy of approaches for mining software repositories in the context of software evolution. *J. Softw. Maint. Evol.*, 19(2):77–131, March 2007.
- [KP08] Barbara Kitchenham and Shari L. Pfleeger. Personal opinion surveys. In Forrest Shull, Janice Singer, and Dag I. K. Sjöberg, editors, *Guide to Advanced Empirical Software Engineering*, pages 63–92, 2008.
- [LS07] 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.
- [Mar11] 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.
- [McC06] 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.
- [MR06] Douglas C. Montgomery and George C. Runger. *Applied Statistics and Probability for Engineers*. Wiley, 2th edition, 2006.
- [MS02] Andreas Muller and Geoffrey Simmons. Exception handling: Common problems and best practice with java 1.4. In *Proceedings of NetObject Days'2002*, October 2002.
- [Nel09] Ian Nelson. Empty catch blocks, June 2009. <http://www.ianfnelson.com/blog/empty-catch-blocks>.
- [R93] R, January 1993. <http://www.r-project.org/>.
- [RM03] 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.
-

- [RS03] 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.
- [Sal07] Dustin Sallings. Empty catch blocks are always wrong, June 2007. <http://www.rockstarprogrammer.org/post/2007/jun/15/empty-catch-blocks-are-always-wrong/>.
- [SAW12] 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.
- [SCA10] 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.
- [SGH10] H.B. Shah, C. Gorg, and M.J. Harrold. Understanding exception handling: Viewpoints of novices and experts. *Software Engineering, IEEE Transactions on*, 36(2):150–161, 2010.
- [Som10] Ian Sommerville. *Software Engineering 9*. Addison-Wesley, 9th edition, 2010.
- [WN08] Westley Weimer and George C. Necula. Exceptional situations and program reliability. *ACM Trans. Program. Lang. Syst.*, 30(2):1–51, 2008.
- [XRR<sup>+</sup>] Jie Xu, Brian Randell, Alexander Romanovsky, Cecilia M F Rubira, Robert J Stroud, and Zhixue Wu. Fault tolerance in concurrent object-oriented software through coordinated error recovery.
- [ZE12] Pingyu Zhang and S. Elbaum. Amplifying tests to validate exception handling code. In *Proceedings of 34th International Conference on Software Engineering*, pages 595–605, June 2012.
- [ZPZ07] 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

## Survey

This Appendix contains the questionnaire used in this study.

Table A.1: Questionnaire about exception handling bugs.

Questionnaire about exception handling bugs
<p><b>This survey's purpose is to help identify aspects related to exception handling bugs (which are caused by code that executes handling instructions): how frequent these bugs happen, how they are documented and how much time it takes to correct them. This work is part of the research of a master program at the Informatics Center of Universidade Federal de Pernambuco advised by Fernando Castor. All participant's identification information will be undisclosed. The estimated time to complete all questions is 15-20 minutes.</b></p> <p><b>To answer this questionnaire please consider the following definition of "bugs related to exception handling code": the bugs that are located specifically within the handlers (catch blocks) or the clean-up action (finally blocks), and also errors arising from any exception that should be being handled (or thrown) and is not being handled (or thrown), thus causing an error in the system.</b></p>
<p><b>1. For how long have you been a Java developer?</b></p> <p><input type="checkbox"/> Less than 2 years</p> <p><input type="checkbox"/> 2 to 5 years</p>
Continued on next page

---

**Table A.1 – continued from previous page**

Questionnaire about exception handling bugs
<input type="checkbox"/> 5 to 7 years <input type="checkbox"/> 7 to 10 years <input type="checkbox"/> More than 10 years
<b>2. What is the approximate size of the project you are currently working on (LoC estimate)? (in case you are not working, please consider your last project you worked on)</b>  <input type="checkbox"/> Less than 20K LOC <input type="checkbox"/> 20K to 50K LOC <input type="checkbox"/> 50K to 100K LOC <input type="checkbox"/> 100K to 200K LOC <input type="checkbox"/> More than 200K LOC
<b>3. Which programming languages have professionally worked with? (you can select more than one answer)</b>  <input type="checkbox"/> JavaScript <input type="checkbox"/> Ruby <input type="checkbox"/> Python <input type="checkbox"/> Java <input type="checkbox"/> PHP <input type="checkbox"/> C <input type="checkbox"/> C++ <input type="checkbox"/> Perl <input type="checkbox"/> Objective-C <input type="checkbox"/> C
<b>4. In the design phase of your projects, what is the importance given to the documentation of exception handling?</b>  <input type="checkbox"/> None
Continued on next page

---



---

**Table A.1 – continued from previous page**

<b>Questionnaire about exception handling bugs</b>
<input type="checkbox"/> Little <input type="checkbox"/> Some <input type="checkbox"/> A lot <input type="checkbox"/> The most
<b>5. Are there any specifications, documented policies or standards that are part of your organization's culture related to the implementation of error handling?</b>  <input type="checkbox"/> Yes <input type="checkbox"/> No
<b>5.1. If you answered YES to the previous question, please describe the policies adopted by your organization:</b>
<b>6. How often are bugs reported at your organization?</b>  <input type="checkbox"/> Never <input type="checkbox"/> Rarely <input type="checkbox"/> Sometimes <input type="checkbox"/> Most of the time <input type="checkbox"/> Always
<b>7. How often are bugs related to exception handling reported at your organization?</b>  <input type="checkbox"/> Never <input type="checkbox"/> Rarely <input type="checkbox"/> Sometimes <input type="checkbox"/> Most of the time <input type="checkbox"/> Always
<b>8. Does your organization use any tool for reporting and keeping</b>
Continued on next page

---

---

**Table A.1 – continued from previous page**

<b>Questionnaire about exception handling bugs</b>
<b>track of bugs?</b>  <input type="checkbox"/> Yes <input type="checkbox"/> No
<b>8.1. If you answered YES in the question above, please describe these tools below:</b>
<b>9. Is there any testing process implemented in your organization?</b>  <input type="checkbox"/> Yes <input type="checkbox"/> No
<b>9.1. If you answered YES to the previous question, please describe this process below:</b>
<b>10. Are there specific tests for the exception handling code in your organization?</b>  <input type="checkbox"/> Yes <input type="checkbox"/> No
<b>11. How often do you find bugs related to exception handling?</b>  <input type="checkbox"/> Never <input type="checkbox"/> Rarely <input type="checkbox"/> Sometimes <input type="checkbox"/> Most of the time <input type="checkbox"/> Always
<b>12. How often do you find bugs that are not related to exception handling?</b>  <input type="checkbox"/> Never <input type="checkbox"/> Rarely <input type="checkbox"/> Sometimes
Continued on next page

---

---

**Table A.1 – continued from previous page**

<b>Questionnaire about exception handling bugs</b>
<input type="checkbox"/> Most of the time <input type="checkbox"/> Always
<b>13. Estimate the percentage of bugs related to exception handling code in your projects: (estimate a value between 0 and 100 %)</b>
<b>14. Have you ever needed to fix bugs related to exception handling?</b>  <input type="checkbox"/> Yes <input type="checkbox"/> No
<b>14.1. If you answered YES to the previous question, please describe some of these situations below:</b>
<b>15. Please select below 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)</b>  <input type="checkbox"/> Lack of a handler that should exist <input type="checkbox"/> No exception thrown in a situation of a known error <input type="checkbox"/> Programming error in the catch block <input type="checkbox"/> Programming error in the finally block Other:
<b>16. What is the average level of difficulty to fix bugs related to exception handling?</b>  <input type="checkbox"/> Very easy <input type="checkbox"/> Easy <input type="checkbox"/> Medium <input type="checkbox"/> Hard <input type="checkbox"/> Very hard
Continued on next page

---

**Table A.1 – continued from previous page**

Questionnaire about exception handling bugs
<p><b>17. What is the average level of difficulty to fix other bugs that are not related to exception handling?</b></p> <p><input type="checkbox"/> Very easy</p> <p><input type="checkbox"/> Easy</p> <p><input type="checkbox"/> Medium</p> <p><input type="checkbox"/> Hard</p> <p><input type="checkbox"/> Very hard</p>
<p><b>18. What is the average priority/severity of reported bugs related to exception handling code?</b></p> <p><input type="checkbox"/> Very low</p> <p><input type="checkbox"/> Low</p> <p><input type="checkbox"/> Medium</p> <p><input type="checkbox"/> High</p> <p><input type="checkbox"/> Very High</p>
<p><b>19. Why do you use exception handling in your projects? (you can select more than one answer)</b></p> <p><input type="checkbox"/> Organization's policies</p> <p><input type="checkbox"/> Language requirement</p> <p><input type="checkbox"/> Importance of feature</p> <p><input type="checkbox"/> Your interest in improving the quality of feature</p> <p><input type="checkbox"/> Your interest in creating ways to tolerate faults</p> <p><input type="checkbox"/> Need to debug a specific part of the code</p> <p><input type="checkbox"/> You do not use exception handling</p> <p>Other:</p>
<p><b>20. What is your opinion about the quality of the code that performs exception handling in your projects compared to the quality of other</b></p>
Continued on next page

---

**Table A.1 – continued from previous page**

<b>Questionnaire about exception handling bugs</b>
<b>parts of the code?</b>  <input type="checkbox"/> Very bad <input type="checkbox"/> Bad <input type="checkbox"/> Reasonable <input type="checkbox"/> Good <input type="checkbox"/> Very good
<b>21. If you would like to know about the results of this questionnaire please fill in your email address in the field below:</b>

# B

## Exception Handling Bugs

This Appendix contains all exception handling bugs found in this study for each system.

### B.1 Exception Handling Bugs of Eclipse

Table B.1: Exception handling bugs of Eclipse.

ID	Classification
258145	Exception not handled
3309	API doc different from code
21203	Error in the handler
20832	Error in the finally block
116223	Exception not handled
6750	Error in the handler
139160	Exception shuldn't be thrown
24134	Exception not thrown
39063	Exception shuldn't be thrown
242292	Exception shuldn't be thrown
40937	Error in the handler
99113	Exception shuldn't be thrown
243715	Exception shuldn't be thrown
363858	Exception not handled
170237	Error in the handler
120865	Exception shuldn't be thrown
Continued on next page	

**Table B.1 – continued from previous page**

<b>ID</b>	<b>Classification</b>
114582	Exception not handled
154667	Exception shuldn't be thrown
129306	Error in the handler
27148	Error in the handler
38495	Error in the handler
298951	Error in the finally block
352447	Exception shuldn't be thrown
35614	Exception not handled
50904	Error in the handler
109118	Exception not handled
120559	Exception not thrown
148970	Exception not handled
306917	General catch block
6528	Exception not handled
6779	Error in the handler
14230	Error in the handler
15182	Exception not handled
21018	Error in the handler
23361	Error in the handler
29790	Exception not handled
31016	Exception not handled
32219	API doc different from code
40020	Exception shuldn't be thrown
47160	Error in the handler
48732	Exception shuldn't be thrown
61075	Error in the handler
63077	Wrong exception thrown
63513	Exception shuldn't be thrown
64299	Error in the handler
67289	Error in the handler
69586	Exception not handled
74934	Exception not thrown
Continued on next page	

**Table B.1 – continued from previous page**

<b>ID</b>	<b>Classification</b>
79395	Wrong exception thrown
94689	Error in the handler
97764	Error in the handler
126727	Exception shuldn't be thrown
132494	Exception not thrown
137619	Exception not handled
138167	Exception not handled
140750	Exception not handled
143013	Exception not handled
143259	Error in the finally block
144450	Error in the handler
178013	Error in the handler
187265	Exception not handled
195823	Error in the handler
196714	Exception not handled
210152	Exception not handled
221723	Empty catch block
222284	Exception not handled
227531	Error in the handler
229480	Error in the handler
230128	Exception not handled
232496	Exception not thrown
245518	Wrong exception thrown
245830	Wrong exception thrown
255974	Wrong exception thrown
284789	Exception not handled
298250	Exception not handled
302455	Exception not handled
305001	Exception not handled
342757	Exception not handled
352665	Error in the handler
398272	Exception not handled
Continued on next page	



**Table B.1 – continued from previous page**

<b>ID</b>	<b>Classification</b>
14588	Exception not handled
41602	API doc different from code
61996	Error in the definition of exception class
392132	Error in the handler
395658	Error in the handler
9395	Error in the finally block
234307	Exception not handled
10679	Exception shuldn't be thrown
37331	Exception not handled
44274	Exception not handled
65237	Error in the handler
95480	Exception not handled

## **B.2 Exception Handling Bugs of Tomcat**

Table B.2: Exception handling bugs of Tomcat.

<b>ID</b>	<b>Classification</b>
13140	Exception not thrown
13138	Exception not thrown
53333	Exception not handled
49240	Exception not handled
42449	Exception not handled
5797	Exception not handled
19238	Exception not handled
46298	Wrong exception thrown
29093	Exception not handled
22691	Exception not handled
48582	Error in the handler
12088	Error in the handler
Continued on next page	

**Table B.2 – continued from previous page**

<b>ID</b>	<b>Classification</b>
40241	General catch block
46967	Exception not handled
52461	Exception not handled
18200	Error in the handler
17687	Error in the handler
16181	Error in the handler
14658	Error in the handler
2500	Error in the handler
43174	Error in the handler
50929	Invalid or non-existent cause associated with exception
5998	Invalid or non-existent cause associated with exception
53421	Invalid or non-existent cause associated with exception
42650	Exception not handled
48644	General catch block
36994	Exception shouldn't be thrown
8013	Exception not handled
5829	Error in the handler
53830	Error in the handler
15404	Exception not thrown
19117	Exception not thrown
16727	Wrong exception thrown
16731	Exception not thrown
17611	Exception not thrown
16129	Exception not thrown
16728	Exception not thrown
17390	Exception not thrown
6196	Wrong exception thrown
49128	Error in the handler
49127	General catch block
54256	Exception not handled
39126	Error in the handler
53545	Error in the handler
Continued on next page	

**Table B.2 – continued from previous page**


<b>ID</b>	<b>Classification</b>
175	Invalid or non-existent cause associated with exception
238	Error in the handler
339	Exception not handled
117	Wrong exception thrown
1316	Error in the handler
644	Error in the handler
434	Error in the handler
41752	Invalid or non-existent cause associated with exception
40133	Wrong exception thrown
2779	Invalid or non-existent cause associated with exception
22113	empty catch block
6130	Exception not handled
8200	Exception not thrown
15754	Exception not thrown
3307	Exception not thrown
18201	Exception not thrown
4023	Error in the handler
19864	Error in the handler
15851	Error in the handler
54087	Exception not handled
53225	Error in the handler
52543	Error in the handler
51893	Wrong exception thrown
48810	Error in the handler
54007	Error in the handler
42434	Error in the handler
48454	Exception not handled
51647	Exception not handled
54284	Wrong exception thrown
53529	Error in the handler
41530	Error in the handler
35984	Error in the handler
Continued on next page	

**Table B.2 – continued from previous page**

<b>ID</b>	<b>Classification</b>
42181	Exception not handled
47499	Error in the handler
51324	Error in the handler
51550	Error in the handler
50571	Error in the handler
48007	Exception not handled
51088	Error in the handler
34080	Invalid or non-existent cause associated with exception
45737	Exception shouldn't be thrown
10907	Error in the handler
26191	Exception not handled
31052	Invalid or non-existent cause associated with exception
3870	Error in the handler
42039	Invalid or non-existent cause associated with exception
54458	Invalid or non-existent cause associated with exception
52091	Error in the finally block
52591	Exception not handled
48179	Error in the handler
51713	Error in the handler
43887	Invalid or non-existent cause associated with exception
44787	Error in the handler
47437	Invalid or non-existent cause associated with exception
12657	Exception not handled
18698	Exception shouldn't be thrown
4501	Wrong exception thrown
5507	Exception not handled
24861	Invalid or non-existent cause associated with exception
24368	Error in the handler
13552	Error in the handler
19049	Invalid or non-existent cause associated with exception
17736	Error in the handler
29387	Error in the handler
Continued on next page	

**Table B.2 – continued from previous page**

<b>ID</b>	<b>Classification</b>
19034	Exception not handled
14283	Exception not handled
4383	Invalid or non-existent cause associated with exception
45603	Error in the handler
15795	Exception shouldn't be thrown
27790	Error in the handler
522	Error in the handler
6332	Invalid or non-existent cause associated with exception
31171	Error in the handler
46223	Wrong exception thrown
18204	Exception not thrown
15967	Exception not thrown
12717	Exception not thrown
12654	Exception not thrown
13094	Exception not thrown
4609	Exception not thrown
15904	Exception not thrown
2655	Exception not thrown
40960	Wrong exception thrown
19114	Invalid or non-existent cause associated with exception



# R Scripts

This Appendix contains all R Scripts used in this study for each system.

## C.1 Eclipse R Script

**Listing C.1** Eclipse.R.

```
1 library (sm)
2 library (vioplot)
3
4 dadosEclipseOthers <- read.csv ("bugs-eclipse-others.out",
   sep="_", header=TRUE, blank.lines.skip=TRUE)
5 dadosEclipseBugs <- read.csv ("bugs-eclipse-92.out", sep="_",
   header=TRUE, blank.lines.skip=TRUE)
6 dadosEclipseSearch <- read.csv ("bugs-eclipse-search.out",
   sep="_", header=TRUE, blank.lines.skip=TRUE)
7
8 eclipseBugOthers <- dadosEclipseOthers$bug_id
9 eclipseCreationDateOthers <- dadosEclipseOthers$creation_
   date
10 eclipseFixDateOthers <- dadosEclipseOthers$fix_date
11 eclipseFixTimeOthers <- dadosEclipseOthers$fix_time
12 eclipseCommentsOthers <- dadosEclipseOthers$comments_
   number
13 eclipseAttachOthers <- dadosEclipseOthers$has_attach
14 eclipsePriorityOthers <- dadosEclipseOthers$priority
```

```
15 eclipseSeverityOthers <- dadosEclipseOthers$severity
16 eclipseStatusOthers <- dadosEclipseOthers$status
17 eclipseResolutionOthers <- dadosEclipseOthers$resolution
18 eclipseAssignedOthers <- dadosEclipseOthers$assigned
19 eclipseReporterOthers <- dadosEclipseOthers$reporter
20
21 eclipseBugBugs <- dadosEclipseBugs$bug_id
22 eclipseCreationDateBugs <- dadosEclipseBugs$creation_date
23 eclipseFixDateBugs <- dadosEclipseBugs$fix_date
24 eclipseFixTimeBugs <- dadosEclipseBugs$fix_time
25 eclipseCommentsBugs <- dadosEclipseBugs$comments_number
26 eclipseAttachBugs <- dadosEclipseBugs$has_attach
27 eclipsePriorityBugs <- dadosEclipseBugs$priority
28 eclipseSeverityBugs <- dadosEclipseBugs$severity
29 eclipseStatusBugs <- dadosEclipseBugs$status
30 eclipseResolutionBugs <- dadosEclipseBugs$resolution
31 eclipseAssignedBugs <- dadosEclipseBugs$assigned
32 eclipseReporterBugs <- dadosEclipseBugs$reporter
33
34 eclipseBugSearch <- dadosEclipseSearch$bug_id
35 eclipseCreationDateSearch <- dadosEclipseSearch$creation_
    date
36 eclipseFixDateSearch <- dadosEclipseSearch$fix_date
37 eclipseFixTimeSearch <- dadosEclipseSearch$fix_time
38 eclipseCommentsSearch <- dadosEclipseSearch$comments_
    number
39 eclipseAttachSearch <- dadosEclipseSearch$has_attach
40 eclipsePrioritySearch <- dadosEclipseSearch$priority
41 eclipseSeveritySearch <- dadosEclipseSearch$severity
42 eclipseStatusSearch <- dadosEclipseSearch$status
43 eclipseResolutionSearch <- dadosEclipseSearch$resolution
44 eclipseAssignedSearch <- dadosEclipseSearch$assigned
45 eclipseReporterSearch <- dadosEclipseSearch$reporter
46
47 ## reporter
```

---

```
48 summary(eclipseReporterOthers)
49 summary(eclipseReporterBugs)
50 summary(eclipseReporterSearch)
51
52 ## assigned
53 summary(eclipseAssignedOthers)
54 summary(eclipseAssignedBugs)
55 summary(eclipseAssignedSearch)
56
57 ## resolution
58 summary(eclipseResolutionOthers)
59 barplot(table(eclipseResolutionOthers), main="Eclipse_
      Resolution", xlab="Others")
60 summary(eclipseResolutionBugs)
61 barplot(table(eclipseResolutionBugs), main="Eclipse_
      Resolution", xlab="Bugs")
62 summary(eclipseResolutionSearch)
63
64 ## status
65 summary(eclipseStatusOthers)
66 barplot(table(eclipseStatusOthers), main="Eclipse_Status",
      xlab="Others")
67 summary(eclipseStatusBugs)
68 barplot(table(eclipseStatusBugs), main="Eclipse_Status",
      xlab="Bugs")
69 summary(eclipseStatusSearch)
70
71 ## messages
72 summary(eclipseCommentsOthers)
73 hist(eclipseCommentsOthers, main="Eclipse_Comments", xlab=
      "Others")
74 boxplot(eclipseCommentsOthers, main="Eclipse_Comments",
      xlab="Others")
75 var(eclipseCommentsOthers)
76 sd(eclipseCommentsOthers)
```



```
77 vioplot(eclipseCommentsOthers , names=c("Others"))
78 title("Eclipse_Comments")
79 summary(eclipseCommentsBugs)
80 hist(eclipseCommentsBugs , main="Eclipse_Comments" , xlab="
    Bugs")
81 boxplot(eclipseCommentsBugs , main="Eclipse_Comments" , xlab
    ="Bugs")
82 var(eclipseCommentsBugs)
83 sd(eclipseCommentsBugs)
84 vioplot(eclipseCommentsBugs , names=c("Bugs"))
85 title("Eclipse_Comments")
86 boxplot(eclipseCommentsOthers , eclipseCommentsBugs , main="
    Eclipse_Comments" , names=c("Others" , "Bugs"))
87 vioplot(eclipseCommentsOthers , eclipseCommentsBugs , names=
    c("Others" , "Bugs"))
88 title("Eclipse_Comments")
89 summary(eclipseCommentsSearch)
90 var(eclipseCommentsSearch)
91 sd(eclipseCommentsSearch)
92
93 ## severity
94 summary(eclipseSeverityOthers)
95 pie(table(eclipseSeverityOthers) , main="Eclipse_Others_
    Severity")
96 barplot(table(eclipseSeverityOthers) , main="Eclipse_
    Severity" , xlab="Others")
97 summary(eclipseSeverityBugs)
98 pie(table(eclipseSeverityBugs) , main="Eclipse_Bugs_
    Severity")
99 barplot(table(eclipseSeverityBugs) , main="Eclipse_Severity
    " , xlab="Bugs")
100 summary(eclipseSeveritySearch)
101
102 ## priority
103 summary(eclipsePriorityOthers)
```

---

```
104 pie(table(eclipsePriorityOthers), main="Eclipse_Others_  
    Priority")  
105 barplot(table(eclipsePriorityOthers), main="Eclipse_  
    Priority", xlab="Others")  
106 summary(eclipsePriorityBugs)  
107 pie(table(eclipsePriorityBugs), main="Eclipse_Bugs_  
    Priority")  
108 barplot(table(eclipsePriorityBugs), main="Eclipse_Priority  
    ", xlab="Bugs")  
109 summary(eclipsePrioritySearch)  
110  
111 ## fix time  
112 summary(eclipseFixTimeOthers)  
113 hist(eclipseFixTimeOthers, main="Eclipse_Others_Fix_Time",  
    xlab="Days")  
114 boxplot(eclipseFixTimeOthers, main="Eclipse_Others_Fix_  
    Time", xlab="Days")  
115 var(eclipseFixTimeOthers)  
116 sd(eclipseFixTimeOthers)  
117 vioplot(eclipseFixTimeOthers, names=c("Others"))  
118 title("Eclipse_Fix_Time")  
119 summary(eclipseFixTimeBugs)  
120 hist(eclipseFixTimeBugs, main="Eclipse_Bugs_Fix_Time",  
    xlab="Days")  
121 boxplot(eclipseFixTimeBugs, main="Eclipse_Bugs_Fix_Time",  
    xlab="Days")  
122 var(eclipseFixTimeBugs)  
123 sd(eclipseFixTimeBugs)  
124 vioplot(eclipseFixTimeBugs, names=c("Bugs"))  
125 title("Eclipse_Fix_Time")  
126 boxplot(eclipseFixTimeOthers, eclipseFixTimeBugs, main="  
    Eclipse_Fix_Time", names=c("Others", "Bugs"))  
127 vioplot(eclipseFixTimeOthers, eclipseFixTimeBugs, names=c(  
    "Others", "Bugs"))  
128 title("Eclipse_Fix_Time")
```

---

```

129 summary(eclipseFixTimeSearch)
130 var(eclipseFixTimeSearch)
131 sd(eclipseFixTimeSearch)
132
133 ## attach
134 summary(eclipseAttachOthers)
135 barplot(table(eclipseAttachOthers)/length(
    eclipseAttachOthers), main="Eclipse_Attachment", xlab="
    Others", names.arg=c("No_attachment", "With_attachment
    "))
136 summary(eclipseAttachBugs)
137 barplot(table(eclipseAttachBugs)/length(eclipseAttachBugs)
    , main="Eclipse_Attachment", xlab="Bugs", names.arg=c("
    No_attachment", "With_attachment"))
138 summary(eclipseAttachSearch)
139
140 ## tests
141 t.test(eclipseFixTimeOthers, eclipseFixTimeBugs)
142 t.test(eclipseFixTimeOthers, eclipseFixTimeBugs,
    alternative = "greater")
143 t.test(eclipseCommentsOthers, eclipseCommentsBugs)
144 t.test(eclipseCommentsOthers, eclipseCommentsBugs,
    alternative = "less")
145 t.test(eclipseCommentsOthers, eclipseCommentsSearch)
146 t.test(eclipseFixTimeOthers, eclipseFixTimeSearch)

```

## C.2 Tomcat R Script

**Listing C.2** Tomcat.R.

```

1 library(sm)
2 library(vioplot)
3
4 dadosTomcatOthers <- read.csv("bugs-tomcat-others.out",
    sep="_", header=TRUE, blank.lines.skip=TRUE)

```

```
5 dadosTomcatBugs <- read.csv("bugs-tomcat-128.out", sep="_"
  , header=TRUE, blank.lines.skip=TRUE)
6 dadosTomcatSearch <- read.csv("bugs-tomcat-search.out",
  sep="_", header=TRUE, blank.lines.skip=TRUE)
7
8 tomcatBugOthers <- dadosTomcatOthers$bug_id
9 tomcatCreationDateOthers <- dadosTomcatOthers$creation_
  date
10 tomcatFixDateOthers <- dadosTomcatOthers$fix_date
11 tomcatFixTimeOthers <- dadosTomcatOthers$fix_time
12 tomcatCommentsOthers <- dadosTomcatOthers$comments_number
13 tomcatAttachOthers <- dadosTomcatOthers$has_attach
14 tomcatPriorityOthers <- dadosTomcatOthers$priority
15 tomcatSeverityOthers <- dadosTomcatOthers$severity
16 tomcatStatusOthers <- dadosTomcatOthers$status
17 tomcatResolutionOthers <- dadosTomcatOthers$resolution
18 tomcatAssignedOthers <- dadosTomcatOthers$assigned
19 tomcatReporterOthers <- dadosTomcatOthers$reporter
20
21 tomcatBugBugs <- dadosTomcatBugs$bug_id
22 tomcatCreationDateBugs <- dadosTomcatBugs$creation_date
23 tomcatFixDateBugs <- dadosTomcatBugs$fix_date
24 tomcatFixTimeBugs <- dadosTomcatBugs$fix_time
25 tomcatCommentsBugs <- dadosTomcatBugs$comments_number
26 tomcatAttachBugs <- dadosTomcatBugs$has_attach
27 tomcatPriorityBugs <- dadosTomcatBugs$priority
28 tomcatSeverityBugs <- dadosTomcatBugs$severity
29 tomcatStatusBugs <- dadosTomcatBugs$status
30 tomcatResolutionBugs <- dadosTomcatBugs$resolution
31 tomcatAssignedBugs <- dadosTomcatBugs$assigned
32 tomcatReporterBugs <- dadosTomcatBugs$reporter
33
34 tomcatBugSearch <- dadosTomcatSearch$bug_id
35 tomcatCreationDateSearch <- dadosTomcatSearch$creation_
  date
```

---

```
36 tomcatFixDateSearch <- dadosTomcatSearch$fix_date
37 tomcatFixTimeSearch <- dadosTomcatSearch$fix_time
38 tomcatCommentsSearch <- dadosTomcatSearch$comments_number
39 tomcatAttachSearch <- dadosTomcatSearch$has_attach
40 tomcatPrioritySearch <- dadosTomcatSearch$priority
41 tomcatSeveritySearch <- dadosTomcatSearch$severity
42 tomcatStatusSearch <- dadosTomcatSearch$status
43 tomcatResolutionSearch <- dadosTomcatSearch$resolution
44 tomcatAssignedSearch <- dadosTomcatSearch$assigned
45 tomcatReporterSearch <- dadosTomcatSearch$reporter
46
47 ## reporter
48 summary(tomcatReporterOthers)
49 summary(tomcatReporterBugs)
50 summary(tomcatReporterSearch)
51
52 ## assigned
53 summary(tomcatAssignedOthers)
54 summary(tomcatAssignedBugs)
55 summary(tomcatAssignedSearch)
56
57 ## resolution
58 summary(tomcatResolutionOthers)
59 barplot(table(tomcatResolutionOthers), main="Tomcat_
    Resolution", xlab="Others")
60 summary(tomcatResolutionBugs)
61 barplot(table(tomcatResolutionBugs), main="Tomcat_
    Resolution", xlab="Bugs")
62 summary(tomcatResolutionSearch)
63
64 ## status
65 summary(tomcatStatusOthers)
66 barplot(table(tomcatStatusOthers), main="Tomcat_Status",
    xlab="Others")
67 summary(tomcatStatusBugs)
```

---

```
68 barplot( table( tomcatStatusBugs ), main="Tomcat_Status" ,  
          xlab="Bugs" )  
69 summary( tomcatStatusSearch )  
70  
71 ## messages  
72 summary( tomcatCommentsOthers )  
73 hist( tomcatCommentsOthers , main="Tomcat_Comments" , xlab="  
        Others" )  
74 boxplot( tomcatCommentsOthers , main="Tomcat_Comments" , xlab=  
          =" Others" )  
75 var( tomcatCommentsOthers )  
76 sd( tomcatCommentsOthers )  
77 vioplot( tomcatCommentsOthers , names=c( " Others" ))  
78 title( "Tomcat_Comments" )  
79 summary( tomcatCommentsBugs )  
80 hist( tomcatCommentsBugs , main="Tomcat_Comments" , xlab="  
        Bugs" )  
81 boxplot( tomcatCommentsBugs , main="Tomcat_Comments" , xlab="  
        Bugs" )  
82 var( tomcatCommentsBugs )  
83 sd( tomcatCommentsBugs )  
84 vioplot( tomcatCommentsBugs , names=c( " Bugs" ))  
85 title( "Tomcat_Comments" )  
86 boxplot( tomcatCommentsOthers , tomcatCommentsBugs , main="Tomcat_Comments" , names=c( " Others" , " Bugs" ))  
87 vioplot( tomcatCommentsOthers , tomcatCommentsBugs , names=c(  
          " Others" , " Bugs" ))  
88 title( "Tomcat_Comments" )  
89 summary( tomcatCommentsSearch )  
90 var( tomcatCommentsSearch )  
91 sd( tomcatCommentsSearch )  
92  
93 ## severity  
94 summary( tomcatSeverityOthers )
```

```
95 pie ( table (tomcatSeverityOthers) , main="Tomcat_Others_
      Severity ")
96 barplot ( table (tomcatSeverityOthers) , main="Tomcat_Severity
      " , xlab="Others" )
97 summary (tomcatSeverityBugs)
98 pie ( table (tomcatSeverityBugs) , main="Tomcat_Bugs_Severity "
      )
99 barplot ( table (tomcatSeverityBugs) , main="Tomcat_Severity " ,
      xlab="Bugs" )
100 summary (tomcatSeveritySearch)
101
102 ## priority
103 summary (tomcatPriorityOthers)
104 pie ( table (tomcatPriorityOthers) , main="Tomcat_Others_
      Priority ")
105 barplot ( table (tomcatPriorityOthers) , main="Tomcat_Priority
      " , xlab="Others" )
106 summary (tomcatPriorityBugs)
107 pie ( table (tomcatPriorityBugs) , main="Tomcat_Bugs_Priority "
      )
108 barplot ( table (tomcatPriorityBugs) , main="Tomcat_Priority " ,
      xlab="Bugs" )
109 summary (tomcatPrioritySearch)
110
111 ## fix time
112 summary (tomcatFixTimeOthers)
113 hist (tomcatFixTimeOthers , main="Tomcat_Others_Fix_Time" ,
      xlab="Days" )
114 boxplot (tomcatFixTimeOthers , main="Tomcat_Others_Fix_Time"
      , xlab="Days" )
115 var (tomcatFixTimeOthers)
116 sd (tomcatFixTimeOthers)
117 vioplot (tomcatFixTimeOthers , names=c ( " Others" ))
118 title ( "Tomcat_Fix_Time" )
119 summary (tomcatFixTimeBugs)
```

---

```
120 hist(tomcatFixTimeBugs , main="Tomcat_Bugs_Fix_Time" , xlab="
    "Days")
121 boxplot(tomcatFixTimeBugs , main="Tomcat_Bugs_Fix_Time" ,
    xlab="Days")
122 var(tomcatFixTimeBugs)
123 sd(tomcatFixTimeBugs)
124 vioplot(tomcatFixTimeBugs , names=c( "Bugs" ))
125 title ( "Tomcat_Fix_Time" )
126 boxplot(tomcatFixTimeOthers , tomcatFixTimeBugs , main="
    Tomcat_Fix_Time" , names=c( "Others" , "Bugs" ))
127 vioplot(tomcatFixTimeOthers , tomcatFixTimeBugs , names=c( "
    Others" , "Bugs" ))
128 title ( "Tomcat_Fix_Time" )
129 summary(tomcatFixTimeSearch)
130 var(tomcatFixTimeSearch)
131 sd(tomcatFixTimeSearch)
132
133 ## attach
134 summary(tomcatAttachOthers)
135 barplot( table(tomcatAttachOthers)/length(
    tomcatAttachOthers) , main="Tomcat_Attachment" , xlab="
    Others" , names.arg=c( "No_attachement" , "With_attachment
    " ))
136 summary(tomcatAttachBugs)
137 barplot( table(tomcatAttachBugs)/length(tomcatAttachBugs) ,
    main="Tomcat_Attachment" , xlab="Bugs" , names.arg=c( "No_
    attachment" , "With_attachment" ))
138 summary(tomcatAttachSearch)
139
140 ## tests
141 t.test(tomcatFixTimeOthers , tomcatFixTimeBugs)
142 t.test(tomcatFixTimeOthers , tomcatFixTimeBugs , alternative
    = "greater")
143 t.test(tomcatFixTimeOthers , tomcatFixTimeBugs , alternative
    = "less")
```

---



```
144 t.test(tomcatCommentsOthers, tomcatCommentsBugs)
145 t.test(tomcatCommentsOthers, tomcatCommentsBugs,
         alternative = "greater")
146 t.test(tomcatCommentsOthers, tomcatCommentsBugs,
         alternative = "less")
147 t.test(tomcatFixTimeOthers, tomcatFixTimeSearch)
148 t.test(tomcatCommentsOthers, tomcatCommentsSearch)
```

---